Set up server-side hooks in src/hooks

Set up server-side hooks in src/hooks.server.ts.

The hooks:

Create a request-specific Supabase client, using the user credentials from the request cookie. This client is used for server-only code.

Check user authentication.

Guard protected pages.

**src/hooks.server.ts**

import { createServerClient } from '@supabase/ssr'

import { type Handle, redirect } from '@sveltejs/kit'

import { sequence } from '@sveltejs/kit/hooks'

import { PUBLIC\_SUPABASE\_URL, PUBLIC\_SUPABASE\_ANON\_KEY } from '$env/static/public'

const supabase: Handle = async ({ event, resolve }) => {

/\*\*

\* Creates a Supabase client specific to this server request.

\*

\* The Supabase client gets the Auth token from the request cookies.

\*/

event.locals.supabase = createServerClient(PUBLIC\_SUPABASE\_URL, PUBLIC\_SUPABASE\_ANON\_KEY, {

cookies: {

getAll: () => event.cookies.getAll(),

/\*\*

\* SvelteKit's cookies API requires `path` to be explicitly set in

\* the cookie options. Setting `path` to `/` replicates previous/

\* standard behavior.

\*/

setAll: (cookiesToSet) => {

cookiesToSet.forEach(({ name, value, options }) => {

event.cookies.set(name, value, { ...options, path: '/' })

})

},

},

})

/\*\*

\* Unlike `supabase.auth.getSession()`, which returns the session \_without\_

\* validating the JWT, this function also calls `getUser()` to validate the

\* JWT before returning the session.

\*/

event.locals.safeGetSession = async () => {

const {

data: { session },

} = await event.locals.supabase.auth.getSession()

if (!session) {

return { session: null, user: null }

}

const {

data: { user },

error,

} = await event.locals.supabase.auth.getUser()

if (error) {

// JWT validation has failed

return { session: null, user: null }

}

return { session, user }

}

return resolve(event, {

filterSerializedResponseHeaders(name) {

/\*\*

\* Supabase libraries use the `content-range` and `x-supabase-api-version`

\* headers, so we need to tell SvelteKit to pass it through.

\*/

return name === 'content-range' || name === 'x-supabase-api-version'

},

})

}

const authGuard: Handle = async ({ event, resolve }) => {

const { session, user } = await event.locals.safeGetSession()

event.locals.session = session

event.locals.user = user

if (!event.locals.session && event.url.pathname.startsWith('/private')) {

redirect(303, '/auth')

}

if (event.locals.session && event.url.pathname === '/auth') {

redirect(303, '/private')

}

return resolve(event)

}

export const handle: Handle = sequence(supabase, authGuard)

Create TypeScript definitions

To prevent TypeScript errors, add type definitions for the new event.locals properties.

src/app.d.ts

import type { Session, SupabaseClient, User } from '@supabase/supabase-js'

declare global {

namespace App {

// interface Error {}

interface Locals {

supabase: SupabaseClient

safeGetSession: () => Promise<{ session: Session | null; user: User | null }>

session: Session | null

user: User | null

}

interface PageData {

session: Session | null

}

// interface PageState {}

// interface Platform {}

}

}

export {}

Create a Supabase client in your root layout

**Create a Supabase client in your root +layout.ts. This client can be used to access Supabase from the client or the server. In order to get access to the Auth token on the server, use a +layout.server.ts file to pass in the session from event.locals.**

**src/routes/+layout.ts**

import { createBrowserClient, createServerClient, isBrowser } from '@supabase/ssr'

import { PUBLIC\_SUPABASE\_ANON\_KEY, PUBLIC\_SUPABASE\_URL } from '$env/static/public'

import type { LayoutLoad } from './$types'

export const load: LayoutLoad = async ({ data, depends, fetch }) => {

/\*\*

\* Declare a dependency so the layout can be invalidated, for example, on

\* session refresh.

\*/

depends('supabase:auth')

const supabase = isBrowser()

? createBrowserClient(PUBLIC\_SUPABASE\_URL, PUBLIC\_SUPABASE\_ANON\_KEY, {

global: {

fetch,

},

})

: createServerClient(PUBLIC\_SUPABASE\_URL, PUBLIC\_SUPABASE\_ANON\_KEY, {

global: {

fetch,

},

cookies: {

getAll() {

return data.cookies

},

},

})

/\*\*

\* It's fine to use `getSession` here, because on the client, `getSession` is

\* safe, and on the server, it reads `session` from the `LayoutData`, which

\* safely checked the session using `safeGetSession`.

\*/

const {

data: { session },

} = await supabase.auth.getSession()

const {

data: { user },

} = await supabase.auth.getUser()

return { session, supabase, user }

}

**src/routes/+layout.server.ts**

import type { LayoutServerLoad } from './$types'

export const load: LayoutServerLoad = async ({ locals: { safeGetSession }, cookies }) => {

const { session } = await safeGetSession()

return {

session,

cookies: cookies.getAll(),

}

}

Listen to Auth events

Set up a listener for Auth events on the client, to handle session refreshes and signouts.

