Click to prove you're human

Auth is a user management and authentication server written in Go that powersSupabase's features such as:Issuing JWTsRow Level Security with PostgRESTUser managementSign in with email, password, magic link, phone numberSign in with external providers (Google, Apple, Facebook, Discord, ...)It is originally based on the excellentGoTrue codebase by Netlify, however both have diverged significantly in features and capabilities.If you wish to contribute to the project, please refer to the contributing guide. Quick StartRunning in ProductionConfigurationEndpoints Create a .env file to store your own custom env vars. See example.envStart the local postgres database in a postgres container: docker-compose -f docker-compose-dev.yml up postgresBuild the auth binary: make build . You should see an output like this:go build -ldflags "-X github.com/supabase/auth/cmd.Version=`git rev-parse HEAD`"GOOS=linux GOARCH=arm64 go build -ldflags "-X github.com/supabase/auth/cmd.Version=`git rev-parse HEAD`" -o gotrue-arm64Execute the auth binary: ./auth Create a .env.docker file to store your own custom env vars. See example.docker.envmake buildmake devdocker ps should show 2 docker containers (auth_postgresql and gotrue_gotrue)That's it! Visit the health checkendpoint to confirm that auth is running. Running an authentication server in production is not an easy feat. Werecommend using Supabase Auth which gets regularsecurity updates.Otherwise, please make sure you setup a process to promptly update to thelatest version. You can do that by following this repository, specifically theReleases and SecurityAdvisories sections. Auth uses the Semantic Versioning scheme. Here are somefurther clarifications on backward compatibility guarantees:Go API compatibilityAuth is not meant to be used as a Go library. There are no guarantees onbackward API compatibility when used this way regardless which version numberchanges.PatchChanges to the patch version guarantees backward compatibility with:Database objects (tables, columns, indexes, functions).REST APIJWT structureConfigurationGuaranteed examples:A column won't change its type.A table won't change its primary key.An index will not be removed.A uniqueness constraint will not be removed.A REST API will not be removed.Parameters to REST APIs will work equivalently as before (or better, if a bughas been fixed).Configuration will not change.Not guaranteed examples:A table may add new columns.Columns in a table may be reordered.Non-unique constraints may be removed (database level checks, null, defaultvalues).JWT may add new properties.MinorChanges to minor version guarantees backward compatibility with:REST APIJWT structureConfigurationExceptions to these guarantees will be made only when serious security issuesare found that can't be remedied in any other way.Guaranteed examples:Existing APIs may be deprecated but continue working for the next few minorversion releases.Configuration changes may become deprecated but continue working for the nextfew minor version releases.Already issued JWTs will be accepted, but new JWTs may be with a differentstructure (but usually similar).Not guaranteed examples:Removal of JWT fields after a deprecation notice.Removal of certain APIs after a deprecation notice.Removal of sign-in with external providers, after a deprecation notice.Deletion, truncation, significant schema changes to tables, indexes, views,functions.We aim to provide a deprecation notice in execution logs for at least two majorversion releases or two weeks if multiple releases go out. Compatibility willbe guaranteed while the notice is live.MajorChanges to the major version do not guarantee any backward compatibility withprevious versions. Certain inherited features from the Netlify codebase are not supported bySupabase and they may be removed without prior notice in the future. This is acomprehensive list of those features:Multi-tenancy via the instances table i.e. GOTRUE_MULTI_INSTANCE_MODEconfiguration parameter.System user (zero UUID user).Super admin via the is_super_admin column.Group information in JWTs via GOTRUE_JWT_ADMIN_GROUP_NAME and otherconfiguration fields.Symmetrics JWTs. In the future it is very likely that Auth will beginissuing asymmetric JWTs (subject to configuration), so do not rely on theassumption that only HS256 signed JWTs will be issued long term.Note that this is not an exhaustive list and it may change. These are some best practices to follow when self-hosting to ensure backwardcompatibility with Auth:Do not modify the schema managed by Auth. You can see all of themigrations in the migrations directory.Do not rely on schema and structure of data in the database. Always useAuth APIs and JWTs to infer information about users.Always run Auth behind a TLS-capable proxy such as a load balancer, CDN,nginx or other similar software. You may configure Auth using either a configuration file named .env,environment variables, or a combination of both. Environment variables are prefixed with GOTRUE_, and will always have precedence over values provided via file. GOTRUE_SITE_URL= - string requiredThe base URL your site is located at. Currently used in combination with other settings to construct URLs used in emails. Any URI that shares a host with SITE_URL is a permitted value for redirect_to params (see /authorize etc.).URI_ALLOW_LIST - stringA comma separated list of URIs (e.g. " ,https://*.foo.example.com, ") which are permitted as valid redirect_to destinations. Defaults to []. Supports wildcard matching through globbing. e.g. https://*.foo.example.com will allow and to be accepted. Globbing is also supported on subdomains. e.g. will allow and to be accepted.For more common glob patterns, check out the following link.OPERATOR_TOKEN - string Multi-instance mode onlyThe shared secret with an operator (usually Netlify) for this microservice. Used to verify requests have been proxied through the operator andthe payload values can be trusted.DISABLE_SIGNUP - boolWhen signup is disabled the only way to create new users is through invites. Defaults to false, all signups enabled.GOTRUE_EXTERNAL_EMAIL_ENABLED - boolUse this to disable email signups (users can still use external oauth providers to sign up / sign in)GOTRUE_EXTERNAL_PHONE_ENABLED - boolUse this to disable phone signups (users can still use external oauth providers to sign up / sign in)GOTRUE_RATE_LIMIT_HEADER - stringHeader on which to rate limit the /token endpoint.GOTRUE_RATE_LIMIT_EMAIL_SENT - stringRate limit the number of emails sent per hr on the following endpoints: /signup, /invite, /magiclink, /recover, /otp, & /user.GOTRUE_PASSWORD_MIN_LENGTH - intMinimum password length, defaults to 6.GOTRUE_PASSWORD_REQUIRED_CHARACTERS - a string of character sets separated by :. A password must contain at least one character of each set to be accepted. To use the : character escape it with \.GOTRUE_SECURITY_REFRESH_TOKEN_ROTATION_ENABLED - boolIf refresh token rotation is enabled, auth will automatically detect malicious attempts to reuse a revoked refresh token. When a malicious attempt is detected, gotrue immediately revokes all tokens that descended from the offending token.GOTRUE_SECURITY_REFRESH_TOKEN_REUSE_INTERVAL - stringThis setting is only applicable if GOTRUE_SECURITY_REFRESH_TOKEN_ROTATION_ENABLED is enabled. The reuse interval for a refresh token allows for exchanging the refresh token multiple times during the interval to support concurrency or offline issues. During the reuse interval, auth will not consider using a revoked token as a malicious attempt and will simply return the child refresh token.Only the previous revoked token can be reused. Using an old refresh token way before the current valid refresh token will trigger the reuse detection. GOTRUE_API_HOST=localhostPORT=9999API_EXTERNAL_URL= API_HOST - stringHostname to listen on.PORT (no prefix) / API_PORT - numberPort number to listen on. Defaults to 8081.API_ENDPOINT - string Multi-instance mode onlyControls what endpoint Netlify can access this API on.API_EXTERNAL_URL - string requiredThe URL on which Gotrue might be accessed at.REQUEST_ID_HEADER - stringIf you wish to inherit a request ID from the incoming request, specify the name in this value. GOTRUE_DB_DRIVER=postgresDATABASE_URL=root@localhost/authDB_DRIVER - string requiredChooses what dialect of database you want. Must be postgres.DATABASE_URL (no prefix) / DB_DATABASE_URL - string requiredConnection string for the database.GOTRUE_DB_MAX_POOL_SIZE - intSets the maximum number of open connections to the database. Defaults to 0 which is equivalent to an "unlimited" number of connections.DB_NAMESPACE - stringAdds a prefix to all table names.Migrations NoteMigrations are applied automatically when you run ./auth. However, you also have the option to rerun the migrations via the