# OnlyCats → Supabase: Full Build Playbook (Auth, Storage, Realtime Likes/Comments/DMs)

This is the comprehensive, copy‑paste‑friendly guide to take your static OnlyCats HTML into a real social app backed by Supabase (Postgres + Auth + Storage + Realtime). Each section has: (A) the exact steps and code, and (B) a short plain-English mental model to keep things clear for brains with ADHD, dyslexia, or autism (hi fam).

## Quick Table of Contents

* 1) Supabase project + CLI
* 2) Database schema (two options: recommended Profiles vs your Users table)
* 3) RLS policies (secure-by-default)
* 4) Storage buckets (public posts, private avatars)
* 5) Front-end wiring (auth, feed, posts, likes, comments, follows, DMs)
* 6) Realtime subscriptions (see changes instantly)
* 7) Search (ILike + trigram indexing)
* 8) Hosting (Supabase static hosting or GitHub Pages)
* 9) Testing checklist
* 10) Extras: counters triggers, seed data, environment setup

## 1) Create your Supabase project (+ optional CLI)

1. Go to the Supabase dashboard → New Project. Name: onlycats. Pick a nearby region.
2. Copy your SUPABASE\_URL and anon key (Settings → API). Never ship the service role key to the browser.
3. Optional: install CLI → `npm install -g supabase`, then in your repo run `supabase init`.

### Plain-English mental model — What’s happening

Supabase gives you a managed Postgres + Auth + Storage + Realtime server. Think: one backend for everything.  
URL + anon key lets your front-end talk to this backend.

## 2) Database schema

You proposed a `users` table with a password hash. With Supabase Auth you \*\*don’t\*\* store password hashes in `public`. There are two safe approaches below. Use \*\*Option A (recommended)\*\* unless you must keep your original.

### Option A (recommended): `auth.users` for credentials + `public.profiles` for public data

-- Enable useful extensions (often already enabled on Supabase)  
create extension if not exists "uuid-ossp";  
create extension if not exists pg\_trgm;  
  
-- Public profile table (1 row per auth user)  
create table if not exists public.profiles (  
 id uuid primary key references auth.users(id) on delete cascade,  
 username text unique not null check (char\_length(username) between 3 and 30),  
 display\_name text,  
 bio text,  
 avatar\_url text,  
 created\_at timestamptz not null default now()  
);  
  
-- Follow graph  
create table if not exists public.follows (  
 follower uuid not null references public.profiles(id) on delete cascade,  
 following uuid not null references public.profiles(id) on delete cascade,  
 created\_at timestamptz not null default now(),  
 primary key (follower, following),  
 check (follower <> following)  
);  
  
-- Posts  
create table if not exists public.posts (  
 id uuid primary key default uuid\_generate\_v4(),  
 author uuid not null references public.profiles(id) on delete cascade,  
 caption text not null,  
 media\_url text,  
 media\_type text check (media\_type in ('image','video', 'none')),  
 likes\_count int not null default 0,  
 comments\_count int not null default 0,  
 created\_at timestamptz not null default now()  
);  
  
-- Comments (supports threads via parent\_comment)  
create table if not exists public.comments (  
 id uuid primary key default uuid\_generate\_v4(),  
 post\_id uuid not null references public.posts(id) on delete cascade,  
 author uuid not null references public.profiles(id) on delete cascade,  
 content text not null,  
 parent\_comment uuid references public.comments(id) on delete cascade,  
 created\_at timestamptz not null default now()  
);  
  
-- Likes (either post OR comment; not both at once)  
create table if not exists public.likes (  
 id uuid primary key default uuid\_generate\_v4(),  
 user\_id uuid not null references public.profiles(id) on delete cascade,  
 post\_id uuid references public.posts(id) on delete cascade,  
 comment\_id uuid references public.comments(id) on delete cascade,  
 created\_at timestamptz not null default now(),  
 check ((post\_id is not null) <> (comment\_id is not null))  
);  
create unique index if not exists uq\_like\_post on public.likes(user\_id, post\_id) where post\_id is not null;  
create unique index if not exists uq\_like\_comment on public.likes(user\_id, comment\_id) where comment\_id is not null;  
  
-- Messages (DMs)  
create table if not exists public.messages (  
 id uuid primary key default uuid\_generate\_v4(),  
 sender uuid not null references public.profiles(id) on delete cascade,  
 receiver uuid not null references public.profiles(id) on delete cascade,  
 content text not null,  
 created\_at timestamptz not null default now()  
);  
  
-- Helpful indexes  
create index if not exists idx\_posts\_created\_at on public.posts(created\_at desc);  
create index if not exists idx\_comments\_post\_id\_created on public.comments(post\_id, created\_at);  
create index if not exists idx\_messages\_pair\_created on public.messages(sender, receiver, created\_at);  
create index if not exists idx\_profiles\_username\_trgm on public.profiles using gin (username gin\_trgm\_ops);

### Plain-English mental model — Mental model

auth.users = private login credentials. profiles = public display info linked by the same UUID. Everything else references profiles.id. Safer and cleaner.

