AWS Fargate Solution Design

You are an AWS Solutions Architect with 20+ years of software development experience. Design a detailed end-to-end solution for the following scenario, including multiple options, trade-offs, and a recommended best option:

1. Context and Current Setup
Cloud provider: AWS
Compute platform: ECS on AWS Fargate (no EC2)
Network:
ECS Services are private, running in private subnets inside a VPC.
Access is through an Application Load Balancer (ALB) in public subnets.
The ALB has an AWS-provided DNS name.
There is also a custom domain configured in Route 53:
Hosted Zone in Route 53 with a CNAME (or ALIAS) pointing to the ALB DNS.
ACM certificate is used for HTTPS termination on the ALB.
Applications:
Frontend: React single-page application (SPA), containerized, deployed on its own ECS Fargate Service.
Backend API: Python FastAPI service, containerized, deployed on another ECS Fargate Service.
Cluster: Both ECS services run in the same ECS cluster.
Authentication & Authorization:
Uses Microsoft Entra ID (Azure AD) for both authentication and authorization.
Single-tenant Entra ID setup (only one Azure AD tenant).
Routing / Domains:
The React UI and FastAPI backend are both reachable via the ALB.
You can assume something like:
https://app.example.com → React SPA (ECS Service 1)
https://api.example.com or https://app.example.com/api → FastAPI backend (ECS Service 2)
The exact path vs. subdomain split can be part of your design options.
2. Requirements
2.1 Authentication / Access Requirements
UI (React SPA):
End-users access the UI URL directly in the browser via the ALB and custom domain.
When they access the UI, they must be prompted to sign in via Microsoft Entra ID if they are not already authenticated.

Once signed in, the SPA should have the necessary tokens to:
Display user info (e.g., name, email).
Call the backend FastAPI service securely on behalf of the user.
Service/API (FastAPI backend):
Must be accessible via:
Browser (e.g., direct GET request to some endpoint).
API clients such as Postman.
The React UI when user is already authenticated (SPA → API).
All these access paths should enforce Microsoft Entra ID authentication:
If calling from SPA, the SPA should attach a valid access token in the Authorization: Bearer header.
If calling from Postman or browser directly, the caller must obtain a token from Entra ID and pass it to the API as a Bearer token.
The API should validate tokens and enforce authorization based on claims (e.g., scopes, roles, groups).
Centralized Authorization in Entra ID:
Both authentication and authorization decisions (at least coarse-grained) should be based on Entra ID:
Users sign in using Entra ID.
User roles, groups, or app-specific permissions are defined and managed in Entra ID.
The FastAPI backend should consume these claims (e.g., roles, groups, scp, app_roles) to make authorization decisions.
Constraints & Prohibitions:
Do NOT use AWS API Gateway.
Do NOT use AWS Cognito.
Keep latency low and connections secure (TLS everywhere, minimal extra hops).
2.2 Non-Functional Requirements
Security:
All traffic over HTTPS only.
Private ECS services: only the ALB should have public access, services stay in private subnets.
Tokens should be short-lived; refresh handled appropriately.
Avoid storing tokens insecurely (e.g., no access tokens in localStorage if that can be avoided; consider secure cookies vs. memory storage tradeoffs).
Performance / Latency:
Avoid unnecessary proxy layers that would add significant latency.
Prefer regional alignment (Entra ID endpoints vs AWS region/ location where reasonable).
Consider ALB OIDC offload tradeoffs vs. app-level token validation.
Developer Experience:
API must be easily testable via Postman:
Document how developers obtain a token from Entra ID (client

credentials, OAuth 2.0 authorization code, device code, etc.).
Show how they configure Postman to call the API with Bearer tokens.
Frontend developers should have a clear React + MSAL (or similar) integration model.
3. Solution Scope
Provide a detailed architecture and solution design, including:

High-level Architecture Diagram Description
Describe (in text) how you would draw an architecture diagram with:
VPC, subnets (public for ALB, private for ECS/Fargate services).
ALB listeners & target groups.
Two ECS Fargate services (React SPA, FastAPI API).
Route 53 hosted zone and DNS routing (e.g., app.example.com, api.example.com).
Microsoft Entra ID (Authorization Server / IdP).
Token flows (SPA → Entra, SPA → API, Postman → Entra → API, browser direct → Entra → API).
Multiple Design Options for Authentication & Integration Present several options (at least 4–5 distinct options) for how to implement authentication and UI→service integration, explicitly considering:
Option A – App-Level Auth with SPA & API Both Integrated Directly with Entra ID
Frontend (React SPA):
Use OAuth 2.0 Authorization Code Flow with PKCE via MSAL.js or equivalent library.
SPA registers as a public client / SPA application in Entra ID.
SPA obtains:
ID token for user identity.
Access token(s) for calling the FastAPI backend (registered as a separate Web/API app in Entra).
Token storage:
Discuss the pros/cons of storing tokens in memory, sessionStorage, or localStorage.
Recommend secure patterns (e.g., in-memory with silent token renew if using MSAL, or using hidden iframe if allowed, or rotating refresh tokens).
Backend (FastAPI):
Register a separate Entra ID application representing the API (with its own client ID, app ID URI, scopes / app roles).
SPA requests specific scopes (e.g., api:///user.read) for calling the API.
FastAPI validates the access token on each request:
Use Entra's OpenID Connect discovery document to fetch JWKS, issuer, etc.

Validate signature, issuer, audience, and scopes.
Extract user identity and authorization claims (roles, groups, custom claims).
Postman / Browser Direct:
Users/developers obtain an access token from Entra ID (via OAuth 2.0 Authorization Code, device code, or client credentials for service-to-service) and pass it as Authorization: Bearer to the API.
Authorization:
Use app roles or Entra AD groups.
Map them to claims in tokens and enforce in FastAPI via decorators/middleware.
Option B – ALB-Level OIDC Authentication (ALB as OIDC Client) + Token Forwarding to Backend
Configure ALB authenticate-oidc action against Microsoft Entra ID.
ALB enforces login before requests reach SPA or API:
Discuss:
One ALB with different listeners/rules:
https://app.example.com → React SPA target group.
https://api.example.com or /api/* → FastAPI target group.
ALB side OIDC:
ALB redirects unauthenticated users to Entra ID.
Entra ID returns an authorization code; ALB exchanges it and obtains tokens.
ALB attaches ID token or user info headers to the backend.
Show challenges and/or trade-offs:
ALB's OIDC integration is generally more suitable for web backends rather than pure SPAs.
Handling API access via Postman: how to bypass ALB auth or utilize x-amzn-oidc-data headers, or alternatively require explicit Bearer tokens anyway.
Token injection vs. direct token validation by backend.
Evaluate whether ALB OIDC is a good fit here, given the requirement that:
The API must be callable by SPA, browser, and Postman with Entra-based tokens.
The SPA is a true frontend application (not server-side rendered).
Option C – Backend-For-Frontend (BFF) Pattern
SPA is served by a minimal backend (Node.js or Python) that:
Handles authentication server-side using Entra ID (Auth Code Flow).
Stores session cookies; backend calls API using on-behalf-of (OBO) flow or direct API tokens.
Explore pros/cons, but clearly note:
This may complicate the current architecture (React SPA running in ECS as static assets).
Good for security (no tokens in browser), but may diverge from the

current Fargate deployment model.
Option D – Hybrid: SPA Uses Auth Code + PKCE; FastAPI Validates Tokens; Postman Uses Same API Registration
Similar to Option A, but explicitly show:
How to configure Entra ID with:
One SPA client app registration.
One API (Web) app registration.
How to share these between:
SPA → API calls.
Postman → API calls.
Browser direct → API calls (if desired).
This might end up being one of the recommended options, so describe it in greater detail.
Option E – httpd (Apache) Web Server Acting as Auth Proxy in the Same ECS Services (for UI and API Separately)
Describe an option where you introduce an Apache httpd reverse proxy in front of both the UI and the API, each in their own ECS Fargate Service:
Overall Idea:
For the UI ECS Service:
Run a container (or multi-container task) where Apache httpd:
Terminates incoming HTTP from the ALB (inside VPC).
Performs OIDC authentication with Microsoft Entra ID using mod_auth_openidc (or similar).
Serves static React build assets (or proxies to a Node server serving the SPA).
Once the user is authenticated, Apache sets secure cookies / headers and forwards authenticated requests to the SPA content.
For the API ECS Service:
Another httpd proxy container in front of FastAPI:
Handles OIDC auth against Entra ID.
Validates the ID/access tokens.
Forwards authenticated requests (with user claims injected as headers) to the FastAPI app behind it on localhost or another container in the same task.
Entra Integration:
Detailed configuration for mod_auth_openidc:
OIDCProviderMetadataURL pointing to Entra's discovery endpoint.
OIDCClientID, OIDCClientSecret (for confidential client flows, especially for the API).
OIDCRedirectURI on the respective domains (app.example.com for UI; api.example.com for API).
Requested scopes (e.g., openid profile email api:///user.read).
How cookies are stored (mod_auth_openidc session cookies) and how access tokens are managed by Apache.
Flow for UI:

