Architecting Modern Authentication: A Deep Dive into Fastify, Prisma, Google OAuth, and Magic Links

I. Executive Summary: Architecting "Better Auth" with Fastify and Prisma

The request for "better auth" encapsulates a significant paradigm shift in application security.[1] It signifies a move away from traditional, password-centric systems toward more secure, user-friendly, and flexible authentication strategies, including passwordless flows (like magic links) and federated identity (like Google Auth). This query also astutely identifies a specific ecosystem of modern, framework-agnostic libraries, such as better-auth and lucia-auth, which are designed to handle these complex flows.[3]

This report delivers on both interpretations of the query. It provides a comprehensive "roll your own" (RYO) implementation guide for the two requested flows—Google OAuth and magic links—built on the high-performance stack of Fastify and Prisma. It also provides an expert analysis of the managed library ecosystem, offering a clear recommendation for a production-ready path.

The selected technology stack is exceptionally well-suited for this task. Fastify provides a high-throughput, low-overhead server with a powerful plugin architecture, making it ideal for a high-performance authentication service.[5] Prisma offers a type-safe, modern ORM that radically simplifies database interactions, ensuring that complex queries for finding, linking, and creating users are secure, robust, and maintainable.[5]

This report is structured to build a complete solution from the ground up:

1. **Foundational Architecture:** Establishing the non-negotiable database schema and server structure required for any multi-provider system.
2. **Google OAuth Deep Dive:** A complete, end-to-end implementation of Google (OAuth 2.0) login, including critical security vulnerability analysis.
3. **Magic Link Deep Dive:** A guide to building a secure passwordless flow, addressing common but complex edge cases like email scanner-proofing.
4. **API Security:** Best practices for securing protected routes and managing sessions within the Fastify framework.
5. **Ecosystem Analysis:** A comparative analysis of the "Roll Your Own" approach versus integrating dedicated libraries like lucia-auth.

II. Foundational Architecture: Server and Database

Before implementing any specific authentication flow, a robust foundation must be laid. This architecture is the most critical component and is designed to support both Google Auth and magic links, as well as future providers, from the outset.

A. Initial Project Setup: Fastify, TypeScript, and Prisma

The project begins with a standard TypeScript, Fastify, and Prisma setup. pnpm is used as the package manager, but npm or yarn are also suitable.[5]

1. **Initialize Project and Install Dependencies:**
   Bash
   ```
   pnpm init
   # Install core dependencies
   pnpm add fastify @prisma/client prisma fastify-type-provider-zod zod
   # Install development dependencies
   pnpm add -D typescript tsx @types/node
   ```
   - fastify-type-provider-zod is included for first-class schema validation and type inference in route handlers.[5]
2. **Initialize Prisma:**
   Bash
   ```
   npx prisma init
   ```
   This creates the prisma/schema.prisma file and a .env file for database credentials.[5] The schema.prisma should be configured for the chosen database (e.g., PostgreSQL).[12]
3. Create Prisma Client Singleton:
   To prevent database connection exhaustion, a singleton pattern should be used for the Prisma client. Create a file src/lib/prisma.ts:
   TypeScript
   ```
   import { PrismaClient } from '@prisma/client';

   const globalForPrisma = globalThis as unknown as { prisma: PrismaClient };

   export const prisma =
     globalForPrisma.prisma ||
     new PrismaClient({
       log: ['query'],
     });
   ```

if (process.env.NODE_ENV!== 'production') globalForPrisma.prisma = prisma;

This pattern ensures that in development, the hot-reloading process reuses the existing client instance.[8]

B. The Multi-Provider Prisma Schema: The Database Core

This schema is the heart of the "better auth" architecture. It is based on the battle-tested, provider-agnostic model popularized by Auth.js, which is explicitly designed to handle multiple authentication methods for a single user.[13]
The Account and VerificationToken models are not optional; they are the database-level enablers for the requested features.

- **Google Auth (OAuth):** The Account model is the mechanism that links a single User record to their unique Google ID (the providerAccountId).[14]
- **Magic Link (Passwordless):** The VerificationToken model is purpose-built to store the secure, one-time-use tokens required for a magic link flow.[12]

Here is the complete, annotated schema.prisma:

Code snippet

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql" // Or "sqlite", "mysql", etc.
  url      = env("DATABASE_URL")
}

// The central user model.
model User {
  id            String    @id @default(cuid())
  name          String?
  email         String?   @unique
  emailVerified DateTime? @map("email_verified")
  image         String?
```

```
  // Relation: A user can have many accounts (e.g., email, google, github)
  accounts Account
  // Relation: A user can have many sessions (for stateful session auth)
  sessions Session

  @@map("users")
}

// The model for linking OAuth providers (e.g., Google) to a User.
model Account {
  id                String  @id @default(cuid())
  userId            String  @map("user_id")
  type              String  // e.g., "oauth", "credentials"
  provider          String  // e.g., "google", "github"
  providerAccountId String  @map("provider_account_id") // The user's ID from the provider
  refresh_token     String? @db.Text
  access_token      String? @db.Text
  expires_at        Int?
  token_type        String?
  scope             String?
  id_token          String? @db.Text
  session_state     String?