### Option B (matches your draft): single `public.users` table (not recommended)

create extension if not exists "uuid-ossp";  
create table if not exists public.users (  
 id uuid primary key default uuid\_generate\_v4(),  
 username text unique not null,  
 email text unique not null,  
 -- Avoid storing password hashes in public when using Supabase Auth.  
 -- If you insist on this layout, DO NOT use Supabase Auth; handle passwords yourself (not advised).  
 password\_hash text,  
 display\_name text,  
 bio text,  
 avatar\_url text,  
 created\_at timestamptz not null default now()  
);  
  
-- Then port the rest of the schema by replacing profiles(...) with users(...)

## 3) Row‑Level Security (RLS) policies

Enable RLS, then add policies so users can read the world but only write their own stuff.

-- Enable RLS  
alter table public.profiles enable row level security;  
alter table public.follows enable row level security;  
alter table public.posts enable row level security;  
alter table public.comments enable row level security;  
alter table public.likes enable row level security;  
alter table public.messages enable row level security;  
  
-- Helpful predicate  
create or replace function auth.uid() returns uuid language sql stable as $$  
 select nullif(current\_setting('request.jwt.claims', true)::jsonb ->> 'sub','')::uuid  
$$;  
  
-- PROFILES  
create policy "Anyone can read profiles" on public.profiles  
 for select using (true);  
create policy "Users can insert their profile" on public.profiles  
 for insert with check (id = auth.uid());  
create policy "Users can update their own profile" on public.profiles  
 for update using (id = auth.uid());  
  
-- FOLLOWS  
create policy "Read follows" on public.follows  
 for select using (true);  
create policy "Follow someone" on public.follows  
 for insert with check (follower = auth.uid());  
create policy "Unfollow" on public.follows  
 for delete using (follower = auth.uid());  
  
-- POSTS  
create policy "Read posts" on public.posts  
 for select using (true);  
create policy "Create post" on public.posts  
 for insert with check (author = auth.uid());  
create policy "Edit own post" on public.posts  
 for update using (author = auth.uid());  
create policy "Delete own post" on public.posts  
 for delete using (author = auth.uid());  
  
-- COMMENTS  
create policy "Read comments" on public.comments  
 for select using (true);  
create policy "Create comment" on public.comments  
 for insert with check (author = auth.uid());  
create policy "Edit own comment" on public.comments  
 for update using (author = auth.uid());  
create policy "Delete own comment" on public.comments  
 for delete using (author = auth.uid());  
  
-- LIKES  
create policy "Read likes" on public.likes  
 for select using (true);  
create policy "Like/unlike by self" on public.likes  
 for insert with check (user\_id = auth.uid());  
create policy "Delete own like" on public.likes  
 for delete using (user\_id = auth.uid());  
  
-- MESSAGES  
create policy "Read own conversations" on public.messages  
 for select using (sender = auth.uid() or receiver = auth.uid());  
create policy "Send message" on public.messages  
 for insert with check (sender = auth.uid());  
create policy "Delete own message (optional)" on public.messages  
 for delete using (sender = auth.uid());

### Plain-English mental model — Mental model

RLS = nightclub bouncer at the database door. These rules say: you can see public stuff, but you only write rows that belong to your user ID.

## 4) Storage for images (posts + avatars)

1. Create bucket `cats` in Storage.
2. For public post media, set bucket to Public. For avatars, keep private and use signed URLs.
3. Folder convention: posts/{post\_id}/file.ext and avatars/{user\_id}.jpg

// JS: upload image then insert post  
const file = fileInput.files[0];  
const postId = crypto.randomUUID();  
const path = `posts/${postId}/${file.name}`;  
  
const { data: up, error: upErr } = await supabase.storage  
 .from('cats')  
 .upload(path, file, { upsert: true });  
if (upErr) throw upErr;  
  
const { data: urlData } = supabase.storage.from('cats').getPublicUrl(path);  
const mediaUrl = urlData.publicUrl;  
  
await supabase.from('posts').insert({  
 id: postId,  
 author: (await supabase.auth.getUser()).data.user.id,  
 caption,  
 media\_url: mediaUrl,  
 media\_type: 'image'  
});

### Plain-English mental model — Mental model

Storage = cloud disk. Public = anyone can see that URL. Private = you hand out time‑limited signed URLs only.

