Authentication and Authorization with OIDC Providers

Introduction to related Auth concepts for web apps and servers.

Showcase of 2 related OIDC flows as well as popular implementation strategies for SPA apps.

stzagkarak 07/25

Content:

1. OAuth 2.0 and OpenID Connect Introduction
2. OAuth 2.0 Authorization Code Flow with PKCE
3. OAuth 2.0 Client Credentials Flow

Authorization vs Authentication

- **Authentication** - *Who are you?*

  Verifies the identity of the user.

  Confirms that the user (or system) is really who they claim to be.

- **Authorization** - *What are you allowed to do?*

  Determines access rights and permissions.

  Decides what an authenticated user is allowed to access or perform.

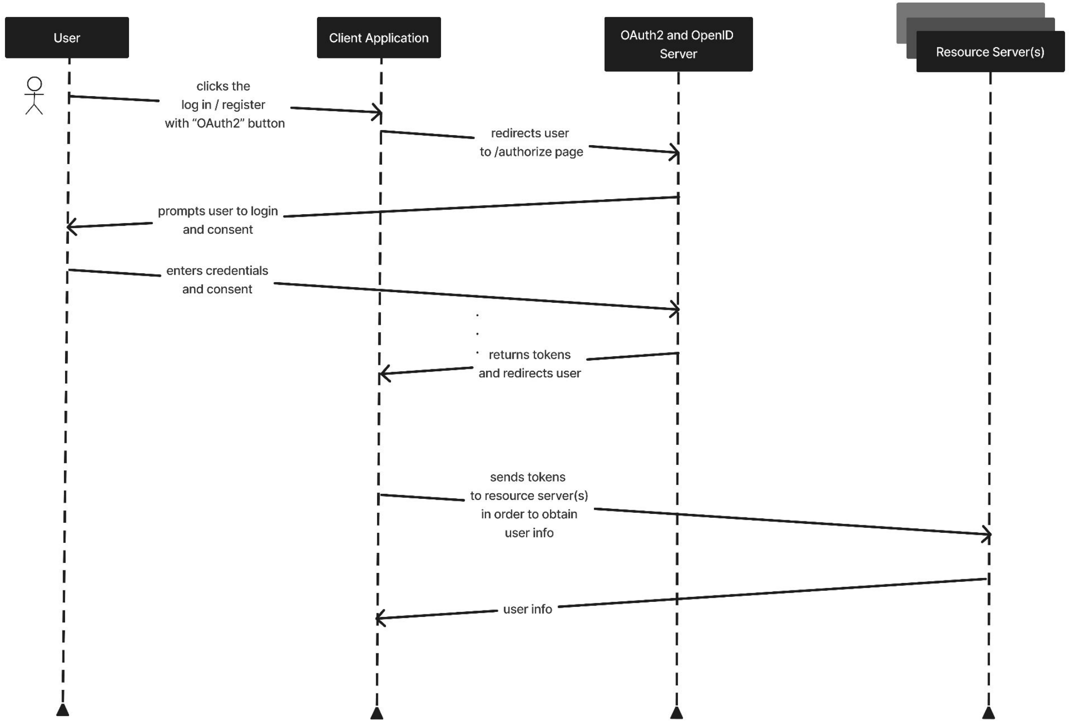
What is OAuth 2.0 ?

- An **Authorization framework** that enables applications to access **user resources** from **resource servers** over HTTP.
- Provides **Authorization,** NOT Authentication.
- Uses **Access Tokens**, no passwords.
- Permissions granted to third party applications are defined by **Scopes**, specifying exactly what the application can access.

Key Terms

- **Resource owner**: End User
- **OAuth2 client**: The application that uses OAuth 2.0 Flows to authenticate its users.
- **Resource Server**: A server holding sensitive user information.
- **Authorization Server**: A server capable of providing access tokens that can be consumed by resource servers.
- **Authorization Flows**: The standardized strategies that are implemented by the OAuth 2.0 compliant Authorization Server.

OAuth Flow (over-simplified)

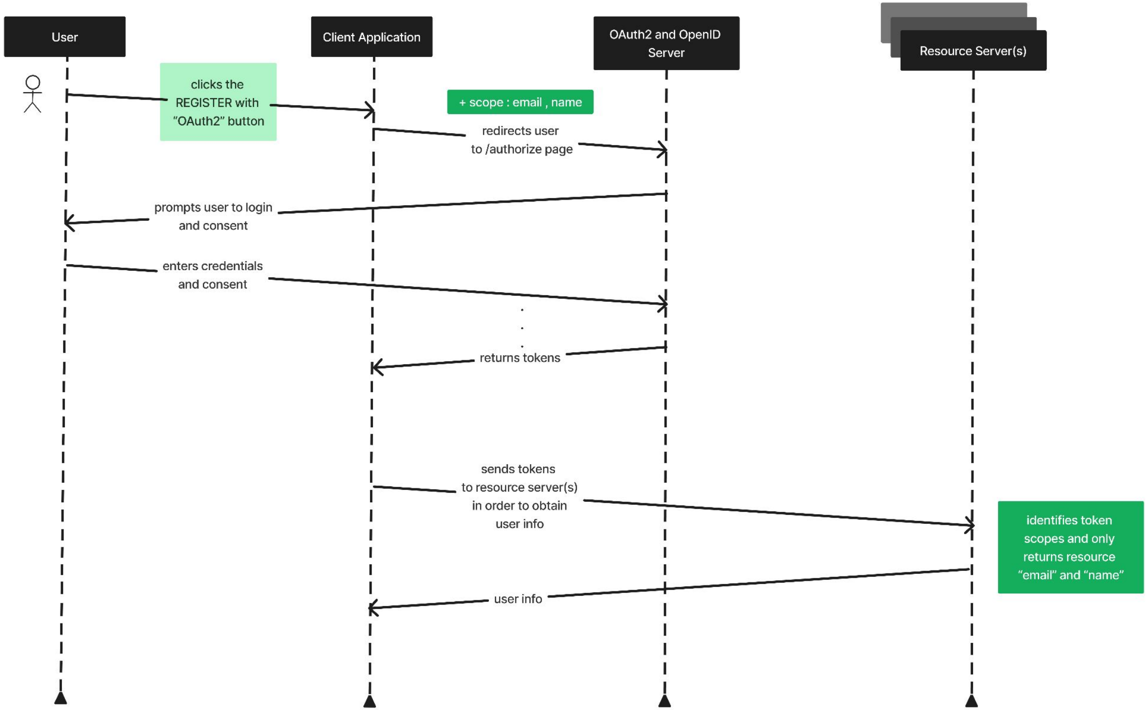

User
Client Application
OAuth2 and OpenID Server
Resource Server(s)
clicks the log in / register with “OAuth2” button
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
returns tokens and redirects user
sends tokens to resource server(s) in order to obtain user info
user info

