**Spotify Track Explorer – Architecture & Build Plan (Spring Boot + Next.js + AWS)**

Goal: A portfolio‑ready app where users sign in to **your app** (not via Spotify), then **link** their Spotify account to fetch and explore their playlists/tracks. Spring Boot provides the API + account linking; Next.js is the client; AWS hosts infra.

**1) Core Idea & Identity Model**

**Two identities, linked:**

* **App user (primary identity):** Your own Users table (email/password or Cognito). This is what the person uses to sign in to *your* app.
* **Spotify account (linked identity):** Optional secondary identity captured via OAuth 2.0 Authorization Code w/ PKCE. Stored per user. You use its tokens to call Spotify **on behalf of** the signed‑in app user.

This prevents being “tethered” to Spotify-only auth: users log into *your* app first, then optionally connect Spotify.

**2) High‑Level Architecture**

[Next.js (UI)] ↔ [Spring Boot API]

| |

| JWT (app auth) | Refresh Spotify tokens on demand

↓ ↓

Browser Spotify Web API

(User-granted scopes)

**Recommended hosting (MVP → Production):**

* **MVP/Low Cost:**
  + Next.js on **Vercel** (or Netlify).
  + Spring Boot on **AWS Elastic Beanstalk** (EC2) or **AWS Lightsail** (simple, cheap) with free-tier t2.micro.
  + **PostgreSQL** on **Amazon RDS** (free tier) or **Neon/Render** for very low dev cost.
  + Secrets in **AWS Secrets Manager**; non-secret config in **SSM Parameter Store**.
* **Production‑ready:**
  + **ECS Fargate** (or EKS) for Spring API behind an **ALB**.
  + **CloudFront** CDN in front of Next.js if self‑hosted; otherwise Vercel’s edge.
  + **RDS Postgres** multi‑AZ.
  + **CloudWatch** metrics/logs + **X‑Ray** tracing.

You can keep Next.js on Vercel and only run Spring + DB on AWS to minimise cost/complexity.

**3) OAuth & Auth Flows**

**3.1 App Authentication (to *your* app)**

Choose one:

1. **Spring Security + JWT** (email/password). Next.js stores an httpOnly session cookie with the JWT.
2. **Amazon Cognito** → Spring validates Cognito JWTs.

For portfolio clarity, option **1)** keeps everything in your repo and is simpler to demo.

**3.2 Spotify Account Linking (per user)**

* Use **Authorization Code with PKCE** (Spotify supports regular code flow; PKCE is good practice for SPA leg).
* Start the flow from your Next.js UI ("Connect Spotify" button) → hit your API /oauth/spotify/authorizewhich returns the Spotify authorize URL (with state bound to **your** user session).
* Spotify redirects to your backend callback /oauth/spotify/callback?code=...&state=... → backend exchanges code for access\_token, refresh\_token, expires\_in.
* Store tokens in spotify\_accounts table, linked to your users.id (NOT as the user’s login).

**3.3 Using the Tokens**

* Next.js calls your API (with **app** JWT) e.g. /me/playlists.
* API loads the user’s stored Spotify tokens, refreshes if expired, then calls Spotify Web API.
* Optionally cache results to DB for faster UI and offline exploration (with a “Refresh from Spotify” button).

**4) Data Model (PostgreSQL)**

-- Primary users of YOUR app

CREATE TABLE users (

id UUID PRIMARY KEY,

email TEXT UNIQUE NOT NULL,

password\_hash TEXT NOT NULL,

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW()

);

-- Linked Spotify accounts (optional per user)

CREATE TABLE spotify\_accounts (

id UUID PRIMARY KEY,

user\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,

spotify\_user\_id TEXT NOT NULL,

display\_name TEXT,

scope TEXT NOT NULL,

access\_token TEXT NOT NULL,

refresh\_token TEXT NOT NULL,

expires\_at TIMESTAMPTZ NOT NULL,

created\_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

UNIQUE (user\_id),

UNIQUE (spotify\_user\_id)

);

-- Optional cached entities

CREATE TABLE playlists (

id TEXT PRIMARY KEY, -- Spotify playlist ID

user\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,

name TEXT NOT NULL,

owner\_spotify\_id TEXT,

snapshot\_id TEXT,

last\_synced TIMESTAMPTZ

);