Browser hits https://app.example.com.
ALB routes to UI target group → httpd container.
mod_auth_openidc detects no session → redirects to Entra login.
After successful login, Entra redirects back to OIDCRedirectURI, Apache completes the OIDC flow, sets session, and then:
Serves SPA static files.
Optionally injects user info into headers or a bootstrap JSON endpoint for SPA to consume.
Discuss whether the SPA itself needs direct access tokens (for calling API) or whether:
The API is called via the same httpd proxy, and httpd handles token exchange / on-behalf-of, or
SPA still uses MSAL and obtains its own token for API, while httpd only protects UI access.
Flow for API:
Client (SPA / Postman / browser) hits https://api.example.com.
ALB routes to API target group → httpd container.
There are two main patterns:
Browser & SPA via session:
mod_auth_openidc authenticates and keeps a session cookie.
For SPA: calls to /api/* might use the same cookie, and httpd injects tokens/claims as headers to FastAPI.
Bearer token mode:
mod_auth_openidc configured in a "token verification" mode:
Expects Authorization: Bearer header from SPA/Postman.
Validates the JWT (signature, issuer, audience, scopes).
On success, forwards request to FastAPI with claims in headers; on failure, returns 401.
Discuss which of these is better, given the requirement that Postman and direct browser calls must be supported.
FastAPI Behavior:
With httpd handling OIDC/token validation, FastAPI can:
Either trust headers from Apache for identity and authorization.
Or still validate Bearer tokens directly (defense in depth).
Discuss security implications of trusting proxy headers (e.g., use of a private network, mTLS or security groups to ensure only httpd can access FastAPI).
Pros and Cons:
Pros:
Centralizes OIDC logic in Apache; simpler app code in FastAPI/ React.
Standard, battle-tested mod_auth_openidc behavior.
Potential to hide tokens from the SPA in some designs (if using pure server-side session for API).
Cons:
Additional complexity (Apache config, session management).

Another component in the path, some added latency.
Need to ensure pattern still supports Postman and direct access appropriately.
Clearly compare this Apache proxy option against the other options and indicate where it shines and where it is not ideal.
For each option (A–E):
Describe:
Token flow (step-by-step).
Where validation occurs (ALB vs. Apache vs. app).
How UI → API integration works.
How Postman / direct calls work.
Provide pros, cons, complexity, security, and latency analysis.
Best-Practice Recommendation Among the presented options, choose and clearly mark one best option for this scenario, considering:
Strong adherence to the requirement to use Entra ID only (no API Gateway, no Cognito).
Clear and consistent Entra ID usage for both SPA and API.
Secure handling of tokens.
Simplicity and maintainability.
Support for UI, browser, Postman, and (if relevant) httpd proxy flows.
Good developer experience.
Likely, the best option will be some variant of:
React SPA using Auth Code + PKCE via MSAL.
FastAPI validating access tokens from Entra ID directly.
Separate Entra App Registrations for SPA and API, with defined scopes/app roles.
Postman configured to acquire tokens for the same API app registration.
Justify this choice deeply and explicitly, and also explicitly mention why you would or would not choose the httpd proxy option vs ALB OIDC or pure app-level auth.
4. Detailed Entra ID Configuration
Provide a step-by-step, detailed configuration guide for Microsoft Entra ID (single-tenant) that supports the recommended architecture:

App Registrations
SPA App (e.g., “MyApp SPA”):
Type: public client SPA.
Redirect URIs:
https://app.example.com
https://app.example.com/auth/callback (or similar, depending on routing).
Implicit grant vs Auth Code with PKCE:

Use Authorization Code with PKCE, not implicit flow.
Grant it permission to call the API application.
Describe required manifest or configuration changes (e.g., spa redirectUris, allowPublicClient, etc.).
API App (e.g., “MyApp API”):
Type: Web/API (confidential client).
Expose an API:
Define Application ID URI (e.g., api://).
Define scopes: e.g., user.read, user.write, admin.
Optionally define app roles for authorization: Reader, Writer, Admin.
Describe how to configure “Authorized client applications” so that the SPA can request tokens for the scopes.
Optionally, if using httpd mod_auth_openidc with confidential client flows:
Describe whether separate app registrations are needed for httpd acting as a confidential client (especially for API proxy patterns).
Permissions, Scopes, and Consent
Configure the SPA app to request the API app’s scopes:
e.g., api:///user.read.
Configure admin consent as needed for organizational rollout.
Discuss delegated permissions vs. application permissions and clarify which are used in this scenario (primarily delegated, since the user signs in).
User Roles / Groups & Authorization
Describe how to:
Define app roles within the API App registration (e.g., role IDs, display names, allowed member types = Users/Groups).
Assign users or groups to app roles in Entra ID enterprise application blade.
Explain how roles appear in the token (e.g., roles claim vs. groups claim).
Discuss token size concerns with many groups; suggest enabling group overage or using app roles for more predictable claim sizes.
Give guidelines on whether to use:
App roles for API-level authorization.
Entra ID groups for coarse RBAC.
Or custom claims (via optional claims or custom extension attributes) when necessary.
FastAPI Integration Provide detailed guidance (pseudo-code or code-level concepts) for how to integrate FastAPI with Entra ID both with and without a proxy:
Without Apache proxy:
Use Python libraries like:
python-jose, PyJWT, msal, or others for token validation.
Configuration in FastAPI:
Entra tenant ID.

Authority / issuer URL, e.g.:
https://login.microsoftonline.com//v2.0
Audience (API Application ID URI or Client ID).
Required scopes (e.g., api:///user.read).
Implement a reusable dependency or middleware that:
Extracts Authorization header.
Validates JWT against Entra's JWKs.
Checks iss, aud, exp, nbf, etc.
Checks that the user has required scope(s)/role(s).
Injects the user principal (with claims) into the FastAPI path operations.
With Apache proxy:
Describe how FastAPI would:
Trust user identity and roles passed via headers set by mod_auth_openidc (e.g., X-User, X-Roles, X-Email).
Optionally still re-validate Bearer tokens for critical endpoints (defense in depth).
Discuss how to implement FastAPI dependencies that read identity from headers instead of from JWT directly.
Provide example patterns like:
A get_current_user dependency.
A decorator or dependency like @requires_role("Admin") or @requires_scope("user.read").
React SPA Integration Describe how to integrate the React SPA with Entra ID:
Use MSAL.js (or a similar Entra ID SPA library).
Configure MSAL with:
Client ID of the SPA app registration.
Authority: https://login.microsoftonline.com/.
Redirect URIs matching Entra registration.
postLogoutRedirectUri.
On initial load:
If not authenticated, redirect to login or show "Sign In".
After login, store the MSAL account and tokens.
For calling the API:
Use acquireTokenSilent (with fallback to acquireTokenRedirect/ acquireTokenPopup) to get an access token for the API scopes.
Attach the access token in the Authorization: Bearer header in fetch/Axios calls.
If using the Apache proxy pattern for UI:
Explain how the SPA would behave if Apache already enforces login.
Clarify whether SPA still needs to obtain tokens for the API or can rely on session cookies / same-origin requests against an httpd-fronted API.
State management:

Where to store the MSAL instance and user account (React context or a global store).
Token storage & security:
Discuss risk of XSS and how that affects token storage choice.
Suggest using MSAL's recommended protected iframe/hidden refresh mechanisms rather than manually persisting tokens.
5. ALB, ECS, and Networking Details
Provide a precise design on the AWS side:

ALB Configuration
Listeners:
HTTPS : 443 with ACM certificate for *.example.com or specific hostnames like app.example.com.
Rules:
Host-based routing:
Host: app.example.com → Target Group: React UI ECS service (or httpd+SPA).
Host: api.example.com → Target Group: FastAPI ECS service (or httpd+FastAPI).
Or path-based if using a single domain:
/ → UI Target Group.
/api/* → API Target Group.
Target groups:
Type: IP or instance as per Fargate best practices.
Health checks configured on appropriate endpoints (e.g., /healthz on FastAPI or Apache).
ECS Fargate Services
Network mode: awsvpc.
Tasks run in private subnets only.
ALB is associated with the ECS services via service -> targetGroupArn.
Security groups:
ALB SG:
Inbound: HTTPS (443) from internet.
Outbound: to ECS tasks' security group.
ECS Tasks SG:
Inbound: from ALB SG on the container port (e.g., 80/8080).
Outbound: to the internet via NAT Gateway (for talking to Entra ID, fetching JWKS, etc.).
If using multi-container tasks for httpd + FastAPI or httpd + SPA:
Describe the task definition with multiple containers on the same ENI, how they communicate (localhost/ports), and which container is the one registered in the target group (httpd).
DNS / Route 53 / Certificates
Route 53 Hosted Zone: example.com.
Records:

app.example.com → ALIAS to ALB DNS.
api.example.com → ALIAS to ALB DNS (if using host-based routing).
ACM:
Request a certificate for app.example.com and api.example.com (or *.example.com) in the same region as the ALB.
Ensure all traffic is HTTPS with HTTP→HTTPS redirection (optional but recommended).
6. API Access from Postman and Browser
Explain in detail how:

Postman Access:
Developers configure an OAuth 2.0 client in Postman:
Authorization URL: https://login.microsoftonline.com//oauth2/v2.0/authorize
Token URL: https://login.microsoftonline.com//oauth2/v2.0/token
Client ID: SPA client or a dedicated "Postman client" app (explain which is better and why).
Scopes: api:///user.read (and others as needed).
Obtain an access token and call:
https://api.example.com/endpoint with Authorization: Bearer .
Discuss any differences if API is fronted by httpd:
How mod_auth_openidc can validate Bearer tokens.
Any headers or configuration changes needed.
Describe typical errors (401, 403) and how to debug (invalid audience, missing scope, wrong tenant, etc.).
Direct Browser Access:
For endpoints intended for human users, describe:
Option of redirecting unauthenticated users to SPA (preferred UX).
Or presenting raw JSON for authenticated tokens (if user already logs in via SPA & sends Bearer tokens).
Clarify how this fits with the chosen architecture and with/without the Apache proxy.
7. Security, Latency, and Operational Considerations
Security:
Key management: tokens are signed by Entra; no custom key mgmt is needed, but FastAPI or httpd must regularly fetch JWKS and cache them.
TLS at ALB; traffic in VPC can be HTTP or mTLS (discuss pros/cons).
CSRF, XSS considerations:
For SPA, emphasize protecting against XSS so tokens aren't exfiltrated.
For Apache session cookies, discuss SameSite, HttpOnly, Secure flags.
Suggest hardening (Content Security Policy, secure React coding

practices).
Logging and monitoring:
Log authentication failures with enough detail (but not tokens).
Use CloudWatch Logs & Metrics; optionally integrate with SIEM.
Latency:
Entra endpoints are internet-accessible; FastAPI/Apache and SPA will talk to them via NAT.
Explain impact and how token caching (and JWKs caching) reduce latency.
Compare latency impacts of:
Direct app-level token validation.
ALB OIDC.
Apache proxy with mod_auth_openidc.
Operations:
Token lifetime and refresh:
Describe default lifetime and how refresh happens in SPA via MSAL, and/or in Apache via mod_auth_openidc.
Rotating Entra settings:
Impact of rotating client secrets (for confidential clients like httpd proxy; SPA is a public client and has no secret).
Rolling deployments and blue/green:
Ensure tokens validated by new versions of API/proxy remain valid.
8. Final Output Expectations
Your final answer should:

Provide a structured, step-by-step solution design with headings and subheadings.
Clearly list:
Multiple design options (including the Apache httpd proxy option).
A recommended best option (with rationale).
Include enough concrete detail (URLs, claims, typical configuration values, Apache directives, etc.) to be practically implementable.
Focus heavily on:
AWS ECS Fargate + ALB network and security setup.
Deep integration with Microsoft Entra ID for both authentication and authorization.
UI (React SPA) → API (FastAPI) integration, including token flows.
How an httpd proxy option would work and how it compares.
Ensuring:
Secure connections.
Low latency.
No use of AWS API Gateway or Cognito.

Claude Sonnet 4.5

End-to-End AWS ECS Fargate + Microsoft Entra ID Authentication & Authorization Solution

Executive Summary

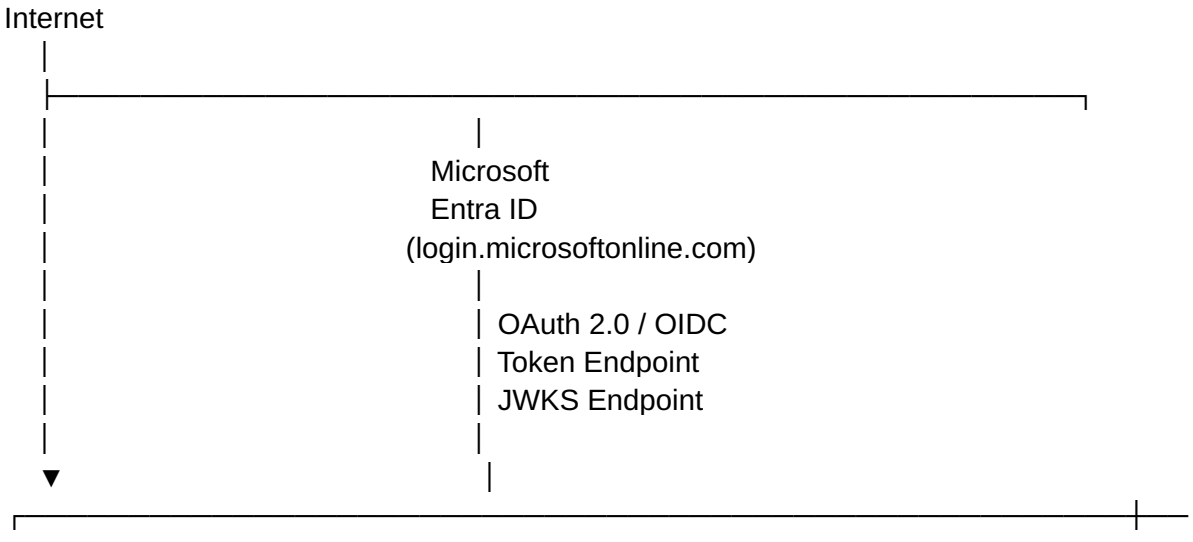
This document provides a comprehensive solution design for securing a React SPA and FastAPI backend running on AWS ECS Fargate, using Microsoft Entra ID (Azure AD) as the sole identity provider. The solution addresses authentication, authorization, API access from multiple clients (browser, SPA, Postman), and presents five distinct architectural options with detailed trade-off analysis.

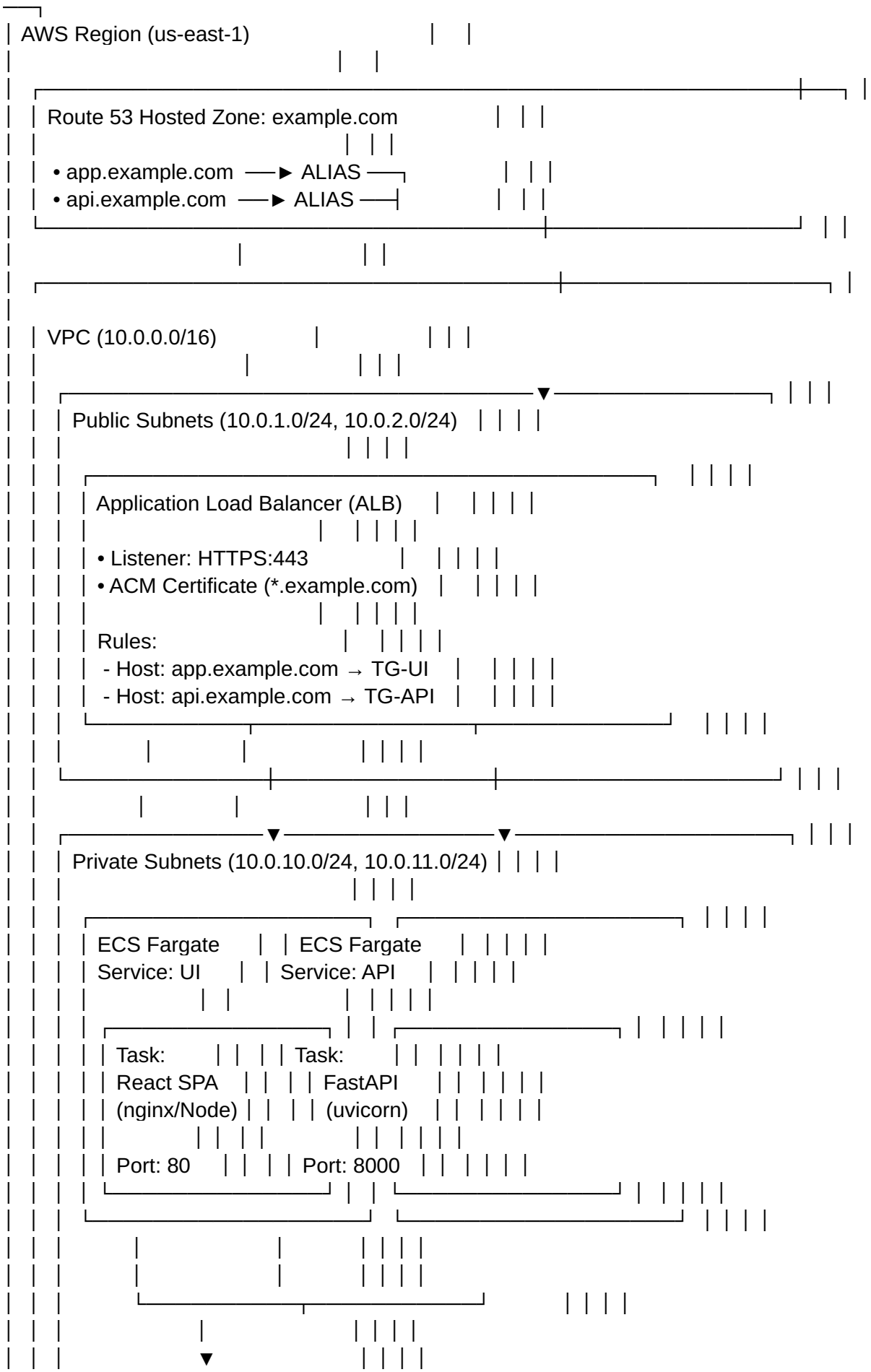
Recommended
Approach: Option D - App-Level Authentication with Direct Entra ID Integration (Auth Code + PKCE for SPA, JWT Validation in FastAPI)

