Documentation

What is Atlas?

What is Atlas?

Atlas is a complete, self-hostable authentication and user-management platform — the kind of drop-in auth you'd reach for Clerk or Auth0 to provide, but one you can also run on your own infrastructure.

You point your app at Atlas and get sign-up and sign-in flows, sessions, multi-factor auth and passkeys, organizations and B2B multi-tenancy, roles and permissions, enterprise SSO (SAML), SCIM provisioning, and webhooks — reachable from a hosted UI, a large SDK lineup, and a REST API.

What you get

- Authentication — password, email codes, magic links, passkeys/WebAuthn, and 100+ social providers (Google, GitHub, Microsoft, Apple, and more).
- Sessions — issued as standard JWTs, verifiable statelessly against your instance's JWKS.
- Organizations & B2B — multi-tenant organizations with memberships, invitations, and per-org roles.
- Authorization — organization roles & permissions (RBAC) plus fine-grained, relationship-based access (FGA, OpenFGA-compatible).
- Enterprise — SSO via SAML, and SCIM directory provisioning.
- MFA — TOTP authenticator apps, SMS, backup codes, and passkeys.
- Webhooks — subscribe to user, session, and organization events.

How it fits together

Atlas exposes two API surfaces, each with its own key:

- The Frontend API (FAPI) runs sign-in/sign-up flows from the browser or a mobile app. It is called with a publishable key (pk_...), which is safe to ship in client code.
- The Backend API (BAPI) is the full management surface — users, organizations, roles, SSO, SCIM, webhooks, billing. It is called from your server with a secret key (sk_...), which must never leave your backend.

Your frontend runs the flow with the publishable key; your backend either verifies the resulting session JWT locally (no round trip) or calls the Backend API with the secret key.

SDKs

Atlas ships official SDKs across the stack:

- Frontend — @atlas/js, @atlas/react, @atlas/nextjs, @atlas/vue, @atlas/nuxt, @atlas/angular, @atlas/svelte, @atlas/react-native.
- Backend — @atlas/backend (Node), plus Python, Go, Ruby, PHP, Java, and .NET.
- Native mobile — Swift (iOS/macOS), Kotlin (Android), and Flutter.

See the SDK overview for a snippet of each.

Next steps

- Create an application and get your keys.
- Protect a route with the React or Next.js SDK.

- Authorize your users with roles, permissions, and FGA.
- Self-host Atlas on your own infrastructure.

Need a hand? Email support@atlasauth.net.

Protect a route with React & Next.js

Protect a route with React & Next.js

This quickstart adds Atlas to a React or Next.js app: a provider, a sign-in UI, and a protected route. It takes a few minutes and uses your publishable key.

React

1. Install

```
npm install @atlas/react
```

2. Wrap your app in the provider

<AtlasProvider> boots the session, completes any OAuth/hosted-page redirect, and schedules token refresh. Give it your publishable key and your instance's Frontend API origin:

```
import { AtlasProvider } from '@atlas/react';

export function Root() {
  return (
    <AtlasProvider
      publishableKey="pk_live_your_key"
      frontendApi="https://accounts.yourapp.com"
    >
      <App />
    </AtlasProvider>
  );
}
```

3. Render sign-in and user state

<SignedIn> / <SignedOut> render by auth state; <SignIn> is the prebuilt flow; <UserButton> is the account menu:

```
import { SignedIn, SignedOut, SignIn, UserButton } from '@atlas/react';

function Header() {
  return (
    <>
      <SignedOut>
        <SignIn />
      </SignedOut>
      <SignedIn>
        <UserButton />
      </SignedIn>
    </>
  );
}
```

4. Read the user from hooks

```
import { useUser, useAuth } from '@atlas/react';

function Profile() {
  const { isLoaded, isSignedIn, user } = useUser();
  const { getToken, signOut } = useAuth();

  if (!isLoaded) return null;          // still booting
  if (!isSignedIn) return <p>Please sign in.</p>;
  return <p>Hello, {user.firstName}</p>;
}
```