## 5) Front‑end wiring (auth, feed, posts, likes, comments, follows, DMs)

### Include Supabase client (CDN ESM)

<script type="module">  
 import { createClient } from 'https://cdn.jsdelivr.net/npm/@supabase/supabase-js/+esm';  
 const supabase = createClient('https://YOUR\_PROJECT.supabase.co', 'YOUR\_ANON\_KEY');  
</script>

### Auth (sign up, login, session)

// Sign up  
async function signUp(email, password, username, displayName) {  
 const { data: sign, error } = await supabase.auth.signUp({ email, password });  
 if (error) throw error;  
 // Create profile row  
 const uid = sign.user.id;  
 await supabase.from('profiles').insert({ id: uid, username, display\_name: displayName });  
}  
  
// Login  
async function login(email, password) {  
 const { error } = await supabase.auth.signInWithPassword({ email, password });  
 if (error) throw error;  
}  
  
// Listen for auth state  
supabase.auth.onAuthStateChange(async (event, session) => {  
 if (session?.user) {  
 // render app UI  
 } else {  
 // render login UI  
 }  
});

### Feed (fetch posts + author info)

// Thanks to FK metadata, you can select nested author fields  
const { data: posts, error } = await supabase  
 .from('posts')  
 .select('id, caption, media\_url, created\_at, likes\_count, comments\_count, author:profiles(id, username, display\_name, avatar\_url)')  
 .order('created\_at', { ascending: false });

### Create post (with or without image)

async function createPost(caption, maybeFile) {  
 const uid = (await supabase.auth.getUser()).data.user.id;  
 let mediaUrl = null, mediaType = 'none';  
  
 if (maybeFile) {  
 const postId = crypto.randomUUID();  
 const path = `posts/${postId}/${maybeFile.name}`;  
 const { error: upErr } = await supabase.storage.from('cats').upload(path, maybeFile, { upsert: true });  
 if (upErr) throw upErr;  
 mediaUrl = supabase.storage.from('cats').getPublicUrl(path).data.publicUrl;  
 mediaType = 'image';  
 }  
  
 const { error } = await supabase.from('posts').insert({ author: uid, caption, media\_url: mediaUrl, media\_type: mediaType });  
 if (error) throw error;  
}

### Likes (toggle)

async function toggleLike(postId) {  
 const uid = (await supabase.auth.getUser()).data.user.id;  
 const { data: existing } = await supabase.from('likes').select('id').eq('user\_id', uid).eq('post\_id', postId).maybeSingle();  
  
 if (existing) {  
 await supabase.from('likes').delete().eq('id', existing.id);  
 } else {  
 await supabase.from('likes').insert({ user\_id: uid, post\_id: postId });  
 }  
}

### Comments (add)

async function addComment(postId, content, parent=null) {  
 const uid = (await supabase.auth.getUser()).data.user.id;  
 const { error } = await supabase.from('comments').insert({ post\_id: postId, author: uid, content, parent\_comment: parent });  
 if (error) throw error;  
}

### Follow / Unfollow

async function follow(userId) {  
 const me = (await supabase.auth.getUser()).data.user.id;  
 await supabase.from('follows').insert({ follower: me, following: userId });  
}  
async function unfollow(userId) {  
 const me = (await supabase.auth.getUser()).data.user.id;  
 await supabase.from('follows').delete().eq('follower', me).eq('following', userId);  
}

### DMs (conversation query)

async function sendMessage(toUserId, text) {  
 const me = (await supabase.auth.getUser()).data.user.id;  
 await supabase.from('messages').insert({ sender: me, receiver: toUserId, content: text });  
}  
  
async function getConversation(otherId) {  
 const me = (await supabase.auth.getUser()).data.user.id;  
 const { data } = await supabase.from('messages')  
 .select('\*')  
 .or(`and(sender.eq.${me},receiver.eq.${otherId}),and(sender.eq.${otherId},receiver.eq.${me})`)  
 .order('created\_at', { ascending: true });  
 return data;  
}

### Plain-English mental model — Mental model

Front‑end = your OnlyCats HTML + JS. Supabase client is the bridge that sends/receives data. You’ll wire button clicks to these functions.

## 6) Realtime subscriptions

1. Ensure Realtime is enabled for your project.
2. In Database → Replication → Realtime, toggle on posts, comments, likes, messages (if needed).

