Use Supabase with Next.js

Learn how to create a Supabase project, add some sample data, and query from a Next.js app.

1 Create a Supabase project

Go to database.new and create a new Supabase project.

When your project is up and running, go to the Table Editor, create a new table and insert some data.

Alternatively, you can run the following snippet in your project's SQL Editor. This will create a `instruments` table with some sample data.

```sql
-- Create the table
create table instruments (
  id bigint primary key generated always as identity,
  name text not null
);
-- Insert some sample data into the table
insert into instruments (name)
values
  ('violin'),
  ('viola'),
  ('cello');

alter table instruments enable row level security;
```

Make the data in your table publicly readable by adding an RLS policy:

```
create policy "public can read instruments"
on public.instruments
for select to anon
using (true);
```

2

Create a Next.js app

Use the `create-next-app` command and the `with-supabase` template, to create a Next.js app pre-configured with:

- Cookie-based Auth
- TypeScript
- Tailwind CSS

```
npx create-next-app -e with-supabase
```

3

Declare Supabase Environment Variables

Rename `.env.example` to `.env.local` and populate with your Supabase connection variables:

Project URL

Loading...

```
Loading...
```

Anon key

Loading...

```
Loading...
```

.env.local

```
NEXT_PUBLIC_SUPABASE_URL=<SUBSTITUTE_SUPABASE_URL>
NEXT_PUBLIC_SUPABASE_ANON_KEY=<SUBSTITUTE_SUPABASE_ANON_KEY>
```

4 Query Supabase data from Next.js

Create a new file at `app/instruments/page.tsx` and populate with the following.

This will select all the rows from the `instruments` table in Supabase and render them on the page.

app/instruments/page.tsx utils/supabase/server.ts

```
import { createClient } from '@/utils/supabase/server';

export default async function Instruments() {
  const supabase = await createClient();
  const { data: instruments } = await supabase.from("instruments").select();

  return <pre>{JSON.stringify(instruments, null, 2)}</pre>
}
```

5 Start the app

Run the development server, go to http://localhost:3000/instruments in a browser and you should see the list of instruments.

```
npm run dev
```

Next steps

- Set up Auth for your app
- Insert more data into your database
- Upload and serve static files using Storage

Edit this page on GitHub

Need some help? Contact support
Latest product updates? See Changelog
Something's not right? Check system status

Use Supabase with Nuxt

Learn how to create a Supabase project, add some sample data to your database, and query the data from a Nuxt app.

1 Create a Supabase project

Go to database.new and create a new Supabase project.

When your project is up and running, go to the Table Editor, create a new table and insert some data.

Alternatively, you can run the following snippet in your project's SQL Editor. This will create a `instruments` table with some sample data.

```sql
-- Create the table
create table instruments (
  id bigint primary key generated always as identity,
  name text not null
);
-- Insert some sample data into the table
insert into instruments (name)
values
  ('violin'),
  ('viola'),
  ('cello');

alter table instruments enable row level security;
```

Make the data in your table publicly readable by adding an RLS policy:

```
create policy "public can read instruments"
on public.instruments
for select to anon
using (true);
```

2

Create a Nuxt app

Create a Nuxt app using the `npx nuxi` command.

Terminal

```
npx nuxi@latest init my-app
```

3

Install the Supabase client library

The fastest way to get started is to use the `supabase-js` client library which provides a convenient interface for working with Supabase from a Nuxt app.

Navigate to the Nuxt app and install `supabase-js`.

Terminal

```
cd my-app && npm install @supabase/supabase-js
```

4

Query data from the app

In `app.vue`, create a Supabase client using your project URL and public API (anon) key:

Project URL

Loading...

```
Loading...
```

Anon key

Loading...

```
Loading...
```

Replace the existing content in your `app.vue` file with the following code.

app.vue

```
<script setup>
import { createClient } from '@supabase/supabase-js'
const supabase = createClient('https://<project>.supabase.co', '<your-anon-ke
const instruments = ref([])

async function getInstruments() {
  const { data } = await supabase.from('instruments').select()
  instruments.value = data
}

onMounted(() => {
  getInstruments()
})
</script>

<template>
  <ul>
    <li v-for="instrument in instruments" :key="instrument.id">{{ instrument.
  </ul>
</template>
```

5 Start the app

Start the app, navigate to http://localhost:3000 in the browser, open the browser console, and you should see the list of instruments.

Terminal

```
npm run dev
```

The community-maintained @nuxtjs/supabase module provides an alternate DX for working with Supabase in Nuxt.