following methods:If built locally: ./auth migrateUsing Docker: docker run --rm auth gotrue migrate LOG_LEVEL=debug # available without GOTRUE prefix (exception)GOTRUE_LOG_FILE=/var/log/go/auth.logLOG_LEVEL - stringControls what log levels are output. Choose from panic, fatal, error, warn, info, or debug. Defaults to info.LOG_FILE - stringIf you wish logs to be written to a file, set log_file to a valid file path. Auth has basic observability built in. It is able to exportOpenTelemetry metrics and traces to a collector. To enable tracing configure these variables:GOTRUE_TRACING_ENABLED - booleanGOTRUE_TRACING_EXPORTER - string only opentelemetry supportedMake sure you also configure the OpenTelemetryExporterconfiguration for your collector or service.For example, if you useHoneycomb.ioyou should set these standard OpenTelemetry OTLP variables:OTEL_SERVICE_NAME=authOTEL_EXPORTER_OTLP_PROTOCOL=grpcOTEL_EXPORTER_OTLP_ENDPOINT= OTEL_EXPORTER_OTLP_HEADERS="x-honeycomb-team=,x-honeycomb-dataset=auth" To enable metrics configure these variables:GOTRUE_METRICS_ENABLED - booleanGOTRUE_METRICS_EXPORTER - string only opentelemetry and prometheussupportedMake sure you also configure the OpenTelemetryExporterconfiguration for your collector or service.If you use the prometheus exporter, the server host and port can beconfigured using these standard OpenTelemetry variables:OTEL_EXPORTER_PROMETHEUS_HOST - IP address, default 0.0.0.0OTEL_EXPORTER_PROMETHEUS_PORT - port number, default 9100The metrics are exported on the / path on the server.If you use the opentelemetry exporter, the metrics are pushed to thecollector.For example, if you useHoneycomb.ioyou should set these standard OpenTelemetry OTLP variables:OTEL_SERVICE_NAME=authOTEL_EXPORTER_OTLP_PROTOCOL=grpcOTEL_EXPORTER_OTLP_ENDPOINT= OTEL_EXPORTER_OTLP_HEADERS="x-honeycomb-team=,x-honeycomb-dataset=auth"Note that Honeycomb.io requires a paid plan to ingest metrics.If you need to debug an issue with traces or metrics not being pushed, you canset DEBUG=true to get more insights from the OpenTelemetry SDK. When using the OpenTelemetry tracing or metrics exporter you can define customresource attributes using the standard OTEL_RESOURCE_ATTRIBUTES environmentvariable.A default attribute auth.version is provided containing the build version. All HTTP calls to the Auth API are traced. Routes use the parametrizedversion of the route, and the values for the route parameters can be found asthe http.route.params. span attribute.For example, the following request:GET /admin/users/4acde936-82dc-4552-b851-831fb8ce0927/will be traced as:http.method = GEThttp.route = /admin/users/{user_id}http.route.params.user_id = 4acde936-82dc-4552-b851-831fb8ce0927 All of the Go runtime metrics are exposed. Some HTTP metrics are also collectedby default. GOTRUE_JWT_SECRET=supersecretvalueGOTRUE_JWT_EXP=3600GOTRUE_JWT_AUD=netlifyJWT_SECRET - string requiredThe secret used to sign JWT tokens with.JWT_EXP - numberHow long tokens are valid for, in seconds. Defaults to 3600 (1 hour).JWT_AUD - stringThe default JWT audience. Use audiences to group users.JWT_ADMIN_GROUP_NAME - stringThe name of the admin group (if enabled). Defaults to admin.JWT_DEFAULT_GROUP_NAME - stringThe default group to assign all new users to. We support apple, azure, bitbucket, discord, facebook, figma, github, gitlab, google, keycloak, linkedin, notion, spotify, slack, twitch, twitter and workos for external authentication.Use the names as the keys underneath external to configure each separately.GOTRUE_EXTERNAL_GITHUB_ENABLED=trueGOTRUE_EXTERNAL_GITHUB_CLIENT_ID=myappclientidGOTRUE_EXTERNAL_GITHUB_SECRET=clientsecretvaluessshGOTRUE_EXTERNAL_GITHUB_REDIRECT_URI= external providers are required, but you must provide the required values if you choose to enable any.EXTERNAL_X_ENABLED - boolWhether this external provider is enabled or notEXTERNAL_X_CLIENT_ID - string requiredThe OAuth2 Client ID registered with the external provider.EXTERNAL_X_SECRET - string requiredThe OAuth2 Client Secret provided by the external provider when you registered.EXTERNAL_X_REDIRECT_URI - string requiredThe URI a OAuth2 provider will redirect to with the code and state values.EXTERNAL_X_URL - stringThe base URL used for constructing the URLs to request authorization and access tokens. Used by gitlab and keycloak. For gitlab it defaults to . For keycloak you need to set this to your instance, for example: To try out external authentication with Apple locally, you will need to do the following:Remap localhost to in your /etc/hosts config.Configure auth to serve HTTPS traffic over localhost by replacing ListenAndServe in api.go with: func (a *API) ListenAndServe(hostAndPort string) { log := logrus.WithField("component", "api") path, err := os.Getwd() if err != nil { log.Println(err) } server := &http.Server{ Addr: hostAndPort, Handler: a.handler, } done := make(chan struct{}) defer close(done) go func() { waitForTermination(log, done) ctx, cancel := context.WithTimeout(context.Background(), time.Minute) defer cancel() server.Shutdown(ctx) }() if err := server.ListenAndServeTLS("PATH_TO_CRT_FILE", "PATH_TO_KEY_FILE"); err != http.ErrServerClosed { log.WithError(err).Fatal("http server listen failed") } }Generate the crt and key file. See here for more information.Generate the GOTRUE_EXTERNAL_APPLE_SECRET by following this post! Sending email is not required, but highly recommended for password recovery.If enabled, you must provide the required values below.GOTRUE_SMTP_HOST=smtp.mandrillapp.comGOTRUE_SMTP_PORT=587GOTRUE_SMTP_USER=smtp-delivery@example.comGOTRUE_SMTP_PASS=correcthorsebatterystapleGOTRUE_SMTP_ADMIN_EMAIL=support@example.comGOTRUE_MAILER_SUBJECTS_CONFIRMATION="Please confirm"SMTP_ADMIN_EMAIL - string requiredThe From email address for all emails sent.SMTP_HOST - string requiredThe mail server hostname to send emails through.SMTP_PORT - number requiredThe port number to connect to the mail server on.SMTP_USER - stringIf the mail server requires authentication, the username to use.SMTP_PASS - stringIf the mail server requires authentication, the password to use.SMTP_MAX_FREQUENCY - numberControls the minimum amount of time that must pass before sending another signup confirmation or password reset email. The value is the number of seconds. Defaults to 900 (15 minutes).SMTP_SENDER_NAME - stringSets the name of the sender. Defaults to the SMTP_ADMIN_EMAIL if not used.MAILER_AUTOCONFIRM - boolIf you do not require email confirmation, you may set this to true. Defaults to false.MAILER_OTP_EXP - numberControls the duration an email link or otp is valid for.MAILER_URLPATHS_INVITE - stringURL path to use in the user invite email. Defaults to /verify.MAILER_URLPATHS_CONFIRMATION - stringURL path to use in the signup confirmation email. Defaults to /verify.MAILER_URLPATHS_RECOVERY - stringURL path to use in the password reset email. Defaults to /verify.MAILER_URLPATHS_EMAIL_CHANGE - stringURL path to use in the email change confirmation email. Defaults to /verify.MAILER_SUBJECTS_INVITE - stringEmail subject to use for user invite. Defaults to You have been invited.MAILER_SUBJECTS_CONFIRMATION - stringEmail subject to use for signup confirmation. Defaults to Confirm Your Signup.MAILER_SUBJECTS_RECOVERY - stringEmail subject to use for password reset. Defaults to Reset Your Password.MAILER_SUBJECTS_MAGIC_LINK - stringEmail subject to use for magic link email. Defaults to Your Magic Link.MAILER_SUBJECTS_EMAIL_CHANGE - stringEmail subject to use for email change confirmation. Defaults to Confirm Email Change.MAILER_TEMPLATES_INVITE - stringURL path to an email template to use when inviting a user. (e.g. SiteURL, Email, and ConfirmationURL variables are available.Default Content (if template is unavailable):You have been invited You have been invited to create a user on {{ .SiteURL }}. Follow this link to accept the invite:Accept the inviteMAILER_TEMPLATES_CONFIRMATION - stringURL path to an email template to use when confirming a signup. (e.g. SiteURL, Email, and ConfirmationURL variables are available.Default Content (if template is unavailable):Confirm your signup Follow this link to confirm your user:Confirm your mailMAILER_TEMPLATES_RECOVERY - stringURL path to an email template to use when resetting a password. (e.g. SiteURL, Email, and ConfirmationURL variables are available.Default Content (if template is unavailable):Reset Password Follow this link to reset the password for your user:Reset PasswordMAILER_TEMPLATES_MAGIC_LINK - stringURL path to an email template to use when sending magic link. (e.g. SiteURL, Email, and ConfirmationURL variables are available.Default Content (if template is unavailable):Magic Link Follow this link to login:Log InMAILER_TEMPLATES_EMAIL_CHANGE - stringURL path to an email template to use when confirming the change of an email