MCP OAuth Demo — Minimal Client/Server/Auth Flow

This project demonstrates a minimal Model Context Protocol (MCP)-style setup using OAuth 2.1 (Auth Code + PKCE): * An MCP Resource Server (tool server) that only responds to requests with a valid access token. * An Authorization Server that handles login & consent and issues tokens. * A Client that discovers the auth server from a 401 WWW-Authenticate, performs the OAuth dance, stores the token, then calls the tool successfully.

This is the foundation for adding an AI Agent (MCP client) and later a Gateway (TES) for fine-grained, policy-based auth and consent automation.

1) Components

Auth (/auth)

OAuth 2.1 authorization server (local demo): serves authorization & token endpoints, issues access tokens.

Server (/server)

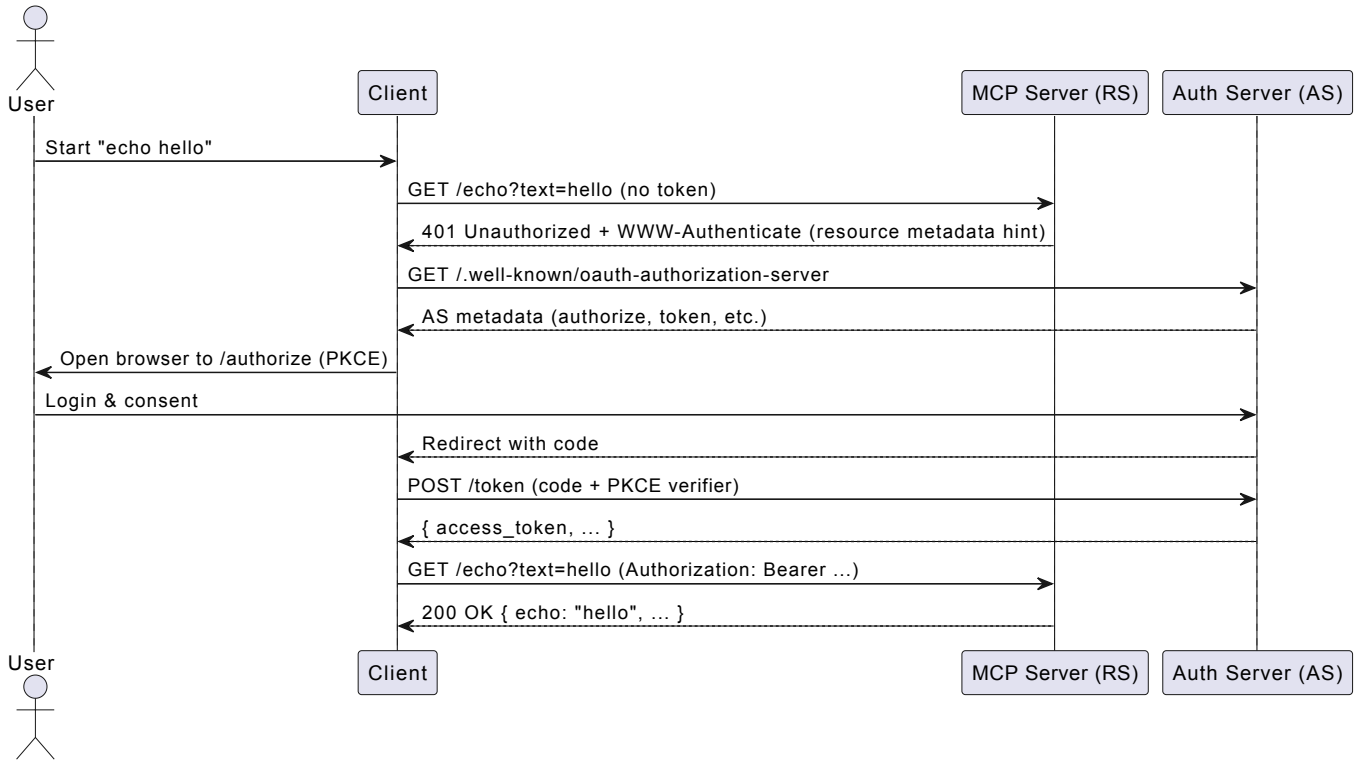
MCP-style tool server (resource server). Exposes a protected GET /echo?text=.... Validates incoming Bearer tokens.