1. OAuth 2.0 and OpenID Connect Introduction

Scopes

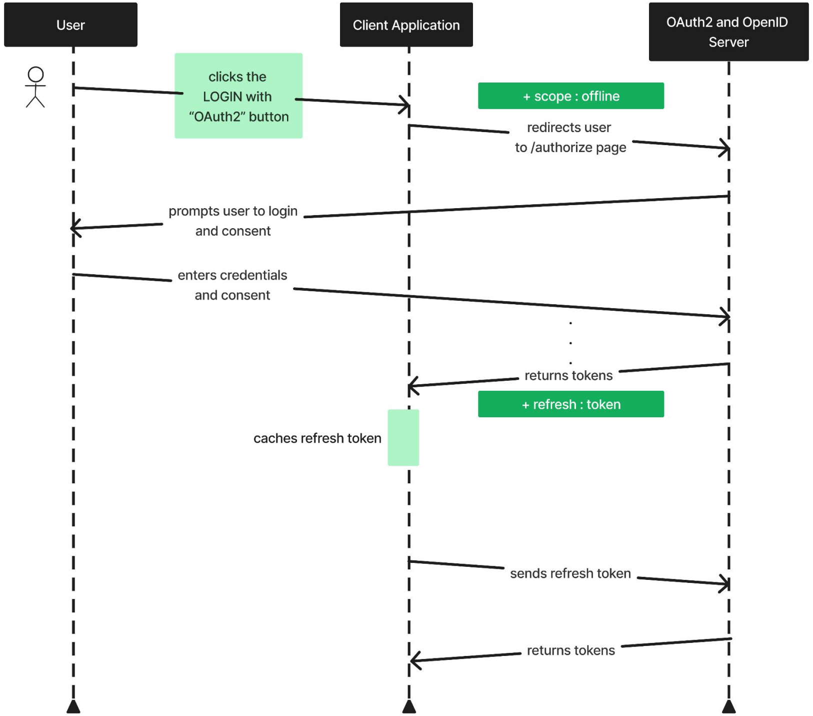
- Define what the client is authorized to access in the name of the user.
- Some popular pre-defined scopes:
  - offline: Allows a third-party application to **issue new pairs of access tokens** on behalf of the user (no need for the user to provide consent again).
  - openid: Allows a third-party application to gain read access to *generic* user information (such as username, photo, roles etc.).
  - email: Used with the `openid` scope to gain access to the primary email of the user.

OAuth Flow (over-simplified + scopes)


User
Client Application
OAuth2 and OpenID Server
Resource Server(s)
clicks the REGISTER with "OAuth2" button
+ scope : email , name
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
returns tokens
sends tokens to resource server(s) in order to obtain user info
identifies token scopes and only returns resource "email" and "name"
user info

1. OAuth 2.0 and OpenID Connect Introduction

OAuth Flow (over-simplified + scopes + refresh scope)

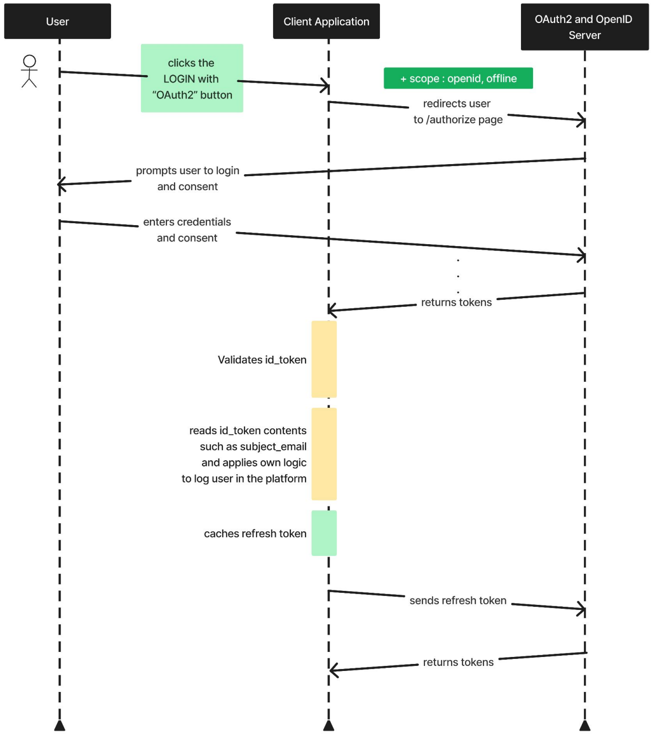


1. OAuth 2.0 and OpenID Connect Introduction

What is OpenID Connect ?

- An **Authentication Protocol,** build on top of the OAuth 2.0 framework.
- Used by third-party applications in order **verify user identities** and **obtain user profile information**.
- From an end-user perspective, the protocol provides a standardized way to login to different applications and websites using a **single set of credentials** (Single Sign On) .
- Provides third-party applications with an **ID token** holding generic user data. Can differentiate between user types by assigning **roles**.
- Authorization Server Implementations (e.g Keycloak) bundle OAuth 2.0 and OIDC (authorization and authentication) in specific flows.

OAuth Flow (over-simplified + offline scope + openid scope)


User
Client Application
OAuth2 and OpenID Server
clicks the LOGIN with "OAuth2" button
+ scope : openid, offline
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
returns tokens
Validates id_token
reads id_token contents such as subject_email and applies own logic to log user in the platform
caches refresh token
sends refresh token
returns tokens

1. OAuth 2.0 and OpenID Connect Introduction

Grants (Standardized Strategies and Flows)

- OAuth 2.0 has several flows that grant authorization, each designed for different application types and security requirements.
- **Many flows have been deprecated** or exist for really specific use cases (e.g devices with low capabilities).
- The goals are always the same, **to authorize and/or authenticate** entities.
- In this presentation we will focus on the Authorization Code Grant and the Client Credentials Grant.

Moving on...

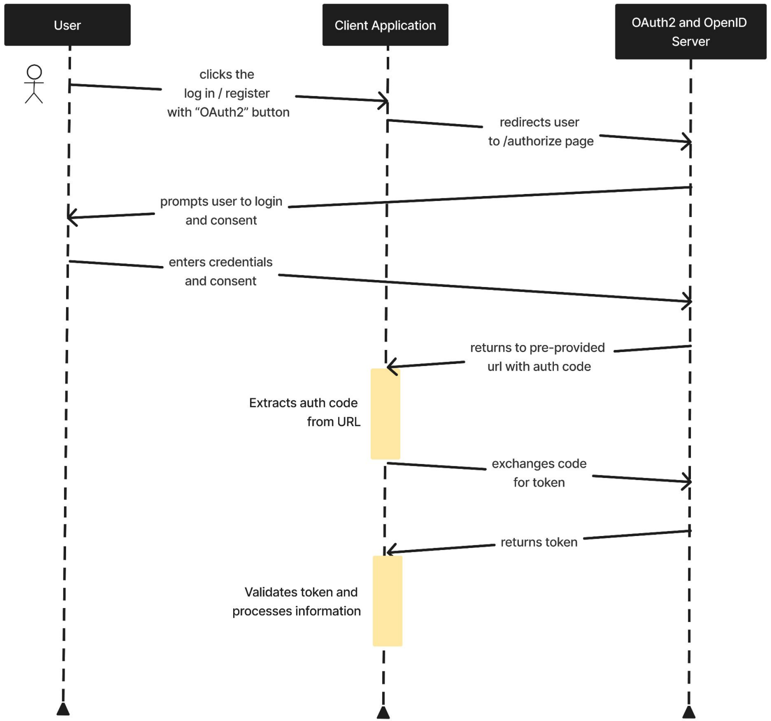
The ***most secure*** way **widely used** way for web applications (server-side, SPAs with protected APIs) to verify identities and setup sessions with clients.