  // Relation: Links this account back to the User
  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  // Unique constraint: A user can only have one account per provider
  @@unique([provider, providerAccountId])
  @@map("accounts")
}

// Optional model for stateful, database-backed sessions.
// Not required if using stateless JWTs.
model Session {
  id           String   @id @default(cuid())
  sessionToken String   @unique @map("session_token")
  userId       String   @map("user_id")
  expires      DateTime

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@map("sessions")
```

```
}

// The model for one-time tokens (e.g., magic links, email verification).
model VerificationToken {
  identifier String   // e.g., the user's email
  token      String   @unique
  expires    DateTime

  @@unique([identifier, token])
  @@map("verification_tokens")
}
```

After defining the schema, run the migration: npx prisma migrate dev --name init.
This schema's design is clarified by understanding the role of each model:

Model	Primary Role	Key Fields	Used By
User	The central record for an individual.	id, email, name	Google Auth, Magic Link
Account	Links a User to an external OAuth provider.	userId, provider, providerAccountId	Google Auth
Session	Stores active, persistent user sessions (optional, for stateful sessions).	userId, sessionToken, expires	(Alternative to JWT)
VerificationToken	Stores one-time tokens for passwordless flows.	identifier, token, expires	Magic Link

C. Application Structure: A Modular Pattern for Routes, Services, and Controllers

A clean separation of concerns is essential for a secure and maintainable authentication system. A modular "service pattern" is strongly recommended.[10]
This structure is not just for organization; it is a security and testability best practice. Auth logic is complex and sensitive.

- auth.routes.ts: Defines API endpoints (e.g., /login, /callback) and their Zod schemas. Its only job is to handle HTTP definitions.[16]
- auth.controller.ts: Handles the FastifyRequest and FastifyReply. It extracts data (body, query) and calls the appropriate service. It does not know how to hash a password or

talk to Prisma.[11]

- auth.service.ts: Contains all business logic (e.g., login(email), verifyMagicToken(token), findOrCreateGoogleUser(profile)). It interacts with Prisma and performs cryptographic operations.

This separation means the security-critical logic in the service layer can be unit-tested in isolation, and controllers remain "dumb," reducing the attack surface.

D. Core Session Plugins: Configuring @fastify/cookie and @fastify/jwt

After a user successfully authenticates (via Google or magic link), they need a persistent session. The standard stateless approach is a JSON Web Token (JWT) stored in a secure cookie.
This is set up by registering two core Fastify plugins:

TypeScript

```
// In your main app.ts or a dedicated plugins/session.ts file

import { FastifyInstance } from 'fastify';
import fastifyCookie from '@fastify/cookie';
import fastifyJwt from '@fastify/jwt';
import fp from 'fastify-plugin';

async function sessionPlugin(app: FastifyInstance) {
  // 1. Register Cookie Plugin
  // This allows reading and setting cookies
  app.register(fastifyCookie);

  // 2. Register JWT Plugin
  // This adds JWT signing and verification methods to Fastify
  app.register(fastifyJwt, {
    secret: process.env.JWT_SECRET as string, // MUST be a strong secret from.env
    cookie: {
      cookieName: 'token', // The name of the cookie
      signed: false, // We sign the JWT payload, not the cookie
    },
  });
}

export default fp(sessionPlugin);
```

This configuration adds app.jwt.sign and app.jwt.verify methods, as well as req.cookies.[5]

III. Implementation Deep Dive 1: Google OAuth 2.0

This section provides a complete, end-to-end implementation of the Google Auth flow using the @fastify/oauth2 plugin.[20]

A. Provider Configuration: Registering @fastify/oauth2 for Google

The @fastify/oauth2 plugin is the Fastify-native choice for handling OAuth flows. It should be registered as a plugin:

TypeScript

```
// In a new file, e.g., plugins/google-oauth.ts
import { FastifyInstance } from 'fastify';
import fp from 'fastify-plugin';
import oauthPlugin from '@fastify/oauth2';

async function googleOAuthPlugin(app: FastifyInstance) {
  app.register(oauthPlugin, {
    name: 'google',
    scope: ['profile', 'email'], // Scopes to request from Google
    credentials: {
      client: {
        id: process.env.GOOGLE_CLIENT_ID as string,
        secret: process.env.GOOGLE_CLIENT_SECRET as string,
      },
      auth: oauthPlugin.GOOGLE_CONFIGURATION,
    },
    // This will create the /api/auth/google route automatically
    startRedirectPath: '/api/auth/google',
    // This is the URL Google will redirect back to
    callbackUri: 'http://localhost:3000/api/auth/google/callback',
  });
}