address. (e.g. SiteURL, Email, NewEmail, and ConfirmationURL variables are available.Default Content (if template is unavailable):Confirm Change of Email Follow this link to confirm the update of your email from {{ .Email }} to {{ .NewEmail }}:Change Email SMS_AUTOCONFIRM - boolIf you do not require phone confirmation, you may set this to true. Defaults to false.SMS_MAX_FREQUENCY - numberControls the minimum amount of time that must pass before sending another sms otp. The value is the number of seconds. Defaults to 60 (1 minute)).SMS_OTP_EXP - numberControls the duration an sms otp is valid for.SMS_OTP_LENGTH - numberControls the number of digits of the sms otp sent.SMS_PROVIDER - stringAvailable options are: twilio, messagebird, textlocal, and vonageThen you can use your twilio credentials:SMS_TWILIO_ACCOUNT_SIDSMS_TWILIO_AUTH_TOKENSMS_TWILIO_MESSAGE_SERVICE_SID - can be set to your twilio sender mobile numberOr Messagebird credentials, which can be obtained in the Dashboard:SMS_MESSAGEBIRD_ACCESS_KEY - your Messagebird access keySMS_MESSAGEBIRD_ORIGINATOR - SMS sender (your Messagebird phone number with + or company name)If enabled, CAPTCHA will check the request body for the captcha_token field and make a verification request to the CAPTCHA provider.SECURITY_CAPTCHA_ENABLED - stringWhether captcha middleware is enabledSECURITY_CAPTCHA_PROVIDER - stringfor now the only options supported are: hcaptcha and turnstileSECURITY_CAPTCHA_SECRET - stringSECURITY_CAPTCHA_TIMEOUT - stringRetrieve from hcaptcha or turnstile account SECURITY_UPDATE_PASSWORD_REQUIRE_REAUTHENTICATION - boolEnforce reauthentication on password update. GOTRUE_EXTERNAL_ANONYMOUS_USERS_ENABLED - boolUse this to enable/disable anonymous sign-ins. Auth exposes the following endpoints: Returns the publicly available settings for this auth instance.{ "external": { "apple": true, "azure": true, "bitbucket": true, "discord": true, "facebook": true, "figma": true, "github": true, "gitlab": true, "google": true, "keycloak": true, "linkedin": true, "notion": true, "slack": true, "spotify": true, "twitch": true, "twitter": true, "workos": true }, "disable_signup": false, "autoconfirm": false} Creates (POST) or Updates (PUT) the user based on the user_id specified. The ban_duration field accepts the following time units: "ns", "us", "ms", "s", "m", "h". See time.ParseDuration for more details on the format used.headers:{ "Authorization": "Bearer eyJhbGciOiJI...M3A90LCkxxtX9oNP9KZO" // requires a role claim that can be set in the GOTRUE_JWT_ADMIN_ROLES env var} body:{ "role": "test-user", "email": "email@example.com", "phone": "12345678", "password": "secret", // only if type = signup "email_confirm": true, "phone_confirm": true, "user_metadata": {}, "app_metadata": {}, "ban_duration": "24h" or "none" // to unban a user} Returns the corresponding email action link based on the type specified. Among other things, the response also contains the query params of the action link as separate JSON fields for convenience (along with the email OTP from which the corresponding token is generated).headers:{ "Authorization": "Bearer eyJhbGciOiJI...M3A90LCkxxtX9oNP9KZO" // admin role required} body:{ "type": "signup" or "magiclink" or "recovery" or "invite", "email": "email@example.com", "password": "secret", // only if type = signup "data": { ... }, // only if type = signup "redirect_to": " " // Redirect URL to send the user to after an email action. Defaults to SITE_URL. }Returns:{ "action_link": " "email_otp": "EMAIL_OTP", "hashed_token": "TOKEN", "verification_type": "TYPE", "redirect_to": "REDIRECT_URL", ...} Register a new user with an email and password.{ "email": "email@example.com", "password": "secret"}returns:{ "id": "11111111-2222-3333-4444-5555555555555", "email": "email@example.com", "confirmation_sent_at": "2016-05-15T20:49:40.882805774-07:00", "created_at": "2016-05-15T19:53:12.368652374-07:00", "updated_at": "2016-05-15T19:53:12.368652374-07:00"} // if sign up is a duplicate then faux data will be returned// as to not leak information about whether a given email// has an account with your service or notRegister a new user with a phone number and password.{ "phone": "12345678", // follows the E.164 format "password": "secret"}Returns:{ "id": "11111111-2222-3333-4444-5555555555555", // if duplicate sign up, this ID will be faux "phone": "12345678", "confirmation_sent_at": "2016-05-15T20:49:40.882805774-07:00", "created_at": "2016-05-15T19:53:12.368652374-07:00", "updated_at": "2016-05-15T19:53:12.368652374-07:00"}if AUTOCONFIRM is enabled and the sign up is a duplicate, then the endpoint will return:{ "code":400, "msg":"User already registered"} Allows a user to resend an existing signup, sms, email_change or phone_change OTP.{ "email": "user@example.com", "type": "signup"}{ "phone": "12345678", "type": "sms"}returns:{ "message_id": "msgid123456"} Invites a new user with an email.This endpoint requires the service_role or supabase_admin JWT set as an Auth Bearer header:e.g.headers: { "Authorization" : "Bearer eyJhbGciOiJI...M3A90LCkxxtX9oNP9KZO"}{ "email": "email@example.com"}Returns:{ "id": "11111111-2222-3333-4444-5555555555555", "email": "email@example.com", "confirmation_sent_at": "2016-05-15T20:49:40.882805774-07:00", "created_at": "2016-05-15T19:53:12.368652374-07:00", "updated_at": "2016-05-15T19:53:12.368652374-07:00", "invited_at": "2016-05-15T19:53:12.368652374-07:00"} Verify a registration or a password recovery. Type can be signup or recovery or inviteand the token is a token returned from either /signup or /recover.{ "type": "signup", "token": "confirmation-code-delivered-in-email"}password is required for signup verification if no existing password exists.Returns:{ "access_token": "jwt-token-representing-the-user", "token_type": "bearer", "expires_in": 3600, "refresh_token": "a-refresh-token", "type": "signup | recovery | invite"}Verify a phone signup or sms otp. Type should be set to sms.{ "type": "sms", "token": "confirmation-otp-delivered-in-sms", "redirect_to": " ", "phone": "phone-number-sms-otp-was-delivered-to"}Returns:{ "access_token": "jwt-token-representing-the-user", "token_type": "bearer", "expires_in": 3600, "refresh_token": "a-refresh-token"} Verify a registration or a password recovery. Type can be signup or recovery or magiclink or inviteand the token is a token returned from either /signup or /recover or /magiclink.query params:{ "type": "signup", "token": "confirmation-code-delivered-in-email", "redirect_to": " "}User will be logged in and redirected to:SITE_URL/#access_token=jwt-token-representing-the-user&token_type=bearer&expires_in=3600&refresh_token=a-refresh-token&type=inviteYour app should detect the query params in the fragment and use them to set the session (supabase-js does this automatically)You can use the type param to redirect the user to a password set form in the case of invite or recovery,or show an account confirmed/welcome message in the case of signup, or direct them to some additional onboarding flow One-Time-Password. Will deliver a magiclink or sms otp to the user depending on whether the request body contains an "email" or "phone" key.If "create_user": true, user will not be automatically signed up if the user doesn't exist.{ "phone": "12345678" // follows the E.164 format "create_user": true}OR// exactly the same as /magiclink{ "email": "email@example.com" "create_user": true}Returns: Magic Link. Will deliver a link (e.g. /verify?type=magiclink&token=fgtyuf68ddqdaDd) to the user based onemail address which they can use to redeem an access_token.By default Magic Links can only be sent once every 60 seconds{ "email": "email@example.com"}Returns: when clicked the magic link will redirect the user to #access_token=x&refresh_token=y&expires_in=z&token_type=bearer&type=magiclink (see /verify above) Password recovery. Will deliver a password recovery mail to the user based onemail address.By default recovery links can only be sent once every 60 seconds{ "email": "email@example.com"}Returns: This is an OAuth2 endpoint that currently implementsthe password and refresh_token grant typesquery params: body:// Email login{ "email": "name@domain.com", "password": "somepassword"} // Phone login{ "phone": "12345678", "password": "somepassword"}orquery params: body:{ "refresh_token": "a-refresh-token"}Once you have an access token, you can access the methods requiring authenticationby settings the Authorization: Bearer YOUR_ACCESS_TOKEN_HERE header.Returns:{ "access_token": "jwt-token-representing-the-user", "token_type": "bearer", "expires_in": 3600, "refresh_token": "a-refresh-token"} Get the JSON object for the logged in user (requires authentication)Returns:{ "id": "11111111-2222-3333-4444-5555555555555", "email": "email@example.com", "confirmation_sent_at": "2016-05-15T20:49:40.882805774-07:00", "created_at": "2016-05-15T19:53:12.368652374-07:00", "updated_at": "2016-05-15T19:53:12.368652374-07:00"} Update a user (Requires authentication). Apart from changing email/password, thismethod can be used to set custom user data. Changing the email will result in a magiclink being sent out.