Falar Documentation - Installation And Setup

Thanks for choosing Falar! This documentation elaborates on the installation and setup process for running Falar.

Firstly, the code has to be extracted to a folder of your choice. For the remainder of this guide, it is assumed that the code is extracted in a folder called **FalarApp**. Also remember to follow the steps in order as the guide is written in a way were some steps assume the previous steps are done in order to work properly,

The following are prerequisites before you start this setup:

- An IDE of your choice (Preferably **VSCode**)
- **NodeJS** and **NPM** (https://nodejs.org/en/download)
- A **Supabase** Project (Self Hosted or on the Supabase Cloud (https://supabase.io)

1. **Install the required dependencies.**

   Dependencies are external open-source packages that the project uses, To install the required dependencies, navigate to the root of the project (in this case, FalarApp) and run the following on a terminal/command prompt:

   ```
   npm install -force
   ```

   After the installation is completed, you can run your project. It will still produce erors since we have not set up the backend yet.

2. **Setup the Environment Variables**

   Before running the project, rename the .env.example file to .env

   Fill in the details of the 3 environment variables in the file.

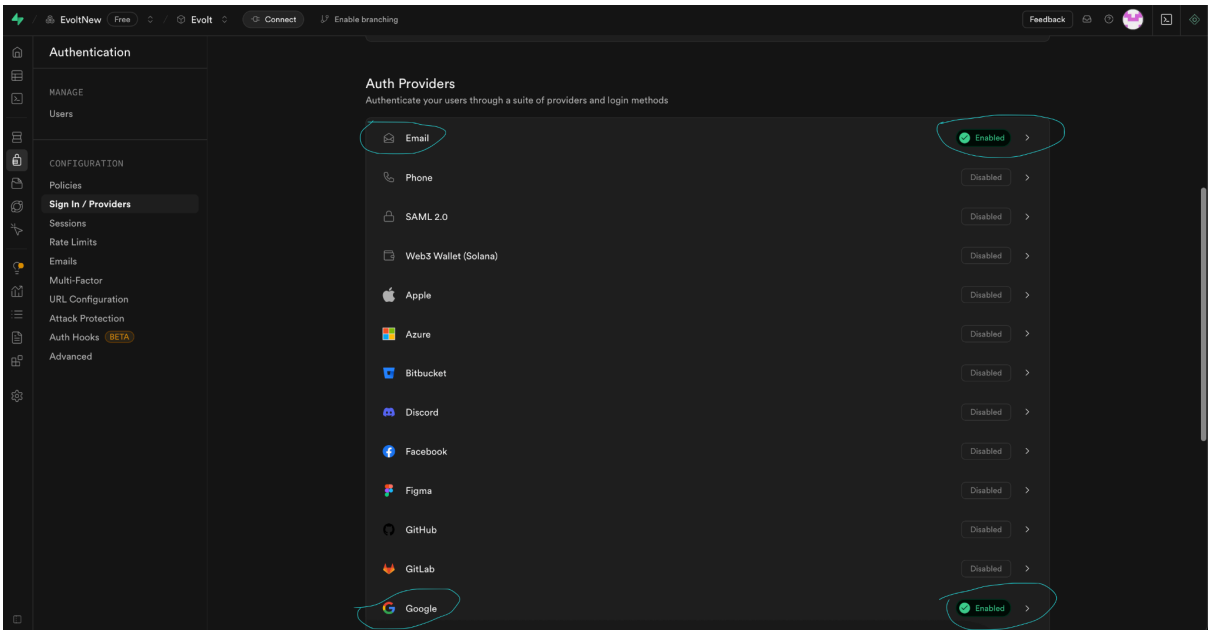
   `NEXT_PUBLIC_SUPABASE_URL` and `NEXT_PUBLIC_SUPABASE_ANON_KEY` can be retrieved from the Supabase dashboard of your project. Follow the official supabase documentation for the same.
   (https://supabase.com/docs/reference/javascript/initializing)

   `TENOR_API_KEY` has to be generated from the Google Cloud Console. Tenor is used in Falar for the GIF feature. Create and setup a project in the Google Cloud Console. We will later use it to provide Google OAuth login and user management for the app.