Authorization Code Grant

Authorization Code Grant

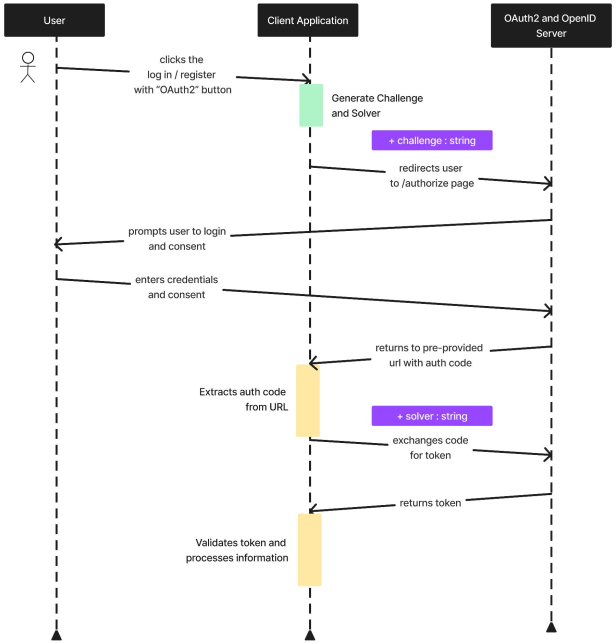
- **Returns both access tokens and ID tokens** (if the `openid` scope is specified), providing both Authorization and Authentication.
- The **most secure** way and **widely used** flow for web applications (server-side, SPAs with protected APIs) to gain access and ID tokens.
- The strategy can also be used securely in mobile apps.
- Used as a **starting point to to verify a user’s identity and establish a secure session** with the frontend client (Stateless JWT or Stateful cookie-session).
- **2-step process** (1 redirect and 1 extra call to obtain tokens).

Authorization Code Flow (simple)


User
Client Application
OAuth2 and OpenID Server
clicks the log in / register with “OAuth2” button
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
returns to pre-provided url with auth code
Extracts auth code from URL
exchanges code for token
returns token
Validates token and processes information

2. OAuth 2.0 Authorization Code Flow with PKCE

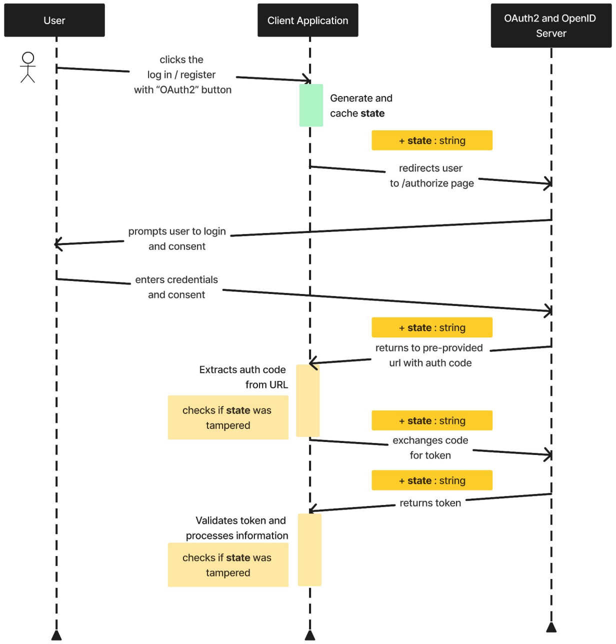
Authorization Code Flow (simple + PKCE)


User
Client Application
OAuth2 and OpenID Server
clicks the log in / register with "OAuth2" button
Generate Challenge and Solver
+ challenge : string
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
returns to pre-provided url with auth code
Extracts auth code from URL
+ solver : string
exchanges code for token
returns token
Validates token and processes information

- Authentication Code Flow with PKCE (proof key for Code exchange):
  - Prevents CSRF and Authorization Code Injection attacks.

2. OAuth 2.0 Authorization Code Flow with PKCE

Authorization Code Flow (simple + state)


User
Client Application
OAuth2 and OpenID Server
clicks the log in / register with "OAuth2" button
Generate and cache **state**
+ **state** : string
redirects user to /authorize page
prompts user to login and consent
enters credentials and consent
+ **state** : string
returns to pre-provided url with auth code
Extracts auth code from URL
checks if **state** was tampered
+ **state** : string
exchanges code for token
+ **state** : string
returns token
Validates token and processes information
checks if **state** was tampered

- Authentication Code Flow with State (proof key for Code exchange):
  - Prevents CSRF attacks. Attacks where the attacker uses the same session as the victim in parallel.