CREATE TABLE tracks (

id TEXT PRIMARY KEY, -- Spotify track ID

name TEXT NOT NULL,

artist TEXT NOT NULL,

album TEXT,

duration\_ms INT,

popularity INT,

preview\_url TEXT

);

CREATE TABLE playlist\_tracks (

playlist\_id TEXT REFERENCES playlists(id) ON DELETE CASCADE,

track\_id TEXT REFERENCES tracks(id) ON DELETE CASCADE,

added\_at TIMESTAMPTZ,

PRIMARY KEY (playlist\_id, track\_id)

);

**5) Spring Boot – Key Pieces**

**Dependencies:**

* spring-boot-starter-web
* spring-boot-starter-security
* spring-security-oauth2-client
* spring-boot-starter-data-jpa
* jjwt (or spring-security-oauth2-jose) for app JWTs
* postgresql

**Config outline:**

# application.yaml

app:

jwt:

issuer: https://track-explorer

secret: ${JWT\_SECRET}

expiry-mins: 60

spotify:

client-id: ${SPOTIFY\_CLIENT\_ID}

client-secret: ${SPOTIFY\_CLIENT\_SECRET}

redirect-uri: https://api.yourapp.com/oauth/spotify/callback

scopes: user-read-email playlist-read-private user-library-read

**Endpoints:**

* POST /auth/register – create app user
* POST /auth/login – login → return JWT (httpOnly cookie)
* GET /auth/me – current user
* GET /oauth/spotify/authorize – returns Spotify authorize URL (with generated PKCE + state)
* GET /oauth/spotify/callback – exchange code → store tokens → redirect to UI
* POST /oauth/spotify/disconnect – delete tokens
* GET /me/spotify/profile – proxy to Spotify /me
* GET /me/playlists – fetch playlists (optionally cache)
* GET /playlists/{id}/tracks – fetch tracks (optionally cache)

**Token refresh:**

* Before each Spotify call, check expires\_at ≤ now + 60s; if so, refresh.
* Store new access\_token, expires\_at.

**Security:**

* Protect app API with JWT filter (OncePerRequestFilter).
* Validate state on Spotify callback vs session/user.
* Store secrets in AWS Secrets Manager (injected to env at deploy).

**6) Next.js – Key Pieces**

* UI: Next.js App Router + Tailwind/daisyUI.
* App auth: simple forms → call /auth/register & /auth/login (set httpOnly cookie from API).
* After login, show **“Connect Spotify”** card if not linked.
* Pages:
  + /dashboard – user profile + connect/disconnect
  + /playlists – list + filter/sort + search
  + /playlists/[id] – tracks table, audio preview, export

**Client calls:**

* Use fetch('/api/...', { credentials: 'include' }) so cookies travel.
* Errors → toast notifications.

**7) AWS Mapping & Cost Notes**

* **Compute:** Elastic Beanstalk (single t2.micro) for Spring API. Scale later.
* **DB:** RDS Postgres (db.t3.micro) – free tier eligible. For dev, consider storage autoscaling off and low allocated storage to reduce cost. Stop when not in use.
* **Secrets:** AWS **Secrets Manager** (Spotify client secret, JWT secret). Rotate later.
* **Logs/Monitoring:** CloudWatch logs + alarms (5xx rate, latency). Health checks.
* **Networking:** Public ALB → EB → EC2. Restrict DB security group to EB.

**Nice-to-haves:**

* **CloudFront** in front of Next.js (if self-hosting UI). If using Vercel, skip.
* **S3** for exports (CSV/JSON of playlists) and images.

**8) Scopes & Features (Phased)**

**Phase 1 (MVP):**

* App auth (JWT)
* Link Spotify
* Fetch & display: profile, playlists, tracks

**Phase 2:**

* Caching to DB + Refresh button
* Search, sort, filters, multi‑playlist merge view
* Export CSV/JSON

**Phase 3:**

* Track analytics (duplicates, energy/tempo via Audio Features)
* Smart playlists (rules engine) → create new playlist via Spotify API (needs playlist-modify-private scope)
* Background sync job (scheduled)

**9) Example Sequence (Link + Fetch)**

**Link Spotify:**

1. User logs in → receives app JWT cookie.
2. Clicks **Connect Spotify** → UI calls GET /oauth/spotify/authorize.
3. API returns Spotify authorize URL (includes state and PKCE verifier hashed to challenge).
4. Browser → Spotify consent → back to /oauth/spotify/callback.
5. API exchanges code → stores tokens → redirects to /dashboard?linked=1.