Edit this page on GitHub

Need some help? Contact support
Latest product updates? See Changelog
Something's not right? Check system status

—

Contributing
Author Styleguide
Open Source
SupaSquad
Privacy Settings

Use Supabase with React

Learn how to create a Supabase project, add some sample data to your database, and query the data from a React app.

1 Create a Supabase project

Go to database.new and create a new Supabase project.

When your project is up and running, go to the Table Editor, create a new table and insert some data.

Alternatively, you can run the following snippet in your project's SQL Editor. This will create a `instruments` table with some sample data.

```sql
-- Create the table
create table instruments (
  id bigint primary key generated always as identity,
  name text not null
);
-- Insert some sample data into the table
insert into instruments (name)
values
  ('violin'),
  ('viola'),
  ('cello');

alter table instruments enable row level security;
```

Make the data in your table publicly readable by adding an RLS policy:

```
create policy "public can read instruments"
on public.instruments
for select to anon
using (true);
```

2 Create a React app

Create a React app using a Vite template.

Terminal

```
npm create vite@latest my-app -- --template react
```

3 Install the Supabase client library

The fastest way to get started is to use the `supabase-js` client library which provides a convenient interface for working with Supabase from a React app.

Navigate to the React app and install `supabase-js`.

Terminal

```
cd my-app && npm install @supabase/supabase-js
```

4 Query data from the app

In `App.jsx`, create a Supabase client using your project URL and public API (anon) key:

Project URL

Loading...

Loading...

Anon key

Loading...

Loading...

Add a `getInstruments` function to fetch the data and display the query result to the page.

src/App.jsx

```
import { useEffect, useState } from "react";
import { createClient } from "@supabase/supabase-js";

const supabase = createClient("https://<project>.supabase.co", "<your-anon-ke

function App() {
  const [instruments, setInstruments] = useState([]);

  useEffect(() => {
    getInstruments();
  }, []);

  async function getInstruments() {
    const { data } = await supabase.from("instruments").select();
    setInstruments(data);
  }

  return (
    <ul>
      {instruments.map((instrument) => (
        <li key={instrument.name}>{instrument.name}</li>
      ))}
    </ul>
  );
}

export default App;
```

5 Start the app

Start the app, go to http://localhost:5173 in a browser, and open the browser console and you should see the list of instruments.

Terminal

```
npm run dev
```

Next steps

- Set up Auth for your app
- Insert more data into your database
- Upload and serve static files using Storage

