BERNSTEIN

TECHNICAL BRIEF

INNOVATION EVIDENCE PLATFORM

Single Sign-On with OpenID Connect

Let your team sign in to your branded Bernstein Space with your own corporate identity provider - Okta, Microsoft Entra, Google, Auth0, AWS Cognito, OneLogin, Keycloak - instead of a separate Bernstein password.

This brief explains OpenID Connect SSO end to end: what it is and why it exists, the concepts and standards from zero, exactly what happens when a member signs in, a step-by-step setup guide for both sides, the providers we support, how to integrate any other compliant provider by hand, the security model, and a troubleshooting playbook your IT team can act on.

UPDATED	AUDIENCE	PROTOCOL
June 2026	IT, security & admins	OpenID Connect · OAuth 2.0

1 · PURPOSE

1 What SSO is for, and why it exists

A **Space** is your white-label tenant on Bernstein: a branded sign-in at `app.bernstein.io/<your-space>` with your logo, your colours and - optionally - **your own login system**. SSO is what connects that login to the corporate identity provider your team already uses.

Without SSO, each member signs in with a Bernstein-managed email and password. With SSO, the member signs in **at your company's identity provider**, and Bernstein trusts that provider's verdict. One account, governed by your IT policies.

What you get

No second password

Members use the corporate account they already have. Nothing new to remember, reset or rotate on the Bernstein side.

Central control

Deactivate an employee in your IdP and Bernstein access is cut immediately - no separate offboarding step to forget.

Your policy, enforced upstream

MFA, password rules and conditional access are applied by your IdP at sign-in time - not re-implemented by us.

Automatic provisioning

A first-time SSO user is created and bound to the Space automatically as a managed member, so admins don't pre-create accounts.

ONE SETTING, PER SPACE

SSO is configured **per Space**. A Space can have **either OpenID Connect or SAML** active - never both at once. This brief covers **OpenID Connect**; SAML follows the same shape with different configuration fields. Bernstein's own non-Space users always use email and password.

2 · CONCEPTS FROM ZERO

2 The concepts and the standards

If you've never configured OIDC, read this once - everything later builds on it. The vocabulary is small and the moving parts are few.

The cast

	Plain meaning	In this setup
IdP / OP	The **Identity Provider** (a.k.a. OpenID Provider) - the system that holds the accounts and checks passwords and MFA.	Your Okta / Entra / Google / Cognito tenant
RP / client	The **Relying Party** - the app that relies on the IdP to authenticate users.	**Bernstein** is the RP
Client ID / Secret	The RP's own credentials *at the IdP* - they identify Bernstein to your provider.	Stored on your Space's OIDC config
End user	The human signing in.	Your Space member

What OpenID Connect actually is

OpenID Connect (OIDC) is a thin **identity layer on top of OAuth 2.0**. OAuth 2.0 answers *"can this app access X?"* (authorization); OIDC adds *"who is this user?"* (authentication) by issuing a signed **ID token**. These are open, widely-implemented standards - not anything Bernstein-specific.

	What it provides
OAuth 2.0 (RFC 6749)	The Authorization Code flow: redirect → one-time code → back-channel token exchange.
OpenID Connect Core 1.0	The `id_token`, the standard claims (`email`, `sub`, `name`) and the `openid` scope.
OIDC Discovery 1.0	The `/.well-known/openid-configuration` document that publishes a provider's endpoints and keys.
JWT (RFC 7519)	The format of the `id_token` - a signed JSON Web Token.
JWK / JWKS (RFC 7517)	The provider's public signing keys, published at a `jwks_uri`, used to verify the token signature.

THE FLOW IN ONE BREATH

Bernstein sends the user to your IdP. The user logs in there. The IdP sends the user back with a one-time **authorization code**. Bernstein swaps that code, server-to-server, for an **ID token**, cryptographically verifies the token's signature against your IdP's published keys, reads the user's email, and signs them in. This is the **Authorization Code flow** - the most secure browser flow, because tokens are exchanged on the back channel and never exposed in a URL.

2 · CONCEPTS FROM ZERO

The four facts Bernstein needs about your IdP

To talk to *any* OIDC provider, Bernstein needs exactly four values, plus two optional ones it can usually discover on its own.