{ "email": "new-email@example.com", "password": "new-password", "phone": "+123456789", "data": { "key": "value", "number": 10, "admin": false }}Returns:{ "id": "11111111-2222-3333-4444-5555555555555", "email": "email@example.com", "email_change_sent_at": "2016-05-15T20:49:40.882805774-07:00", "phone": "+123456789", "phone_change_sent_at": "2016-05-15T20:49:40.882805774-07:00", "created_at": "2016-05-15T19:53:12.368652374-07:00", "updated_at": "2016-05-15T19:53:12.368652374-07:00"}If GOTRUE_SECURITY_UPDATE_PASSWORD_REQUIRE_REAUTHENTICATION is enabled, the user will need to reauthenticate first.{ "password": "new-password", "nonce": "123456"} Sends a nonce to the user's email (preferred) or phone. This endpoint requires the user to be logged in / authenticated first. The user needs to have either an email or phone number for the nonce to be sent successfully.headers: { "Authorization" : "Bearer eyJhbGciOiJI...M3A90LCkxxtX9oNP9KZO"} Logout a user (Requires authentication).This will revoke all refresh tokens for the user. Remember that the JWT tokenswill still be valid for stateless auth until they expires. Get access_token from external oauth providerquery params:provider=apple | azure | bitbucket | discord | facebook | figma | github | gitlab | google | keycloak | linkedin | notion | slack | spotify | twitch | twitter | workosscopes=Redirects to provider and then to /callbackFor apple specific setup see: External provider should redirect to hereRedirects to #access_token=&refresh_token=&provider_token=&expires_in=3600&provider=If additional scopes were requested then provider_token will be populated, you can use this to fetch additional data from the provider or interact with their services 1. Overview Supabase is a backend as a service(BaaS) platform, which is an open source alternative to firebase, but it has better services interms of cost and setup. there are many services including, but not limited to, authentication, postgres database, storage and more. For this guide, i will guide you step by step from creating and configuring a fully automated auth + db, including syncing user data, i am using nextjs but you can use any frameworks of your choice. 2. Prerequisite nextjs 14 or 15 (app router) 3. Supabase setup Step 1: Go to supabase and signup, then create a new project by selecting your organization. Step 2 : Once you created, go to APIs page and copy your url and anon key and store them in .env.local, name them like this: NEXT_PUBLIC_SUPABASE_URL, NEXT_PUBLIC_SUPABASE_ANON_KEY. Step 3: Now, i am going to use Google as a provider so you need to create a project in Google cloud platform and get your Oauth credentials(client_id and client_secret). Follow this guide on how to obtain them. Step 4 : Great, you came all this way, go to Authentication => Providers => Google, insert your client_id and client_secret and copy callback url, go back to Google cloud and peste it in Authorized redirect URLs section in Credentials page of your project. Now enable the provider and save it. weve finished the basic setup for auth now. Step 5 : Go to database section, click connect button on top, get your connection string so that you can push the table schema from your local repo. after pushing your schema, it will create a table for you. So what we want is when a new user signs up using google, he will be added to the auth and synced to the database automatically. there are no webhooks to do this, but you can use triggers and functions to call the db and insert user data. Go to Sql editor section and create a function, this will be called by the trigger when a user signs up, which in turn inserts user data to the table. make sure to change fields as your need:create or replace function public.sync_user() returns trigger as $$begin insert into public."User" ("supabaseUserId", email, "fullName", "avatarUrl") values (new.id, -- maps to supabaseUserId new.email, coalesce(new.raw_user_meta_data->>'full_name', null), coalesce(new.raw_user_meta_data->>'avatar_url', null)) on conflict ("supabaseUserId") do nothing; return new;end;$$ language plpgsql security definer; Now, create a trigger, which will be called on a new user sign ups:create trigger on_auth_user_createdafter insert on auth.usersfor each rowexecute function public.sync_user(); execute both of queries and now you have a fully automated user data sync with supabase . 4. Implementation in Next.js Step 1 : create a new nextjs app using: npx create-next-app@latest Step 2 : once installed, open it up in your favorite editor and install supabase package using: npm install @supabase/supabase-js Step 3 : create a file named supabase.ts, and peste the following code:import { createClient } from "@supabase/supabase-js";const supabaseUrl = process.env.NEXT_PUBLIC_SUPABASE_URL!;const supabaseKey = process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!;export const supabase = createClient(supabaseUrl, supabaseKey); Step 4 : create a global store for managing auth state, in my case i am using zustand but you can use your own state manager and implement like mine:import { create } from "zustand";import { supabase } from "@/lib/supabase";type AuthState = { user: any | null; isLoading: boolean; setUser: (user: any | null) => void; setLoading: (loading: boolean) => void; signOut: () => Promise;};export const useAuthStore = create((set) => ({ user: null, isLoading: true, setUser: (user) => set({ user }), setLoading: (loading) => set({ isLoading: loading }), signOut: async () => { await supabase.auth.signOut(); set({ user: null }); },})); Step 5 : create a sign in page, app/(auth)/signin:"use client"import { Button } from "@/components/ui/button";import { supabase } from "@/lib/supabase"export default function GoogleSignIn() { const handleSignIn = async () => { const { data, error } = await supabase.auth.signInWithOAuth({ provider: 'google', options: { redirectTo: `${window.location.origin}` } }) console.log("Supadata", data); if (error) console.error(error) } return (Sign in with Google)} Step 6 : create a sign out page: app/(auth)/signout:"use client"import { useAuthStore } from "@/store/authStore";import { useRouter } from "next/navigation";export default function SignOutButton() { const { signOut } = useAuthStore(); const router = useRouter(); const handleSignOut = async () => { await signOut(); router.push("/signin"); }; return Sign Out;} Step 7 : create AuthProvider.ts file to manage auth state globally:"use client";import { useEffect } from "react";import { supabase } from "@/lib/supabase";import { useAuthStore } from "@/store/authStore";export default function AuthProvider({ children }: { children: React.ReactNode }) { const setUser = useAuthStore((state) => state.setUser); const setLoading = useAuthStore((state) => state.setLoading); useEffect(() => { const initializeAuth = async () => { setLoading(true); const { data: { session }, error } = await supabase.auth.getSession(); if (error) { console.error("Error fetching session:", error); setUser(null); } else if (session) { setUser(session.user); } else { setUser(null); } setLoading(false); }; initializeAuth(); // Listen for authentication changes const { data: authListener } = supabase.auth.onAuthStateChange((_, session) => { setUser(session?.user || null); }); return () => { authListener.subscription.unsubscribe(); }; }, [setUser, setLoading]); return {children};} Step 8 : create Authenticated.ts file which controls access to pages:"use client";import { useEffect } from "react";import { useRouter } from "next/navigation";import { useAuthStore } from "@/store/authStore";export default function Authenticated({ children }: { children: React.ReactNode }) { const user = useAuthStore((state) => state.user); const isLoading = useAuthStore((state) => state.isLoading); const router = useRouter(); useEffect(() => { if (!isLoading && !user) { router.replace("/signin"); } }, [user, isLoading, router]); return {children};} Step 9 : Now wrap root layout.ts using AuthProvider and Authenticated (optional, you can use on specific pages):import "./globals.css";import AuthProvider from "@/components/AuthProvider";import Authenticated from "@/components/Authenticated";export default function RootLayout({ children,}: Readonly) { return ({children});} 5. Conclusion In this article, you learned how to: Set up Supabase authentication in a Next.js app. Use Google as an authentication provider. Automatically sync user data to the PostgreSQL database.With the default options, the module requires a log-in page and a confirm page to handle the PKCE authorization code flow. If you want to understand how it