2. OAuth 2.0 Authorization Code Flow with PKCE

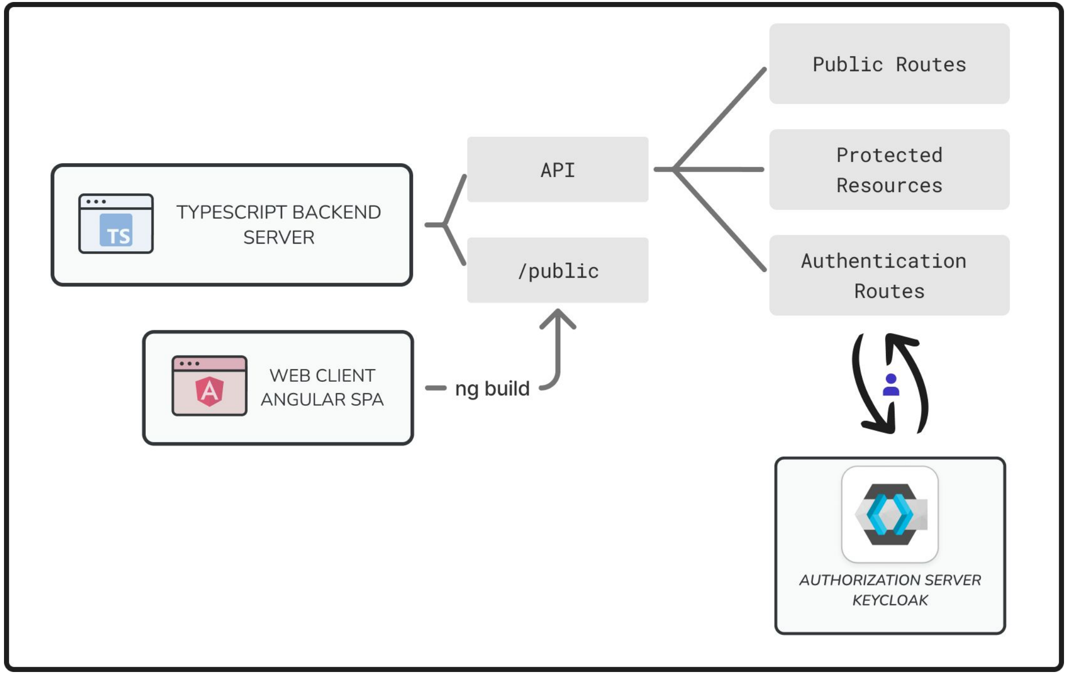
Requirements

- The third-party **application must be registered as a realm client** on the service provider’s infrastructure.
    - Can be **initiated by the admin**, manually (using the Keycloak an admin web interface)
    - Can be **initiated by the application itself** (using the client-registration endpoint and OIDC dynamic client registration)
- The third-party **application must register valid endpoints for client callbacks** after login, after logout.

Goals

- The goal is to **use the Authorization Code Grant** (with the `openid` scope) **in order to verify the identity of a user**.
- Having verified the identity of the user, **we want to create a secure stateless session with our frontend client** (Angular) using JWT tokens.

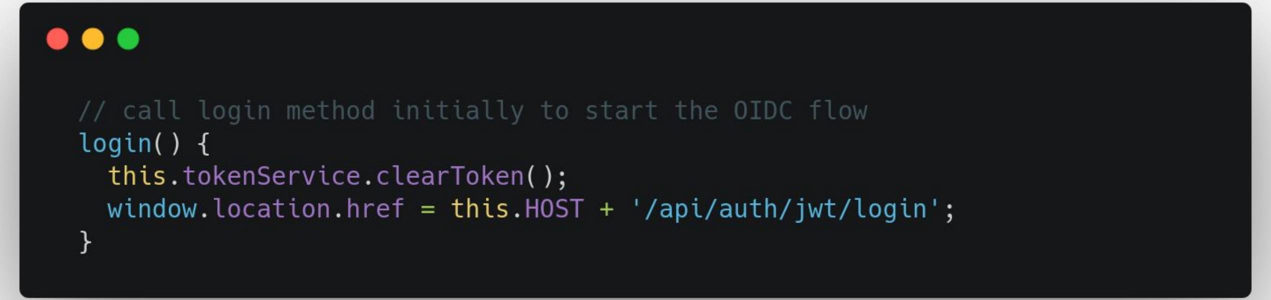
Application Diagram



2. OAuth 2.0 Authorization Code Flow with PKCE

Login Flow (1 / 3)

The frontend **SPA client** (Angular in our case) **redirects to a specific backend route** (here /api/auth/jwt/login) in order to initiate the login process.



```
// call login method initially to start the OIDC flow
login() {
  this.tokenService.clearToken();
  window.location.href = this.HOST + '/api/auth/jwt/login';
}
```

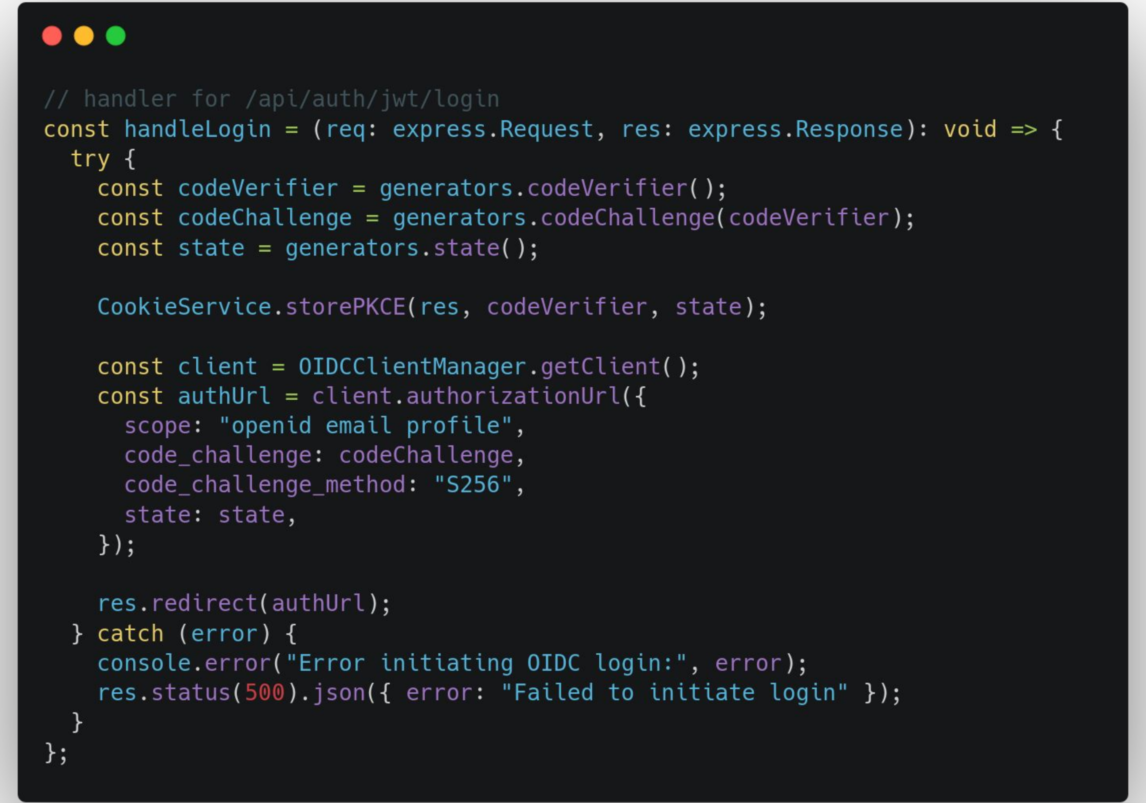
file: ../homepage.ts

Login Flow (2 / 3)

The backend application initiates the Authorization Code Grant.

- The backend **redirects** to the Authorization Server's login Screen.
- Upon successful user login, the Authorization Server returns to the backend application with an "Authorization Code".
- The server then redirects to the Authorization server while providing the Authorization Code in order to receive a pair of access,id,refresh tokens for the signed in user.
- The Authorization server returns to a specific backend route (here `/api/auth/jwt/callback`) with the tokens included.
- The backend generates its own JWT tokens to initiate a secure session with browser client code (Angular SPA). It **redirects** the user back to the SPA with the generated, short lived, session JWT token as a URL parameter.