Client (/client)

A simple app that:

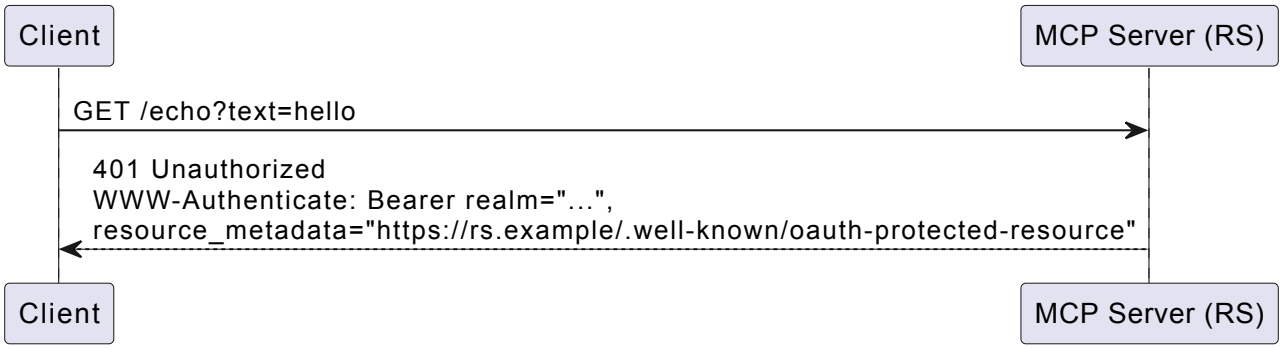
1. calls GET /echo without a token
2. reads the WWW-Authenticate hint
3. performs Authorization Code + PKCE in the browser
4. exchanges code → token
5. retries the call with the token (success)

2) High-Level Flow (at a glance)