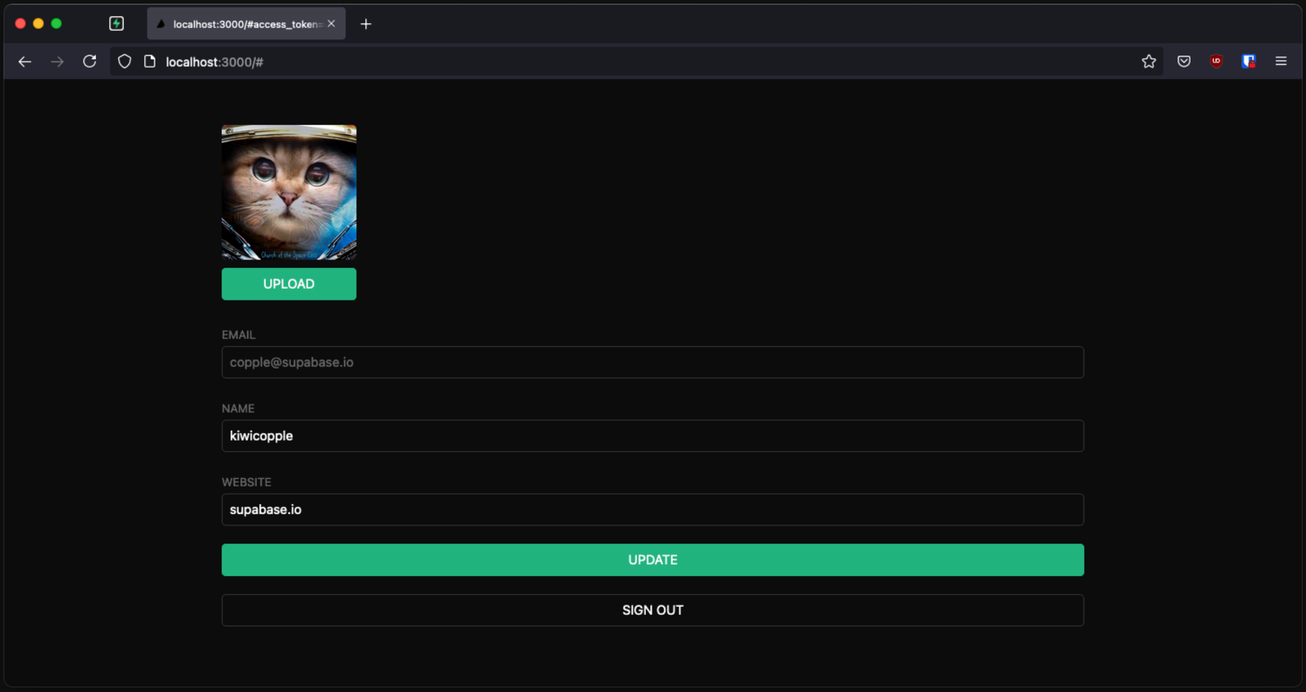
Edit this page on GitHub

Need some help? Contact support
Latest product updates? See Changelog
Something's not right? Check system status

Build a User Management App with Next.js

This tutorial demonstrates how to build a basic user management app. The app authenticates and identifies the user, stores their profile information in the database, and allows the user to log in, update their profile details, and upload a profile photo. The app uses:

- Supabase Database - a Postgres database for storing your user data and Row Level Security so data is protected and users can only access their own information.
- Supabase Auth - allow users to sign up and log in.
- Supabase Storage - users can upload a profile photo.



If you get stuck while working through this guide, refer to the full example on GitHub.

Project setup

Before we start building we're going to set up our Database and API. This is as simple as starting a new Project in Supabase and then creating a "schema" inside the database.

Create a project

1. Create a new project in the Supabase Dashboard.
2. Enter your project details.
3. Wait for the new database to launch.

Set up the database schema

Now we are going to set up the database schema. We can use the "User Management Starter" quickstart in the SQL Editor, or you can just copy/paste the SQL from below and run it yourself.

Dashboard SQL

1. Go to the SQL Editor page in the Dashboard.
2. Click **User Management Starter**.
3. Click **Run**.

You can pull the database schema down to your local project by running the `db pull` command. Read the local development docs for detailed instructions.

```
supabase link --project-ref <project-id>
# You can get <project-id> from your project's dashboard URL: https://supabase
```

```
supabase db pull
```

Get the API keys

Now that you've created some database tables, you are ready to insert data using the auto-generated API.
We just need to get the Project URL and `anon` key from the API settings.

1. Go to the API Settings page in the Dashboard.
2. Find your Project `URL`, `anon`, and `service_role` keys on this page.

Building the app

Let's start building the Next.js app from scratch.

Initialize a Next.js app

We can use `create-next-app` to initialize an app called `supabase-nextjs`:

JavaScript TypeScript

```
npx create-next-app@latest --use-npm supabase-nextjs
cd supabase-nextjs
```

Then install the Supabase client library: supabase-js

```
npm install @supabase/supabase-js
```

And finally we want to save the environment variables in a `.env.local`.
Create a `.env.local` file at the root of the project, and paste the API URL and the `anon` key that you copied earlier.

```
NEXT_PUBLIC_SUPABASE_URL=YOUR_SUPABASE_URL
NEXT_PUBLIC_SUPABASE_ANON_KEY=YOUR_SUPABASE_ANON_KEY
```

App styling (optional)

An optional step is to update the CSS file `app/globals.css` to make the app look nice.
You can find the full contents of this file here.

Supabase Server-Side Auth

Next.js is a highly versatile framework offering pre-rendering at build time (SSG), server-side rendering at request time (SSR), API routes, and middleware edge-functions.

To better integrate with the framework, we've created the `@supabase/ssr` package for Server-Side Auth. It has all the functionalities to quickly configure your Supabase project to use cookies for storing user sessions. See the Next.js Server-Side Auth guide for more information.

Install the package for Next.js.

```
npm install @supabase/ssr
```

Supabase utilities

There are two different types of clients in Supabase:

1. **Client Component client** - To access Supabase from Client Components, which run in the browser.

2 **Server Component client** - To access Supabase from Server Components, Server Actions, and Route Handlers, which run only on the server.

It is recommended to create the following essential utilities files for creating clients, and organize them within `utils/supabase` at the root of the project.