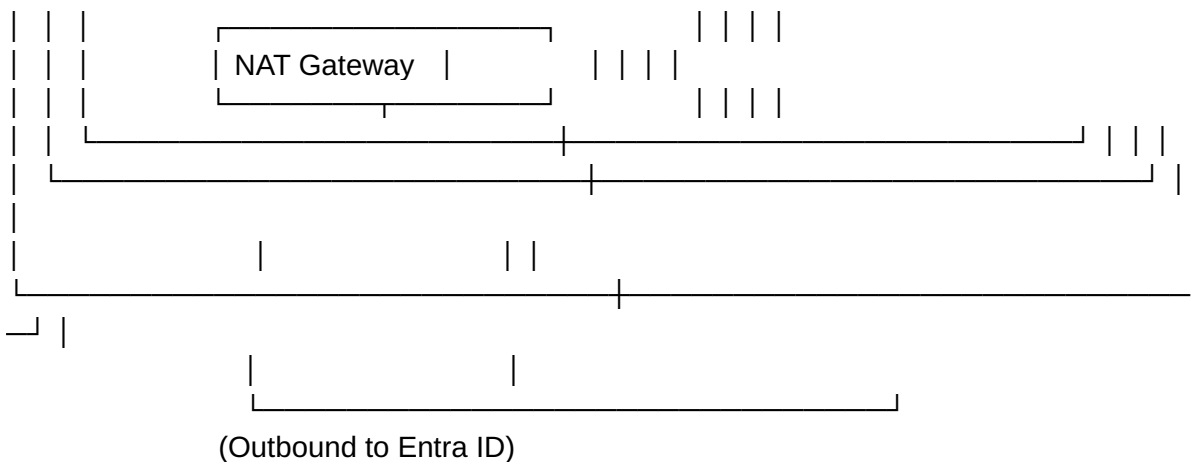
---

1. High-Level Architecture Diagram Description

Components & Layout




AWS Region (us-east-1)
Route 53 Hosted Zone: example.com
• app.example.com ──► ALIAS
• api.example.com ──► ALIAS
VPC (10.0.0.0/16)
Public Subnets (10.0.1.0/24, 10.0.2.0/24)
Application Load Balancer (ALB)
• Listener: HTTPS:443
• ACM Certificate (*.example.com)
Rules:
- Host: app.example.com → TG-UI
- Host: api.example.com → TG-API
Private Subnets (10.0.10.0/24, 10.0.11.0/24)
ECS Fargate
Service: UI
Task:
React SPA
(nginx/Node)
Port: 80
ECS Fargate
Service: API
Task:
FastAPI
(uvicorn)
Port: 8000



Token Flows

Flow 1: User Accesses React SPA

1. User navigates to https://app.example.com
2. ALB routes to React SPA ECS service
3. SPA loads in browser, MSAL.js detects no authentication
4. MSAL redirects to Entra ID: https://login.microsoftonline.com/<tenant>/oauth2/v2.0/authorize
5. User authenticates with Entra ID
6. Entra redirects back to https://app.example.com/auth/callback with authorization code
7. MSAL exchanges code for tokens (ID token + access token for API)
8. SPA stores tokens in memory via MSAL
9. User is authenticated

Flow 2: SPA Calls FastAPI Backend

1. SPA needs to call API endpoint
2. MSAL acquireTokenSilent() retrieves access token for API scope
3. SPA makes request: GET https://api.example.com/users with Authorization: Bearer <access_token>
4. ALB routes to FastAPI ECS service
5. FastAPI validates JWT:
    - Fetches JWKS from Entra ID (cached)
    - Validates signature, issuer, audience, expiration
    - Extracts claims (roles, scopes, user identity)
6. FastAPI processes request and returns response

Flow 3: Postman/Direct API Access

1. Developer obtains access token from Entra ID (OAuth 2.0 flow in Postman)
2. Postman sends: GET https://api.example.com/users with Authorization: Bearer <access_token>
3. Same validation flow as Flow 2

2. Design Options Analysis

Option A: App-Level Auth with SPA & API Both Integrated Directly with Entra ID

Architecture

Frontend (React SPA):

- Uses MSAL.js 2.x or 3.x library
- Implements OAuth 2.0 Authorization Code Flow with PKCE
- Registered as a public client (SPA) in Entra ID
- Obtains:
    - ID token for user identity display
    - Access token for calling FastAPI (with specific API scopes)

Backend (FastAPI):

- Separate Entra ID app registration (Web/API type)
- Exposes API with Application ID URI: api://<api-app-id>
- Defines scopes: user.read, user.write, admin
- Validates JWT on every request using Python libraries

Detailed Token Flow

Step-by-Step SPA Authentication:

1. **Initial Page Load:**
   ```javascript
   // MSAL Configuration
   const msalConfig = {
   auth: {
   clientId: '',
   authority: 'https://login.microsoftonline.com/',
   redirectUri: 'https://app.example.com/auth/callback',
   postLogoutRedirectUri: 'https://app.example.com'
   },
   cache: {
   cacheLocation: 'sessionStorage', // or 'localStorage' or 'memory'
   ```

```
    storeAuthStateInCookie: false
    }
    };
const msalInstance = new PublicClientApplication(msalConfig);
```

1. **Login Initiation:**
   ```javascript
   const loginRequest = {
   scopes: ['openid', 'profile', 'email', 'api:///user.read']
   };
// Redirect to Entra ID
await msalInstance.loginRedirect(loginRequest);
```

1. **Handle Redirect Callback:**
   javascript
   ```
   msalInstance.handleRedirectPromise()
     .then(response => {
       if (response) {
         // User is authenticated
         const account = response.account;
         // ID token available in response.idToken
       }
     });
   ```
2. **Acquire Token for API Call:**
   ```javascript
   const tokenRequest = {
   scopes: ['api:///user.read'],
   account: msalInstance.getAllAccounts()[0]
   };
try {
const response = await msalInstance.acquireTokenSilent(tokenRequest);
const accessToken = response.accessToken;


 // Call API
 fetch('https://api.example.com/users', {
   headers: {
     'Authorization': `Bearer ${accessToken}`
   }
 });


} catch (error) {
// Silent acquisition failed, use interactive method
await msalInstance.acquireTokenRedirect(tokenRequest);
}
```

FastAPI Token Validation:

```
from fastapi import FastAPI, Depends, HTTPException, Security
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
import requests
from functools import lru_cache
from typing import Dict, List
import time

app = FastAPI()
security = HTTPBearer()

# Configuration
TENANT_ID = "<tenant-id>"
CLIENT_ID = "<api-app-id>"

# API's client ID
ISSUER = f"https://login.microsoftonline.com/{TENANT_ID}/v2.0"
JWKS_URI = f"https://login.microsoftonline.com/{TENANT_ID}/discovery/v2.0/keys"
AUDIENCE = f"api://{CLIENT_ID}"

# Cache JWKS for 24 hours
@lru_cache(maxsize=1)
def get_jwks(cache_time: int):
    response = requests.get(JWKS_URI)
    return response.json()

def get_current_jwks():
    # Refresh cache every 24 hours
    cache_key = int(time.time() / 86400)
    return get_jwks(cache_key)

def validate_token(credentials: HTTPAuthorizationCredentials = Security(security)) -> Dict:
    token = credentials.credentials

    try:
        # Get signing keys
        jwks = get_current_jwks()

        # Decode header to get kid
        unverified_header = jwt.get_unverified_header(token)

        # Find the right key
        rsa_key = {}
        for key in jwks["keys"]:
            if key["kid"] == unverified_header["kid"]:
                rsa_key = {
                    "kty": key["kty"],
                    "kid": key["kid"],
                    "use": key["use"],
                    "n": key["n"],
```

```
                "e": key["e"]
            }
            break

    if not rsa_key:
        raise HTTPException(status_code=401, detail="Unable to find appropriate key")

    # Validate token
    payload = jwt.decode(
        token,
        rsa_key,
        algorithms=["RS256"],
        audience=AUDIENCE,
        issuer=ISSUER,
        options={
            "verify_signature": True,
            "verify_exp": True,
            "verify_nbf": True,
            "verify_iat": True,
            "verify_aud": True,
            "verify_iss": True
        }
    )

    return payload

  except JWTError as e:
    raise HTTPException(status_code=401, detail=f"Token validation failed: {str(e)}")

def require_scope(required_scope: str):
  def scope_checker(token_payload: Dict = Depends(validate_token)):
    scopes = token_payload.get("scp", "").split() or token_payload.get("roles", [])
    if required_scope not in scopes:
      raise HTTPException(status_code=403, detail=f"Missing required scope:
{required_scope}")
    return token_payload
  return scope_checker

def require_role(required_role: str):
  def role_checker(token_payload: Dict = Depends(validate_token)):
    roles = token_payload.get("roles", [])
    if required_role not in roles:
      raise HTTPException(status_code=403, detail=f"Missing required role:
{required_role}")
    return token_payload
  return role_checker

# Example endpoints
@app.get("/users")
async def get_users(token_payload: Dict = Depends(require_scope("user.read"))):
```

```
    user_id = token_payload.get("oid")

# Object ID
    user_email = token_payload.get("preferred_username")
    return {
        "message": "Users retrieved successfully",
        "user": {
            "id": user_id,
            "email": user_email
        }
    }

@app.post("/admin/settings")
async def update_settings(token_payload: Dict = Depends(require_role("Admin"))):
    return {"message": "Settings updated"}

@app.get("/health")
async def health():
    return {"status": "healthy"}
```