export default fp(googleOAuthPlugin);
```

This code registers the plugin and makes app.google available for use.[21]

B. Security Imperative: Understanding and Mitigating CSRF (CVE-2023-31999)

This is a non-negotiable security requirement. Versions of @fastify/oauth2 prior to 7.2.0 contained a critical Cross-Site Request Forgery (CSRF) vulnerability.[22]

- **The Vulnerability:** The OAuth 2.0 state parameter is the *only* defense against CSRF in this flow. It's a unique, secret string our server generates, sends to Google, and expects to receive back. We must verify the returned state matches the one we sent.[25] The vulnerability was that older plugin versions used a *static, single state* for all users, generated at application startup.[22]
- **The Attack:** An attacker could start their *own* login, get the *known* static state, and forge a malicious link. If a victim (logged into Google) clicked this link, their Google account would be connected to the *attacker's* account on our application.
- **The Fix:** Ensure you are using the latest version (pnpm i @fastify/oauth2@latest). Version 7.2.0 and later fix this by default. The plugin now correctly generates a *unique, per-request* state and stores it securely in a cookie, making the flow safe.[22]

C. The Google Authentication Flow: A Step-by-Step Implementation

This flow uses the modular structure defined in Section II-C.
1. The Login Route (/api/auth/google)
This route is created automatically by the @fastify/oauth2 plugin's startRedirectPath option.21 A frontend application simply needs to link to it: Login with Google.
2. The Callback Route (/api/auth/google/callback)
This is where our custom logic resides.

- src/modules/auth/auth.routes.ts:
  TypeScript
  ```
  //...
  app.get(
    '/google/callback',
    { schema: {... } }, // Add Zod schema for query params
    authController.googleCallback
  );
  //...
  ```

- src/modules/auth/auth.controller.ts:
  TypeScript

```
    //...
    async googleCallback(req: FastifyRequest, reply: FastifyReply) {
      const code = (req.query as { code: string }).code;
      const jwt = await authService.handleGoogleCallback(code, req.fastify);

      // Set the JWT in a secure cookie
      reply
      .setCookie('token', jwt, {
          httpOnly: true, // Prevents JS access
          secure: process.env.NODE_ENV === 'production', // HTTPS only
          sameSite: 'lax', // Mitigates CSRF
          path: '/',
        })
      .redirect('http://localhost:5173/dashboard'); // Redirect to frontend
    }
    //...
```

3. Callback Service Logic (src/modules/auth/auth.service.ts)
This service method performs the heavy lifting.

TypeScript

```
//...
import { FastifyInstance } from 'fastify';
import { prisma } from '../../lib/prisma';

//...
async handleGoogleCallback(code: string, app: FastifyInstance) {
  // 1. Exchange the code for an access token
  const { token } = await app.google.getAccessTokenFromAuthorizationCodeFlow({
    code,
  });

  // 2. Use the access token to fetch the user's profile from Google
  const googleProfileResponse = await fetch(
    'https://www.googleapis.com/oauth2/v2/userinfo',
    {
      headers: {
        Authorization: `Bearer ${token.access_token}`,
      },
    }
  );
```

```
  const profile = await googleProfileResponse.json();

  // 3. Find or create the user in the database
  const user = await this.findOrCreateGoogleUser(profile);

  // 4. Issue a session (JWT)
  const jwt = app.jwt.sign({
    userId: user.id,
    email: user.email,
    name: user.name,
  });

  return jwt;
}
//...
```

4. User Provisioning: The prisma.upsert "Find or Create" Pattern
This private service method is the core of the provisioning logic. It uses prisma.upsert to atomically find or create the user and their associated account, preventing race conditions.27

TypeScript

```
//... in auth.service.ts
async findOrCreateGoogleUser(profile: {
  id: string;
  email: string;
  name: string;
  picture: string;
}) {
  const account = await prisma.account.upsert({
    // 1. Find the account by its provider and providerAccountId
    where: {
      provider_providerAccountId: {
        provider: 'google',
        providerAccountId: profile.id, // The unique Google User ID
      },
    },
    // 2. If found, do nothing (empty update)
    update: {},
    // 3. If not found, create the Account AND connect-or-create the User
    create: {
```

```
    provider: 'google',
    providerAccountId: profile.id,
    type: 'oauth',
    user: {
      // This is the magic:
      connectOrCreate: {
        // Try to find a User with this email
        where: { email: profile.email },
        // If not found, create a new User with this data
        create: {
          email: profile.email,
          name: profile.name,
          image: profile.picture,
        },
      },
    },
  },
  include: { user: true }, // Return the full user object
 });