Next.js (App Router)

1. Install

```
npm install @atlas/nextjs
```

2. Protect routes with middleware

atlasMiddleware verifies the session and redirects signed-out users away from anything not listed as public. Unlisted routes are protected by default, so a new page is safe from the moment you add it:

```
// middleware.ts
import { atlasMiddleware } from '@atlas/nextjs';

export default atlasMiddleware({
  jwksUrl: 'https://accounts.yourapp.com/.well-known/jwks.json',
  issuer: 'https://accounts.yourapp.com',
  publicRoutes: ['/', '/sign-in(.*)', '/sign-up(.*)'],
  signInUrl: '/sign-in',
});

export const config = { matcher: ['/((?!_next|.*\\..*).*)'] };
```

3. Read auth in a server component or route handler

Create an auth() helper once, then call it wherever you need the session. It exposes has() and protect() for authorization — the same condition set as React's <Protect>:

```
// atlas.ts
import { createAuthHelper } from '@atlas/nextjs';

export const auth = createAuthHelper({
  jwksUrl: 'https://accounts.yourapp.com/.well-known/jwks.json',
  issuer: 'https://accounts.yourapp.com',
});
```

```
// app/api/billing/route.ts
import { auth } from '@/atlas';

export async function POST(request: Request) {
  const a = await auth(request);
  a.protect({ permission: 'org:billing:manage' }); // throws !’ 403 if not held
  // …do the privileged work
}
```

Next

- Verify sessions on your backend for non-JS servers.
- Authorization — roles, permissions, and fine-grained access.

SDKs

SDKs

Atlas ships official SDKs across the whole stack. Frontend SDKs use your publishable key (pk_...) and talk to the Frontend API; backend SDKs use your secret key (sk_...) and talk to the Backend API, plus verify session JWTs locally.

The lineup

Tier	SDKs
Frontend (web)	@atlas/js, @atlas/react, @atlas/nextjs, @atlas/vue, @atlas/nuxt, @atlas/angular, @atlas/svelte, @atlas/react-native
Backend	@atlas/backend (Node), Python, Go, Ruby, PHP, Java, .NET
Native mobile	Swift (iOS/macOS), Kotlin (Android), Flutter

Every frontend SDK wraps the framework-agnostic @atlas/js client, so they all drive the same endpoints and can't diverge in what the server sees. Every backend SDK mirrors @atlas/backend namespace-for-namespace over the same 45 Backend API resource areas.

Frontend

@atlas/js — framework-agnostic

```
import { FapiClient, advance, initialFlow } from '@atlas/js';

const client = new FapiClient({ publishableKey: 'pk_live_…' });
let flow = initialFlow;
flow = await advance(client, flow, { identifier: 'ada@example.com' });
flow = await advance(client, flow, { strategy: 'password', password });
```

@atlas/react

```
import { AtlasProvider, SignedIn, SignIn, UserButton, useUser } from '@atlas/
react';

<AtlasProvider publishableKey="pk_live_…" frontendApi="https://
accounts.yourapp.com">
  <SignedIn><UserButton /></SignedIn>
</AtlasProvider>
```

@atlas/nextjs

```
import { atlasMiddleware } from '@atlas/nextjs';

export default atlasMiddleware({
  jwksUrl: 'https://accounts.yourapp.com/.well-known/jwks.json',
  issuer: 'https://accounts.yourapp.com',
  publicRoutes: ['/', '/sign-in(.*)'],
});
```

@atlas/vue

```
import { createApp } from 'vue';
import { createAtlas } from '@atlas/vue';

createApp(App).use(createAtlas({ publishableKey: 'pk_live_…' })).mount('#app');
// then in components: const { isSignedIn, user } = useUser();
```