Token Storage Options

Option A1: In-Memory Storage (Recommended)

- **Implementation:** MSAL stores tokens in JavaScript memory only
- **Pros:**
    - Most secure against XSS attacks
    - Tokens never persisted to disk
    - Automatic cleanup on page refresh
- **Cons:**
    - Tokens lost on page refresh (requires silent token renewal)
    - Slightly more complex implementation
- **Best for:** High-security applications

Option A2: sessionStorage

- **Implementation:** cacheLocation: 'sessionStorage' in MSAL config
- **Pros:**
    - Survives page refreshes within same session
    - Cleared when tab/browser closes
    - Better UX than in-memory
- **Cons:**
    - Vulnerable to XSS attacks
    - Accessible via JavaScript
- **Best for:** Balanced security/UX

Option A3: localStorage

- **Implementation:** cacheLocation: 'localStorage' in MSAL config
- **Pros:**
    - Persists across browser sessions
    - Best UX (user stays logged in)
- **Cons:**
    - Most vulnerable to XSS
    - Tokens persist indefinitely until cleared
- **Best for:** Low-risk internal applications

Recommendation: Use **sessionStorage** with proper XSS protections (Content Security Policy, input sanitization, React's built-in XSS protection).

Postman Configuration

Step 1: Configure OAuth 2.0 in Postman

Authorization Type: OAuth 2.0
Grant Type: Authorization Code (with PKCE)

Auth URL: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize
Access Token URL: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
Client ID: <spa-client-id> (or dedicated Postman client ID)
Client Secret: (leave empty for public client)
Scope: api://<api-app-id>/user.read openid profile
State: <random-string>
Code Challenge Method: SHA-256

Step 2: Get New Access Token

- Click "Get New Access Token"
- Browser opens for Entra ID login
- After authentication, token is returned to Postman
- Use token in requests to https://api.example.com/*

Alternative: Device Code Flow (Better for CLI/Postman)

```
# Step 1: Initiate device code flow
POST https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/devicecode
Content-Type: application/x-www-form-urlencoded

client_id=<spa-client-id>&
scope=api://<api-app-id>/user.read
```

```
# Response includes user_code and verification_uri
# User visits verification_uri and enters user_code

# Step 2: Poll for token
POST https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:device_code&
client_id=<spa-client-id>&
device_code=<device_code_from_step1>
```

Pros & Cons

Pros:

- **Clean separation of concerns:** SPA handles UI auth, API handles validation
- **Standard OAuth 2.0/OIDC flows:** Well-documented, widely supported
- **No additional infrastructure:** No proxy layers, direct integration
- **Low latency:** Direct token validation, JWKS caching minimizes external calls
- **Flexible token storage:** Choose security vs. UX trade-off
- **Easy Postman integration:** Same token flow works for all clients
- **Granular authorization:** Scopes and roles enforced at API level
- **Stateless API:** No session management required

Cons:

- **Token exposure in browser:** Access tokens visible in browser memory/ storage
- **XSS vulnerability:** If XSS exists, tokens can be stolen
- **Client-side complexity:** SPA must handle token acquisition, renewal, errors
- **CORS considerations:** API must handle CORS for SPA calls
- **Token refresh UX:** Silent renewal can fail, requiring user interaction

Complexity: Medium
Security: High (with proper XSS protections)
Latency: Low (direct calls, cached JWKS)
Developer Experience: Good (standard libraries, clear patterns)

Option B: ALB-Level OIDC Authentication + Token Forwarding

Architecture

ALB Configuration:

- ALB listener rules include authenticate-oidc action
- ALB acts as OIDC client (confidential client with Entra ID)
- ALB handles OAuth 2.0 Authorization Code Flow
- After authentication, ALB forwards requests with special headers

Detailed Configuration

ALB Listener Rule for UI:

```
{
  "Type": "authenticate-oidc",
  "Order": 1,
  "AuthenticateOidcConfig": {
    "Issuer": "https://login.microsoftonline.com/<tenant-id>/v2.0",
    "AuthorizationEndpoint": "https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/
authorize",
    "TokenEndpoint": "https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token",
    "UserInfoEndpoint": "https://graph.microsoft.com/oidc/userinfo",
    "ClientId": "<alb-client-id>",
    "ClientSecret": "<alb-client-secret>",
    "Scope": "openid profile email",
    "SessionCookieName": "AWSELBAuthSessionCookie",
    "SessionTimeout": 3600,
    "OnUnauthenticatedRequest": "authenticate"
  }
},
{
  "Type": "forward",
  "Order": 2,
  "TargetGroupArn": "arn:aws:elasticloadbalancing:..."
}
```

Headers Forwarded by ALB:

- x-amzn-oidc-accesstoken: Access token (if available)
- x-amzn-oidc-identity: User identity
- x-amzn-oidc-data: JWT with user claims (signed by ALB)

Token Flow

1. User navigates to https://app.example.com
2. ALB detects no session cookie
3. ALB redirects to Entra ID authorization endpoint
4. User authenticates with Entra ID
5. Entra redirects back to ALB with authorization code
6. ALB exchanges code for tokens
7. ALB sets session cookie (AWSELBAuthSessionCookie)
8. ALB forwards request to ECS with headers
9. React SPA loads (already authenticated from ALB perspective)

Challenges & Issues

Challenge 1: SPA Token Access

- ALB doesn't expose tokens to the SPA directly
- SPA cannot call API with Bearer tokens (doesn't have access token)
- **Workaround:**
    - API also fronted by same ALB with OIDC
    - SPA makes same-origin requests (cookies automatically sent)
    - Not true RESTful API pattern

Challenge 2: Postman Access

- Postman cannot easily bypass ALB OIDC
- Would need to:
    - Obtain session cookie from ALB (complex)
    - Or use separate path without OIDC (security gap)
- **Workaround:** Create separate ALB listener rule for /api/v1/* without OIDC, requiring Bearer tokens (defeats purpose of ALB OIDC)

Challenge 3: API Validation

- FastAPI must validate x-amzn-oidc-data header
- This JWT is signed by ALB, not Entra ID
- Requires fetching ALB's public key from AWS
- Additional complexity

Challenge 4: SPA Architecture Mismatch

- ALB OIDC designed for server-side rendered apps
- React SPA is client-side rendered
- SPA doesn't benefit from ALB's token management
- SPA still needs MSAL for proper token handling if calling external APIs

Pros & Cons

Pros:

- **Centralized authentication:** ALB handles all OIDC flows
- **No app-level auth code:** Simpler app code (in theory)
- **Session management:** ALB manages sessions via cookies

Cons:

- **Poor fit for SPA architecture:** Designed for server-side apps
- **Difficult Postman integration:** No clear path for API clients
- **Token inaccessibility:** SPA can't get access tokens for API calls
- **Complex validation:** Must validate ALB-signed JWTs, not Entra tokens
- **Limited flexibility:** Hard to customize auth flows
- **Cookie-based only:** Can't use Bearer tokens effectively
- **CORS complications:** Cookie-based auth with CORS is complex
- **Not RESTful:** API becomes session-dependent

Complexity: High (for this use case)
Security: Medium (relies on ALB session cookies)
Latency: Medium (ALB adds auth layer)
Developer Experience: Poor (doesn't match requirements)

Recommendation: **Not suitable for this scenario** due to SPA architecture and Postman requirement.

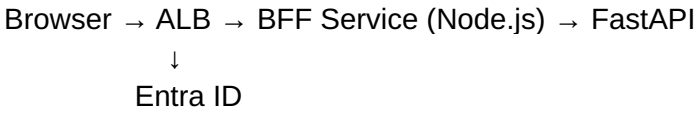
Option C: Backend-For-Frontend (BFF) Pattern

Architecture

New Component: BFF Service

- Node.js or Python backend serving the SPA
- Handles authentication server-side
- Stores session in secure HTTP-only cookies
- Proxies API calls using On-Behalf-Of (OBO) flow or service tokens

Modified Architecture:



Detailed Flow

Step 1: User Authentication

1. User navigates to https://app.example.com
2. BFF detects no session
3. BFF initiates OAuth 2.0 Authorization Code Flow (confidential client)
4. User redirects to Entra ID
5. After auth, Entra redirects to BFF with code
6. BFF exchanges code for tokens (access, refresh, ID)
7. BFF stores tokens server-side (Redis, DynamoDB, or in-memory)
8. BFF sets HTTP-only session cookie
9. BFF serves React SPA

Step 2: SPA → API Call

1. SPA makes request to BFF: GET /api/users (with session cookie)
2. BFF validates session
3. BFF retrieves access token from server-side storage
4. BFF calls FastAPI: GET https://api.example.com/users with Authorization: Bearer <token>
5. FastAPI validates token and responds
6. BFF forwards response to SPA

Step 3: Token Refresh

- BFF handles refresh token flow transparently
- No client-side token management

Implementation Considerations

BFF Service (Node.js Example):

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const OIDCStrategy = require('passport-azure-ad').OIDCStrategy;

const app = express();

// Session configuration
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: true,
    httpOnly: true,
    sameSite: 'lax',
```