 return account.user;
}
//...
```

This single, idempotent query [27] correctly handles:

- A brand new user signing up.
- An existing user logging in.
- An existing user (who maybe signed up via magic link) linking their Google account for the first time.

IV. Implementation Deep Dive 2: Passwordless Magic Links

This flow is more complex than it appears due to critical security and usability edge cases. It uses the VerificationToken model from our schema.[14]

A. Architectural Flow

The flow is as follows [15]:

1. User POSTS to /api/auth/magic-link with their email.
2. Server generates a secure, short-lived, one-time-use token.
3. Server saves a **hash** of this token in the VerificationToken table.
4. Server emails the **raw** token to the user in a link.[29]
5. User clicks the link: GET /api/auth/verify?token=...
6. Server validates the token, ensures it's not expired, and **deletes it** to prevent reuse.[30]
7. Server issues a session (JWT) and sets the cookie.

B. Implementation: Step-by-Step Code

1. The Request Route (/api/auth/magic-link)

- src/modules/auth/auth.routes.ts:
  TypeScript
  ```
  app.post(
    '/magic-link',
    { schema: { body: z.object({ email: z.string().email() }) } },
    authController.requestMagicLink
  );
  ```

- src/modules/auth/auth.controller.ts:
  TypeScript
  ```
  async requestMagicLink(req: FastifyRequest<{ Body: { email: string } }>, reply: FastifyReply) {
    await authService.generateAndSendMagicLink(req.body.email);
    // Always return 202 Accepted, even if the email doesn't exist,
    // to prevent user enumeration attacks.
    return reply.code(202).send({ message: 'If your account exists, a login link has been sent.' });
  }
  ```

2. Service Logic: Generating and Sending the Link

- src/modules/auth/auth.service.ts:
  TypeScript
  ```
  import crypto from 'crypto';
  import nodemailer from 'nodemailer'; // Requires `pnpm add nodemailer`

  //...
  async generateAndSendMagicLink(email: string) {
    // 1. Find or create user.
    const user = await prisma.user.upsert({
      where: { email },
  ```

```
  create: { email },
  update: {},
});

// 2. Generate a secure, unguessable token.
const token = crypto.randomBytes(32).toString('hex');

// 3. BEST PRACTICE: Hash the token before storing it.
// We verify by hashing the incoming token and comparing to this.
const hashedToken = crypto
.createHash('sha256')
.update(token)
.digest('hex');

// 4. Store the hashed token in the database.
// Set an expiry (e.g., 15 minutes).
await prisma.verificationToken.create({
  data: {
    identifier: email,
    token: hashedToken,
    expires: new Date(Date.now() + 15 * 60 * 1000), // 15 minutes
  },
});

// 5. Email the *raw* token to the user.
const magicLink = `http://localhost:3000/api/auth/verify?token=${token}`;

// Nodemailer setup (example)
const transporter = nodemailer.createTransport({
  // Configure with your email provider (e.g., Resend, SendGrid, SMTP)
  // This example uses Ethereal for testing
  host: 'smtp.ethereal.email',
  port: 587,
  secure: false,
  auth: {
    user: 'example@ethereal.email',
    pass: 'password',
  },
});

await transporter.sendMail({
  from: '"My App" <noreply@myapp.com>',
  to: email,
```

```
    subject: 'Your Magic Login Link',
    html: `Click here to log in: <a href="${magicLink}">${magicLink}</a>`,
  });
}
//...
```

3. The Verification Route (/api/auth/verify)

- src/modules/auth/auth.routes.ts:
  TypeScript
  ```
  app.get(
    '/verify',
    { schema: { querystring: z.object({ token: z.string() }) } },
    authController.verifyMagicLink
  );
  ```

- src/modules/auth/auth.controller.ts:
  TypeScript
  ```
  async verifyMagicLink(req: FastifyRequest<{ Querystring: { token: string } }>, reply:
  FastifyReply) {
    try {
      const jwt = await authService.verifyMagicToken(req.query.token, req.fastify);

      // Set cookie and redirect (same as Google callback)
      reply
      .setCookie('token', jwt, {
          httpOnly: true, secure: true, sameSite: 'lax', path: '/',
        })
      .redirect('http://localhost:5173/dashboard');
    } catch (error) {
      // Handle invalid or expired token
      reply.redirect('http://localhost:5173/login?error=invalid_token');
    }
  }
  ```

4. Token Verification Logic (src/modules/auth/auth.service.ts)

TypeScript

```
//...
async verifyMagicToken(token: string, app: FastifyInstance) {
  // 1. Hash the incoming token to match what's in the DB.
  const hashedToken = crypto
```

```
.createHash('sha256')
.update(token)
.digest('hex');

// 2. Find and delete the token in a transaction.
// This is critical to ensure it's one-time-use.
const dbToken = await prisma.$transaction(async (tx) => {
  const foundToken = await tx.verificationToken.findUnique({
    where: { token: hashedToken },
  });

  if (!foundToken) {
    throw new Error('Invalid token');
  }

  // Delete it immediately to prevent reuse
  await tx.verificationToken.delete({
    where: { token: hashedToken },
  });

  return foundToken;
});

// 3. Check for expiry
if (new Date() > dbToken.expires) {
  throw new Error('Expired token');
}