Login Flow (2/3)

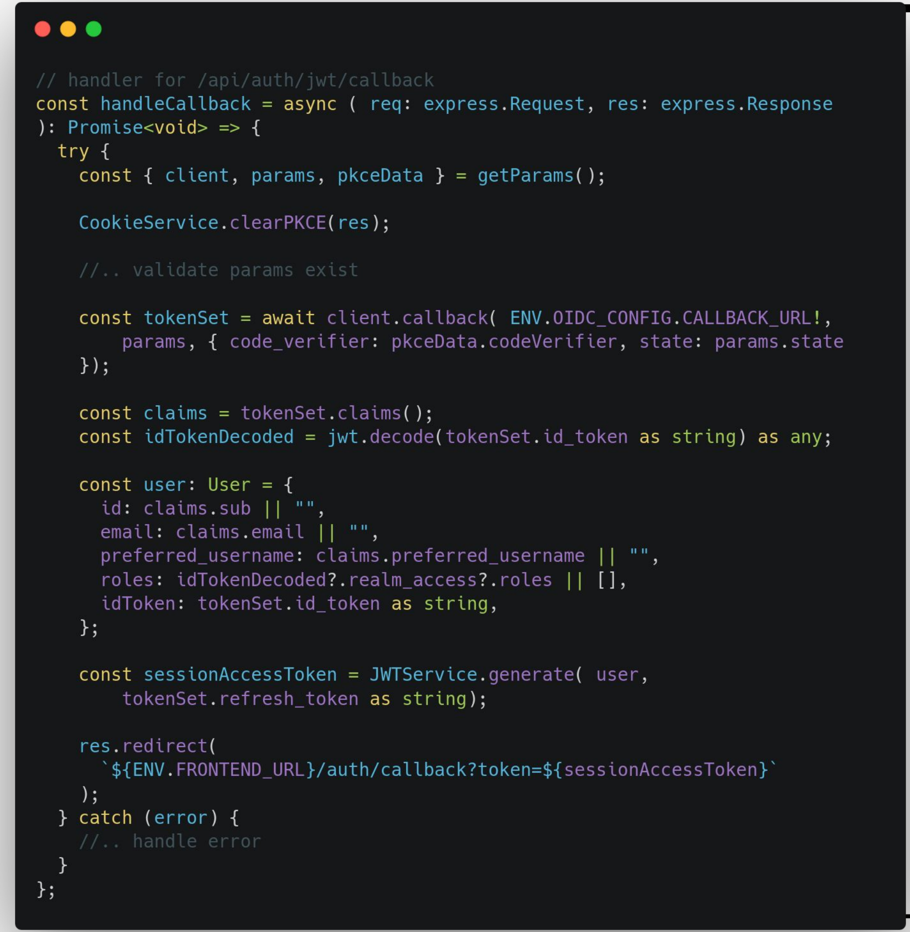


```
// handler for /api/auth/jwt/login
const handleLogin = (req: express.Request, res: express.Response): void => {
  try {
    const codeVerifier = generators.codeVerifier();
    const codeChallenge = generators.codeChallenge(codeVerifier);
    const state = generators.state();

    CookieService.storePKCE(res, codeVerifier, state);

    const client = OIDCClientManager.getClient();
    const authUrl = client.authorizationUrl({
      scope: "openid email profile",
      code_challenge: codeChallenge,
      code_challenge_method: "S256",
      state: state,
    });

    res.redirect(authUrl);
  } catch (error) {
    console.error("Error initiating OIDC login:", error);
    res.status(500).json({ error: "Failed to initiate login" });
  }
};
```

2. OAuth 2.0 Authorization Code Flow with PKCE

file: /backend/src/index.ts

Login Flow (2 / 3)

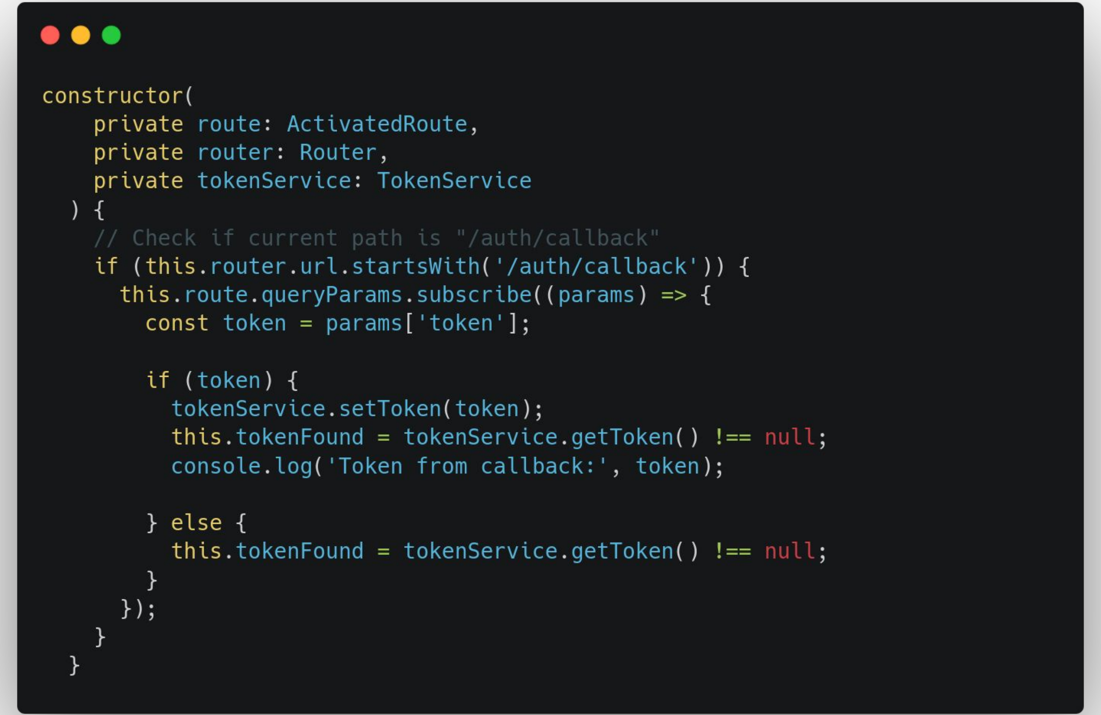


```
// handler for /api/auth/jwt/callback
const handleCallback = async ( req: express.Request, res: express.Response
): Promise<void> => {
 try {
  const { client, params, pkceData } = getParams();

  CookieService.clearPKCE(res);

  //.. validate params exist

  const tokenSet = await client.callback( ENV.OIDC_CONFIG.CALLBACK_URL!,
    params, { code_verifier: pkceData.codeVerifier, state: params.state
  });

  const claims = tokenSet.claims();
  const idTokenDecoded = jwt.decode(tokenSet.id_token as string) as any;

  const user: User = {
   id: claims.sub || "",
   email: claims.email || "",
   preferred_username: claims.preferred_username || "",
   roles: idTokenDecoded?.realm_access?.roles || [],
   idToken: tokenSet.id_token as string,
  };

  const sessionAccessToken = JWTService.generate( user,
    tokenSet.refresh_token as string);

  res.redirect(
   `${ENV.FRONTEND_URL}/auth/callback?token=${sessionAccessToken}`
  );
 } catch (error) {
  //.. handle error
 }
};
```