// New posts  
supabase.channel('rt-posts')  
 .on('postgres\_changes', { event: 'INSERT', schema: 'public', table: 'posts' }, payload => {  
 const post = payload.new;  
 // prepend to feed  
 })  
 .subscribe();  
  
// Likes on a specific post  
supabase.channel('rt-likes')  
 .on('postgres\_changes', { event: '\*', schema: 'public', table: 'likes' }, payload => {  
 // update like count UI if it matches the open post  
 })  
 .subscribe();  
  
// New comments  
supabase.channel('rt-comments')  
 .on('postgres\_changes', { event: 'INSERT', schema: 'public', table: 'comments' }, payload => {  
 // push into comment list  
 })  
 .subscribe();  
  
// DMs for me  
const me = (await supabase.auth.getUser()).data.user.id;  
supabase.channel('rt-dms')  
 .on('postgres\_changes', { event: 'INSERT', schema: 'public', table: 'messages', filter: `receiver=eq.${me}` }, payload => {  
 // show incoming message toast  
 })  
 .subscribe();

### Plain-English mental model — Mental model

Realtime = Supabase pushes DB changes over websockets, so your UI updates like magic.

## 7) Search (simple + fast)

// Simple case-insensitive search by username substring  
async function searchUsers(term) {  
 const { data } = await supabase  
 .from('profiles')  
 .select('id, username, display\_name, avatar\_url')  
 .ilike('username', `%${term}%`);  
 return data;  
}

-- Speed up search using trigram index (already added earlier)  
create extension if not exists pg\_trgm;  
create index if not exists idx\_profiles\_username\_trgm on public.profiles using gin (username gin\_trgm\_ops);

### Plain-English mental model — Mental model

ILike = contains search. Trigram index = speed boost so it stays snappy.

## 8) Hosting your front‑end

* Option A: Supabase Storage static hosting (upload your /public folder to a bucket and serve).
* Option B: GitHub Pages/Netlify/Vercel (any static host). Your app talks to Supabase via URL + anon key.

### Minimal static deploy script (Storage)

// deploy.js  
import { createClient } from 'https://cdn.jsdelivr.net/npm/@supabase/supabase-js/+esm';  
  
const SUPABASE\_URL = process.env.SUPABASE\_URL;  
const SUPABASE\_KEY = process.env.SUPABASE\_SERVICE\_ROLE; // use service role only in local deploy script, never ship to browser  
const BUCKET = 'site';  
  
const supabase = createClient(SUPABASE\_URL, SUPABASE\_KEY);  
import fs from 'fs'; import path from 'path';  
  
async function uploadDir(dir, prefix='') {  
 const files = fs.readdirSync(dir);  
 for (const f of files) {  
 const full = path.join(dir, f);  
 const stat = fs.statSync(full);  
 if (stat.isDirectory()) await uploadDir(full, `${prefix}${f}/`);  
 else {  
 const data = fs.readFileSync(full);  
 const { error } = await supabase.storage.from(BUCKET).upload(`${prefix}${f}`, data, { upsert: true });  
 if (error) console.error('Upload error', f, error.message);  
 }  
 }  
}  
await uploadDir('public/');  
console.log('Deployed to Storage bucket. Make the bucket public and open index.html via the public URL.');

## 9) Testing checklist (grab classmates, open 2 browsers)

* Sign up 2+ users, set usernames/avatars.
* Post an image and a text-only post. Confirm both render.
* Like/unlike from both users. Watch counts update in realtime.
* Comment and reply; verify nested threads show up.
* Follow/unfollow and check follower counts.
* Send DMs; confirm messages stream in realtime to the receiver tab.
* Search users by partial username; confirm results.
* Refresh; ensure sessions persist and RLS still allows your operations.

## 10) Extras & polish

### 10.1 Auto-create profile row after signup (edge function or DB trigger)

-- DB trigger to create a profile row when a new auth user registers  
create or replace function public.handle\_new\_user()  
returns trigger as $$  
begin  
 insert into public.profiles (id, username, display\_name)  
 values (new.id, split\_part(new.email, '@', 1), split\_part(new.email, '@', 1));  
 return new;  
end;  
$$ language plpgsql security definer;  
  
drop trigger if exists on\_auth\_user\_created on auth.users;  
create trigger on\_auth\_user\_created  
after insert on auth.users  
for each row execute procedure public.handle\_new\_user();

### Plain-English mental model — Mental model

As soon as a new account is created in auth, we make a matching public profile row automatically.