   Link to Google Cloud Console: https://console.cloud.google.com/
   Link to Tenor API Key: https://developers.google.com/tenor/guides/quickstart
3. **Login Configuration**

The next step is to configure the project properly in the Supabase console. First step in the set-up process to complete is the authentication flow.

For this, firstly enable Email/Password and Google OAuth in the auth providers as shown in the image:



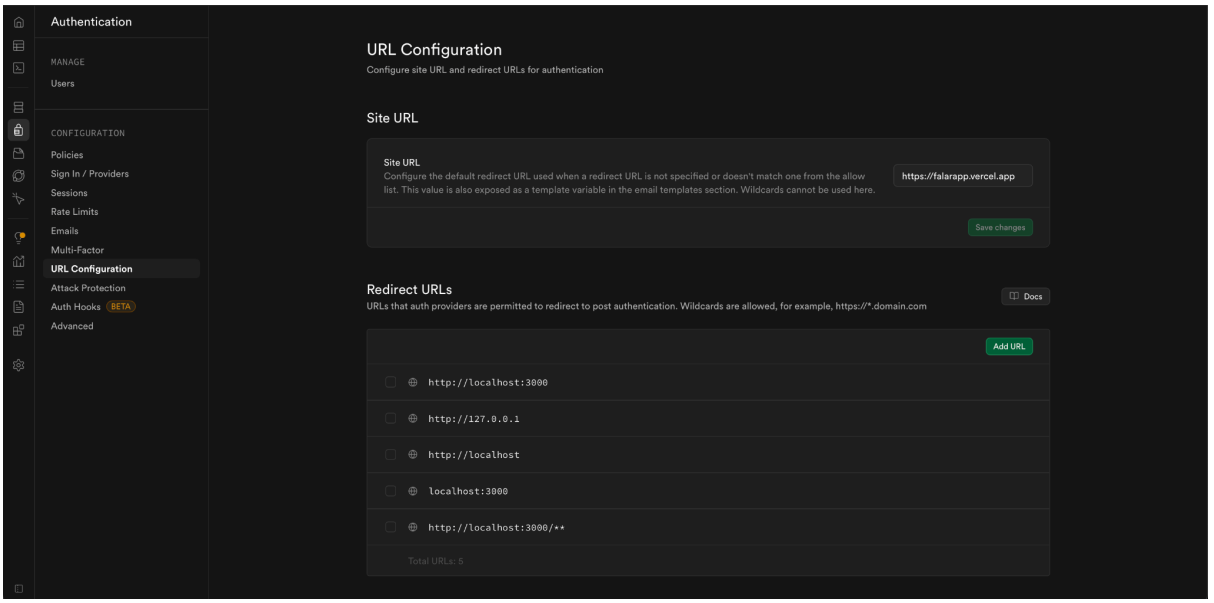
To setup Google OAuth, you need the client ID and client Secret from the Google Console. Follow Supabase's guide for retrieving them and paste their values properly in the place given.

Link:
https://supabase.com/docs/guides/auth/social-login/auth-google#configure-your-services-id

The next thing to do is to configure the URLs:

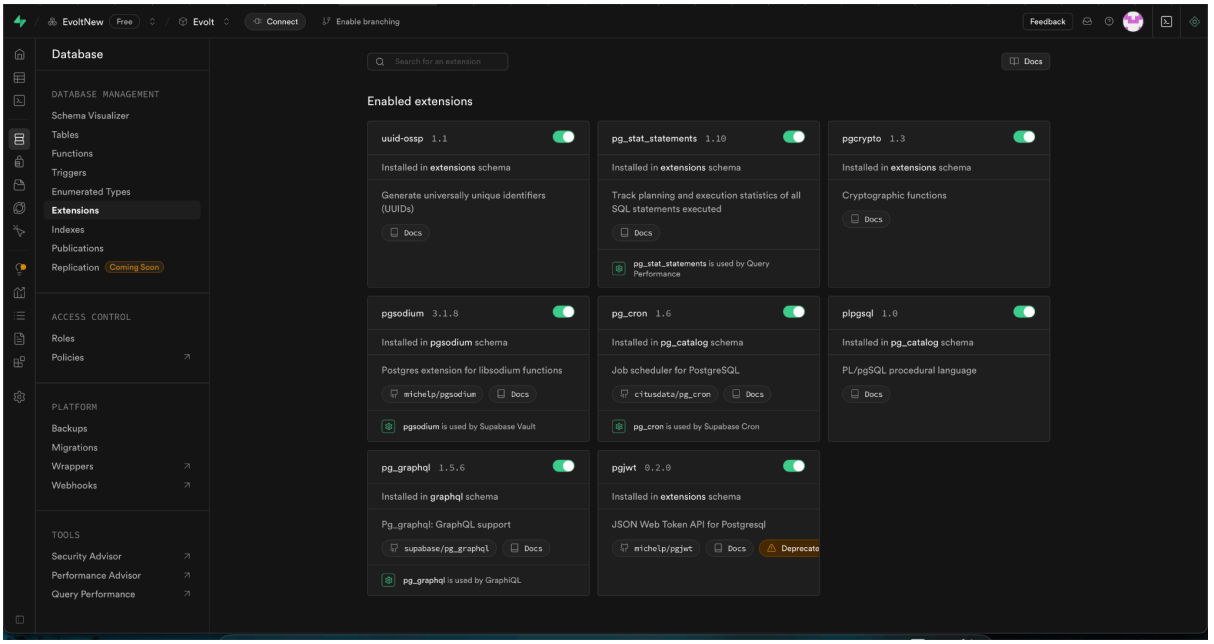
Add the hosted version's URL in the Site URL space (This can be a custom domain, or as in the image shown, a vercel [hosting provider's] domain).

Then add all the redirect URLs shown in the image below. **Do not miss out any of the URLs** or your site might not work properly.



4. Database Setup

Go to Database -> Extensions and ensure the following extensions are enabled (If not, enable the missing extensions before continuing):



Next, head over to the SQL Editor in the sidebar and run the following queries one by one by copy pasting them to generate the required database tables:

(For the simplicity of the initial setup process, the guide assumes that RLS is disabled. This is not very secure. You have to manually configure the RLS policies

based on your needs if you want security for the data. It can also be done by us if you are having active support. Please contact us by our support mail on the product page.)

User table:

```
create table public.user (   id uuid not null default auth.uid (),   created_at timestamp with time zone not null default now(),   name text null,   handle text not null,   about text null,   image text null,   followers text[] null,   following text[] null,   private boolean null,   email text not null,   cover text null,   bookmarks text[] null default '{}'::text[],   liked text[] null default '{}'::text[],   isresume boolean not null default false,   quickiebookmarks text[] null default '{}'::text[],   quickieliked text[] null default '{}'::text[],   blocked text[] null default '{}'::text[],   blockedby text[] null default '{}'::text[],   notifications bigint null default '0'::bigint,   constraint user_pkey primary key (id),   constraint user_email_key unique (email),   constraint user_handle_key unique (handle) ) TABLESPACE pg_default;
```

Posts table:

```
create table public.posts (   id bigint generated by default as identity not null,   created_at timestamp with time zone not null default now(),   poster uuid null default auth.uid (),   likes bigint null default '0'::bigint,   replies bigint null default '0'::bigint,   content text not null,   image text[] null,   title text null,   handle text null,   liked text[] null default '{}'::text[],   excerpt text not null,   cover text null default '/bg.jpg'::text,   bookmarked text[] null default '{}'::text[],   constraint posts_pkey primary key (id),   constraint posts_poster_fkey foreign KEY (poster) references "user" (id) on update CASCADE on delete CASCADE ) TABLESPACE pg_default;
```

Trending Table:

```
create table public.trending (   date date not null,   0 json null default '{}'::json,   3 json null default '{}'::json,   6 json null default '{}'::json,   9 json null default '{}'::json,   12 json null default '{}'::json,   15 json null default '{}'::json,   18 json null default '{}'::json,   21 json null default '{}'::json,   constraint trending_pkey primary key (date) ) TABLESPACE pg_default;
```