2. OAuth 2.0 Authorization Code Flow with PKCE

file: /backend/src/index.ts

Login Flow (3 / 3)



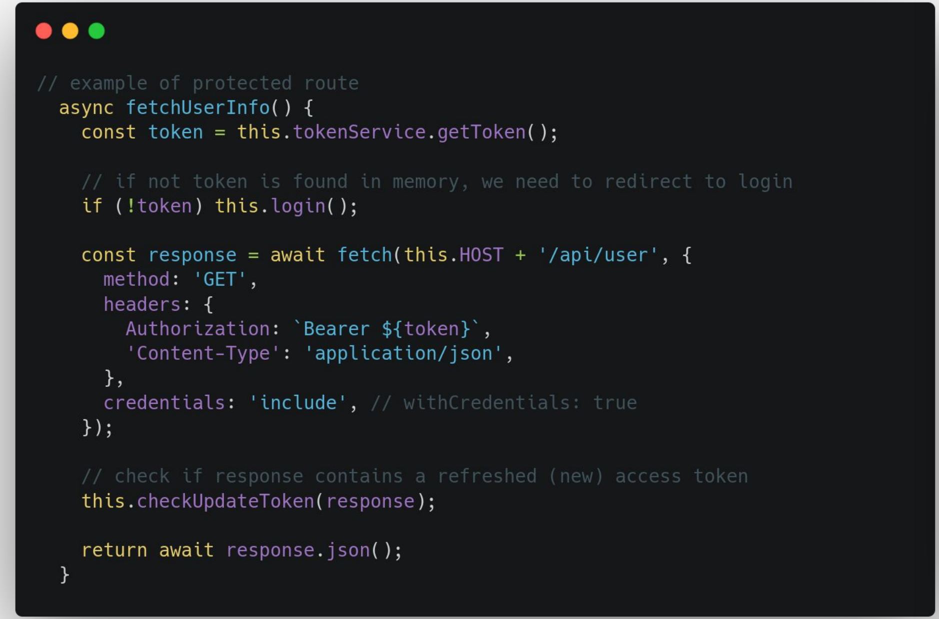
```
constructor(
    private route: ActivatedRoute,
    private router: Router,
    private tokenService: TokenService
  ) {
    // Check if current path is "/auth/callback"
    if (this.router.url.startsWith('/auth/callback')) {
      this.route.queryParams.subscribe((params) => {
        const token = params['token'];

        if (token) {
          tokenService.setToken(token);
          this.tokenFound = tokenService.getToken() !== null;
          console.log('Token from callback:', token);

        } else {
          this.tokenFound = tokenService.getToken() !== null;
        }
      });
    }
  }
```

The Frontend SPA Application **captures the URL parameter** and keeps the token in memory.

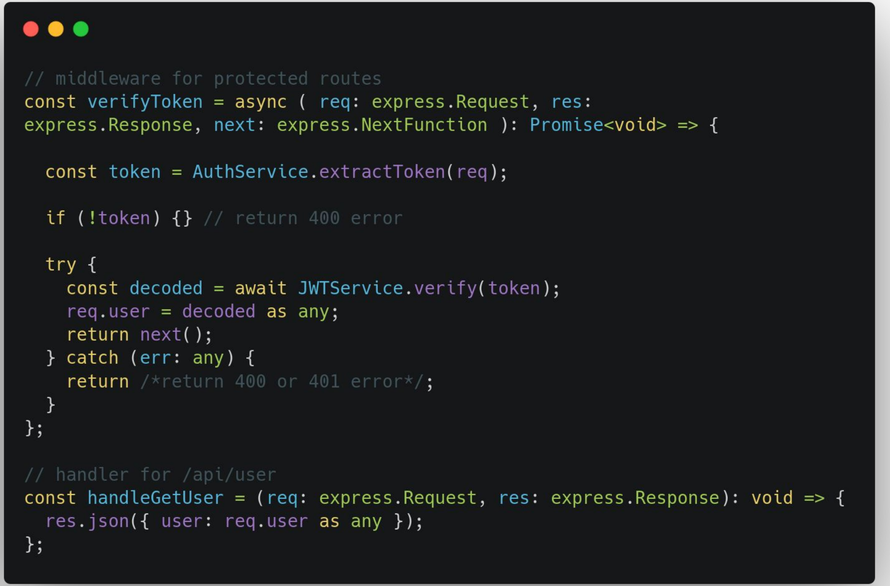
- Insecure practice to save the token in localstorage on the same thread.
- Modern approach is to either:
    - Just **keep the token in memory**.
    - Use Web Workers and store the variable in their localstorage.

file: ../homepage.ts

Calling Protected Endpoints



```
// example of protected route
  async fetchUserInfo() {
    const token = this.tokenService.getToken();

    // if not token is found in memory, we need to redirect to login
    if (!token) this.login();

    const response = await fetch(this.HOST + '/api/user', {
      method: 'GET',
      headers: {
        Authorization: `Bearer ${token}`,
        'Content-Type': 'application/json',
      },
      credentials: 'include', // withCredentials: true
    });

    // check if response contains a refreshed (new) access token
    this.checkUpdateToken(response);

    return await response.json();
  }
```



```
// middleware for protected routes
const verifyToken = async ( req: express.Request, res:
express.Response, next: express.NextFunction ): Promise<void> => {

  const token = AuthService.extractToken(req);

  if (!token) {} // return 400 error

  try {
    const decoded = await JWTService.verify(token);
    req.user = decoded as any;
    return next();
  } catch (err: any) {
    return /*return 400 or 401 error*/;
  }
};

// handler for /api/user
const handleGetUser = (req: express.Request, res: express.Response): void => {
  res.json({ user: req.user as any });
};
```

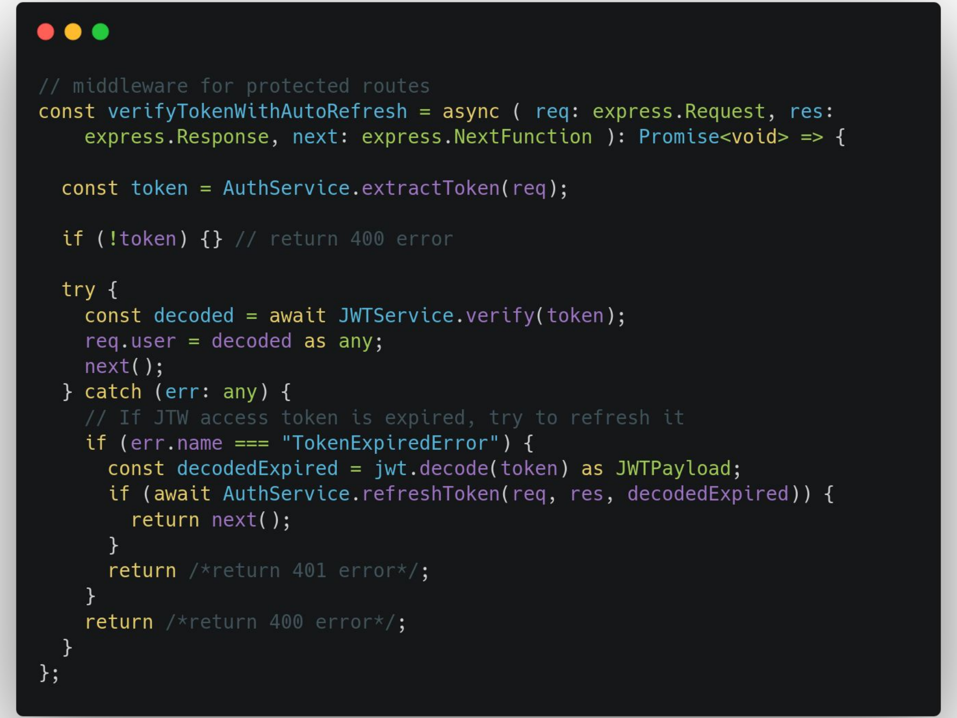
files: ../homepage.ts, /backend/index.ts