// 4. Find the user associated with the token
const user = await prisma.user.findUnique({
  where: { email: dbToken.identifier },
});

if (!user) {
  throw new Error('User not found');
}

// 5. Issue a session (JWT)
const jwt = app.jwt.sign({
  userId: user.id,
  email: user.email,
  name: user.name,
});
```

```
  return jwt;
}
//...
```

C. Security & Usability: Handling Email Scanners and Link Expiry

The implementation above has a critical usability flaw. Many corporate email clients (like Microsoft ATP Safe Links) automatically "click" links in emails to scan them for malware.[32] This scanner's GET request will hit our /api/auth/verify endpoint, consume the one-time-use token, and make the link dead for the actual user.

The Solution: Two-Step Verification

The GET request from the email must *not* create a session or consume the token.

1. **Modify the Flow:** The GET /api/auth/verify?token=... route should be changed. It should find the token, check its expiry, but **not delete it**.
2. **Redirect to Frontend:** This GET handler should then redirect to a frontend page: reply.redirect(http://localhost:5173/auth/magic-link/confirm?token=${token}`).
3. **Frontend Page:** This page renders a simple button: "Click to Log In."
4. **New Endpoint:** This button triggers a POST /api/auth/magic-link/consume request, sending the *same token* in the body.
5. **Consume Token:** This new POST handler performs the logic from the verifyMagicToken function (find, check expiry, **and delete**), then issues the session cookie.

This defeats email scanners, as they only make the initial GET request and do not execute the JavaScript required to click the button and make the subsequent POST request.

D. Alternative Pattern: Using a Temporary UnconfirmedUser Table

An alternative pattern, excellent for *initial sign-up verification*, involves a temporary table.[30]

- **Schema:** model UnconfirmedUser {... }.[30]
- **Flow:**
    1. POST /register creates an UnconfirmedUser record.
    2. An email is sent with the UnconfirmedUser.id: .../confirm-email?user_id=....[30]
    3. GET /confirm-email finds the UnconfirmedUser by its ID.
    4. It checks the createdAt timestamp for expiry (e.g., 15 minutes).[30]
    5. If valid, it *copies* the data to the main User table.
    6. It **deletes** the UnconfirmedUser record, making the link one-time-use.[30]

This pattern is very secure for sign-ups but is less ideal for a "log in" (magic link) flow for existing users, as it conflates registration with authentication. The VerificationToken model is the cleaner, more appropriate choice for the "log in" use case.

V. API Security: Protecting Routes and Managing Sessions

Once the user has a JWT, the API must have a mechanism to protect endpoints and manage that session.

A. Creating a Reusable Authentication Decorator (app.authenticate)

The Fastify-native way to create reusable authentication logic is with a decorator.[11] This decorator is registered in a plugin.
Step 1: Type Augmentation
First, we must tell TypeScript that the FastifyRequest object will have a user property.11

TypeScript

```
// types/fastify.d.ts
import '@fastify/jwt';

type UserPayload = {
  userId: string;
  email: string;
  name: string;
};

declare module 'fastify' {
  interface FastifyRequest {
    user: UserPayload;
  }
}

declare module '@fastify/jwt' {
  interface FastifyJWT {
    payload: UserPayload;
    user: UserPayload;
  }
}
```

Step 2: The authenticate Decorator Plugin

TypeScript

```
// plugins/auth.ts
import { FastifyInstance, FastifyReply, FastifyRequest } from 'fastify';
import fp from 'fastify-plugin';

async function authPlugin(app: FastifyInstance) {
  // Decorate the request object with a 'user' property, initialized to null
  app.decorateRequest('user', null);

  // Decorate the Fastify instance with an 'authenticate' function
  app.decorate('authenticate', async (
    req: FastifyRequest,
    reply: FastifyReply
  ) => {
    try {
      // 1. Get the token from the cookie
      const token = req.cookies.token; // 'token' is the cookieName from JWT setup

      if (!token) {
        return reply.status(401).send({ message: 'Authentication required' });
      }

      // 2. Verify the token
      // This automatically uses the secret from the @fastify/jwt setup
      const decoded = req.jwt.verify<FastifyJWT['user']>(token);

      // 3. Attach the user payload to the request
      req.user = decoded;

    } catch (err) {
      reply.clearCookie('token'); // Clear invalid token
      return reply.status(401).send({ message: 'Invalid or expired token' });
    }
  });
}

export default fp(authPlugin);
```

This plugin provides a simple app.authenticate function that can be used as a hook.[11]

B. Applying Protection: Using the preHandler Hook

With the decorator registered, protecting any route is trivial using the preHandler hook.[11]

TypeScript