	What it is	Example
Client ID	Bernstein's identifier at your IdP	`19k3f8uvhvm7lu7o4negrpdpnk`
Client Secret	Bernstein's secret at your IdP	••••••••
Authentication URL	The IdP's *authorization* endpoint - where we send the user	`…/oauth2/authorize`
Token URL	The IdP's *token* endpoint - where we swap the code	`…/oauth2/token`
Issuer (optional)	The IdP's canonical identity URL - the trust anchor used to verify and discover keys	`https://…/<tenant>`
JWKS URL (optional)	Where the IdP publishes its public signing keys	`…/.well-known/jwks.json`

Issuer and JWKS URL are auto-detected when left blank (see §6). Set them by hand only when a provider publishes discovery on a different host than its token endpoint - most notably AWS Cognito.

3 · THE SIGN-IN FLOW

3 What actually happens when a member signs in

From the member clicking "Log in" to landing in their vault, the flow runs a fixed, well-understood sequence. The security-critical step is #4 - where Bernstein verifies the token's signature before trusting anyone.

1 · Space loads

The member visits app.bernstein.io/<your-space>. Bernstein sees the Space uses OpenID and shows the SSO sign-in button.

2 · Redirect to your IdP

Clicking "Log in" sends the member to your IdP's authorization endpoint, carrying our client_id, the redirect URI, the scopes and a single-use state token.

3 · Authenticate

The member logs in at your IdP's own page - password, MFA, conditional access - all governed by your policies, not ours.

4 · Code → token, verified

The IdP redirects back with a one-time code. Bernstein swaps it server-to-server for an ID token, then verifies its signature, issuer and audience.

5 · Find or create the user

Bernstein reads the email claim and binds the member to the Space - creating a managed account on first sign-in, or matching an existing member.

6 · Signed in

The browser is handed a short-lived, single-use code (never the access token), exchanges it, and the member lands in the vault lobby.

TWO ANTI-ABUSE GUARDS BAKED INTO THE FLOW

A single-use **state** token ties each login attempt to its callback and is destroyed on use - blocking cross-site request forgery and code replay. And the access token is **never placed in a URL**: the callback hands the browser a short-lived one-time code, which the app immediately scrubs from the address bar and exchanges for the real session token over a header.

Scopes requested: `openid email profile`. The `email` claim is required (it's how members are matched); `given_name`, `family_name` and `name` are used when present to populate the profile.

4 · SETUP GUIDE

4 Setup, step by step, from zero

You configure SSO in **two places**: at your **Identity Provider** (register Bernstein as an app) and in **Bernstein** (enter your IdP's details). Do the IdP side first, so you have the values to paste.

Part A · At your Identity Provider

1 · Create an app / client

Of type **Web application**, using the **Authorization Code** grant. (Providers call it "App client", "Application" or "OIDC client".)

2 · Register the redirect URI

Paste Bernstein's callback / reply URL (next callout). Copy the exact value from the Bernstein admin panel to avoid typos.

3 · Enable the scopes

Turn on `openid`, `email` and `profile` so the ID token carries the email we match on.

4 · Collect the values

Client ID, Client Secret, the Authorization endpoint and the Token endpoint. (Issuer and JWKS URL are optional - let Bernstein auto-detect them.)

THE REDIRECT URI TO REGISTER AT YOUR IDP

`https://app.bernstein.io/api/sso/v1/spaces/<SPACE_ID>/openid/callback`

The exact value is shown in the Bernstein SSO admin panel as **"Callback url"** - copy it from there. <SPACE_ID> is the Space's UUID, not its slug. A mismatch here is the single most common cause of a failed setup.

Part B · In Bernstein

1 · Open the panel

Sign in as a Space owner/admin and open Administration → Single Sign-On → OpenID Connect.

2 · Fill the form

Client Id, Client secret, Authentication url and Token url. Leave Issuer and JWKS url blank to auto-detect.

3 · Save

On save, Bernstein attempts discovery and auto-fills Issuer and JWKS when it can (see §6) - it never fails the save if discovery is unreachable.

4 · Test

Click the Test badge before activating. It opens the live sign-in URL in a new tab so you can complete a real round-trip against your IdP.

5 · Activate

Flip the toggle. If SAML is active for this Space, disable it first - only one protocol can be active at a time.

6 · Done

The Space switches to OpenID and every visitor to app.bernstein.io/<your-space> is offered SSO.