@atlas/nuxt

```
// nuxt.config.ts
export default defineNuxtConfig({
  modules: ['@atlas/nuxt'],
  atlas: { publishableKey: 'pk_live_…', frontendApi: 'https://
accounts.yourapp.com' },
});
```

@atlas/angular

```
import { provideAtlas } from '@atlas/angular';

bootstrapApplication(AppComponent, {
  providers: [provideAtlas({ publishableKey: 'pk_test_…' })],
});
```

@atlas/svelte

```
<script lang="ts">
  import { AtlasProvider } from '@atlas/svelte';
</script>

<AtlasProvider publishableKey="pk_live_…" frontendApi="https://
accounts.yourapp.com">
  <slot />
</AtlasProvider>
```

@atlas/react-native

```
import { AtlasProvider } from '@atlas/react-native';
// React Native has no cookie jar — pass a TokenStorage adapter (e.g. expo-
secure-store).
<AtlasProvider publishableKey="pk_live_…" storage={secureStorage}>{children}</
AtlasProvider>
```

Backend

@atlas/backend (Node)

```
import { AtlasBackend, createAtlasClient } from '@atlas/backend';

const atlas = createAtlasClient({ secretKey: 'sk_live_…' });
const user = await atlas.users.create({ email_address: 'ada@example.com' });
```

Python — atlas-backend

```
from atlas_backend import AtlasClient
atlas = AtlasClient("sk_live_...")
user = atlas.users.create({"email_address": "ada@example.com"})
```

Go — atlas-go

```
import atlas "github.com/atlas-auth/atlas-go"
// client over the sk_ Backend API; std-lib only, context-first methods.
```

Ruby — atlas-auth

```
require "atlas"
atlas = Atlas::Client.new(ENV.fetch("ATLAS_SECRET_KEY"))
user = atlas.users.create(email_address: "ada@example.com")
```

PHP — atlas-auth/atlas-php

```
use Atlas\Client;
$atlas = new Client('sk_live_…');
$user = $atlas->users->get('user_123');
```

Java — net.atlasauth:atlas-java

```
AtlasClient atlas = AtlasClient.builder().secretKey("sk_live_…").build();
// plus SessionVerifier for local JWT verification
```

.NET — Atlas.Sdk

```
using var atlas = new AtlasClient("sk_live_...");
User user = await atlas.Users.GetAsync("user_123");
```

Native mobile

These are client SDKs — they use a publishable key and talk to the Frontend API, mirroring @atlas/js shape-for-shape.

Swift

```
import Atlas
let atlas = AtlasClient(publishableKey: "pk_live_…", frontendApi:
"accounts.your-domain.com")
let user = try await atlas.signIn(email: "ada@example.com", password: "…")
```

Kotlin / Android

```
// net.atlasauth:atlas-android — OkHttp + coroutines, mirrors the Swift SDK.
val atlas = AtlasClient(publishableKey = "pk_live_…", frontendApi =
"accounts.your-domain.com")
```

Flutter / Dart — atlas_auth

```
final client = AtlasClient(publishableKey: 'pk_live_…', frontendApi:
'accounts.example.com');
final user = await client.signIn(email: 'ada@example.com', password: '…');
```

Self-hosting Atlas

Self-hosting Atlas

Atlas is self-hostable — you can run the entire platform on your own infrastructure and keep every user, key, and session inside your own boundary. This page is a high-level map of a deployment; your dashboard and repository carry the exact, version-specific steps.

What runs

An Atlas deployment is three services plus two datastores:

Component	What it does
api	The Frontend API (FAPI), Backend API (BAPI), and dashboard API.
web	The dashboard single-page app (served as static files).
worker	Background jobs — webhook delivery, sweeps, and other async work.
Postgres	The system of record: users, sessions, organizations, keys, configuration.
Redis	Caching and ephemeral coordination. Treated as losable — sign-in keeps working from Postgres if Redis is lost.