**Fetch playlists:**

1. UI calls GET /me/playlists with app cookie.
2. API loads tokens, refreshes if needed, calls Spotify /me/playlists.
3. Optionally upserts into playlists & playlist\_tracks.
4. Returns clean JSON to UI.

**10) Minimal Spring Boot Snippets (Pseudo‑Code)**

// SpotifyOAuthController.java

@GetMapping("/oauth/spotify/authorize")

public AuthorizeUrl getAuthorizeUrl(@AuthenticationPrincipal AppUser user) {

var state = stateService.issue(user.getId());

var pkce = pkceService.create(); // codeVerifier + codeChallenge

var url = UriComponentsBuilder

.fromUriString("https://accounts.spotify.com/authorize")

.queryParam("client\_id", clientId)

.queryParam("response\_type", "code")

.queryParam("redirect\_uri", redirectUri)

.queryParam("scope", scopes)

.queryParam("state", state)

.queryParam("code\_challenge", pkce.getChallenge())

.queryParam("code\_challenge\_method", "S256")

.build(true).toUriString();

cache.store(state, pkce.getVerifier());

return new AuthorizeUrl(url);

}

@GetMapping("/oauth/spotify/callback")

public ResponseEntity<?> callback(@RequestParam String code, @RequestParam String state) {

var userId = stateService.consume(state);

var verifier = cache.takeVerifier(state);

var token = spotify.exchangeCodeForToken(code, verifier);

spotifyAccounts.save(

userId,

token.getSpotifyUserId(),

token.getAccessToken(),

token.getRefreshToken(),

Instant.now().plusSeconds(token.getExpiresIn()),

token.getScope()

);

return redirect("/dashboard?linked=1");

}

// SpotifyProxyService.java

public <T> T call(UserId userId, UriBuilder uri, Class<T> type) {

var acct = repo.findByUserId(userId);

if (acct.isExpiredSoon()) {

var refreshed = spotify.refresh(acct.refreshToken());

repo.updateTokens(refreshed);

acct = refreshed;

}

var req = HttpRequest.newBuilder(uri.build())

.header("Authorization", "Bearer " + acct.accessToken())

.build();

return http.send(req, type);

}

**11) Environment & Secrets**

* SPOTIFY\_CLIENT\_ID / SPOTIFY\_CLIENT\_SECRET → **Secrets Manager**
* JWT\_SECRET → **Secrets Manager**
* APP\_BASE\_URL, API\_BASE\_URL → Parameter Store / env vars
* Spotify console: add redirect URL: https://api.yourapp.com/oauth/spotify/callback

**12) Dev & Ops Tips**

* Keep Spotify scopes minimal; add more in Phase 3.
* Handle 401 from Spotify by refreshing once; if still 401 → force relink.
* Backoff on rate limits (Retry-After).
* Log user‑id and request‑id for traceability; don’t log raw tokens.
* For demos, seed a “mock user” with cached playlist JSON if no Spotify link yet.

**13) Roadmap Checklist**

* Spring Boot project skeleton (Gradle/Maven)
* JWT auth (register/login/me)
* DB schema (Liquibase/Flyway)
* Spotify authorize + callback + token storage
* /me/playlists and /playlists/{id}/tracks
* Next.js UI (login, connect, list, detail)
* AWS EB deploy + RDS + Secrets Manager
* Basic monitoring & alarms
* Caching + refresh button
* Export features + S3 (optional)
* Analytics + smart playlists (Phase 3)

**14) Why This Meets Spotify Limits**

* You **own** login & session (JWT/Cognito). Spotify is **just** a linked provider.
* You store & refresh tokens server‑side; client never sees Spotify tokens.
* You can cache to reduce rate‑limit pressure and provide fast UX.

**15) Next Steps For You**

1. Pick app auth: **Spring JWT** (simple) or **Cognito** (AWS‑native).
2. Create Spotify app in Dev Dashboard; add redirect URL.
3. Stand up API locally; wire /oauth/spotify/\*.
4. Build minimal Next.js pages: Login → Dashboard → Connect → Playlists.
5. Deploy API to EB + RDS; wire secrets; test end‑to‑end.

You’ll have a clean, professional, portfolio‑ready app that clearly demonstrates Java (Spring), JavaScript (Next.js), and sensible AWS usage.