JavaScript TypeScript

Create a `client.js` and a `server.js` with the following functionalities for client-side Supabase and server-side Supabase, respectively.

utils/supabase/client.js utils/supabase/server.js

```
import { createBrowserClient } from '@supabase/ssr'

export function createClient() {
  // Create a supabase client on the browser with project's credentials
  return createBrowserClient(
    process.env.NEXT_PUBLIC_SUPABASE_URL,
    process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY
  )
}
```

Next.js middleware

Since Server Components can't write cookies, you need middleware to refresh expired Auth tokens and store them. This is accomplished by:

- Refreshing the Auth token with the call to `supabase.auth.getUser`.
- Passing the refreshed Auth token to Server Components through `request.cookies.set`, so they don't attempt to refresh the same token themselves.
- Passing the refreshed Auth token to the browser, so it replaces the old token. This is done with `response.cookies.set`.

You could also add a matcher, so that the middleware only runs on route that access Supabase. For more information, check out this documentation.

Be careful when protecting pages. The server gets the user session from the cookies, which can be spoofed by anyone.

Always use `supabase.auth.getUser()` to protect pages and user data.

Never trust `supabase.auth.getSession()` inside server code such as middleware. It isn't guaranteed to revalidate the Auth token.

It's safe to trust `getUser()` because it sends a request to the Supabase Auth server every time to revalidate the Auth token.

JavaScript TypeScript

Create a `middleware.js` file at the project root and another one within the `utils/supabase` folder. The `utils/supabase` file contains the logic for updating the session. This is used by the `middleware.js` file, which is a Next.js convention.

middleware.js utils/supabase/middleware.js

```
import { updateSession } from '@/utils/supabase/middleware'

export async function middleware(request) {
  // update user's auth session
  return await updateSession(request)
}

export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones starting with:
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico (favicon file)
     * Feel free to modify this pattern to include more paths.
     */
    '/((?!_next/static|_next/image|favicon.ico|.*\\.(?:svg|png|jpg|jpeg|gif|w
  ],
}
```

Set up a login page

Login and signup form

Create a login/signup page for your application:

JavaScript TypeScript

Create a new folder named `login`, containing a `page.jsx` file with a login/signup form.

app/login/page.jsx

```
import { login, signup } from './actions'

export default function LoginPage() {
  return (
    <form>
      <label htmlFor="email">Email:</label>
      <input id="email" name="email" type="email" required />
      <label htmlFor="password">Password:</label>
      <input id="password" name="password" type="password" required />
      <button formAction={login}>Log in</button>
      <button formAction={signup}>Sign up</button>
    </form>
  )
}
```

Navigate to `http://localhost:3000/login`. You should see your login form, but it's not yet hooked up to the actual login function. Next, you need to create the login/signup actions. They will:

- Retrieve the user's information.
- Send that information to Supabase as a signup request, which in turns will send a confirmation email.
- Handle any error that arises.

Note that cookies is called before any calls to Supabase, which opts fetch calls out of Next.js's caching. This is important for authenticated data fetches, to ensure that users get access only to their own data.

See the Next.js docs to learn more about opting out of data caching.

JavaScript TypeScript

app/login/actions.js app/error/page.jsx

```js
'use server'

import { revalidatePath } from 'next/cache'
import { redirect } from 'next/navigation'

import { createClient } from '@/utils/supabase/server'

export async function login(formData) {
  const supabase = await createClient()

  // type-casting here for convenience
  // in practice, you should validate your inputs
  const data = {
    email: formData.get('email'),
    password: formData.get('password'),
  }

  const { error } = await supabase.auth.signInWithPassword(data)

  if (error) {
    redirect('/error')
  }

  revalidatePath('/', 'layout')
  redirect('/account')
}

export async function signup(formData) {
  const supabase = await createClient()

  const data = {
    email: formData.get('email'),
    password: formData.get('password'),
  }

  const { error } = await supabase.auth.signUp(data)

  if (error) {
    redirect('/error')
  }

  revalidatePath('/', 'layout')
  redirect('/account')
}
```

When you enter your email and password, you will receive an email with the title **Confirm Your Signup**. Congrats 🎉 !!!

Email template

Change the email template to support a server-side authentication flow.

Before we proceed, let's change the email template to support sending a token hash:

- Go to the Auth templates page in your dashboard.
- Select `Confirm signup` template.
- Change `{{ .ConfirmationURL }}` to `{{ .SiteURL }}/auth/confirm?token_hash={{ .TokenHash }}&type=email`.

Did you know? You could also customize emails sent out to new users, including the email's looks, content, and query parameters. Check out the settings of your project.