One origin

The dashboard SPA and the API must be served same-site over HTTPS. The refresh-token cookie is Secure, SameSite=Lax, and path-scoped to the API, so a split-origin setup won't hold a session. Put both behind one host and route by path (for example, /v1 and /.well-known to the API, everything else to the web app).

Configuration essentials

Configuration is environment-driven. The values you must set:

- DATABASE_URL — your Postgres connection string (Atlas uses a dedicated database).
- REDIS_URL — your Redis connection string.
- DATA_ENCRYPTION_KEY — a 32+ byte base64 root key for envelope encryption of secrets at rest. Generate it once and keep it stable: rotating it makes existing encrypted provider secrets undecryptable. Generate with openssl rand -base64 48.
- ATLAS_ORIGIN / host — the public HTTPS origin the deployment is served from.

Email is per-instance, not env

Outbound email is configured per instance as an encrypted bring-your-own-key provider row (dashboard !’ Messaging), not as a global environment secret — so each instance sends from its own verified domain. A simple env-managed mailer (Resend / SES / Brevo / Mailgun) exists as an optional path for the most basic self-host, but the per-instance provider is the norm.

Bring it up

- Create the database on your Postgres.
- Deploy the api, web, and worker images behind your single HTTPS origin.

- Migrate the database using the shipped migration entrypoint.
- Bootstrap the platform instance and an admin user (the bootstrap script prints a generated password when you don't supply one).
- Sign in to the dashboard at your origin with the admin credentials, then create your first application.

Operations

Atlas is built to be operated: key signing supports ordinary rotation (old public key kept published briefly so in-flight tokens still verify) and an emergency rotation that drops the old key with zero grace. Sessions can be revoked instance-wide by bumping a sessions version. Kill switches — disabling a flaky OAuth provider, for instance — are config flags that take effect on the next request, not redeploys. Audit logs are append-only.

Running Atlas in production? Email support@atlasauth.net.

Authorization

Authorization

Authentication proves who someone is. Authorization decides what they may do. Atlas gives you two tools for it, and most apps use both:

- Roles & permissions (RBAC) — coarse, per-organization. "Admins can manage billing; members can't." A handful of roles that apply across an org.
- Fine-grained access (FGA) — per-object, relationship-based (Google-Zanzibar / OpenFGA style). "Alice can edit document 123; Bob can only view it." Scales to millions of per-object grants and inheritance.

One rule governs both: the server is the boundary. Client helpers decide what a user sees; only a check on the request decides what they can do. Every example below that gates UI has a matching server check.

1. Permission keys and role keys

Keys are the contract — they travel in the session token and get hardcoded into your authorization checks, so their shape is fixed and they are immutable once created. The two shapes differ on purpose:

- A role key is org:<name> — e.g. org:admin, org:member. It names who someone is.
- A permission key is org:<resource>:<action> — e.g. org:billing:manage. It names what they may do.

So <Protect role="org:admin"> and <Protect permission="org:billing:manage"> both work — the arity of the key tells you which you're checking. The sys_ prefix is reserved for Atlas's built-in permissions (e.g. org:sys_memberships:manage).

2. Roles & permissions in the dashboard

Under Authorization !’ Roles & permissions, every instance is seeded with two system roles — org:admin (holds every permission) and org:member (read-only) — which you can't delete but can relabel. Add your own roles and permissions; the "Add permission" field offers a menu of conventional keys so you pick a well-shaped one instead of guessing. Group-to-role grants let your IdP drive role assignment through SCIM.

3. What's in the session token

When a user has an active organization, their role and the union of its permissions (direct and any granted through a group) are minted into the session JWT:

```
{
  "sub": "user_2a…",
  "org_id": "org_9f…",
  "org_role": "org:admin",
  "org_permissions": ["org:sys_memberships:manage", "org:billing:manage"]
}
```

Your code reads authorization straight off the token — no call back to Atlas on the hot path. A permission change takes effect within one token lifetime (the same bound as session revocation).