```
    maxAge: 3600000
  }
}));

// Passport OIDC strategy
passport.use(new OIDCStrategy({
    identityMetadata: `https://login.microsoftonline.com/${TENANT_ID}/v2.0/.well-known/
openid-configuration`,
    clientID: BFF_CLIENT_ID,
    clientSecret: BFF_CLIENT_SECRET,
    responseType: 'code',
    responseMode: 'form_post',
    redirectUrl: 'https://app.example.com/auth/callback',
    scope: ['openid', 'profile', 'email', 'offline_access', `api://${API_CLIENT_ID}/user.read`]
  },
  (iss, sub, profile, accessToken, refreshToken, done) => {
    // Store tokens in session or database
    profile.accessToken = accessToken;
    profile.refreshToken = refreshToken;
    return done(null, profile);
  }
));

// Serve React SPA
app.use(express.static('build'));

// API proxy with token injection
app.get('/api/*', ensureAuthenticated, async (req, res) => {
  const accessToken = req.user.accessToken;

  const apiResponse = await fetch(`https://api.example.com${req.path}`, {
    headers: {
      'Authorization': `Bearer ${accessToken}`
    }
  });

  const data = await apiResponse.json();
  res.json(data);
});

function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  }
  res.redirect('/auth/login');
}
```

ECS Task Definition:

- Single task with BFF container (replaces simple SPA container)

- BFF serves static React assets and handles auth
- Increased container complexity

Pros & Cons

Pros:

- **Maximum security:** Tokens never exposed to browser
- **No XSS risk for tokens:** Stored server-side only
- **Simplified SPA:** No auth logic in frontend
- **Automatic token refresh:** Handled by BFF
- **HTTP-only cookies:** More secure than localStorage

Cons:

- **Architectural change:** Requires new BFF service
- **Increased complexity:** Additional service to maintain
- **Stateful backend:** Requires session storage (Redis/DynamoDB)
- **Postman access unclear:** Still needs separate token flow
- **Latency increase:** Additional hop (Browser → BFF → API)
- **Scaling considerations:** Session affinity or distributed session store
- **Diverges from current setup:** React SPA no longer standalone
- **CORS still needed:** If API accessed directly by other clients

Complexity: High (new service, session management)
Security: Very High (tokens never in browser)
Latency: Medium-High (additional proxy layer)
Developer Experience: Medium (simpler frontend, complex backend)

Recommendation: ⚠ Consider only if security requirements mandate no tokens in browser. Significant architectural change from current setup.

Option D: Hybrid - SPA Auth Code + PKCE; FastAPI Direct Validation; Shared API Registration (RECOMMENDED)

Architecture

This is a refined version of Option A with explicit configuration for all access patterns.

Key Principles:

- React SPA uses MSAL with Auth Code + PKCE (public client)

- FastAPI validates JWT directly (no proxy)
- Single API app registration shared by all clients
- Explicit support for SPA, Postman, and browser access

Entra ID App Registrations

Registration 1: SPA Application

- **Name:** MyApp-SPA
- **Type:** Single-page application
- **Client ID:** <spa-client-id>
- **Redirect URIs:**
    - https://app.example.com
    - https://app.example.com/auth/callback
    - http://localhost:3000 (for local development)
- **Platform:** SPA (enables PKCE, disables client secret)
- **Implicit grant:** Disabled (use Auth Code + PKCE)
- **API Permissions:**
    - MyApp-API → user.read (delegated)
    - MyApp-API → user.write (delegated)
    - Microsoft Graph → User.Read (delegated) - optional for profile info

Registration 2: API Application

- **Name:** MyApp-API
- **Type:** Web API
- **Client ID:** <api-client-id>
- **Application ID URI:** api://<api-client-id>
- **Exposed API:**
    - Scope: user.read
        - Display name: "Read user data"
        - Description: "Allows the app to read user data"
        - Admin consent: Not required
        - Enabled: Yes
    - Scope: user.write
        - Display name: "Write user data"
        - Description: "Allows the app to write user data"
        - Admin consent: Required
        - Enabled: Yes
    - Scope: admin
        - Display name: "Admin access"
        - Description: "Full admin access"
        - Admin consent: Required
        - Enabled: Yes

- **App Roles (for authorization):**
  json
  ```
  {
    "allowedMemberTypes": ["User"],
    "description": "Read-only access",
    "displayName": "Reader",
    "id": "<guid-1>",
    "isEnabled": true,
    "value": "Reader"
  },
  {
    "allowedMemberTypes": ["User"],
    "description": "Read and write access",
    "displayName": "Writer",
    "id": "<guid-2>",
    "isEnabled": true,
    "value": "Writer"
  },
  {
    "allowedMemberTypes": ["User"],
    "description": "Full administrative access",
    "displayName": "Admin",
    "id": "<guid-3>",
    "isEnabled": true,
    "value": "Admin"
  }
  ```
- **Authorized client applications:**
  - Add <spa-client-id> to pre-authorize SPA for all scopes

Optional Registration 3: Postman/Developer Client

- **Name:** MyApp-Postman
- **Type:** Public client / Mobile and desktop applications
- **Client ID:** <postman-client-id>
- **Redirect URIs:**
  - https://oauth.pstmn.io/v1/callback (Postman)
  - http://localhost (for device code flow)
- **API Permissions:** Same as SPA
- **Why separate?** Allows tracking/auditing of Postman usage separately

Detailed Implementation

React SPA (Complete Example):

```
// src/authConfig.js
export const msalConfig = {
  auth: {
    clientId: '<spa-client-id>',
```

```
    authority: 'https://login.microsoftonline.com/<tenant-id>',
    redirectUri: 'https://app.example.com/auth/callback',
    postLogoutRedirectUri: 'https://app.example.com',
    navigateToLoginRequestUrl: true
  },
  cache: {
    cacheLocation: 'sessionStorage',
    storeAuthStateInCookie: false
  },
  system: {
    loggerOptions: {
      loggerCallback: (level, message, containsPii) => {
        if (containsPii) return;
        console.log(message);
      },
      logLevel: 'Info'
    }
  }
};

export const loginRequest = {
  scopes: ['openid', 'profile', 'email']
};

export const apiRequest = {
  scopes: ['api://<api-client-id>/user.read']
};

// src/App.js
import { MsalProvider, useMsal, useIsAuthenticated } from '@azure/msal-react';
import { PublicClientApplication } from '@azure/msal-browser';
import { msalConfig } from './authConfig';

const msalInstance = new PublicClientApplication(msalConfig);

function App() {
  return (
    <MsalProvider instance={msalInstance}>
      <MainApp />
    </MsalProvider>
  );
}

function MainApp() {
  const { instance, accounts } = useMsal();
  const isAuthenticated = useIsAuthenticated();

  const handleLogin = () => {
    instance.loginRedirect(loginRequest);
  };
```

```
  const handleLogout = () => {
    instance.logoutRedirect();
  };

  if (!isAuthenticated) {
    return (
      <div>
        <h1>Welcome to MyApp</h1>
        <button onClick={handleLogin}>Sign In</button>
      </div>
    );
  }

  return (
    <div>
      <h1>Welcome, {accounts[0].name}</h1>
      <button onClick={handleLogout}>Sign Out</button>
      <UserList />
    </div>
  );
}

// src/components/UserList.js
import { useMsal } from '@azure/msal-react';
import { apiRequest } from '../authConfig';
import { useState, useEffect } from 'react';

function UserList() {
  const { instance, accounts } = useMsal();
  const [users, setUsers] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetchUsers();
  }, []);

  const fetchUsers = async () => {
    try {
      // Acquire token silently
      const response = await instance.acquireTokenSilent({
        ...apiRequest,
        account: accounts[0]
      });

      // Call API
      const apiResponse = await fetch('https://api.example.com/users', {
        headers: {
          'Authorization': `Bearer ${response.accessToken}`,
          'Content-Type': 'application/json'
```

```
        }
      });

      if (!apiResponse.ok) {
        throw new Error(`API error: ${apiResponse.status}`);
      }

      const data = await apiResponse.json();
      setUsers(data);
    } catch (err) {
      if (err.name === 'InteractionRequiredAuthError') {
        // Silent token acquisition failed, use interactive method
        instance.acquireTokenRedirect(apiRequest);
      } else {
        setError(err.message);
      }
    }
  };

  if (error) {
    return <div>Error: {error}</div>;
  }

  return (
    <div>
      <h2>Users</h2>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

