Diaspora Backend — Auth Base Documentation (Detailed)

Generated: 2025-12-26 19:49

This document is the baseline documentation for the authentication foundation of the backend. It explains what each core file does, why it exists, and how the pieces work together.

Core Auth Architecture

- **Access Token (JWT)**: short-lived (10–15 min). Used for authorizing API calls. Stored in httpOnly cookie.
- **Refresh Token (JWT)**: long-lived (7–30 days). Used only to get a new access token. Stored in httpOnly cookie.
- **Refresh token rotation**: every refresh request revokes the old refresh token record and creates a new one.
- **DB persistence**: refresh token is **hashed** and stored in MySQL so it can be revoked and cannot be leaked as plaintext.
- **Role-based auth**: API routes can be protected by authentication and by role.

How a request flows through the system

```
(1) Client calls /auth/login with email+password
    -> auth.controller issues accessToken + refreshToken
    -> stores hashed refresh token in DB
    -> sets cookies (httpOnly)

(2) Client calls protected API (e.g., /investor/ping)
    -> requireAuth reads accessToken cookie, verifies JWT, sets req.user
    -> (optional) requireRole checks req.user.role

(3) When accessToken expires, client calls /auth/refresh
    -> reads refreshToken cookie
    -> verifies refresh JWT
    -> finds matching hashed token row in DB (bcrypt compare)
    -> revokes old row, creates new row (rotation)
    -> sets new cookies (access + refresh)

(4) Client calls /auth/logout
    -> clears cookies
    -> best-effort revokes refresh token row in DB
```

src/controllers/auth.controller.js

Why this file exists

- Contains all **business logic** for authentication endpoints: register, login, refresh, logout, me.
- Calls model layer (User, RefreshToken) and utility helpers (tokens, cookies, hashing, response).
- Keeps route files clean: routes only map URLs to controller functions.

What we do inside it

- **register**: Validates input, creates User, generates JWTs, stores hashed refresh token row, sets cookies.
- **login**: Validates credentials using bcrypt, generates JWTs, stores hashed refresh token row, sets cookies.
- **me**: Returns current authenticated user profile (safe fields). Used by frontend to restore session.
- **refresh**: Rotates refresh token: verify refresh JWT, match DB hash, revoke old token row, issue new tokens, set cookies.
- **logout**: Clears cookies and revokes the matching refresh token row (best effort).

Key snippet / shape

```
// Controller responsibilities (pseudo)
const accessToken = signAccessToken({ sub: user.id, role: user.role });
const refreshToken = signRefreshToken({ sub: user.id, role: user.role });

await RefreshToken.create({
  userId: user.id,
  tokenHash: await hashRefreshToken(refreshToken),
  expiresAt,
  revokedAt: null
});

setAuthCookies(res, { accessToken, refreshToken });
```

How it is used with other files

- Called by: **src/routes/auth.routes.js**
- Uses: **src/utils/token.js** to sign/verify tokens
- Uses: **src/utils/cookies.js** to set/clear cookies
- Uses: **src/utils/hashToken.js** to hash/compare refresh tokens
- Uses: **src/utils/response.js** for consistent responses
- Uses: Sequelize models **User** and **RefreshToken** for persistence

src/routes/auth.routes.js

Why this file exists

- Defines the public API surface for auth endpoints and maps them to controller handlers.
- This file should stay **thin**: no business logic, only routing + middleware composition.
- Separating routes from controllers makes the project scalable and testable.

What we do inside it

- Routes created:
- • POST /auth/register
- • POST /auth/login
- • POST /auth/refresh
- • POST /auth/logout
- • GET /auth/me (protected by requireAuth)

Key snippet / shape

```
const router = require("express").Router();
const { register, login, refresh, logout, me } = require("../controllers/auth.controller");
const requireAuth = require("../middlewares/requireAuth");

router.post("/register", register);
router.post("/login", login);
router.post("/refresh", refresh);
router.post("/logout", logout);
router.get("/me", requireAuth, me);

module.exports = router;
```

How it is used with other files

- Imports controller handlers from **src/controllers/auth.controller.js**
- Protects `/auth/me` using **src/middlewares/requireAuth.js**
- Mounted from **app.js** using something like `app.use('/auth', authRoutes)`

src/utils/response.js

Why this file exists

- Provides consistent API response formatting through two helpers: **OK** and **FAIL**.
- Prevents repeating boilerplate `res.status(...).json(...)` across controllers.
- Keeps frontend integration predictable (always `success`, `message`, and `data`/`error`).