**src/routes/+layout.svelte**

<script>

import { invalidate } from '$app/navigation'

import { onMount } from 'svelte'

let { data, children } = $props()

let { session, supabase } = $derived(data)

onMount(() => {

const { data } = supabase.auth.onAuthStateChange((\_, newSession) => {

if (newSession?.expires\_at !== session?.expires\_at) {

invalidate('supabase:auth')

}

})

return () => data.subscription.unsubscribe()

})

</script>

{@render children()}

PAGE EXAMPLE

This example page calls Supabase from the server to get a list of countries from the database.

This is an example of a public page that uses publicly readable data.

**src/routes/+page.server.ts**

import type { PageServerLoad } from './$types'

export const load: PageServerLoad = async ({ locals: { supabase } }) => {

const { data: countries } = await supabase.from('countries').select('name').limit(5).order('name')

return { countries: countries ?? [] }

}

**src/routes/+page.svelte**

<script>

let { data } = $props()

let { countries } = $derived(data)

</script>

<h1>Welcome to Supabase!</h1>

<ul>

{#each countries as country}

<li>{country.name}</li>

{/each}

</ul>

Change the Auth confirmation path

If you have email confirmation turned on (the default), a new user will receive an email confirmation after signing up.

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

Go to the Auth templates page in your dashboard. In the Confirm signup template, change {{ .ConfirmationURL }} to {{ .SiteURL }}/auth/confirm?token\_hash={{ .TokenHash }}&type=email.

LOGIN PAGE

**src/routes/auth/+page.server.ts**

import { redirect } from '@sveltejs/kit'

import type { Actions } from './$types'

export const actions: Actions = {

signup: async ({ request, locals: { supabase } }) => {

const formData = await request.formData()

const email = formData.get('email') as string

const password = formData.get('password') as string

const { error } = await supabase.auth.signUp({ email, password })

if (error) {

console.error(error)

redirect(303, '/auth/error')

} else {

redirect(303, '/')

}

},

login: async ({ request, locals: { supabase } }) => {

const formData = await request.formData()

const email = formData.get('email') as string

const password = formData.get('password') as string

const { error } = await supabase.auth.signInWithPassword({ email, password })

if (error) {

console.error(error)

redirect(303, '/auth/error')

} else {

redirect(303, '/private')

}

},

}

**src/routes/auth/+page.svelte**

<form method="POST" action="?/login">

<label>

Email

<input name="email" type="email" />

</label>

<label>

Password

<input name="password" type="password" />

</label>

<button>Login</button>

<button formaction="?/signup">Sign up</button>

</form>

**src/routes/auth/+layout.svelte**

<script>

let { children } = $props()

</script>

<header>

<nav>

<a href="/">Home</a>

</nav>

</header>

{@render children()}

**src/routes/auth/error/+page.svelte**

<p>Login error</p>

Create the signup confirmation route

Finish the signup flow by creating the API route to handle email verification.

**src/routes/auth/confirm/+server.ts**

import type { EmailOtpType } from '@supabase/supabase-js'

import { redirect } from '@sveltejs/kit'

import type { RequestHandler } from './$types'

export const GET: RequestHandler = async ({ url, locals: { supabase } }) => {

const token\_hash = url.searchParams.get('token\_hash')

const type = url.searchParams.get('type') as EmailOtpType | null

const next = url.searchParams.get('next') ?? '/'

/\*\*

\* Clean up the redirect URL by deleting the Auth flow parameters.

\*

\* `next` is preserved for now, because it's needed in the error case.

\*/

const redirectTo = new URL(url)

redirectTo.pathname = next

redirectTo.searchParams.delete('token\_hash')

redirectTo.searchParams.delete('type')

if (token\_hash && type) {

const { error } = await supabase.auth.verifyOtp({ type, token\_hash })

if (!error) {

redirectTo.searchParams.delete('next')

redirect(303, redirectTo)

}

}

redirectTo.pathname = '/auth/error'

redirect(303, redirectTo)

}

private routes

Create private routes that can only be accessed by authenticated users. The routes in the private directory are protected by the route guard in hooks.server.ts.

To ensure that hooks.server.ts runs for every nested path, put a +layout.server.ts file in the private directory. This file can be empty, but must exist to protect routes that don't have their own +layout|page.server.ts.

**src/routes/private/+layout.server.ts**

/\*\*

\* This file is necessary to ensure protection of all routes in the `private`

\* directory. It makes the routes in this directory \_dynamic\_ routes, which

\* send a server request, and thus trigger `hooks.server.ts`.