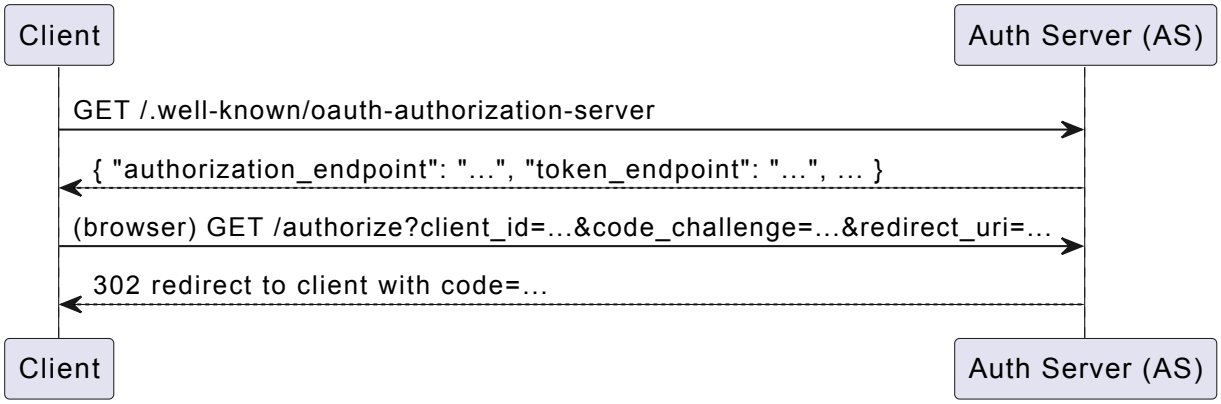
3) Detailed Request/Response Flow

3.1 First Call (No Token)



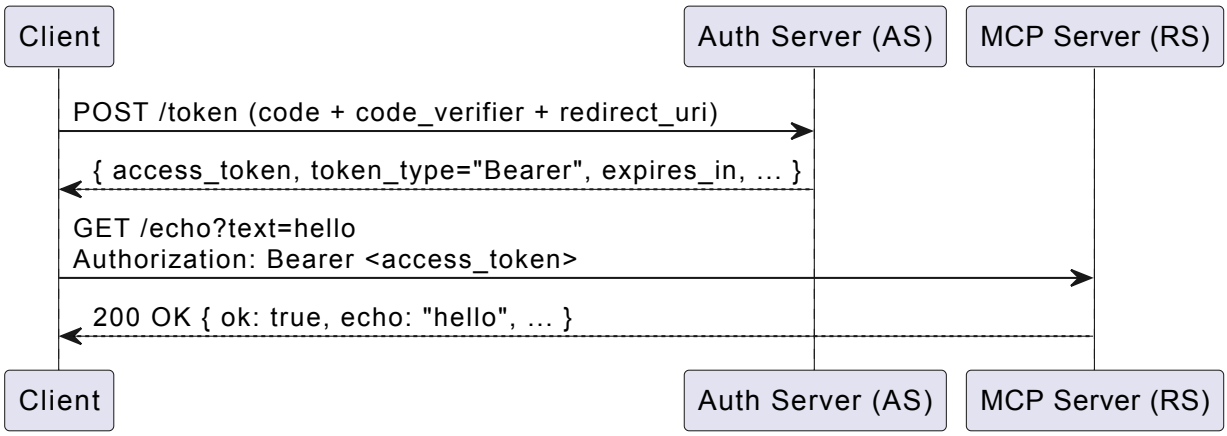
What happens * The Resource Server refuses the request and points the client to its metadata (or directly to the authorization server) via WWW-Authenticate.

3.2 Discovery + OAuth Authorization



What happens * Client fetches AS metadata so it knows where to send the user. * Opens the browser to the authorization endpoint (with PKCE). * After login/consent, the AS redirects back with a code.

3.3 Token Exchange + Protected Call



What happens

- Client exchanges code → token using the PKCE code_verifier.
- Client retries the tool call with Authorization: Bearer … and succeeds.

4) Token Validation (server-side)

On each protected request, the MCP server: 1. Extracts Authorization: Bearer . 2. Validates the JWT (signing key / issuer / audience / expiry). 3. (Optional) Introspects the token at the AS. 4. Enforces any required scopes/claims. 5. Proceeds with the tool action (e.g., echo) if valid; otherwise returns 401/403.

5) Local Development

From the repo root:

Auth server

cd auth npm i npm run dev

Runs on e.g. http://localhost:9000.

MCP server (resource)

cd server npm i npm run dev

Runs on e.g. http://localhost:9090.

Client

cd client npm i npm run dev

This client will: * call the MCP server, * open your browser for login/consent, * exchange the code for a token, * retry the protected call.

If you prefer, add a root script (optional):

npm i -D concurrently npm run dev

to boot services together.

6) What This Demo Proves

```
* OAuth 2.1 auth-code + PKCE end-to-end, including metadata discovery.
* Protected resource semantics for an MCP-style tool server.
* A working client that learns where to authenticate from the server (not
hardcoded).
```

7) What's Next (Roadmap)

```
1.  Swap the simple client for an AI Agent (as the MCP client)
•   Use Python (e.g., LangGraph or a minimal ADK-style loop) to call the
MCP server.
•   Keep the same OAuth flow: the agent "client" gets the 401, performs
the code flow (headless or with a small local redirect handler), stores
the token, and calls again.
2.  Introduce the Gateway (TES) for fine-grained authorization
•   Intercept the agent's MCP requests.
•   Enforce per-tool / per-action policies (context-aware).
•   Automate consent where policy allows; otherwise prompt.
•   Optionally mint short-lived assertion tokens for hop-by-hop isolation.
3.  Dynamic Client Registration (RFC 7591)
•   Register clients on the fly against the AS.
•   Supports multi-server ecosystems without manual client provisioning.
4.  Token Refresh + Rotation
•   Add refresh token flows, short lifetimes, and key rotation to harden
the demo.
5.  Audit & Logs
•   Emit structured logs from client/server/auth for traceability.
```

8) Troubleshooting

```
•   ERR_MODULE_NOT_FOUND jose
```

Run npm i jose in the folder that uses it (auth/server). • Imports vs module type If you see "ECMAScript imports in CommonJS" errors, ensure: • package.json has "type": "module" (or set TS module: nodenext) • Your ts-node / tsconfig matches (e.g., moduleResolution: nodenext). • Port conflicts Change the default ports in .env or server configs if something else is running.