```
// In any routes file, e.g., src/modules/user/user.routes.ts

app.get(
  '/me',
  {
    preHandler: [app.authenticate], // This is the protection
  },
  async (req, reply) => {
    // If this handler runs, we are 100% sure
    // req.user is populated and the token is valid.
    return req.user;
  }
);
```

C. Best Practices for Secure Cookies

A stolen JWT is as bad as a stolen password. The cookie transporting it *must* be secured. When setting the cookie in the controller, all of the following attributes are mandatory for a secure-by-default setup [11]:

- httpOnly: true: **This is the most important.** It prevents client-side JavaScript (document.cookie) from accessing the token, which is the primary defense against XSS (Cross-Site Scripting) token theft.[11]
- secure: true: Ensures the cookie is *only* sent over HTTPS. This prevents Man-in-the-Middle (MITM) attacks from sniffing the token on public Wi-Fi. This should be set to process.env.NODE_ENV === 'production'.[11]
- sameSite: 'lax' (or 'strict'): A powerful defense against CSRF attacks. It controls when the browser sends the cookie with cross-origin requests. lax is a strong, balanced default.[18]
- path: '/': Ensures the cookie is sent for all routes on the domain.

VI. The "Better Auth" Ecosystem: A Comparative Analysis

The "Roll Your Own" (RYO) approach detailed in Sections III-V provides maximum control but also carries a significant security and maintenance burden. For most production applications, a managed library is a superior choice.

A. Analysis: "Roll Your Own" (RYO) vs. Managed Libraries

- **Roll Your Own (RYO):**
  - **Pros:** Full control over every detail, no external dependencies, and a deep understanding of the underlying security principles.
  - **Cons:** Extremely high complexity and security burden. It is easy to miss critical edge cases like the CSRF vulnerability [22] or the email scanner problem.[32]
- **Managed Libraries:**
  - **Pros:** Outsource complex security logic to experts. These libraries handle session management, token/state generation, provider-specific quirks, and security patches.
  - **Cons:** Less control, and you are tied to the library's opinions and update cycle.

B. Option 1: Integrating lucia-auth

lucia-auth is a modern, framework-agnostic, and highly-regarded authentication library. It is arguably the best-in-class choice for this specific stack.[4]

- **First-Class Fastify Support:** It provides an official Fastify middleware (lucia/middleware).[35]
- **Prisma Adapter:** It has an official Prisma adapter (@lucia-auth/adapter-prisma).[4]
- **Flexible & Unopinionated:** It is praised for being simpler and more flexible than other options.[36] It provides the core auth functions but allows you to build your *own* routes and controllers, fitting perfectly into the modular pattern established in this report.
- **Excellent Documentation:** The Lucia documentation includes step-by-step guides for the *exact* flows requested: GitHub/Google OAuth [37] and Email Verification Links (Magic Links).[31]

Example lucia-auth Setup:

TypeScript

```
// src/lib/auth.ts
import { lucia } from 'lucia';
import { fastify } from 'lucia/middleware';
import { prisma } from '@lucia-auth/adapter-prisma';
import { prisma as prismaClient } from './prisma'; // Your singleton client

export const auth = lucia({
  env: process.env.NODE_ENV === 'production'? 'PROD' : 'DEV',
  middleware: fastify(), // Integrates with Fastify
  adapter: prisma(prismaClient), // Connects to Prisma

  getUserAttributes: (data) => {
    return {
      email: data.email,
      name: data.name,
      image: data.image,
    };
  },
});

export type Auth = typeof auth;
```

This auth object then provides all the necessary methods (e.g., auth.createSession, auth.validateSession) to build the auth routes.

C. Option 2: Integrating better-auth

This is the specific library the "better Auth" query may refer to.[1] It is a valid option that also fits the stack, providing a Prisma adapter (better-auth/adapters/prisma) [1] and a Fastify plugin (fastify-better-auth).[3] It supports email/password and social logins. However, lucia-auth currently has a larger, more active community and more comprehensive documentation for these specific advanced patterns.

D. Option 3: Auth.js (formerly NextAuth.js)

While Auth.js is the market leader for React-based auth and defined the multi-provider schema we used [14], its support for non-Next.js backends is not mature. Support for Fastify is listed as "Experimental Release" or "Open PR".[39] This makes it a high-risk, non-production-ready choice for a pure Fastify backend.

E. Expert Recommendation: Choosing the Right Strategy

- **For Learning / Full Control:** Use the **"Roll Your Own" (RYO)** approach detailed in this report (Sections II-V). It is the only way to fully understand the security principles at play.
- **For Production / Speed:** Use **lucia-auth**. It is the best-in-class, production-ready, managed library for the Fastify + Prisma stack. It handles the complex and high-risk session/token minutiae securely, while giving you the flexibility to build your API routes and services as you see fit.

VII. Conclusion and Final Repository Structure

This report has provided a comprehensive architecture for implementing "better auth" with Fastify and Prisma. The solution hinges on several key architectural decisions:

1. **A Multi-Provider Schema:** Adopting the Auth.js-style Prisma schema (with User, Account, and VerificationToken models) is the non-negotiable foundation for supporting both OAuth and passwordless flows.[13]
2. **Modular Service-Based Architecture:** Separating logic into routes, controllers, and services is critical for building testable, secure, and maintainable authentication code.[16]
3. **Secure RYO Implementation:** When rolling your own, critical security details cannot be overlooked. This includes patching the @fastify/oauth2 CSRF vulnerability [22], hashing magic link tokens [40], and implementing a two-step verification to defeat email scanners.[32]
4. **Fastify-Native Security:** API protection should be handled natively using Fastify decorators (app.authenticate) applied via preHandler hooks, and session JWTs must be transported in httpOnly, secure, sameSite cookies.[18]
5. **A Pragmatic Ecosystem:** While building RYO is instructive, a production system is better served by lucia-auth, which provides a secure, "batteries-included" solution that integrates perfectly with Fastify and Prisma.[4]

A final, clean repository structure based on these principles would be as follows:

```
/
├── prisma/
│   ├── schema.prisma      # Your multi-provider schema
│   └── migrations/
├── src/
│   ├── lib/
```

```
│   │   ├── prisma.ts       # Prisma client singleton
│   │   └── auth.ts         # (If using Lucia) Lucia-auth instance
│   ├── modules/
│   │   ├── auth/
│   │   │   ├── auth.routes.ts     # API routes (e.g., /google/callback, /magic-link)
│   │   │   ├── auth.controller.ts  # Request/Reply handling
│   │   │   └── auth.service.ts    # Business logic (Prisma, crypto, email)
│   │   └── user/
│   │       ├── user.routes.ts     # Protected routes (e.g., /me)
│   │       ├── user.controller.ts
│   │       └── user.service.ts
│   ├── plugins/
│   │   ├── auth.ts          # The `app.authenticate` decorator plugin
│   │   ├── cookie.ts        # @fastify/cookie setup
│   │   ├── jwt.ts           # @fastify/jwt setup
│   │   └── google-oauth.ts    # @fastify/oauth2 setup
│   ├── types/
│   │   └── fastify.d.ts      # Fastify request/JWT type augmentation
│   └── app.ts               # Main Fastify server, registers plugins & modules
└── package.json
└── tsconfig.json
└──.env                  # GOOGLE_CLIENT_ID, JWT_SECRET, DATABASE_URL
```

Works cited

1. Building (Better Auth) in Fastify: Multi-Tenant SaaS and Secure API Authentication - Peerlist, accessed on November 3, 2025, https://peerlist.io/shrey_/articles/building-better-auth-in-fastify-multitenant-saas-and-secure-api-authentication
2. How to use Prisma ORM with Better-Auth and Next.js, accessed on November 3, 2025, https://www.prisma.io/docs/guides/betterauth-nextjs
3. flaviodelgrosso/fastify-better-auth: Easy integration of BetterAuth library in Fastify applications 🛡️ - GitHub, accessed on November 3, 2025, https://github.com/flaviodelgrosso/fastify-better-auth
4. Getting started - Lucia Auth, accessed on November 3, 2025, https://v2.lucia-auth.com/getting-started/
5. How to set up a Node.js API with Fastify and Prisma - DEV Community, accessed on November 3, 2025, https://dev.to/micaelmi/setting-up-a-nodejs-api-90j
6. Fastify: Fast and low overhead web framework, for Node.js, accessed on November 3, 2025, https://fastify.dev/
7. Building Node.js Apps with Fastify: A Beginner's Guide | Better Stack Community, accessed on November 3, 2025, https://betterstack.com/community/guides/scaling-nodejs/introduction-to-fastify/
8. Token based authentication with Fastify, JWT, and Typescript - thatarif, accessed

on November 3, 2025, https://thatarif.in/posts/token-based-authentication-with-fastify-jwt/
9. Basic CRUD with Prisma, Fastify and Node - DEV Community, accessed on November 3, 2025, https://dev.to/faraib/basic-crud-with-prisma-fastify-and-node-4lk3
10. token based auth with fastify, prisma, jwt and typescript - GitHub, accessed on November 3, 2025, https://github.com/arifimran5/jwt-auth-fastify
11. Token based authentication with Fastify, JWT, and Typescript | by ..., accessed on November 3, 2025, https://medium.com/@atatijr/token-based-authentication-with-fastify-jwt-and-typescript-1fa5cccc63c5
12. Simple Next.js Magic Link JWT Authentication with Prisma, PostgreSQL, and Resend, accessed on November 3, 2025, https://dev.to/diegocasmo/simple-nextjs-magic-link-jwt-authentication-with-prisma-postgresql-and-resend-21l
13. Credentials Authentication with Database Sessions Using Auth.js, SvelteKit and Prisma, accessed on November 3, 2025, https://dev.to/carstenlebek/credentials-authentication-with-database-sessions-using-authjs-sveltekit-and-prisma-2epm
14. Prisma - Auth.js, accessed on November 3, 2025, https://authjs.dev/getting-started/adapters/prisma
15. Authentication with AuthJS, Resend, Typescript, Nextjs, Prisma, | by Priyankashah - Medium, accessed on November 3, 2025, https://medium.com/@priyankashah3107/authentication-with-authjs-resend-typescript-nextjs-prisma-de49227f7b7f
16. sujeet-agrahari/node-fastify-architecture: A modular folder ... - GitHub, accessed on November 3, 2025, https://github.com/sujeet-agrahari/node-fastify-architecture