works under the hood, you can read this section.All you need to do is to create a login.vue and confirm.vue page in the pages folder.For advanced users who want to implement the auth behaviour themselves, you can disable or override the redirect options.Log-in page - /loginEach time a user is trying to access a page that needs authentication, he will automatically be redirected to the configured log in page. If you want to allow access to "public" page, you just need to add them in the exclude redirect option. Alternatively, you can enable the redirect only for certain routes using the include redirect option.div]:my-2.5 [&_ul]:my-2.5 [&_ol]:my-2.5 [&>*]:last:!mb-0 [&_ul]:ps-4.5 [&_ol]:ps-4.5 [&_li]:my-0 transition-colors border border-warning/25 bg-warning/10 text-warning-600 dark:text-warning-300 [&_a]:text-warning [&_a]:hover:border-warning [&_code]:text-warning-600 dark:[&_code]:text-warning-300 [&_code]:border-warning/25 [&_a]:hover:[&>code]:border-warning [&_a]:hover:[&>code]:text-warning [&>ul]:marker:text-warning/50>Ensure to activate the authentication providers you want in the Supabase Dashboard under Authentication -> Providers.The log-in page initiates the log-in method(s) you choose from the available authorization methods provided by Supabase, it could looks like:pages/login.vueconst supabase = useSupabaseClient()const email = ref('')const signInWithOtp = async () => { const { error } = await supabase.auth.signInWithOtp({ email: email.value, options: { emailRedirectTo: ' , } }) if (error) console.log(error)} Sign In with E-Mail Once the authorization flow is triggered using the auth wrapper of the useSupabaseClient composable, the session management is handled automatically and the user will be redirected to the page you specify in the redirect option (/confirm by default).Confirm page - /confirmThe confirmation page receives the supabase callback which contains session information. The supabase client automatically detects and handles this, and once the session is confirmed the user value will automatically be updated. From there you can redirect to the appropriate page.div]:my-2.5 [&_ul]:my-2.5 [&_ol]:my-2.5 [&>*]:last:!mb-0 [&_ul]:ps-4.5 [&_ol]:ps-4.5 [&_li]:my-0 transition-colors border border-success/25 bg-success/10 text-success-600 dark:text-success-300 [&_a]:text-success [&_a]:hover:border-success [&_code]:text-success-600 dark:[&_code]:text-success-300 [&_code]:border-success/25 [&_a]:hover:[&>code]:border-success [&_a]:hover:[&>code]:text-success [&>ul]:marker:text-success/50>The redirect URL must be configured in your Supabase dashboard under Authentication -> URL Configuration -> Redirect URLs.pages/confirm.vueconst user = useSupabaseUser()watch(user, () => { if (user.value) { // Redirect to protected page return navigateTo('/') }}, { immediate: true }) Waiting for login...You can easily handle redirection to the initial requested route after login using the useSupabaseCookieRedirect composable and the saveRedirectToCookie option.By setting the saveRedirectToCookie option to true, the module will automatically save the current path to a cookie when the user is redirected to the login page. When the user logs in, you can then retrieve the saved path from the cookie and redirect the user to it on the /confirm page:pages/confirm.vueconst user = useSupabaseUser()const redirectInfo = useSupabaseCookieRedirect()watch(user, () => { if (user.value) { // Get redirect path, and clear it from the cookie const path = redirectInfo.pluck() // Redirect to the saved path, or fallback to home return navigateTo(path || '/') }}, { immediate: true }) Waiting for login...If you want to manually set the redirect path, you can do so by disabling saveRedirectToCookie, and then set the value using the useSupabaseCookieRedirect composable directly.Reset PasswordYou can easily handle password reset request for the user. The flow consist of 2 steps:pages/password/reset.vueconst supabase = useSupabaseClient()const email = ref('')const requestResetPassword = async () => { const { data, error } = await supabase.auth.resetPasswordForEmail(email.value, { redirectTo: ' , }) if (error) console.log(error)} Reset Password div]:my-2.5 [&_ul]:my-2.5 [&_ol]:my-2.5 [&>*]:last:!mb-0 [&_ul]:ps-4.5 [&_ol]:ps-4.5 [&_li]:my-0 transition-colors border border-success/25 bg-success/10 text-success-600 dark:text-success-300 [&_a]:text-success [&_a]:hover:border-success [&_code]:text-success-600 dark:[&_code]:text-success-300 [&_code]:border-success/25 [&_a]:hover:[&>code]:border-success [&_a]:hover:[&>code]:text-success [&>ul]:marker:text-success/50>You can configure the URL that the user is redirected to with the redirectTo parameter.pages/password/update.vueconst supabase = useSupabaseClient()const newPassword = ref('')const updateUserPassword = async () => { const { data, error } = await supabase.auth.updateUser({ password: newPassword.value }) if (error) console.log(error)} Update Password You can also watch for the event PASSWORD_RECOVERY that will be emitted when the password recovery link is clicked. You can use onAuthStateChange() to listen and invoke a callback function on these events.pages/password/update.vueconst supabase = useSupabaseClient()const password = generateRandomPassword()watch(newPassword, () => { supabase.auth.onAuthStateChange(async (event, session) => { if (event == "PASSWORD_RECOVERY") { const { data, error } = await supabase.auth .updateUser({ password }) if (data) alert("Password updated successfully!") if (error) alert("There was an error updating your password.") } })})If you want to learn more about it, you can read this section. Today I'll teach you how to use Supabase to add a complete, simple, and secure authentication service to your project. If you are unfamiliar with Supabase, it is a Backend-as-a-Service (BaaS) that provides essential features such as a RESTful API connected to a Postgress database, serverless functions, and, most importantly, authentication, the topic of this blogpost.If you're interested to learn more about the RESTful API, check out my previous blogpost where I walk you through the process of creating a comprehensive CRUD application with Supabase.Table of contentsIntroductionSupabase setupFront-end setupConclusionIntroductionThe authentication process might be tough to grasp and even more difficult to include into your solution. This approach is now a piece of cake thanks to Supabase, which I talked about in a previous blogpost.The authentication solution provided by Supabase is a mix of JWT (JSON Web Token) and API key. Essentially, as long as a user makes API requests without a JWT, he will be deemed an anonymous user and will have restricted access to the API, in our case only the login and register API.If, on the other hand, the user uses a JWT to make API calls, he will be authorized to make any API calls within the scope of the role he has been assigned.Today, I'll be adding their authentication service to the previous blogpost's dashboard app built with Remix. In a nutshell, this is a dashboard that allows you to manage a list of video games. The client makes API calls to a database housed on Supabase.If you are unfamiliar with Supabase, I strongly advise you to read my my previous blogpost first because I will be working on a previously constructed Supabase project!The goals:1/ Add a login and a register page.2/ Prevent unauthorized users from accessing the dashboard.3/ Only allow authorized users to access the dashboard by sending requests with a JWT.Supabase setupBefore we dive into the code, we need to make some changes to the Supabase dashboard. First, we need to enable authentication and then apply policies to the database so that only authenticated users can interact with the dashboard.So first make sure to have enabled email authentication, to keep it simple I won't handle email confirmation. Navigate to Authentication > Providers via the left menu bar.To begin, ensure that email authentication is enabled; to keep things simple, I will not handle email confirmation. Using the left menu bar, navigate to Authentication > Providers.There are numerous providers available; in this example, we will use email authentication. The user must authenticate using the standard email/password combination.Supabase authentication settingsNext, we will add a policy to our database in order to restrict access to our database API. The following policy is straightforward; it grants read/write access to all authenticated users; however, keep in mind that you can go even farther.Allow any authenticated users to have access to our databaseThat's all there is to it! We are now ready to implement authentication in our front-end application.Front-end setupBefore we can truly implement the authentication process in our Remix app, as we did in Supabase, we need to make certain changes, which we will go over before continuing on.Implementing the authentication helperSupabase provides us with various authentication helpers to integrate into our app via a package that can be found here.It should be noted that Remix also provides utilities to assist us in handling user sessions, but because this article is about Supabase, we will use their helper.First, install the Remix package, which will handle session cookie management behind the scenes. They also provide clear and straightforward documentation on how to build it, which we will follow but modify somewhat.