Quickies Table:

```
create table public.quickies (   id bigint generated by default as identity not null,   created_at timestamp with time zone not null default now(),   poster uuid null default auth.uid (),   likes bigint null default '0'::bigint,   comments bigint null default '0'::bigint,   content text not null,   image text[] null,   handle text null,   liked text[] null default '{}'::text[],   bookmarked text[] null default '{}'::text[],   parent bigint null default '0'::bigint,   "to" bigint null default '0'::bigint,   involved text[] null,   quote boolean null default false,   quoteid bigint null default '0'::bigint,   constraint quickies_pkey primary key (id),   constraint quickies_poster_fkey foreign KEY (poster) references "user" (id) on update CASCADE on delete CASCADE ) TABLESPACE pg_default;
```

Comments Table:

```
create table public.comments (   id bigint not null,   content text null,   likes bigint null default '0'::bigint,   liked text[] null default '{}'::text[],   time timestamp with time zone null default now(),   comment_id bigint generated by default as identity not null,   poster uuid null default auth.uid (),   handle text null,   constraint comments_pkey primary key (comment_id),   constraint comments_poster_fkey foreign KEY (poster) references "user" (id) on update CASCADE on delete CASCADE ) TABLESPACE pg_default;
```

Deactivated Table:

```
create table public.deactivated (   id uuid not null default
auth.uid (),   created_at timestamp with time zone not null
default now(),   name text null,   handle text not null,
about text null,   image text null,   followers text[] null,
following text[] null,   private boolean null,   email text
not null,   cover text null,   bookmarks text[] null default
'{}'::text[],   liked text[] null default '{}'::text[],
isresume boolean not null default false,   quickiebookmarks
text[] null default '{}'::text[],   quickieliked text[] null
default '{}'::text[],   notifications bigint null default
'0'::bigint,   blocked text[] null default '{}'::text[],
blockedby bigint[] null default '{}'::bigint[],   constraint
deactivated_pkey primary key (id),   constraint
deactivated_email_key unique (email),   constraint
deactivated_handle_key unique (handle) ) TABLESPACE
pg_default;
```

Used Handles Table:

```
create table public.usedhandles (   id bigint generated by
default as identity not null,   created_at timestamp with time
zone not null default now(),   handle text null,   constraint
usedhandles_pkey primary key (id) ) TABLESPACE pg_default;
```

Hashtags Table:

```
create table public.hashtags (   id bigint generated by
default as identity not null,   hashtag text null,   posts
bigint[] null default '{}'::bigint[],   postcount bigint null,
constraint hashtags_pkey primary key (id) ) TABLESPACE
pg_default;
```

Notifications Table (NOTE: Realtime has to be enabled at the top menubar for this table to receive real-time notification updates)

```
create table public.notifications (   id bigint generated by
default as identity not null,   title text null,   description
text null,   image text null,   type text null,   url text
null,   "to" uuid null,   seen boolean null default false,
postid bigint null,   userid uuid null,   constraint
notifications_pkey primary key (id),   constraint
notifications_to_fkey foreign KEY ("to") references "user"
(id) on delete CASCADE,   constraint notifications_userid_fkey
foreign KEY (userid) references "user" (id) on delete CASCADE
) TABLESPACE pg_default;
```

5. **CRON For Auto-Deletion:**

   The deactivated accounts' data are automatically deleted once every 30 days. To schedule this, a CRON job has to be created. To create the cron job to facilitate the same, run the following SQL query in the SQL editor:

```
select cron.schedule (      'deactivate-account',      '*/55 *
* * * *',      $$ delete from deactivated where created_at <
now() - interval '1 month' $$  );
```

6. **Function for checking if handle exists (during account creation):**

   To check if an user's handle is already existing in the users table when a new user creates an account, create the following function by running this on the SQL Editor:

```
create or replace function check_handle_existence(p_handle
text)  returns table(handle text) as $$  begin    return query
select u.handle from "user" u where u.handle = p_handle
union    select uh.handle from usedhandles uh where uh.handle
= p_handle;  end;  $$ language plpgsql;
```

7. **Functions for searching:**

   Run the following SQL queries to create functions for searching across the database:

   Posts:

```
CREATE FUNCTION title_excerpt_content(public.posts) RETURNS
text AS $$   SELECT $1.title || ' ' || $1.excerpt || ' ' ||
$1.content; $$ LANGUAGE SQL;
```

Users:

```
create function name_handle_about(public.user) returns text as
$$    select $1.name || ' ' || $1.handle || ' ' || $1.about;
$$ language sql immutable;
```

Now, the Database and Authentication are completely set up and the backend is almost done. Only storage has to be configured.

