🔐 Web App Security Layers – Documentation

✅ Layer 1: Middleware – Authentication Layer (Token Validation)

Purpose:

To **check if a request contains a valid token** (authentication) before proceeding to protected pages or APIs.**Runs before request is processed** (server-side). Uses `verifyToken()` to check **if the user is authenticated**.

How it works:

- Runs on **every request** (server-side only).
- Uses `verifyToken()` to check the JWT.
- Redirects or blocks the request if token is missing/invalid.
- Can **skip public pages** (e.g., `/login`, `/about`) via route matching logic.

Strengths:

- Globally applies to all requests.
- Stops invalid users from accessing protected routes or APIs early.
- Fast and efficient.

Weaknesses:

- Runs **only on initial request** (server-side).
- ❌ **Does not protect client-side navigation** (e.g., `router.push('/dashboard')`).
- ❌ Cannot stop users from **manually entering URLs** after logging in.
- ❌ Cannot check **user roles** — only verifies token.

👉 **This is why we need Layer 2.**

✅ Layer 2: Page-Level Guards – Authorization Layer (Auth + Role Guards)

Purpose:

To **protect routes/components on the client side**, and ensure the user is:

- Authenticated (has a valid token)
- Authorized (has the correct role)

How it works:

- Runs in **React/Next.js pages or layouts**.
- `authGuard.tsx` → checks token presence/expiration.
- `roleGuard.tsx` → checks if the user has the required role.
- Redirects or blocks UI rendering for unauthorized access.

Strengths:

- Prevents unauthorized access via **client-side navigation**.
- Works with `router.push`, browser back/forward, direct access.
- Blocks protected pages from rendering on the frontend.

Weaknesses:

- ❌ **Does not secure backend APIs** (can still be called via tools like Postman).
- ❌ Can be bypassed if someone manually calls API endpoints without going through the UI.

👉 **This is why we need Layer 3.**

✅ Layer 3: API-Level Security – Server-Side Enforcement

Purpose:
To **protect API endpoints** on the server so that only:

- Authenticated users
- With valid roles
  can perform specific backend operations.

How it works:

- Each API handler is wrapped with `withAuth(handler)` or similar logic.
- Token is verified before executing logic.
- Can include role-based checks (`if user.role !== 'admin' return 403`).

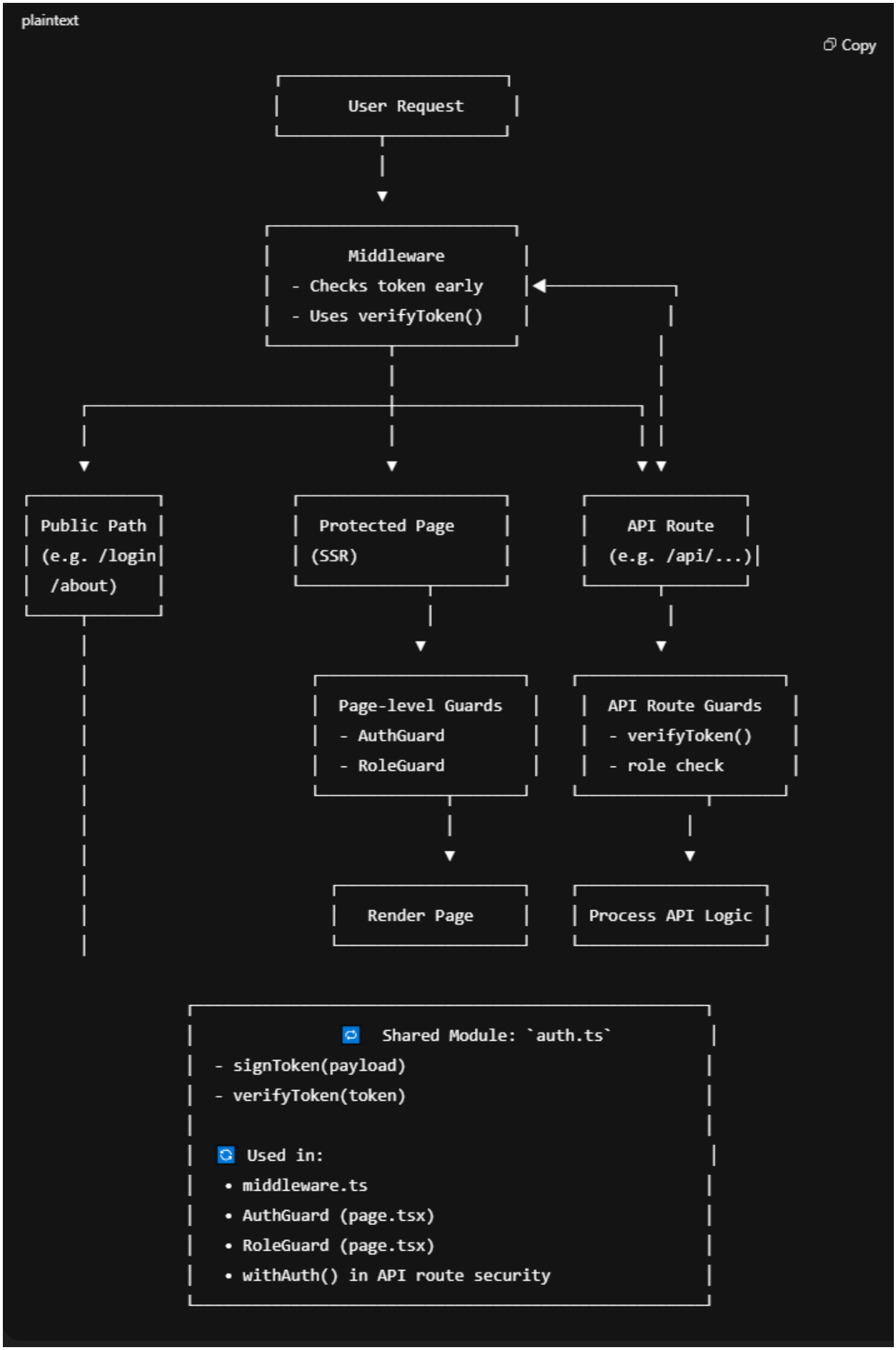
Strengths:

- Final layer of defense — ensures only authorized users can **call backend APIs**.
- Protects against **API abuse**, token tampering, or bypassing UI logic.
- Server-side only — cannot be bypassed by frontend manipulation.