Session Refresh

- The **backend also manages token renewal** when the session JWT access token expires.

- Calls the refresh endpoint on the Authentication Server and if a new tokenset is returned from the Authorization Server, new JWT token is issued and returned to the client.

- Otherwise, returns a hint for the frontend to login again as the client is in an invalid state or the oidc session has expired.

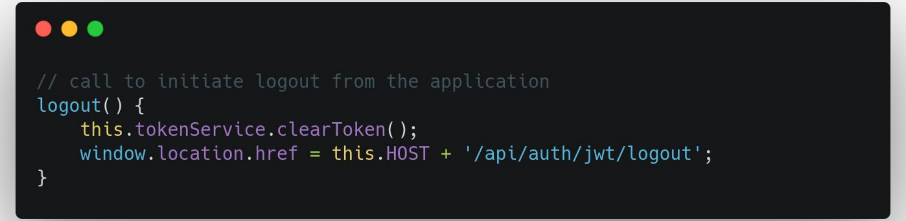


```
// middleware for protected routes
const verifyTokenWithAutoRefresh = async ( req: express.Request, res:
    express.Response, next: express.NextFunction ): Promise<void> => {

  const token = AuthService.extractToken(req);

  if (!token) {} // return 400 error

  try {
    const decoded = await JWTService.verify(token);
    req.user = decoded as any;
    next();
  } catch (err: any) {
    // If JTW access token is expired, try to refresh it
    if (err.name === "TokenExpiredError") {
      const decodedExpired = jwt.decode(token) as JWTPayload;
      if (await AuthService.refreshToken(req, res, decodedExpired)) {
        return next();
      }
      return /*return 401 error*/;
    }
    return /*return 400 error*/;
  }
};
```

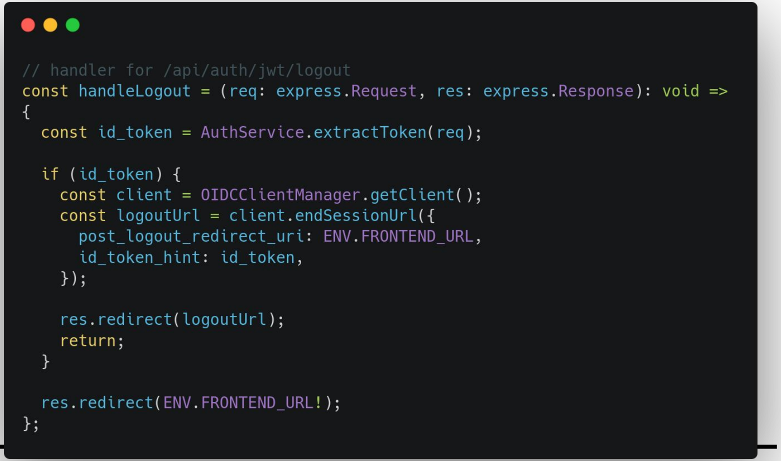
file: /backend/src/index.ts

Logout Flow

- The SPA **redirects** to a specific backend route (here `/api/auth/jwt/logout`).
- If an access token is provided, the backend **redirects** to the Authorization Server endSession endpoint.
- After successful logout, the Authorization Server **redirects** to a valid registered URL.



```
// call to initiate logout from the application
logout() {
    this.tokenService.clearToken();
    window.location.href = this.HOST + '/api/auth/jwt/logout';
}
```



```
// handler for /api/auth/jwt/logout
const handleLogout = (req: express.Request, res: express.Response): void =>
{
  const id_token = AuthService.extractToken(req);

  if (id_token) {
    const client = OIDCClientManager.getClient();
    const logoutUrl = client.endSessionUrl({
      post_logout_redirect_uri: ENV.FRONTEND_URL,
      id_token_hint: id_token,
    });

    res.redirect(logoutUrl);
    return;
  }

  res.redirect(ENV.FRONTEND_URL!);
};
```

files: ../homepage.ts, /backend/index.ts

Moving on...

Authorize Server-to-Server communication and data exchange.

Client Credentials Grant

Client Credentials Grant

- Designed for **machine-to-machine** (Server-to-Server) communication, where an application (the client) needs to access a resource server **without user interaction**.
- **The client authenticates itself** directly using its client ID and client secret to request an access token from the authorization server.
- Often used for backend services, cron jobs, daemons, or **APIs talking to APIs**.
- Simple, Scalable, Secure.
- Resource Server: Hosts the protected resources (e.g., data or services).
- Validates the access token issued by the authorization server before granting access.

Client Credentials Grant

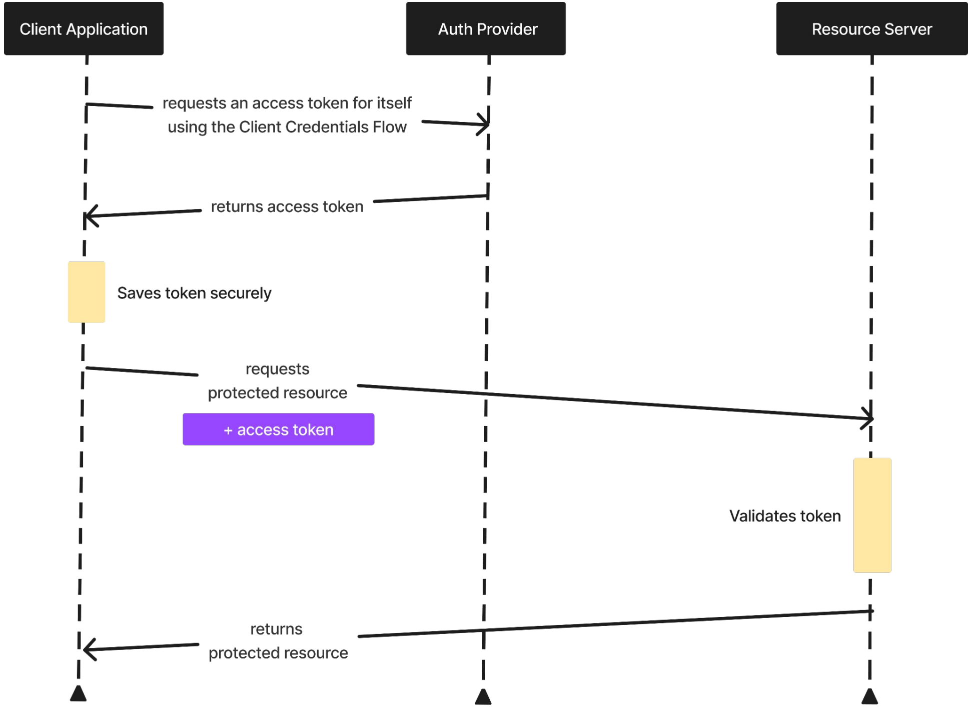
- Resource Server:
  - **Hosts the protected resources** (e.g., data or services).
  - Can **validate the access token** issued by the authorization server in 2 ways:
    1. Using the **introspect endpoint** of the authorization server.
    2. **Locally**, using the public certs provided by the Authorization Server.