What we do inside it

- **OK**: success responses. Includes `success: true`, `message`, `data`.
- **FAIL**: error responses. Includes `success: false`, `message`, `error.code` (and optional details).

Key snippet / shape

```
function OK(res, message, data=null, status=200) {
  return res.status(status).json({ success: true, message, data });
}

function FAIL(res, message, code="UNKNOWN_ERROR", status=400) {
  return res.status(status).json({ success: false, message, error: { code } });
}
```

How it is used with other files

- Imported by controllers and middlewares to keep response format consistent.
- Improves frontend integration: same JSON shape across all endpoints.

src/utils/token.js

Why this file exists

- All JWT operations are centralized here.
- Ensures consistent signing/verification across the project.
- Prevents mistakes like verifying refresh tokens with the access secret (or vice versa).

What we do inside it

- **signAccessToken**: uses ACCESS_TOKEN_SECRET and ACCESS_TOKEN_EXPIRES_IN.
- **signRefreshToken**: uses REFRESH_TOKEN_SECRET and REFRESH_TOKEN_EXPIRES_IN.
- **verifyAccessToken**: validates access token signature/expiry.
- **verifyRefreshToken**: validates refresh token signature/expiry.

Key snippet / shape

```
const jwt = require("jsonwebtoken");

function signAccessToken(payload) {
  return jwt.sign(payload, process.env.ACCESS_TOKEN_SECRET, {
    expiresIn: process.env.ACCESS_TOKEN_EXPIRES_IN || "15m",
  });
}

function signRefreshToken(payload) {
  return jwt.sign(payload, process.env.REFRESH_TOKEN_SECRET, {
    expiresIn: process.env.REFRESH_TOKEN_EXPIRES_IN || "30d",
  });
}

function verifyRefreshToken(token) {
  return jwt.verify(token, process.env.REFRESH_TOKEN_SECRET);
}
```

How it is used with other files

- Imported by auth controller and requireAuth middleware.
- Relies on `.env` secrets: ACCESS_TOKEN_SECRET and REFRESH_TOKEN_SECRET.

src/utils/cookies.js

Why this file exists

- Central place to set and clear auth cookies.
- Ensures cookies are always configured with correct security flags (httpOnly, secure, sameSite).
- Aligns cookie lifetimes with token lifetimes (maxAge based on env).

What we do inside it

- **setAuthCookies**: sets `accessToken` and `refreshToken` httpOnly cookies.
- **clearAuthCookies**: removes both cookies.
- Uses env flags so dev and production behave correctly:
  - COOKIE_SECURE=true in production (HTTPS only)
  - COOKIE_SAMESITE=none for cross-site setups (Next.js on different domain).

Key snippet / shape

```
function setAuthCookies(res, { accessToken, refreshToken }) {
  res.cookie("accessToken", accessToken, { httpOnly: true, maxAge: accessMs, secure, sameSite
  });
  res.cookie("refreshToken", refreshToken, { httpOnly: true, maxAge: refreshMs, secure, sameSite
  });
}

function clearAuthCookies(res) {
  res.clearCookie("accessToken", opts);
  res.clearCookie("refreshToken", opts);
}
```

How it is used with other files

- Imported by auth controller to set/clear httpOnly cookies.
- Relies on `.env` flags for secure/sameSite and expiry strings.

src/utils/hashToken.js

Why this file exists

- Handles hashing and comparing refresh tokens securely using bcrypt.
- This is required because we store refresh tokens hashed in DB (industry standard).
- If the database is leaked, attackers cannot use stored hashes to impersonate users.

What we do inside it

- **hashRefreshToken(token)**: returns bcrypt hash, stored in RefreshToken.tokenHash.
- **compareRefreshToken(token, tokenHash)**: checks if an incoming refresh token matches a stored hash.

Key snippet / shape

```
const bcrypt = require("bcryptjs");

async function hashRefreshToken(token) {
  return bcrypt.hash(token, 12);
}

async function compareRefreshToken(token, tokenHash) {
  return bcrypt.compare(token, tokenHash);
}

module.exports = { hashRefreshToken, compareRefreshToken };
```

How it is used with other files

- Imported by auth controller for refresh token storage and matching.
- Ensures refresh token values are never stored plaintext in the DB.

Why we avoided extra packages (example: cookie-parser)

Initially, we avoided adding extra dependencies because the project requirement was to keep packages minimal. For cookies we only needed to read two values, so a small utility was sufficient. Going forward, we can use stable, widely adopted packages when it meaningfully reduces manual code.