$ npm install @supabase/auth-helpers-remixIn my previous blogpost about Supabase, I created a function to use the supabase client ONLY in the server-side, but now we will need to use a supabase client in the client-side as well, so we will implement it.First let's update the server-side supabase client:supabase.tsimport { createServerClient } from "@supabase/auth-helpers-remix";import type { Database } from "~/types/database";// Initialize a supabase client to be use on the server-sideexport const getSupabaseServerClient = (request: Request, response: Response) => createServerClient(process.env.API_URL!, process.env.API_KEY!, { request, response, });Now we are going to update the root.tsx file which serves as the entry point of the application. We are going to create a client-side supabase client in it and expose it to every routes of the application.We will also build logic to synchronize our client and server supabase clients in the event that the client-side becomes desynchronized.First let's return the API_KEY and API_URL and the session from the server to the client with the loader function:root.tsx// Return the user session and env values to the clientexport const loader = async ({ request }: LoaderArgs) => { const response = new Response(); const env = { API_URL: process.env.API_URL!, API_KEY: process.env.API_KEY!, }; const supabase = createServerClient(process.env.API_URL!, process.env.API_KEY!, { request, response }); // Retrieves the user session to synchronize it with the client const { data: { session }, } = await supabase.auth.getSession(); return json({ env, session }, { headers: response.headers });};Now we will retrieve those values in the client component, store the Supabase client in a state, then implement the synchronization logic in a useEffect, and finally pass the supabase client to every routes via the Outlet context:root.tsxexport default function App() { const submit = useSubmit(); const { env, session } = useLoaderData(); const serverAccessToken = session?.access_token; // Create a single instance of the supabase client to be used on the client side const [supabase] = useState(createBrowserClient(env.API_URL, env.API_KEY)); // Synchronize the client-side supabase instance with the server-side one useEffect(() => { const { data: { subscription }, } = supabase.auth.onAuthStateChange((event, session) => { // If the tokens are not the same, this means client and server are not synchronized // Thus we send a get request to trigger the loader which will provide the client a new Supabase client instance if (session?.access_token !== serverAccessToken) { submit(null, { method: "get" }); } }); return () => { subscription.unsubscribe(); }; }, [serverAccessToken, submit, supabase.auth]); return ({/** Pass the supabase client down to every child components via the outlet context */});}We can now conduct API calls to the Supabase API from the client, and we have ensured that the client and server remain synced.The next step is to develop a login and a register page to authenticate a user in our application. Let's start with the registration page; you'll see a games folder and a games.tsx file below the files tree within the route folder; we'll go over this later.routesgames$gameId.tsxcreate.tsxindex.tsxgames.tsxindex.tsxlogin.tsxregister.tsxLet's create the register page.First, we will check the server to see if the user has already been authenticated; if so, he will be redirected to the dashboard:register.tsxexport const loader = async ({ request }: LoaderArgs) => { const response = new Response(); const supabase = getSupabaseServerClient(request, response); const { data: { session }, } = await supabase.auth.getSession(); if (session) { return redirect("/games", { headers: response.headers }); } return null;};Now, in the client component, we will retrieve the supabase client from the Outlet context in order to execute the authentication API call to Supabase from the client; if the call is successful, the user will be sent to the dashboard; otherwise, an error message will be displayed:register.tsxexport default function Register() { const { supabase } = useOutletContext(); const submit = useSubmit(); const [registerError, setRegisterError] = useState(); const methods = useForm({ defaultValues: { email: "", password: "", }, resolver: yupResolver(registerFormValidation), }); const onSubmit: SubmitHandler = async (values) => { const { error } = await supabase.auth.signUp(values); if (error) { setRegisterError(error.message); } else { // Trigger the current route loader via a get request, this will redirect the user to the dashboard submit(null, { method: "get" }); } }; return (// Your form...);}Moving on to the login page, you can use the same logic as the registration page, with the only significant difference being in your form submit handler function, which we will refer to as the sign in API route rather than the register API route: const onSubmit: SubmitHandler = async (values) => { const { error } = await supabase.auth.signInWithPassword(values); if (error) { setLoginError(error.message); } else { // Send a get request to trigger the current route loader submit(null, { method: "get" }); } };Great! We now have routes to register and authenticate a user, and thanks to the helper package we previously installed, we don't need to deal with session/cookies. You can check the application tab in Chrome Dev Tools and see that a cookie is created when you are successfully authenticated.Let's dive into the last part of this tutorial, which will block unauthenticated users from accessing the dashboard, because you may still access it without being authenticated, which we don't want.Let's take a look back at the routes files tree:routesgames { // Handle session changes }); Step 8: Implement Email Confirmations (Optional) If you enabled email confirmations in Step 3, make sure you handle the confirmation URL in your application. Check for a `confirmation_sent_at` timestamp in your users table to confirm the user's email status. Step 9: Deploy and Test Your Application Deploy your application using your preferred hosting service. Test the registration and login functionalities to ensure everything is working as expected. Step 10: Monitor and Maintain Regularly check the Supabase dashboard for any issues or updates. Monitor user activity and manage your database as your application grows.Supabase Auth makes it easy to implement authentication and authorization in your app. We provide client SDKs and API endpoints to help you create and manage users.Your users can use many popular Auth methods, including password, magic link, one-time password (OTP), social login, and single sign-on (SSO). Authentication and authorization are the core responsibilities of any Auth system.Authentication means checking that a user is who they say they are.Authorization means checking what resources a user is allowed to access.Supabase Auth uses JSON Web Tokens (JWTs) for authentication. For a complete reference of all JWT fields, see the JWT Fields Reference. Auth integrates with Supabase's database features, making it easy to use Row Level Security (RLS) for authorization. You can use Supabase Auth as a standalone product, but it's also built to integrate with the Supabase ecosystem.Auth uses your project's Postgres database under the hood, storing user data and other Auth information in a special schema. You can connect this data to your own tables using triggers and foreign key references.Auth also enables access control to your database's automatically generated REST API. When using Supabase SDKs, your data requests are automatically sent with the user's Auth Token. The Auth Token scopes database access on a row-by-row level when used along with RLS policies. Supabase Auth works with many popular Auth methods, including Social and Phone Auth using third-party providers. See the following sections for a list of supported third-party providers. Charges apply to Monthly Active Users (MAU), Monthly Active Third-Party Users (Third-Party MAU), and Monthly Active SSO Users (SSO MAU) and Advanced MFA Add-ons. For a detailed breakdown of how these charges are calculated, refer to the following pages:Pricing MAUPricing Third-Party MAUPricing SSO MAUAdvanced MFA - PhoneEdit this page on GitHub