4. Check on the server (the real boundary)

Verify the session, then gate the action. The verified result carries has() and protect() bound to the claims — has() returns a boolean, protect() throws ForbiddenError (map it to 401/403):

```
import { AtlasBackend } from '@atlas/backend';

const atlas = new AtlasBackend({
  jwksUrl: 'https://accounts.yourapp.com/.well-known/jwks.json',
  issuer: 'https://accounts.yourapp.com',
```

```
});

const auth = await atlas.authenticateRequest(req);
if (!auth.ok) return res.status(401).end();

// boolean form — full condition set
if (auth.has({ permission: 'org:billing:manage' })) { /* … */ }
if (auth.has({ anyPermission: ['org:billing:manage', 'org:billing:read'] })) { /
* … */ }

// assertion form — throws ForbiddenError when unmet
auth.protect({ role: 'org:admin' });
```

In Next.js, auth() exposes the same has() / protect() in server components and route handlers:

```
import { auth } from '@/atlas'; // your createAuthHelper() instance

export async function POST(request: Request) {
  const a = await auth(request);
  a.protect({ permission: 'org:billing:manage' }); // 403 if not held
  // …do the privileged work
}
```

5. Gate UI on the client

In React, render conditionally with <Protect> or useAuth().has(). This is a rendering aid, not a security boundary — always keep the matching server check from step 4.

```
import { Protect, useAuth } from '@atlas/react';

<Protect permission="org:billing:manage" fallback={<Locked />}>
  <BillingSettings />
</Protect>

// or imperatively
const { has } = useAuth();
if (has({ anyPermission: ['org:billing:manage', 'org:billing:read'] })) { /* …
*/ }
```

@atlas/react-native exposes the same useAuth().has(condition), and @atlas/js ships a framework-agnostic has(claims, condition) for anything else. All of them evaluate the identical condition — a check written for one surface behaves the same everywhere.

6. Manage roles from your backend

Everything on the Roles screen is also a REST API, so provisioning can be automated:

```
import { createAtlasClient } from '@atlas/backend';
const atlas = createAtlasClient({ secretKey: 'sk_live_…' });

await atlas.roles.create({
  key: 'org:auditor',
  name: 'Auditor',
  permissions: ['org:logs:read', 'org:billing:read'],
});

// Retire a role that people still hold — move its members first, atomically:
await atlas.roles.delete('role_123', { reassignTo: 'role_member' });

// Custom permissions
await atlas.permissions.create({ key: 'org:reports:export', name: 'Export
reports' });
await atlas.permissions.delete('perm_456'); // 409s if a role still grants it
```

7. Fine-grained access (FGA)

When "who can touch this specific object" outgrows roles, reach for FGA. You define a model (types and relations), write tuples (facts like "alice is an editor of doc:1"), and check a relation before allowing an action. It supports inheritance ("editors of a folder can edit its documents"), groups, and public wildcards.

```
const atlas = createAtlasClient({ secretKey: 'sk_live_…' });

// One-time: create a store + model (types & relations), then bind the store:
const { id } = await atlas.fga.stores.create({ name: 'app' });
const fga = atlas.fga.store(id);

// Write relationship tuples as facts change:
await fga.write({ writes: { tuple_keys: [
  { user: 'user:alice', relation: 'editor', object: 'doc:1' },
  { user: 'group:eng#member', relation: 'viewer', object: 'doc:1' },
] } });

// Check on the request path:
const { allowed } = await fga.check({ user: 'user:alice', relation: 'editor',
object: 'doc:1' });

// Answer many at once (e.g. filtering a list) in one round trip:
const batch = await fga.batchCheck({ checks: [
  { user: 'user:alice', relation: 'viewer', object: 'doc:1', correlation_id:
'1' },
  { user: 'user:alice', relation: 'viewer', object: 'doc:2', correlation_id:
'2' },
] });

// Or ask which objects a user can reach:
const { objects } = await fga.listObjects({ user: 'user:alice', relation:
'viewer', type: 'doc' });
```

The model is OpenFGA-compatible, so a model authored in the OpenFGA playground ports straight in. Writes are validated against the model's allowed user types, so a typo'd tuple that would silently never match is rejected up front.