\*\*/

**src/routes/private/+layout.svelte**

<script>

let { data, children } = $props()

let { supabase } = $derived(data)

const logout = async () => {

const { error } = await supabase.auth.signOut()

if (error) {

console.error(error)

}

}

</script>

<header>

<nav>

<a href="/">Home</a>

</nav>

<button onclick={logout}>Logout</button>

</header>

<main>

{@render children()}

</main>

**SQL**

**-- Run this SQL against your database to create a `notes` table.**

create table notes (

id bigint primary key generated always as identity,

created\_at timestamp with time zone not null default now(),

user\_id uuid references auth.users on delete cascade not null default auth.uid(),

note text not null

);

alter table notes enable row level security;

revoke all on table notes from authenticated;

revoke all on table notes from anon;

grant all (note) on table notes to authenticated;

grant select (id) on table notes to authenticated;

grant delete on table notes to authenticated;

create policy "Users can access and modify their own notes"

on notes

for all

to authenticated

using ((select auth.uid()) = user\_id);

**src/routes/private/+page.server.ts**

import type { PageServerLoad } from './$types'

export const load: PageServerLoad = async ({ depends, locals: { supabase } }) => {

depends('supabase:db:notes')

const { data: notes } = await supabase.from('notes').select('id,note').order('id')

return { notes: notes ?? [] }

}

**src/routes/private/+page.svelte**

<script lang="ts">

import { invalidate } from '$app/navigation'

import type { EventHandler } from 'svelte/elements'

import type { PageData } from './$types'

let { data } = $props()

let { notes, supabase, user } = $derived(data)

const handleSubmit: EventHandler<SubmitEvent, HTMLFormElement> = async (evt) => {

evt.preventDefault()

if (!evt.target) return

const form = evt.target as HTMLFormElement

const note = (new FormData(form).get('note') ?? '') as string

if (!note) return

const { error } = await supabase.from('notes').insert({ note })

if (error) console.error(error)

invalidate('supabase:db:notes')

form.reset()

}

</script>

<h1>Private page for user: {user?.email}</h1>

<h2>Notes</h2>

<ul>

{#each notes as note}

<li>{note.note}</li>

{/each}

</ul>

<form onsubmit={handleSubmit}>

<label>

Add a note

<input name="note" type="text" />

</label>

</form>

GOOGLE AUTH

**Application code#**

To use your own application code for the signin button, call the signInWithOAuth method (or the equivalent for your language).

Make sure you're using the right supabase client in the following code.

If you're not using Server-Side Rendering or cookie-based Auth, you can directly use the createClient from @supabase/supabase-js. If you're using Server-Side Rendering, see the Server-Side Auth guide for instructions on creating your Supabase client.

supabase.auth.signInWithOAuth({

provider: 'google',

})

For an implicit flow, that's all you need to do. The user will be taken to Google's consent screen, and finally redirected to your app with an access and refresh token pair representing their session.

For a PKCE flow, for example in Server-Side Auth, you need an extra step to handle the code exchange. When calling signInWithOAuth, provide a redirectTo URL which points to a callback route. This redirect URL should be added to your redirect allow list.

In the browser, signInWithOAuth automatically redirects to the OAuth provider's authentication endpoint, which then redirects to your endpoint.

**await supabase.auth.signInWithOAuth({**

**provider,**

**options: {**

**redirectTo: `http://example.com/auth/callback`,**

**},**

**})**

At the callback endpoint, handle the code exchange to save the user session.

**Create a new file at src/routes/auth/callback/+server.js and populate with the following:**

**src/routes/auth/callback/+server.js**

import { redirect } from '@sveltejs/kit';

export const GET = async (event) => {

const {

url,

locals: { supabase }

} = event;

const code = url.searchParams.get('code') as string;

const next = url.searchParams.get('next') ?? '/';

if (code) {

const { error } = await supabase.auth.exchangeCodeForSession(code)

if (!error) {

throw redirect(303, `/${next.slice(1)}`);

}

}

// return the user to an error page with instructions

throw redirect(303, '/auth/auth-code-error');

};

After a successful code exchange, the user's session will be saved to cookies.

Saving Google tokens#

The tokens saved by your application are the Supabase Auth tokens. Your app might additionally need the Google OAuth 2.0 tokens to access Google services on the user's behalf.

On initial login, you can extract the provider\_token from the session and store it in a secure storage medium. The session is available in the returned data from signInWithOAuth (implicit flow) and exchangeCodeForSession (PKCE flow).

Google does not send out a refresh token by default, so you will need to pass parameters like these to signInWithOAuth() in order to extract the provider\_refresh\_token:

**const { data, error } = await supabase.auth.signInWithOAuth({**

**provider: 'google',**

**options: {**

**queryParams: {**

**access\_type: 'offline',**

**prompt: 'consent',**

**},**

**},**

**})**