Enable social logins with the click of a button. Google, Facebook, GitHub, Azure (Microsoft), Gitlab, Twitter, Discord, and many more.There are two parts to every Auth system:Authentication: should this person be allowed in? If yes, who are they?Authorization: once they are in, what are they allowed to do?Supabase Auth is designed to work either as a standalone product, or deeply integrated with the other Supabase products.Postgres is at the heart of everything we do, and the Auth system follows this principle. We leverage Postgres' built-in Auth functionality wherever possible.You can authenticate your users in several ways:Email & password.Magic links (one-click logins).Social providers.Phone logins.We provide a suite of Providers and login methods.You can enable third-providers with the click of a button by navigating to Authentication > Settings > External OAuth Providers and inputting your Client ID and Secret for each.When you need granular authorization rules, nothing beats PostgreSQL's Row Level Security (RLS).Policies are PostgreSQL's rule engine. They are incredibly powerful and flexible, allowing you to write complex SQL rules which fit your unique business needs.Get started with our Row Level Security Guides.Authentication only gets you so far. When you need granular authorization rules, nothing beats PostgreSQL's Row Level Security (RLS). Supabase makes it simple to turn RLS on and off.Policies are PostgreSQL's rule engine. They are incredibly powerful and flexible, allowing you to write complex SQL rules which fit your unique business needs.With policies, your database becomes the rules engine. Instead of repetitively filtering your queries, like this ...const loggedInUserId = 'd0714948'let { data, error } = await supabase .from('users') .select('user_id, name') .eq('user_id', loggedInUserId)// console.log(data)// => { id: 'd0714948', name: 'Jane' }... you can simply define a rule on your database table, auth.uid() = user_id, and your request will return the rows which pass the rule, even when you remove the filter from your middleware:let { data, error } = await supabase.from('users').select('user_id, name')// console.log(data)// Still => { id: 'd0714948', name: 'Jane' }A user signs up. Supabase creates a new user in the auth.users table.Supabase returns a new JWT, which contains the user's UUID.Every request to your database also sends the JWT.Postgres inspects the JWT to determine the user making the request.The user's UID can be used in policies to restrict access to rows.Supabase provides a special function in Postgres, auth.uid(), which extracts the user's UID from the JWT. This is especially useful when creating policies.Supabase makes it simple to manage your users.When users sign up, Supabase assigns them a unique ID. You can reference this ID anywhere in your database. For example, you might create a profiles table referencing id in the auth.users table using a user_id field.Supabase provides the routes to sign up, log in,log out, and manage users in your apps and websites.Sign in: app.supabase.comLast updated February 26, 2024IntroductionAuthentication is a cornerstone of modern web and mobile applications, ensuring that users can securely access their accounts and data. Supabase simplifies the authentication process by providing a powerful, easy-to-implement authentication system that supports everything from email and password login to third-party providers. This guide will walk you through the steps to integrate Supabase Auth into your application, providing your users with a seamless and secure authentication experience.Step-by-Step GuideSet Up Your Supabase ProjectEnable Authentication MethodsIn the Supabase dashboard, go to the "Authentication" section and select "Settings". Here, you can enable various authentication methods, including email/password, magic links, and third-party providers like Google, GitHub, and more.Configure Third-party Providers (Optional)If you choose to enable third-party authentication, you'll need to set up OAuth credentials with each provider and input these into your Supabase project settings. Each provider has a different setup process, so refer to the Supabase documentation for detailed instructions.Install the Supabase Client LibraryInitialize Supabase in Your ApplicationImplement Sign Up and Sign InHandle Authentication StateSecure Your Application with Row Level SecurityEnable Row Level Security (RLS) on your database tables to ensure that users can only access data they are permitted to see. This is done directly in the PostgreSQL interface within the Supabase dashboard.Thoroughly test your authentication flow, including sign-up, sign-in, password recovery, and third-party logins, to ensure everything works smoothly and securely.ConclusionIntegrating authentication with Supabase provides a robust, secure, and scalable solution for managing user access in your application. By following these steps, you can quickly implement a variety of authentication methods, giving your users a seamless experience while keeping their data safe. As you continue to develop your application, explore Supabase's advanced Auth features to further enhance security and user experience.Was this article helpful? In this article, you'll learn the basic key concepts that'll help you grasp how authentication and authorization work. You'll start by learning what authentication and authorization are, and then learn how to implement authentication in your applications using Supabase auth.Table of Contents You'll need the following to make the most out of this article:Basic programming knowledge.A Supabase project to follow along.And a text editor to try out the example code snippets. In simple terms, authentication is the process of a user identifying themselves to a system and the system confirming that the user is who they claim to be. On the other hand, authorization is the process of the system determining which parts of the application the user is allowed to view or interact with, and which parts the user is not allowed to access. A flowchart depicting the user authentication processThe first time a user interacts with a system, they will be requested to register. Typically, the user will provide a piece of information and a secret that is meant to be known only by them and the system. This is the registration part of the authentication process.The next time the user interacts with the same system, they will be required to provide the identifying information along with the previously defined secret in order to verify their identity. The device the user initiates the interaction from is the client and the system is the server. Once the system verifies the user, it sends over some information to the client about the user. Because this process takes time and requires some action from the user, the client will store this information and send it back to the system whenever the user needs to access the system. This reduces friction by not requiring the user to actively re-authenticate every single time. This creates a user session. A sequence diagram showing session management in a client-server architectureThe server can pass the user's information to the client in two ways through tokens or session ids.In the case of tokens, the server generates a signed token and passes it to the client. This token is typically a JWT and may contain information regarding the user. The client will store this token and send it back to the server every time the user makes a request. The server is able to verify the integrity of the token because it signed it. This is referred to as stateless authentication because the token is self contained and the server does not need to store session data in a database or cache. In the case of cookies, the server creates a record of the user session in a database or a cache that will include a session ID. The server send this session ID to the client.The client stores this session ID in a cookie and sends it back to the server whenever the user makes a request. The session ID is a random string that acts as pointer to the actual user record in the database. When the server receives this cookie, it matches the session ID it contains to its session records, and then matches that record to the user data in the database. This is referred to as stateful authentication because a database look-up is needed. An authentication factor refers to a type of credential that can be used to verify a user's identity. There are 3 factors typically used in the authentication process, and they are:Something you know: an examples is a password.Something you have: an example is a token sent to your phone.Something you are: an example is your fingerprint. Authentication strategies refer to the processes used to verify a user. Different types of authentication strategies include: This refers to