8. Which one do I use?

- A small, fixed set of roles that apply across an organization !’ Roles & permissions.
- Permissions that vary per object, per relationship, or need inheritance !’ FGA.
- Both together is normal: RBAC for "is this person an org admin," FGA for "can this person open record #98472."

Create an application

Create an application

An application (also called an instance) is a single Atlas environment: its own users, connections, keys, and JWKS. You typically run one for development and one for production.

1. Create it

Sign in to the Atlas dashboard and create an application. Give it a name; Atlas provisions the instance, a signing key, and a hosted Frontend API for it.

2. Find your keys

Every instance gives you two keys, on the API keys screen:

- Publishable key — pk_test_… (development) or pk_live_… (production). It identifies the instance and is safe to ship in your frontend.
- Secret key — sk_test_… / sk_live_…. It authenticates your backend to the Backend API. Never expose it in client code.

The test / live prefix tells you which environment a key belongs to, so a development key can never touch production data by accident.

3. Configure sign-in methods

Under User & authentication, turn on the methods you want — password, email verification codes, magic links, and passkeys. Under Connections, enable social providers (Google, GitHub, Microsoft, Apple, and 100+ others); paste each provider's OAuth client id and secret, and register this redirect URL with them:

`https://atlasauth.net/v1/oauth/callback/<provider>`

4. Note your Frontend API

Your instance has a Frontend API origin (its FAPI host) that the client SDKs talk to. You'll pass it — along with the publishable key — when you initialize a frontend SDK. In the dashboard it's shown next to your keys.

Next

- Your API keys — what each key does and where it's used.
- Protect a route — wire up the React or Next.js SDK.

Verify sessions on your backend

Verify sessions on your backend

Atlas sessions are standard JWTs. Your backend proves a request is authentic by verifying that token — locally, against your instance's public JWKS, with no call back to Atlas on the hot path. This is the same design in every backend SDK.

The idea

- The frontend sends the session JWT (from the __session cookie, or an Authorization: Bearer header).
- Your backend fetches and caches your instance's JWKS from https://<your-frontend-api>/.well-known/jwks.json.
- It verifies the token's signature, issuer, and expiry, then reads the claims — sub (user id), org_id, org_role, org_permissions.

Because verification is local, it costs nothing per request and keeps working even if Atlas is briefly unreachable. A revoked session or a permission change takes effect within one token lifetime.

Node (@atlas/backend)

```
import { AtlasBackend } from '@atlas/backend';

const atlas = new AtlasBackend({
  jwksUrl: 'https://accounts.yourapp.com/.well-known/jwks.json',
  issuer: 'https://accounts.yourapp.com',
});

const auth = await atlas.authenticateRequest(req);
if (!auth.ok) return res.status(401).end();

// auth.claims.sub, auth.claims.org_id, …
if (auth.has({ permission: 'org:billing:manage' })) { /* … */ }
```

Other languages

Every backend SDK ships the same local session verifier and a typed client over the secret-key Backend API. Point the verifier at your JWKS URL and issuer:

- Python — atlas-backend
- Go — atlas-go
- Ruby — atlas-auth
- PHP — atlas-auth/atlas-php (SessionVerifier)
- Java — net.atlasauth:atlas-java (SessionVerifier)
- .NET — Atlas.Sdk (AtlasBackend)

See the SDK overview for install and a snippet of each.

Calling the Backend API directly

You don't need an SDK — the Backend API is plain REST under /v1, authenticated with your secret key as a bearer token:

```
curl https://atlasauth.net/v1/users \
  -H "Authorization: Bearer sk_live_your_key"
```

It covers users, organizations, sessions, roles & permissions, webhooks, enterprise SSO/SAML, SCIM provisioning, and billing — each endpoint scoped to a capability so a key only does what you grant it.