9. Configuring Storage:

Storage is used to store user's profile pictures and images in their posts and quickies. It has to be configured as following:

Create 4 buckets with the following folders (the first is the bucket name and the names that follow 'with' are the folder names):

covers with **public**, **quickies** with **images**, **posts** with **cover** and **public**, **profile** with **public**

Now, policies have to be configured to allow the buckets to be accessible. We will make them public buckets for simplicity. This is not very secure but this data will have to be accessed by users that are anonymous (not logged in) also. So they are made public for the sake of code and set-up simplicity.

Click '**New Policy**' for the Covers bucket and click the '**Create Policy From Scratch**' option:

Ensure to name your policy. You can name your policy as you like it.

In the **Allowed Operations** section, ensure the following are selected as shown:

Then, fill in the **Policy Definition** with the following text:

```
((bucket_id = 'covers'::text) AND (storage.extension(name) =
'jpg'::text) AND (lower((storage.foldername(name))[1]) =
'public'::text) AND (auth.role() = 'anon'::text))
```

The same procedure has to be repeated for other buckets also.

Next, for the **Posts** bucket, similarly click **New Policy** > **From Scratch** > Enable all the **Allowed Operations** and then fill the **Policy Definition** with the following text:

```
((bucket_id = 'posts'::text) AND (storage.extension(name) =
'jpg'::text) AND (auth.role() = 'anon'::text))
```

Now, for the **Quickies** bucket, **3 policies** have to be created following the same procedure as above:

New Policy > **From Scratch** > Enable all the **Allowed Operations** and then fill the **Policy Definition** with the following text:

```
((bucket_id = 'quickies'::text) AND (storage.extension(name) =
'jpeg'::text) AND (lower((storage.foldername(name))[1]) =
'images'::text) AND (auth.role() = 'anon'::text))
```

Now again,

New Policy > **From Scratch** > Enable all the **Allowed Operations** and then fill the **Policy Definition** with the following text:

```
((bucket_id = 'quickies'::text) AND (storage.extension(name) =
'jpg'::text) AND (lower((storage.foldername(name))[1]) =
'images'::text) AND (auth.role() = 'anon'::text))
```

And once more,

```
((bucket_id = 'quickies'::text) AND (storage.extension(name) =
'png'::text) AND (lower((storage.foldername(name))[1]) =
'images'::text) AND (auth.role() = 'anon'::text))
```

Now finally, for the final bucket, the **profile** bucket:

New Policy > **From Scratch** > Enable all the **Allowed Operations** and then fill the **Policy Definition** with the following text:

```
((bucket_id = 'profile'::text) AND (storage.extension(name) =
'jpg'::text) AND (lower((storage.foldername(name))[1]) =
'public'::text) AND (auth.role() = 'anon'::text))
```

The project set up is finally done and can be run now:

To run the project, type the following in the command prompt/terminal:

```
npm run dev
```

Now, go to the following IP address, http://127.0.0.1 and the project will be visible live!

To deploy the project, you can generate a production build by running the following in command prompt or terminal:

```
npm run build
```

I personally recommend deploying the project to a free hosting site like vercel.com or netlify.com by setting up automatic deployments through **GitHub**. The documentation to do so is elaborated on their respective sites.

Customization:

The App's Name and Subtitle can be configured in the file `config/config.js`. These are global changes and will be reflected throughout the app.

Color scheme can be modified by setting a `customprimary` color in the same `config.js` file. The user can also select their preferred color schemes. The list of colorschemes is available in `utils/themecontext.tsx` file where the colors array holds all the color schemes.

Landing page can be customized by editing `components/landing.tsx`

For any further support, enquiries, issues or potential misleads in the documentation, please feel free to contact me at balafalarapp@gmail.com

Thanks!

Bala