Behind the panel, the admin API is the standard REST shape: GET / POST / PUT on `/api/meta/v1/spaces/:space_id/openid`. Permitted fields: `client_id`, `client_secret`, `authentication_url`, `token_url`, `issuer`, `jwks_uri`, `active`. Updates are field-by-field, so you can correct a single value without re-entering the secret.

5 · PROVIDERS

5 Supported providers, and where to find the values

OIDC is a standard, so **any compliant provider works**. The table below points to the two endpoints for the common ones - replace the placeholders with your tenant's host.

	Authentication URL / Token URL	Notes
Okta	`<org>.okta.com/oauth2/v1/authorize` · `…/token`	Issuer auto-detects from the host root.
Microsoft Entra ID (Azure AD)	`login.microsoftonline.com/<tenant>/oauth2/v2.0/authorize` · `…/token`	Discovery auto-detects from the token-path parent.
Google	`accounts.google.com/o/oauth2/v2/auth` · `oauth2.googleapis.com/token`	Issuer = `https://accounts.google.com`; auto-detects.
Auth0	`<tenant>.auth0.com/authorize` · `…/oauth/token`	Issuer = `https://<tenant>.auth0.com/`.
OneLogin	`<sub>.onelogin.com/oidc/2/auth` · `…/token`	Auto-detects.
Keycloak	`<host>/realms/<realm>/protocol/openid-connect/auth` · `…/token`	Issuer = `<host>/realms/<realm>`.
AWS Cognito	`<domain>.auth.<region>.amazoncognito.com/oauth2/authorize` · `…/token`	⚠ **Set Issuer manually**: `cognito-idp.<region>.amazonaws.com/<userPoolId>`. Discovery is not served on the `auth.*` host.

Replace placeholders with your tenant's values. Cognito is the one provider that needs the Issuer set by hand - everything else auto-detects.

MANUAL INTEGRATION WITH ANY OTHER PROVIDER

Not in the table? Open your provider's discovery document at `<issuer>/.well-known/openid-configuration`. It's a JSON file containing `authorization_endpoint` (→ Authentication URL), `token_endpoint` (→ Token URL), `issuer` and `jwks_uri`. Paste those values into the Bernstein form. **That is the whole integration** - there is nothing provider-specific in Bernstein.

6 · AUTO-DISCOVERY

6 How Bernstein fills Issuer & JWKS itself

Verifying an ID token requires the IdP's **issuer** and **JWKS URL**. So operators don't have to hunt these down, Bernstein discovers them automatically, trying the likely locations in order until one returns a document with both values.

1 · Canonical

`<issuer>/.well-known/openid-configuration` - the only candidate that resolves AWS Cognito, once an issuer is set.

2 · Token-path parent

Walks up from the token endpoint - this is how Microsoft Entra (Azure) discovery is found.

3 · Token host root

Discovery at the host root - covers Google, Okta, Auth0 and OneLogin.

	Behaviour	Why
On save	Best-effort - fills blank Issuer/JWKS and **never fails the save** if discovery is down.	So you can configure even if the IdP is momentarily unreachable.
At sign-in	**Fail-closed** - if issuer and JWKS still can't be resolved, the sign-in is rejected (`sso_error=unavailable`).	**We never verify a token against an unpinned issuer or key set.**

THE ONE CASE AUTO-DETECT CAN'T SOLVE: AWS COGNITO WITH NO ISSUER SET

Cognito's discovery document lives on `cognito-idp.<region>.amazonaws.com/<userPoolId>`, which is *not derivable* from its hosted-UI token host. The fix is one-time: paste the **Issuer** (`https://cognito-idp.<region>.amazonaws.com/<userPoolId>`) and save - the JWKS URL then auto-detects from it.

7 · SECURITY MODEL

7 Why each guard exists

SSO is a trust boundary, so the design is conservative: tokens are verified, never assumed; codes are single-use; and a token is never placed where it could leak. The guarantees below hold for every sign-in.