Your API keys

Your API keys

Atlas has two API surfaces, and each has its own key. Getting these right is the whole security model, so it's worth one page.

Publishable key — pk_...

The publishable key identifies your instance to the Frontend API (FAPI). It is designed to be public: it ships in your browser bundle or mobile app, and it's what every frontend SDK (@atlas/js, @atlas/react, @atlas/nextjs, Swift, Kotlin, Flutter, …) is initialized with.

```
import { FapiClient } from '@atlas/js';

const client = new FapiClient({ publishableKey: 'pk_live_your_key' });
```

A publishable key can only do what an anonymous or signed-in end user is allowed to do — start a sign-in, submit a factor, read the current user. It cannot manage other users or read your instance's data.

Secret key — sk_...

The secret key authenticates your backend to the Backend API (BAPI). It is the full management surface — users, organizations, roles, sessions, SSO, SCIM, webhooks, billing — so it must never appear in client code, a repo, or a browser network tab. Keep it in a server-side environment variable.

```
curl https://atlasauth.net/v1/users \
  -H "Authorization: Bearer sk_live_your_key"
```

Every Backend API request is a plain bearer token — no signing, no handshake.

test vs live

Keys are prefixed by environment: pk_test_ / sk_test_ belong to a development instance, pk_live_ / sk_live_ to production. They address different instances with different data, so you cannot accidentally read or mutate production from a development key.

Sessions are JWTs

When a user signs in, Atlas issues a session as a standard JWT. Your backend can verify it locally — with no call back to Atlas — against your instance's public JWKS at:

```
https://<your-frontend-api>/.well-known/jwks.json
```

That's how every backend SDK's session verifier works: fetch and cache the JWKS, verify the token's signature, issuer, and expiry, and read the claims. See Verify sessions on your backend.

If a key leaks

Roll it from the dashboard's API keys screen. Rolling a secret key immediately invalidates the old one. For a suspected signing-key compromise, Atlas supports emergency key rotation that drops the old public key with no grace period.

Your first sign-in

Your first sign-in

The fastest way to see Atlas working is to render a real sign-in box. There are two ways: a drop-in embeddable widget (no build step), or the JavaScript SDK for full control.

Option A — the embeddable widget

The widget is a custom element that renders a complete, themeable sign-in flow — password, email codes, passkeys, and every social provider you've enabled. Add the script and one tag:

```
<script src="https://atlasauth.net/embed.js"></script>

<atlas-sign-in
  data-atlas-key="pk_live_your_key"
  data-atlas-fapi="https://accounts.yourapp.com">
</atlas-sign-in>
```

It handles the whole flow and sets a session on your domain. Use <atlas-sign-up> for a registration screen.

Option B — drive it with the SDK

For full control, the JavaScript SDK talks to the same endpoints the widget uses. It exposes a small client plus a flow helper that advances an attempt one step at a time — identifier, then factor, then complete — whatever the strategy (password, email code, passkey, OAuth):

```
npm install @atlas/js
```

```
import { FapiClient, advance, initialFlow } from '@atlas/js';

const client = new FapiClient({ publishableKey: 'pk_live_your_key' });

let flow = initialFlow;
flow = await advance(client, flow, { identifier: 'ada@example.com' });
flow = await advance(client, flow, { strategy: 'password', password });
// when flow.attempt.status === 'complete', exchange its ticket for a session
```

The session lives in an HttpOnly cookie the browser sends automatically; the short-lived JWT is held only in memory for the life of the tab. Nothing is written to localStorage.

Using React or Next.js?

Skip the manual flow — @atlas/react and @atlas/nextjs wrap all of this with prebuilt components (<SignIn />, <UserButton />) and hooks (useUser, useAuth). See Protect a route with React & Next.js.