Layer	Runs Where	Protects	Weakness It Solves
1. Middleware	Server (early)	Initial request access	Blocks unauthenticated access globally
2. Page Guard	Client-side	Route-level UI rendering	Blocks client-side routing by token/role
3. API Guard	Server (API)	Backend logic & data access	Prevents unauthorized backend API calls

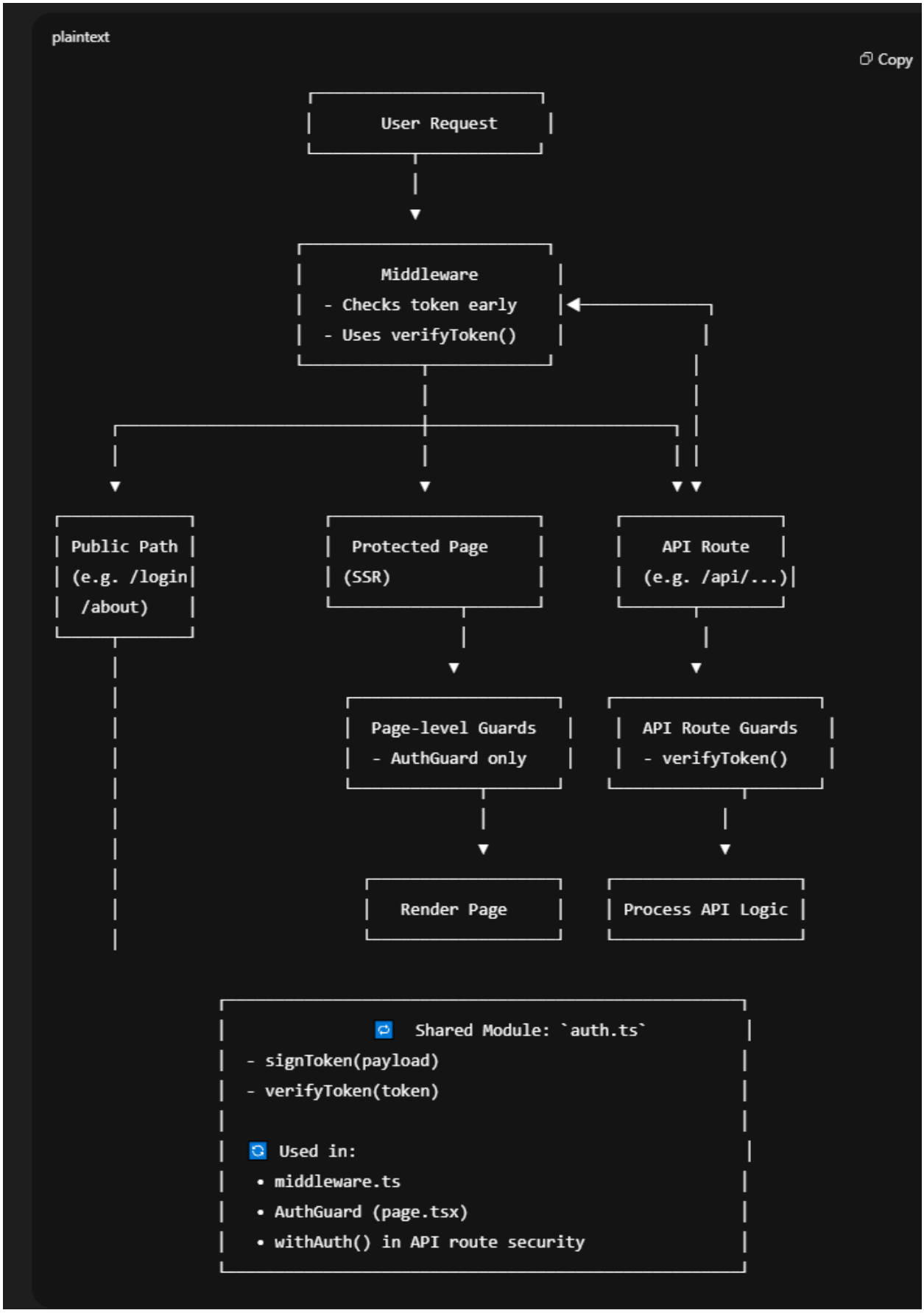
plaintext

Copy



plaintext

Copy



For using middleware - First level security

1. Common lib/auth.ts file
2. Common middleware.ts file - uses verifytoken()
3. Login file api/login.ts - uses signtoken()

For using guards - for page level security

1. Common lib/auth.ts file
2. Authguard file - // lib/authGuard.tsx
3. Roleguard file - // lib/roleGuard.tsx
4. Using them file - app/dashboard/page.tsx

For using withAuth - For API level security

1. Common lib/auth.ts file
2. /lib/withAuth.ts file - for common file to use verifytoken()
3. /app/api/tasks/route.ts - for using withauth file in every api call

If no role needed…you will skip the role part

In nestjs like backend ..its managed if just you add authguard,and roleguard…..they will automatically manage API level security…as you did in HireSmart project in gitHub