the traditional way of users identifying themselves by providing a text-based secret that is user defined. Typically, the system handles the entire process on its servers and is responsible for security and reliability. In this approach, the system verifies the users identity without requiring user defined passwords. The system will, instead, generate a one time password (OTP) and send to the user. This OTP is then used in place of a password to gain access to the system. Examples include magic links, where the system sends a code to the users email. The system attempts to verify the user is who they claim to be by requiring an extra piece of information after the primary authentication has checked out. This can be an OTP sent to the user via email or SMS, or it can be by requiring the users' biometric information before the system grants access. This is similar to 2FA, except that the system will use more than one extra method to verify the users identity. The extra methods or factors used in both MFA and 2FA are usually external to the system, such as an SMS requiring a phone. OAuth is an authorization framework that allows clients to access information from an external server on the user's behalf. The external server prompts the user for permission to share the requested resources with the client. After user permits the action, the external server issues an access token to the client. The client then gives this access token to the original server, which verifies the token's validity and manages access to the requested resources. OAuth 2.0 is the latest version of OAuth and is the more widely used framework. OAuth 2.0 extends support for non-browser based systems. Social Authentication is based on OAuth 2.0 but in this case, the external server that the client redirects the user to is typically a social media platform. This is the type of authentication process carried out whenever you see a "Continue with Twitter/X" button on an authentication page. SAML stands for Security Assertion Markup Language. It is a standard for passing authentication and authorization information between systems. One system acts as the requesting system or the service provider (SP) and the other system holds the requested information or acts as the Identity Provider (IdP). On receiving this request, the identity provider will generate some statements in SAML form that contains some user information. The service provider then uses this information to decide how to handle the user in relation with its protected resources. SSO refers to Single Sign On. This is an authentication strategy that lets users sign in through one system/application that then lets them access multiple applications within the same network. This improves the user experience by not requiring the user to log in to different related applications. An example of this is Google workspace. You don't need to log into Docs separately if you are already logged into your Gmail account. SSO is facilitated by SAML as SAML provides a standard authentication mechanism and allows different systems to trust each other. Authentication involves handling, moving, and storing sensitive user information in relation to protected server resources. This makes security and best practices an important aspect of an authentication system.There are some basic steps you can take to greatly increase the security of your authentication systems. These include:Enforcing stronger passwords.Requiring the user to register an extra factor to enable 2FA.Encrypting sensitive data as it is being transferred via HTTPS.Storing passwords in an encrypted manner.Using standard authentication frameworks like OAuth 2.0.There are certain compliances that your system should consider when handling sensitive user data beyond specific authentication information. This is even more so if operating in certain countries or handling enterprise applications. These compliances include:GDPR: This compliance enforces standards around data handling including confidentiality and integrity.HIPAA: This compliance applies to medical data. It expects high levels of integrity.SOC: This is a framework more generally required of cloud technologies. It is based on the American Institute of CPAs and covers aspects of privacy, security, availability, integrity and confidentiality.Keeping all this in mind, you will find that it is often easier to use dedicated authentication services for your applications instead of rolling out your own auth.There are lots of options for this, including dedicated authentication services such as Clerk and Auth0, and Backend-as-a-Service such as Supabase and Firebase. In this case, let's take a look at the Supabase authentication offering. Supabase is an open source Backend as a Service (BaaS) platform that makes developing a backend for your applications very easy and fast. It is based on open source technologies and actively supports the open source ecosystem. Supabase offers common services that most backend applications will require. These services are:Database: This is a fully featured Postgres database.Authentication: This is an enterprise ready authentication service that is based on a fork of the goTrue server.Realtime: This is an API that adds the ability to use real-time capabilities in your applications.Storage: This is a storage service which is an s3 wrapper.Edge Functions: These are serverless functions that run on the edge. Powered by the Deno runtime.Vector: This is a vector database that makes it easier to work with embeddings in your AI applications.Supabase is SOC2, HIPAA and GDPR compliant, self-host-able and open source. Furthermore, their authentication service exposes many strategies, giving you full control over your data and can be used independently of their other offerings. This and their auto documenting API makes it a very good choice for your applications. The first step is to set up your Supabase project's auth settings. You can enable the exact authentication methods you want to use via the settings. There are three ways you can start using Supabase auth in your project: You can directly use the authentication service in your applications by calling the auth endpoint and passing the user information to it. You can also get, update and delete your users. The API is automatically available when you create a project via the Supabase console and can be called like so: const res = await fetch("https:///auth/v1/signup", { method: "POST", headers: { authorization: "Bearer YOUR_SUPABASE_KEY", "content-type": "application/json", }, body: JSON.stringify({ email: "user-email", password: "user-password", }),}); Supabase offers a few SDKs (software development kits) in different programming languages meant to make interacting with your Supabase project straightforward. Languages officially supported include Dart and JavaScript, with Python and others having strong community support. The procedure for getting started involves adding the SDK as a dependency, then connecting your application to your Supabase project. In the case of the JavaScript SDK, this would look something like this:npm install @supabase/supabase-js Supabaseimport { createClient } from '@supabase/supabase-js' const supabaseUrl = 'https://.supabase.co'const supabaseKey = process.env.SUPABASE_ANON_KEYconst supabase = createClient(supabaseUrl, supabaseKey)Then you can access the authentication methods under the auth namespace like so:const { data, error } = await supabase.auth.signUp({ email: 'user email', password: 'user password',}) Supabase provides helper libraries to make authentication using their service even easier. These libraries provide customizable login screens that support magic links, password based and social login strategies. Currently, the libraries are available for JavaScript and Flutter. Supabase also provides a separate SSR (Server Side Rendering) package for applications that use server side frameworks and require a Supabase access token to be available to them.To start using React Auth UI, as an example, first you need to install the dependencies as shown below:npm install @supabase/supabase-js @supabase/auth-ui-react @supabase/auth-ui-sharedThen you can start using the library after initializing Supabase as in the SDK example above. Here is some sample code that shows how to use the auth UI library in a React application:import { useEffect } from "react";import { useNavigate } from "react-router-dom";import { Auth } from "@supabase/auth-ui-react";import { ThemeSupa } from "@supabase/auth-ui-shared";import { supa } from "../constants"; const AuthUi = () => { const navigate = useNavigate(); useEffect(() => { const { data: { subscription }, } = supa.auth.onAuthStateChange((event) => { if (event === "SIGNED_IN") { navigate("/authenticated"); } }); return () => subscription.unsubscribe(); }, [navigate]); return (