	What it does	Why it matters
Authorization Code flow	Tokens travel on the back channel; only a one-time code is ever in a redirect URL.	Removes the largest browser-side exposure surface.
Signed-token verification	The ID token signature is verified against your IdP's published keys; `iss` and `aud` are checked; an unsigned `alg:none` token is rejected.	A forged or unsigned token cannot impersonate a user. Pinned issuer/JWKS is mandatory.
Single-use state	A token created at redirect and destroyed at callback.	Blocks cross-site request forgery and authorization-code replay.
No bearer in URL	The callback hands the app a short-lived single-use code, not the access token; the app scrubs it from the address bar at once.	The session token never lands in browser history, logs or referrers.
Cross-tenant binding	A federated identity from a different IdP can't be claimed by another Space; a local password account may adopt SSO only if its email is confirmed.	Stops one tenant from hijacking another tenant's identity.
Membership gate	**An existing user must already be a member of the Space, or sign-in is refused.**	**SSO authenticates - it doesn't silently grant access to non-members.**

WHAT SSO DOES, AND WHAT IT DOESN'T

SSO decides **who can sign in** and ties that to your corporate directory. It does not change Bernstein's evidence model: your files are still encrypted client-side with zero-knowledge encryption, and your content never leaves your control. SSO governs the front door; the vault behind it is unchanged.

8 · TROUBLESHOOTING

8 Troubleshooting playbook

When a sign-in fails, Bernstein redirects the member to the Space login with a clear notice and an `sso_error` code, instead of a blank error page. Match the symptom below to its cause and fix.

Bernstein-side errors (carry an sso_error code)

	Likely cause	Fix
unavailable	Issuer / JWKS couldn't be discovered. Classic AWS Cognito case (discovery isn't on the hosted-UI host), or the IdP returned no `email` claim, or `iss`/`aud` didn't match.	Set the Issuer for Cognito; enable the `email` scope/claim; confirm the Client ID matches the IdP app.
not_member	The authenticated email isn't a Space member, belongs to a different IdP, or is an unconfirmed local account.	Invite or confirm the user in the Space; ensure the same IdP.
expired	The state token was missing, stale or replayed - e.g. a reused old callback link, a double-submit, or a very slow login.	Simply start the sign-in again.

IdP-side errors (shown by your provider, not Bernstein)

	Likely cause	Fix
redirect_uri_mismatch	The callback URL registered at the IdP doesn't match Bernstein's.	Copy the exact **Callback url** from the admin panel into the IdP app config.
invalid_client / unauthorized_client	Wrong Client ID or Secret, or the Authorization-Code grant / web-app type isn't enabled.	Re-copy the credentials; enable the code grant and web-application type.
invalid_scope / login loops	The scopes `openid email profile` aren't enabled on the IdP app.	Enable those three scopes.
"User does not exist"	**Shown on the IdP's own page - the account isn't provisioned in your user directory. Bernstein never creates IdP users.**	**Create or confirm the user in your IdP (e.g. the Cognito user pool).**

Rule of thumb: an sso_error=… in the Bernstein URL is ours to fix in the config; an error word on your provider's own page is in the IdP app setup. The redirect URI and the three scopes account for the large majority of first-time failures.

9 · REFERENCE

9 Glossary & next steps

A one-line reminder of each term used in this brief, for quick reference during setup.

	Meaning
OIDC	OpenID Connect - an identity layer over OAuth 2.0.
IdP / OP	Identity Provider / OpenID Provider - your Okta, Entra, Google, Cognito tenant.
RP / client	Relying Party - Bernstein, the app relying on your IdP.
Authorization Code flow	Redirect to IdP → one-time code → back-channel token exchange.
id_token	A signed JWT asserting who the user is (carries `email`, `sub`, …).
Issuer (iss)	The IdP's canonical URL - the trust anchor for verification.
Audience (aud)	Who the token is for - must equal Bernstein's Client ID.
Scopes	What we ask for: `openid email profile`.
JWKS	The IdP's published public signing keys (`jwks_uri`).
Discovery	`<issuer>/.well-known/openid-configuration` - the machine-readable IdP config.
Managed user	A Space member auto-provisioned via SSO on first sign-in.
State	A single-use anti-CSRF/replay token tying a login attempt to its callback.

OpenID Connect SSO LIVE

In production on every white-label Space, with auto-discovery and the full security model above.

SAML 2.0 SSO LIVE

Available in parallel for organisations standardised on SAML - same per-Space model, different configuration fields.

Enterprise & white-label tailoring ON REQUEST

Guided onboarding, custom claim mapping and provisioning packs for enterprise and white-label deployments.

Need help wiring up a specific provider, or want this brief folded into a procurement or security-review pack? Contact your Bernstein partner - www.bernstein.io.