ReaderMagicLinkService

September 4, 2026

ReaderMagicLinkService

Kind: Service

Source: `atloria-monorepo/apps/api/src/reader-account/reader-magic-link.service.ts`

C3 magic-link auth — reader-identity claim source 2 ('account').

Mints the SAME docs-reader token as A3/D4 via ReaderTokenService.mintUnion (never a parallel identity): verify() upserts ReaderAccount + the per-project ReaderProjectProfile and re-mints with `sub` + `account` merged onto whatever grants/attrs the reader already held.

One-time-code semantics mirror auth/saml/sso-code.service: random 256-bit code, Redis-stored (keyed by its sha256 so a Redis dump leaks nothing redeemable), single-use via atomic GETDEL, short TTL. Abuse control mirrors the A3 grant flows (per-IP/per-email Redis windows) on top of the login throttle family applied at the controller.

`ReaderMagicLinkService` implements passwordless reader authentication using short-lived, single-use magic-link codes. It verifies a reader’s email, upserts the shared `ReaderAccount` and project-specific `ReaderProjectProfile`, then mints the existing union reader token through `ReaderTokenService.mintUnion` rather than creating a separate identity type.

Methods

Method	Signature	Returns	Description
`start`	`start(slugWithId: string, rawEmail: string, ip: string, existingToken: string, returnTo: string)`	`Promise<{ sent: true }>`	

Method	Signature	Returns	Description
`verify`	`verify(slugWithId: string, code: string, ip: string, headerToken: string)`	`Promise<{ token: string; email: string; returnTo: string }>`	Single-use redeem → upsert account + project profile → ONE docs-reader token with `sub` / `account` UNIONED onto the reader's existing claims.
`sanitizeReturnTo`	`sanitizeReturnTo(raw: string | undefined, projectId: string, slugWithId: string)`	`Promise<string>`	

Dependencies

- `PrismaService`
- `RedisService`
- `ReaderTokenService`
- `DocsAccessService`
- `EmailService`

Where it refuses work

- `ReaderMagicLinkService` stops the work with `NotFoundException` when `!ctx` — “Project not found”, in 2 places.
- `ReaderMagicLinkService` stops the work with `BadRequestException` when `!ReaderMagicLinkService.EMAIL_RE.test(email)` — “Enter a valid email address.”.
- `ReaderMagicLinkService` stops the work with `ServiceUnavailableException` when `!(await this.redis.exists(key))` — “Sign-in is temporarily unavailable — please try again.”.
- `ReaderMagicLinkService` stops the work with `UnauthorizedException` when `!payload || payload.projectId !== ctx.projectId`.
- `ReaderMagicLinkService` stops the work with `HttpException` when `n > max` — “Too many attempts — try again shortly.”.
- `ReaderMagicLinkService` stops the work with an early return when `!code || typeof code !== 'string' || code.length > 128`.

When something fails

- `ReaderMagicLinkService` handles failure in 5 places: it turns it into a return value in 3, lets it reach the caller in 1, and discards it silently in 1. A failure discarded silently leaves no trace for whoever debugs this later.

Diagram

```
sequenceDiagram
  participant Client
  participant Controller
  participant MagicLink as ReaderMagicLinkService
  participant Redis
  participant Email
  participant Account as ReaderAccount/Profile
  participant Tokens as ReaderTokenService

  Client->>Controller: POST /reader-account/magic-link/start
  Controller->>MagicLink: start(email, project, returnTo, ip)
  MagicLink->>MagicLink: sanitizeReturnTo(returnTo)
  MagicLink->>Redis: Check per-IP and per-email rate limits
  MagicLink->>MagicLink: Generate random 256-bit code
  MagicLink->>Redis: Store SHA-256(code) with short TTL
  MagicLink->>Email: Send magic-link URL containing code
  MagicLink-->>Controller: { sent: true }
  Controller-->>Client: Confirmation response

  Client->>Controller: GET/POST magic-link verification with code
  Controller->>MagicLink: verify(code, project)
  MagicLink->>Redis: Atomic GETDEL(SHA-256(code))
  Redis-->>MagicLink: Stored payload or missing/expired value
  MagicLink->>Account: Upsert ReaderAccount and ReaderProjectProfile
  MagicLink->>Tokens: mintUnion(existing grants/attrs + sub + account)
  Tokens-->>MagicLink: Reader token
  MagicLink-->>Controller: { token, email, returnTo }
  Controller-->>Client: Authenticated reader response
```

Usage

```
import { Injectable } from '@nestjs/common';
import { ReaderMagicLinkService } from './reader-magic-link.service';

@Injectable()
export class ReaderAuthController {
  constructor(
    private readonly readerMagicLinkService: ReaderMagicLinkService,
  ) {}
```

```
  async requestMagicLink(input: {
    email: string;
    projectId: string;
    returnTo?: string;
    ip: string;
  }) {
    return this.readerMagicLinkService.start();
  }

  async verifyMagicLink(code: string) {
    const { token, email, returnTo } =
      await this.readerMagicLinkService.verify();

    return {
      accessToken: token,
      reader: { email },
      returnTo,
    };
  }
}
```

AI Coding Instructions

- Always mint authenticated reader tokens through `ReaderTokenService.mintUnion`; preserve existing reader grants and attributes while adding `sub` and `account` claims.
- Store only the SHA-256 hash of the generated 256-bit magic-link code in Redis; never persist the redeemable raw code.
- Preserve single-use semantics by consuming codes with an atomic Redis `GETDEL`, not separate `GET` and `DEL` operations.
- Apply both per-IP and per-email Redis rate-limit windows in addition to controller-level login throttling.
- Route user-provided redirect destinations through `sanitizeReturnTo()` before storing or returning them to prevent unsafe redirects.

Relationships

- DEPENDS_ON → `PrismaService`
- DEPENDS_ON → `RedisService`
- DEPENDS_ON → `ReaderTokenService`
- DEPENDS_ON → `DocsAccessService`
- DEPENDS_ON → `EmailService`

Referenced By

- `ReaderAccountController` (DEPENDS_ON)
- `ReaderAccountModule` (MODULE_PROVIDES)