Confirmation endpoint

As we are working in a server-side rendering (SSR) environment, it is necessary to create a server endpoint responsible for exchanging the `token_hash` for a session.

In the following code snippet, we perform the following steps:

- Retrieve the code sent back from the Supabase Auth server using the `token_hash` query parameter.
- Exchange this code for a session, which we store in our chosen storage mechanism (in this case, cookies).
- Finally, we redirect the user to the `account` page.

JavaScript TypeScript

app/auth/confirm/route.js

```javascript
import { NextResponse } from 'next/server'

import { createClient } from '@/utils/supabase/server'

// Creating a handler to a GET request to route /auth/confirm
export async function GET(request) {
  const { searchParams } = new URL(request.url)
  const token_hash = searchParams.get('token_hash')
  const type = searchParams.get('type')
  const next = '/account'

  // Create redirect link without the secret token
  const redirectTo = request.nextUrl.clone()
  redirectTo.pathname = next
  redirectTo.searchParams.delete('token_hash')
  redirectTo.searchParams.delete('type')

  if (token_hash && type) {
    const supabase = await createClient()

    const { error } = await supabase.auth.verifyOtp({
      type,
      token_hash,
    })
    if (!error) {
      redirectTo.searchParams.delete('next')
      return NextResponse.redirect(redirectTo)
    }
  }

  // return the user to an error page with some instructions
  redirectTo.pathname = '/error'
  return NextResponse.redirect(redirectTo)
}
```

Account page

After a user is signed in we can allow them to edit their profile details and manage their account.

Let's create a new component for that called `AccountForm` within the `app/account` folder.

JavaScript TypeScript

app/account/account-form.jsx

```
'use client'
import { useCallback, useEffect, useState } from 'react'
import { createClient } from '@/utils/supabase/client'

export default function AccountForm({ user }) {
  const supabase = createClient()
  const [loading, setLoading] = useState(true)
  const [fullname, setFullname] = useState(null)
  const [username, setUsername] = useState(null)
  const [website, setWebsite] = useState(null)
  const [avatar_url, setAvatarUrl] = useState(null)

  const getProfile = useCallback(async () => {
    try {
      setLoading(true)

      const { data, error, status } = await supabase
        .from('profiles')
        .select(`full_name, username, website, avatar_url`)
        .eq('id', user?.id)
        .single()

      if (error && status !== 406) {
        throw error
      }

      if (data) {
        setFullname(data.full_name)
        setUsername(data.username)
        setWebsite(data.website)
        setAvatarUrl(data.avatar_url)
      }
    } catch (error) {
      alert('Error loading user data!')
    } finally {
      setLoading(false)
    }
  }, [user, supabase])

  useEffect(() => {
    getProfile()
  }, [user, getProfile])

```

```
    async function updateProfile({ username, website, avatar_url }) {
      try {
        setLoading(true)

        const { error } = await supabase.from('profiles').upsert({
          id: user?.id,
          full_name: fullname,
          username,
          website,
          avatar_url,
          updated_at: new Date().toISOString(),
        })
        if (error) throw error
        alert('Profile updated!')
      } catch (error) {
        alert('Error updating the data!')
      } finally {
        setLoading(false)
      }
    }

    return (
      <div className="form-widget">
        <div>
          <label htmlFor="email">Email</label>
          <input id="email" type="text" value={user?.email} disabled />
        </div>
        <div>
          <label htmlFor="fullName">Full Name</label>
          <input
            id="fullName"
            type="text"
            value={fullname || ''}
            onChange={(e) => setFullname(e.target.value)}
          />
        </div>
        <div>
          <label htmlFor="username">Username</label>
          <input
            id="username"
            type="text"
            value={username || ''}
            onChange={(e) => setUsername(e.target.value)}
          />
        </div>
        <div>
          <label htmlFor="website">Website</label>
          <input
            id="website"
            type="url"
            value={website || ''}
            onChange={(e) => setWebsite(e.target.value)}
          />
        </div>

        <div>
          <button
            className="button primary block"
            onClick={() => updateProfile({ fullname, username, website, avatar
            disabled={loading}
          >
            {loading ? 'Loading ...' : 'Update'}
          </button>
        </div>

        <div>
          <form action="/auth/signout" method="post">
            <button className="button block" type="submit">
              Sign out
            </button>
          </form>
        </div>
      </div>
    )
  }
```

Create an account page for the `AccountForm` component we just created

JavaScript TypeScript

app/account/page.jsx

```
import AccountForm from './account-form'
import { createClient } from '@/utils/supabase/server'

export default async function Account() {
  const supabase = await createClient()

  const {
    data: { user },
  } = await supabase.auth.getUser()

  return <AccountForm user={user} />
}
```

Sign out

Let's create a route handler to handle the signout from the server side. Make sure to check if the user is logged in first!