FastAPI (Complete Example with Enhanced Authorization):

```
# app/auth.py
from fastapi import Depends, HTTPException, Security
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from jose import jwt, JWTError
import requests
from functools import lru_cache, wraps
from typing import Dict, List, Optional, Callable
import time
from pydantic import BaseModel

security = HTTPBearer()
```

```
# Configuration
TENANT_ID = "<tenant-id>"
API_CLIENT_ID = "<api-client-id>"
ISSUER = f"https://login.microsoftonline.com/{TENANT_ID}/v2.0"
JWKS_URI = f"https://login.microsoftonline.com/{TENANT_ID}/discovery/v2.0/keys"
AUDIENCE = f"api://{API_CLIENT_ID}"

class TokenPayload(BaseModel):
    oid: str

# Object ID (user ID)
    preferred_username: Optional[str]
    name: Optional[str]
    scp: Optional[str]

# Scopes (space-separated)
    roles: Optional[List[str]]

# App roles
    groups: Optional[List[str]]

# Group IDs
    iss: str
    aud: str
    exp: int
    nbf: int
    iat: int

# Cache JWKS for 24 hours
@lru_cache(maxsize=1)
def get_jwks(cache_time: int):
    """Fetch and cache JWKS from Entra ID"""
    response = requests.get(JWKS_URI, timeout=10)
    response.raise_for_status()
    return response.json()

def get_current_jwks():
    """Get JWKS with daily cache refresh"""
    cache_key = int(time.time() / 86400)
    return get_jwks(cache_key)

def validate_token(credentials: HTTPAuthorizationCredentials = Security(security)) ->
TokenPayload:
    """
    Validate JWT access token from Entra ID
    Returns token payload with user claims
    """
    token = credentials.credentials
```

```
try:
    # Get signing keys
    jwks = get_current_jwks()

    # Decode header to get kid (key ID)
    unverified_header = jwt.get_unverified_header(token)

    # Find the matching key
    rsa_key = {}
    for key in jwks["keys"]:
        if key["kid"] == unverified_header["kid"]:
            rsa_key = {
                "kty": key["kty"],
                "kid": key["kid"],
                "use": key["use"],
                "n": key["n"],
                "e": key["e"]
            }
            break

    if not rsa_key:
        raise HTTPException(
            status_code=401,
            detail="Unable to find appropriate signing key",
            headers={"WWW-Authenticate": "Bearer"}
        )

    # Validate and decode token
    payload = jwt.decode(
        token,
        rsa_key,
        algorithms=["RS256"],
        audience=AUDIENCE,
        issuer=ISSUER,
        options={
            "verify_signature": True,
            "verify_exp": True,
            "verify_nbf": True,
            "verify_iat": True,
            "verify_aud": True,
            "verify_iss": True
        }
    )

    return TokenPayload(**payload)

except JWTError as e:
    raise HTTPException(
        status_code=401,
        detail=f"Token validation failed: {str(e)}",
```

```
            headers={"WWW-Authenticate": "Bearer"}
        )
    except Exception as e:
        raise HTTPException(
            status_code=401,
            detail=f"Authentication error: {str(e)}",
            headers={"WWW-Authenticate": "Bearer"}
        )

def require_scope(*required_scopes: str):
    """
    Dependency to require specific OAuth scopes
    Usage: @app.get("/endpoint", dependencies=[Depends(require_scope("user.read"))])
    """
    def scope_checker(token_payload: TokenPayload = Depends(validate_token)):
        # Get scopes from token (space-separated string)
        token_scopes = token_payload.scp.split() if token_payload.scp else []

        # Check if any required scope is present
        if not any(scope in token_scopes for scope in required_scopes):
            raise HTTPException(
                status_code=403,
                detail=f"Missing required scope. Required: {required_scopes}, Present:
{token_scopes}"
            )

        return token_payload

    return scope_checker

def require_role(*required_roles: str):
    """
    Dependency to require specific app roles
    Usage: @app.get("/endpoint", dependencies=[Depends(require_role("Admin"))])
    """
    def role_checker(token_payload: TokenPayload = Depends(validate_token)):
        token_roles = token_payload.roles or []

        if not any(role in token_roles for role in required_roles):
            raise HTTPException(
                status_code=403,
                detail=f"Missing required role. Required: {required_roles}, Present: {token_roles}"
            )

        return token_payload

    return role_checker

def require_group(*required_groups: str):
    """
```

```
    Dependency to require membership in specific Entra ID groups
    Usage: @app.get("/endpoint", dependencies=[Depends(require_group("<group-id>"))])
    """
    def group_checker(token_payload: TokenPayload = Depends(validate_token)):
        token_groups = token_payload.groups or []

        if not any(group in token_groups for group in required_groups):
            raise HTTPException(
                status_code=403,
                detail=f"Missing required group membership"
            )

        return token_payload

    return group_checker

def get_current_user(token_payload: TokenPayload = Depends(validate_token)) -> Dict:
    """
    Dependency to get current authenticated user info
    Usage: user = Depends(get_current_user)
    """
    return {
        "id": token_payload.oid,
        "email": token_payload.preferred_username,
        "name": token_payload.name,
        "scopes": token_payload.scp.split() if token_payload.scp else [],
        "roles": token_payload.roles or [],
        "groups": token_payload.groups or []
    }

# app/main.py
from fastapi import FastAPI, Depends
from fastapi.middleware.cors import CORSMiddleware
from app.auth import (
    validate_token,
    require_scope,
    require_role,
    get_current_user,
    TokenPayload
)
from typing import Dict, List

app = FastAPI(title="MyApp API")

# CORS configuration for SPA
app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        "https://app.example.com",
        "http://localhost:3000"
```

```
# For local development
    ],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
    expose_headers=["*"]
)

@app.get("/health")
async def health():
    """Public health check endpoint"""
    return {"status": "healthy"}

@app.get("/users")
async def get_users(
    current_user: Dict = Depends(get_current_user),
    token: TokenPayload = Depends(require_scope("user.read"))
):
    """
    Get list of users
    Requires: user.read scope
    """
    return {
        "users": [
            {"id": "1", "name": "Alice"},
            {"id": "2", "name": "Bob"}
        ],
        "requested_by": current_user
    }

@app.post("/users")
async def create_user(
    user_data: Dict,
    current_user: Dict = Depends(get_current_user),
    token: TokenPayload = Depends(require_scope("user.write"))
):
    """
    Create a new user
    Requires: user.write scope
    """
    return {
        "message": "User created",
        "data": user_data,
        "created_by": current_user
    }

@app.get("/admin/settings")
async def get_admin_settings(
    current_user: Dict = Depends(get_current_user),
```

```
    token: TokenPayload = Depends(require_role("Admin"))
):
    """
    Get admin settings
    Requires: Admin role
    """
    return {
        "settings": {"feature_x": True},
        "accessed_by": current_user
    }

@app.delete("/admin/users/{user_id}")
async def delete_user(
    user_id: str,
    current_user: Dict = Depends(get_current_user),
    token: TokenPayload = Depends(require_role("Admin", "Writer"))
):
    """
    Delete a user
    Requires: Admin OR Writer role
    """
    return {
        "message": f"User {user_id} deleted",
        "deleted_by": current_user
    }

@app.get("/me")
async def get_current_user_info(current_user: Dict = Depends(get_current_user)):
    """
    Get current authenticated user information
    Requires: Valid authentication (any scope)
    """
    return current_user

# Error handlers
@app.exception_handler(401)
async def unauthorized_handler(request, exc):
    return {
        "error": "unauthorized",
        "message": "Invalid or missing authentication token",
        "detail": str(exc.detail) if hasattr(exc, 'detail') else None
    }

@app.exception_handler(403)
async def forbidden_handler(request, exc):
    return {
        "error": "forbidden",
        "message": "Insufficient permissions",
        "detail": str(exc.detail) if hasattr(exc, 'detail') else None
    }
```

Postman Configuration (Detailed):

Method 1: Using SPA Client ID (Recommended for Testing)

1. **Create New Request**
2. **Authorization Tab:**
   - Type: OAuth 2.0
   - Add auth data to: Request Headers
3. **Configure New Token:**
   - Token Name: MyApp API Token
   - Grant Type: Authorization Code (With PKCE)
   - Callback URL: https://oauth.pstmn.io/v1/callback (check "Authorize using browser")
   - Auth URL: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/authorize
   - Access Token URL: https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
   - Client ID: <spa-client-id>
   - Client Secret: (leave empty)
   - Scope: api://<api-client-id>/user.read openid profile
   - State: 12345
   - Client Authentication: Send as Basic Auth header
4. **Get New Access Token**
   - Browser opens for login
   - Authenticate with Entra ID
   - Token returned to Postman
5. **Use Token:**
   - Request URL: https://api.example.com/users
   - Token automatically added as Authorization: Bearer <token>

Method 2: Using Dedicated Postman Client (Better for Production)

Same as above, but use <postman-client-id> instead of SPA client ID. This allows:

- Separate tracking of Postman usage
- Different token lifetimes
- Ability to revoke Postman access without affecting SPA

Method 3: Device Code Flow (Best for CLI/Automation)

```
# Step 1: Get device code
```

```
curl -X POST \
  https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/devicecode \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d 'client_id=<spa-client-id>&scope=api://<api-client-id>/user.read'

# Response:
# {
#   "user_code": "ABCD1234",
#   "device_code": "...",
#   "verification_uri": "https://microsoft.com/devicelogin",
#   "expires_in": 900,
#   "interval": 5
# }

# Step 2: User visits verification_uri and enters user_code

# Step 3: Poll for token (in Postman Pre-request Script or manually)
curl -X POST \
  https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d 'grant_type=urn:ietf:params:oauth:grant-type:device_code&client_id=<spa-client-
id>&device_code=<device_code>'

# Use returned access_token in Authorization header
```

Browser Direct Access

Scenario 1: Authenticated User Calling API Directly

- User already logged into SPA
- Opens new tab: https://api.example.com/users
- **Result:** 401 Unauthorized (no Bearer token in request)
- **Solution:** Not typical use case; redirect to SPA or show API documentation