### 10.2 Keep likes\_count & comments\_count in sync (triggers)

-- Likes counters  
create or replace function public.bump\_likes() returns trigger as $$  
begin  
 update public.posts set likes\_count = likes\_count + 1 where id = new.post\_id;  
 return null;  
end; $$ language plpgsql;  
  
create or replace function public.drop\_likes() returns trigger as $$  
begin  
 update public.posts set likes\_count = greatest(likes\_count - 1, 0) where id = old.post\_id;  
 return null;  
end; $$ language plpgsql;  
  
drop trigger if exists trg\_like\_insert on public.likes;  
create trigger trg\_like\_insert after insert on public.likes for each row execute procedure public.bump\_likes();  
  
drop trigger if exists trg\_like\_delete on public.likes;  
create trigger trg\_like\_delete after delete on public.likes for each row execute procedure public.drop\_likes();  
  
-- Comments counters  
create or replace function public.bump\_comments() returns trigger as $$  
begin  
 update public.posts set comments\_count = comments\_count + 1 where id = new.post\_id;  
 return null;  
end; $$ language plpgsql;  
  
create or replace function public.drop\_comments() returns trigger as $$  
begin  
 update public.posts set comments\_count = greatest(comments\_count - 1, 0) where id = old.post\_id;  
 return null;  
end; $$ language plpgsql;  
  
drop trigger if exists trg\_comment\_insert on public.comments;  
create trigger trg\_comment\_insert after insert on public.comments for each row execute procedure public.bump\_comments();  
  
drop trigger if exists trg\_comment\_delete on public.comments;  
create trigger trg\_comment\_delete after delete on public.comments for each row execute procedure public.drop\_comments();

### 10.3 Seed data (safe for dev)

-- Create two fake users (Option A expects auth signups; for quick demo, you can insert profiles directly with known UUIDs)  
insert into public.profiles (id, username, display\_name, avatar\_url)  
values  
 ('11111111-1111-1111-1111-111111111111','starboy','Starboy The Ponderer','https://picsum.photos/seed/cat1/200'),  
 ('22222222-2222-2222-2222-222222222222','amethyst','Amethyst Girl','https://picsum.photos/seed/cat2/200');  
  
insert into public.posts (author, caption, media\_url, media\_type)  
values  
 ('11111111-1111-1111-1111-111111111111','Hello from OnlyCats!','https://picsum.photos/seed/catA/600','image'),  
 ('22222222-2222-2222-2222-222222222222','Text-only vibes', null,'none');

### 10.4 Environment setup example

# .env (used locally by tools/deploy scripts; never commit secrets)  
SUPABASE\_URL=https://YOUR\_PROJECT.supabase.co  
SUPABASE\_ANON=YOUR\_ANON\_KEY  
SUPABASE\_SERVICE\_ROLE=YOUR\_SERVICE\_ROLE\_KEY # only for server-side scripts, not for browser

### 10.5 Hook into your existing OnlyCats HTML skeleton

Add this `<script type="module">…</script>` at the bottom of your HTML to connect buttons to Supabase. Replace IDs to match your DOM.

<script type="module">  
 import { createClient } from 'https://cdn.jsdelivr.net/npm/@supabase/supabase-js/+esm';  
 const supabase = createClient('https://YOUR\_PROJECT.supabase.co', 'YOUR\_ANON\_KEY');  
  
 // Example: wire up a form to create posts  
 const form = document.querySelector('#create-post-form');  
 const fileInput = document.querySelector('#file-input');  
 form.addEventListener('submit', async (e) => {  
 e.preventDefault();  
 const caption = document.querySelector('#post-caption').value.trim();  
 const file = fileInput?.files?.[0] ?? null;  
 await createPost(caption, file);  
 });  
  
 async function createPost(caption, maybeFile) {  
 const uid = (await supabase.auth.getUser()).data.user.id;  
 let mediaUrl = null, mediaType = 'none';  
 if (maybeFile) {  
 const postId = crypto.randomUUID();  
 const path = `posts/${postId}/${maybeFile.name}`;  
 const { error: upErr } = await supabase.storage.from('cats').upload(path, maybeFile, { upsert: true });  
 if (!upErr) mediaUrl = supabase.storage.from('cats').getPublicUrl(path).data.publicUrl, mediaType = 'image';  
 }  
 const { error } = await supabase.from('posts').insert({ author: uid, caption, media\_url: mediaUrl, media\_type: mediaType });  
 if (error) alert(error.message);  
 }  
</script>

You’re set. Run the SQL in Supabase → SQL Editor, turn on the RLS policies, configure Storage, then wire your front-end using the snippets above.