17. Fastify Clean Architecture - YouTube, accessed on November 3, 2025, https://www.youtube.com/watch?v=5zYzjzoTok4
18. JWT utils for Fastify - GitHub, accessed on November 3, 2025, https://github.com/fastify/fastify-jwt
19. node.js - Fastify Decorator not acting as a preHandler in Fastify ..., accessed on November 3, 2025, https://stackoverflow.com/questions/79068825/fastify-decorator-not-acting-as-a-prehandler-in-fastify-rest-api
20. fastify/fastify-oauth2: Enable to perform login using oauth2 protocol - GitHub, accessed on November 3, 2025, https://github.com/fastify/fastify-oauth2
21. Google OAuth2 with Fastify + TypeScript From Scratch - DEV ..., accessed on November 3, 2025, https://dev.to/fozooni/google-oauth2-with-fastify-typescript-from-scratch-1a57
22. @fastify/oauth2 vulnerable to Cross Site Request Forgery due to ..., accessed on November 3, 2025, https://github.com/advisories/GHSA-g8x5-p9qc-cf95
23. Cross-site Request Forgery (CSRF) in @fastify/oauth2 | CVE-2023 ..., accessed on November 3, 2025,

https://security.snyk.io/vuln/SNYK-JS-FASTIFYOAUTH2-5752413
24. CVE-2023-31999 Detail - NVD, accessed on November 3, 2025, https://nvd.nist.gov/vuln/detail/CVE-2023-31999
25. node.js - OAuth2 callback - identify user? - Stack Overflow, accessed on November 3, 2025, https://stackoverflow.com/questions/55893028/oauth2-callback-identify-user
26. Prevent Attacks and Redirect Users with OAuth 2.0 State Parameters - Auth0, accessed on November 3, 2025, https://auth0.com/docs/secure/attack-protection/state-parameters
27. How to find or create a record with Prisma? - Stack Overflow, accessed on November 3, 2025, https://stackoverflow.com/questions/71524243/how-to-find-or-create-a-record-with-prisma
28. Magic Link Authentication: Building a Cross-Device Authentication System (Part 2) - Medium, accessed on November 3, 2025, https://medium.com/@mbaochajonathan/magic-link-authentication-building-a-cross-device-authentication-system-part-2-aa791fa48ea8
29. Passwordless Auth in Node.js with Magic Links (Step-by-Step) - YouTube, accessed on November 3, 2025, https://www.youtube.com/watch?v=bi8TYKRJwHU
30. Secure Your Sign-Ups: Email Validation in Node.js with Fastify ..., accessed on November 3, 2025, https://dev.to/micaelmi/secure-your-sign-ups-email-validation-in-nodejs-with-fastify-prisma-and-nodemailer-39pn
31. Email authentication with verification links in Express · Lucia, accessed on November 3, 2025, https://v2.lucia-auth.com/guidebook/email-verification-links/express/
32. Magic Links and issues with tokens expiring, security issues - Tips - Bubble Forum, accessed on November 3, 2025, https://forum.bubble.io/t/magic-links-and-issues-with-tokens-expiring-security-issues/293467
33. qlaffont/fastify-auth-prisma - GitHub, accessed on November 3, 2025, https://github.com/qlaffont/fastify-auth-prisma
34. Run multiple auth functions in Fastify - GitHub, accessed on November 3, 2025, https://github.com/fastify/fastify-auth
35. Getting started in Fastify - Lucia Auth, accessed on November 3, 2025, https://v2.lucia-auth.com/getting-started/fastify/
36. Lucia Auth vs Auth.js - Avishka Devinda, accessed on November 3, 2025, https://avishka.dev/blog/lucia-vs-auth-js
37. GitHub OAuth · Lucia, accessed on November 3, 2025, https://v2.lucia-auth.com/guidebook/github-oauth
38. Implementing Google Authentication in a NextJS 14 Application with AuthJS 5, MongoDB, and Prisma ORM | by Sazzadur Rahman | Medium, accessed on November 3, 2025, https://medium.com/@sazzadur/implementing-google-authentication-in-a-nextjs

-14-application-with-authjs-5-mongodb-and-prisma-bbfcb38b3eea
39. Integrations - Auth.js, accessed on November 3, 2025, https://authjs.dev/getting-started/integrations
40. How to securly verify user by passwordless magic link? - Stack Overflow, accessed on November 3, 2025, https://stackoverflow.com/questions/76600446/how-to-securly-verify-user-by-passwordless-magic-link