Scenario 2: API Documentation/Swagger UI

```
# Add Swagger UI with OAuth2 support
from fastapi.openapi.utils import get_openapi

def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema

    openapi_schema = get_openapi(
        title="MyApp API",
        version="1.0.0",
```

```
        description="API with Entra ID authentication",
        routes=app.routes,
    )

    openapi_schema["components"]["securitySchemes"] = {
        "OAuth2": {
            "type": "oauth2",
            "flows": {
                "authorizationCode": {
                    "authorizationUrl": f"https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.0/
authorize",
                    "tokenUrl": f"https://login.microsoftonline.com/{TENANT_ID}/oauth2/v2.0/token",
                    "scopes": {
                        f"api://{API_CLIENT_ID}/user.read": "Read user data",
                        f"api://{API_CLIENT_ID}/user.write": "Write user data"
                    }
                }
            }
        }
    }

    app.openapi_schema = openapi_schema
    return app.openapi_schema

app.openapi = custom_openapi

# Access Swagger UI at https://api.example.com/docs
# Users can authenticate via OAuth2 and test endpoints
```

Pros & Cons

Pros:

- **Complete solution:** Addresses all requirements (SPA, Postman, browser)
- **Standard OAuth 2.0:** Well-documented, industry best practice
- **Clean architecture:** Clear separation of concerns
- **Low latency:** Direct validation, minimal hops
- **Flexible authorization:** Scopes, roles, groups all supported
- **Easy testing:** Postman integration straightforward
- **Stateless API:** No session management required
- **Scalable:** No shared state between API instances
- **Developer-friendly:** Clear patterns, good libraries (MSAL, FastAPI)
- **Production-ready:** Battle-tested approach used by major applications

Cons:

- **Token in browser:** Access tokens stored in sessionStorage (XSS risk)
- **Client complexity:** SPA must handle token acquisition and renewal

- **CORS required:** API must handle CORS for SPA calls
- **Multiple registrations:** Need to manage SPA and API app registrations

Complexity: Medium
Security: High (with proper XSS protections)
Latency: Low (direct calls, cached JWKS)
Developer Experience: Excellent (standard patterns, good tooling)

Recommendation: BEST
OPTION - Meets all requirements with industry-standard approach.

Option E: Apache httpd (mod_auth_openidc) as Auth Proxy

Architecture

Key Concept:

- Apache httpd container runs in front of both UI and API services
- Uses mod_auth_openidc for OIDC authentication with Entra ID
- Two deployment patterns:
    - **Pattern E1:** Separate httpd for UI and API (two ECS services)
    - **Pattern E2:** Multi-container tasks (httpd + app in same task)

Pattern E1: Separate httpd Services

ECS Architecture:

```
ALB
 ├─ app.example.com → Target Group: UI-Proxy Service
 │                    └─ httpd (mod_auth_openidc) → serves React static files
 │
 └─ api.example.com → Target Group: API-Proxy Service
                      └─ httpd (mod_auth_openidc) → proxies to FastAPI Service
                                                    └─ FastAPI (internal)
```

Three ECS Services:

1. **UI-Proxy Service:** httpd serving React SPA with OIDC auth
2. **API-Proxy Service:** httpd validating tokens and proxying to FastAPI
3. **FastAPI Service:** Internal service (no ALB, accessed via service discovery)

Pattern E2: Multi-Container Tasks

ECS Architecture:

```
ALB
 ├─ app.example.com → Target Group: UI Service
 │                  └─ Task: [httpd container, nginx container with React]
 │                       └─ httpd:80 → nginx:8080 (localhost)
 │
 └─ api.example.com → Target Group: API Service
                    └─ Task: [httpd container, FastAPI container]
                         └─ httpd:80 → FastAPI:8000 (localhost)
```

Two ECS Services with Multi-Container Tasks:

1. **UI Service:** httpd + nginx (serving React)
2. **API Service:** httpd + FastAPI

Detailed Configuration

Apache httpd Configuration for UI (app.example.com):

```
# /etc/httpd/conf.d/oidc.conf

LoadModule auth_openidc_module modules/mod_auth_openidc.so

# OIDC Provider Configuration
OIDCProviderMetadataURL https://login.microsoftonline.com/<tenant-id>/v2.0/.well-known/
openid-configuration
OIDCClientID <ui-client-id>
OIDCClientSecret <ui-client-secret>
OIDCRedirectURI https://app.example.com/oauth2/callback
OIDCCryptoPassphrase <random-passphrase>

# Scope configuration
OIDCScope "openid profile email api://<api-client-id>/user.read"

# Session configuration
OIDCSessionType server-cache
OIDCSessionInactivityTimeout 3600
OIDCSessionMaxDuration 86400

# Cache configuration (for multi-instance deployments)
```

```
OIDCCacheType redis
OIDCRedisCacheServer redis.internal.example.com:6379

# Response type
OIDCResponseType code
OIDCResponseMode form_post

# Token handling
OIDCPassAccessToken On
OIDCPassIDTokenAs payload
OIDCPassRefreshToken On

# Virtual Host
<VirtualHost *:80>
    ServerName app.example.com

    # Protect all paths
    <Location />
        AuthType openid-connect
        Require valid-user
    </Location>

    # Public health check (no auth)
    <Location /health>
        AuthType None
        Require all granted
    </Location>

    # Serve React SPA static files
    DocumentRoot /var/www/html
    <Directory /var/www/html>
        Options -Indexes +FollowSymLinks
        AllowOverride None
        Require all granted

        # SPA routing - redirect all to index.html
        RewriteEngine On
        RewriteBase /
        RewriteRule ^index\.html$ - [L]
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteRule . /index.html [L]
    </Directory>

    # Expose user info endpoint for SPA
    <Location /auth/userinfo>
        AuthType openid-connect
        Require valid-user

        SetHandler application/json
```

```
        Header set Content-Type "application/json"

        # Return user info as JSON
        OIDCInfoHook userinfo
    </Location>

    # Optional: Inject user info into HTML
    <Location /auth/bootstrap>
        AuthType openid-connect
        Require valid-user

        SetHandler application/json
        Header set Content-Type "application/json"

        # Custom script to return user + tokens
        SetEnv OIDC_CLAIM_preferred_username %{OIDC_CLAIM_preferred_username}e
        SetEnv OIDC_CLAIM_name %{OIDC_CLAIM_name}e
        SetEnv OIDC_access_token %{OIDC_access_token}e
    </Location>
</VirtualHost>
```

Apache httpd Configuration for API (api.example.com):

```
# /etc/httpd/conf.d/api-oidc.conf

LoadModule auth_openidc_module modules/mod_auth_openidc.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so

# OIDC Provider Configuration
OIDCProviderMetadataURL https://login.microsoftonline.com/<tenant-id>/v2.0/.well-known/
openid-configuration
OIDCClientID <api-client-id>
OIDCClientSecret <api-client-secret>
OIDCRedirectURI https://api.example.com/oauth2/callback
OIDCCryptoPassphrase <random-passphrase>

# OAuth 2.0 Resource Server Mode (Bearer Token Validation)
OIDCOAuthIntrospectionEndpoint https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/
introspect
OIDCOAuthClientID <api-client-id>
OIDCOAuthClientSecret <api-client-secret>

# Accept Bearer tokens
OIDCOAuthAcceptTokenAs header
OIDCOAuthTokenExpiryClaim exp

# Scope validation
```

```
OIDCOAuthVerifySharedKeys <jwks-uri>

# Session configuration (for browser-based access)
OIDCSessionType server-cache
OIDCCacheType redis
OIDCRedisCacheServer redis.internal.example.com:6379

<VirtualHost *:80>
    ServerName api.example.com

    # Public health check
    <Location /health>
        AuthType None
        Require all granted
        ProxyPass http://localhost:8000/health
        ProxyPassReverse http://localhost:8000/health
    </Location>

    # API endpoints - support both Bearer tokens and session cookies
    <Location />
        # Try OAuth Bearer token first, fall back to OIDC session
        AuthType oauth20
        Require valid-user

        # Alternative: Use openid-connect for session-based
        # AuthType openid-connect
        # Require valid-user

        # Proxy to FastAPI backend
        ProxyPass http://localhost:8000/
        ProxyPassReverse http://localhost:8000/

        # Forward user claims as headers
        RequestHeader set X-User-ID %{OIDC_CLAIM_oid}e
        RequestHeader set X-User-Email %{OIDC_CLAIM_preferred_username}e
        RequestHeader set X-User-Name %{OIDC_CLAIM_name}e
        RequestHeader set X-User-Roles %{OIDC_CLAIM_roles}e
        RequestHeader set X-User-Scopes %{OIDC_CLAIM_scp}e
        RequestHeader set X-Access-Token %{OIDC_access_token}e
    </Location>
</VirtualHost>
```

Dockerfile for httpd with mod_auth_openidc:

```dockerfile
FROM httpd:2.4
```

Install mod_auth_openidc

```
RUN apt-get update && \
apt-get install -y libapache2-mod-auth-openidc && \
apt-get clean
```

Enable required modules

```
RUN a2enmod auth_openidc proxy proxy_http headers rewrite ssl
```

Copy configuration

```
COPY oidc.conf /etc/apache2/conf-available/
RUN a2enconf oidc
```