JavaScript TypeScript

app/auth/signout/route.js

```
import { createClient } from '@/utils/supabase/server'
import { revalidatePath } from 'next/cache'
import { NextResponse } from 'next/server'

export async function POST(req) {
  const supabase = await createClient()

  // Check if a user's logged in
  const {
    data: { user },
  } = await supabase.auth.getUser()

  if (user) {
    await supabase.auth.signOut()
  }

  revalidatePath('/', 'layout')
  return NextResponse.redirect(new URL('/login', req.url), {
    status: 302,
  })
}
```

Launch!

Now that we have all the pages, route handlers and components in place, let's run this in a terminal window:

```
npm run dev
```

And then open the browser to localhost:3000 and you should see the completed app.

Bonus: Profile photos

Every Supabase project is configured with Storage for managing large files like photos and videos.

Create an upload widget

Let's create an avatar widget for the user so that they can upload a profile photo. We can start by creating a new component:

JavaScript TypeScript

app/account/avatar.jsx

```
'use client'
import React, { useEffect, useState } from 'react'
import { createClient } from '@/utils/supabase/client'
import Image from 'next/image'

export default function Avatar({ uid, url, size, onUpload }) {
  const supabase = createClient()
  const [avatarUrl, setAvatarUrl] = useState(url)
  const [uploading, setUploading] = useState(false)

  useEffect(() => {
    async function downloadImage(path) {
      try {
        const { data, error } = await supabase.storage.from('avatars').downlo
        if (error) {
          throw error
        }

        const url = URL.createObjectURL(data)
        setAvatarUrl(url)
      } catch (error) {
        console.log('Error downloading image: ', error)
      }
    }

    if (url) downloadImage(url)
  }, [url, supabase])

  const uploadAvatar = async (event) => {
    try {
      setUploading(true)

      if (!event.target.files || event.target.files.length === 0) {
        throw new Error('You must select an image to upload.')
      }

      const file = event.target.files[0]
      const fileExt = file.name.split('.').pop()
      const filePath = `${uid}-${Math.random()}.${fileExt}`

      const { error: uploadError } = await supabase.storage.from('avatars').u

      if (uploadError) {
        throw uploadError
      }

      onUpload(filePath)
    } catch (error) {
      alert('Error uploading avatar!')
    } finally {
      setUploading(false)
    }
  }

  return (
    <div>
      {avatarUrl ? (
        <Image
          width={size}
          height={size}
          src={avatarUrl}
          alt="Avatar"
          className="avatar image"
          style={{ height: size, width: size }}
        />
      ) : (
        <div className="avatar no-image" style={{ height: size, width: size }
      )}
      <div style={{ width: size }}>
```

```
          <label className="button primary block" htmlFor="single">
            {uploading ? 'Uploading ...' : 'Upload'}
          </label>
          <input
            style={{
              visibility: 'hidden',
              position: 'absolute',
            }}
            type="file"
            id="single"
            accept="image/*"
            onChange={uploadAvatar}
            disabled={uploading}
          />
        </div>
      </div>
    )
  }
```

Add the new widget

And then we can add the widget to the `AccountForm` component:

JavaScript TypeScript

app/account/account-form.jsx

```
// Import the new component
import Avatar from './avatar'

// ...

return (
  <div className="form-widget">
    {/* Add to the body */}
    <Avatar
      uid={user?.id}
      url={avatar_url}
      size={150}
      onUpload={(url) => {
        setAvatarUrl(url)
        updateProfile({ fullname, username, website, avatar_url: url })
      }}
      />
      {/* ... */}
    </div>
  )
```

At this stage you have a fully functional application!

See also

- See the complete example on GitHub and deploy it to Vercel
- Build a Twitter Clone with the Next.js App Router and Supabase - free egghead course
- Explore the pre-built Auth UI for React
- Explore the Auth Helpers for Next.js
- Explore the Supabase Cache Helpers
- See the Next.js Subscription Payments Starter template on GitHub

Edit this page on GitHub

Need some help? Contact support

Latest product updates? See Changelog

Something's not right? Check system status

—

Contributing

Author Styleguide

Open Source

SupaSquad

Privacy Settings