Client Credentials Grant

- Client (Application):
  - The machine or service that wants to access a protected resource.
  - **Executes the Client Credentials Flow** in order to gain access tokens.
  - **Authenticates itself** using its client ID and secret.

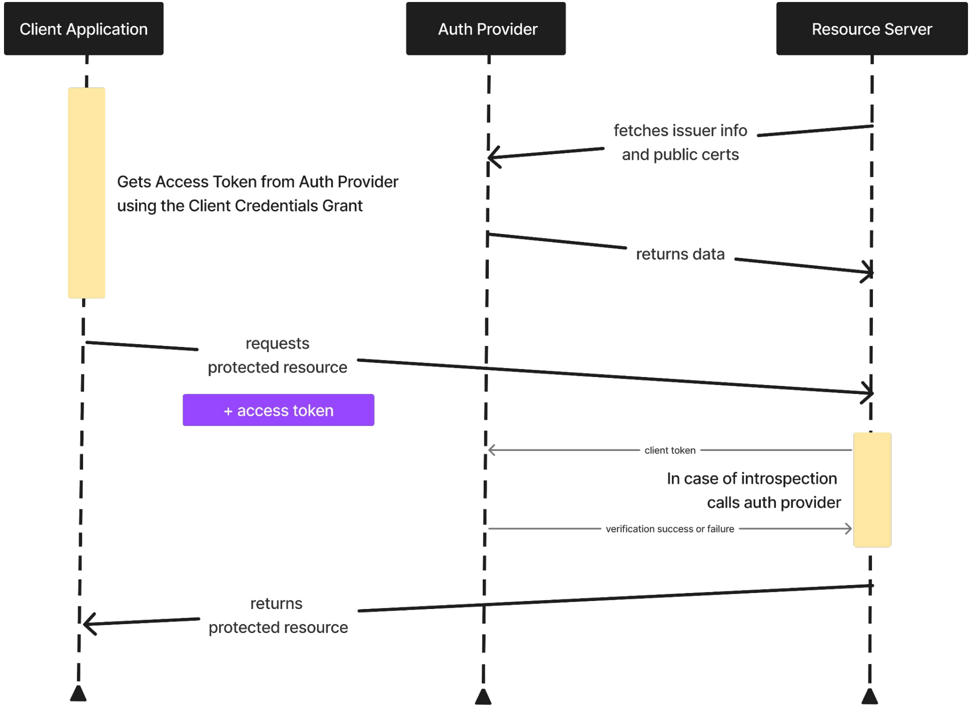
- Both clients (resource server and requesting server) need to registered to the OAuth 2.0 provider.

Client Credentials Flow (Application Client)


Client Application
Auth Provider
Resource Server
requests an access token for itself using the Client Credentials Flow
returns access token
Saves token securely
requests protected resource
+ access token
Validates token
returns protected resource

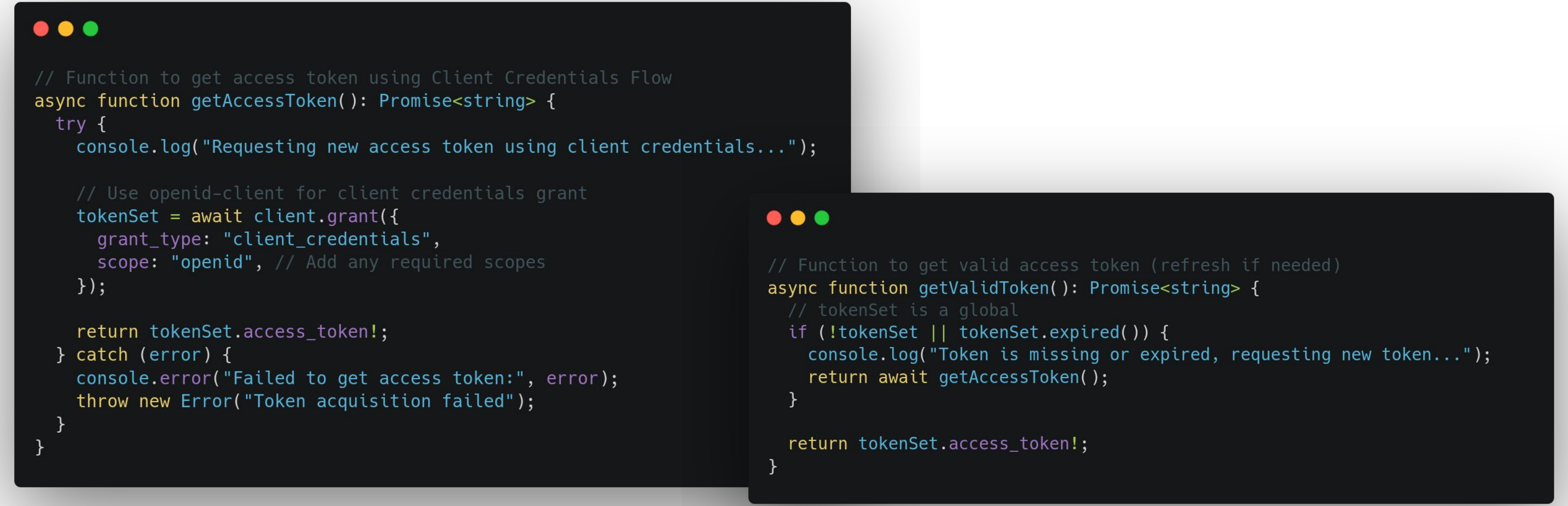
3. OAuth 2.0 Client Credentials Flow

Client Credentials Flow (Resource Server)


Client Application
Auth Provider
Resource Server
fetches issuer info and public certs
Gets Access Token from Auth Provider using the Client Credentials Grant
returns data
requests protected resource
+ access token
client token
In case of introspection calls auth provider
verification success or failure
returns protected resource

3. OAuth 2.0 Client Credentials Flow

Client Application (1 / 2)

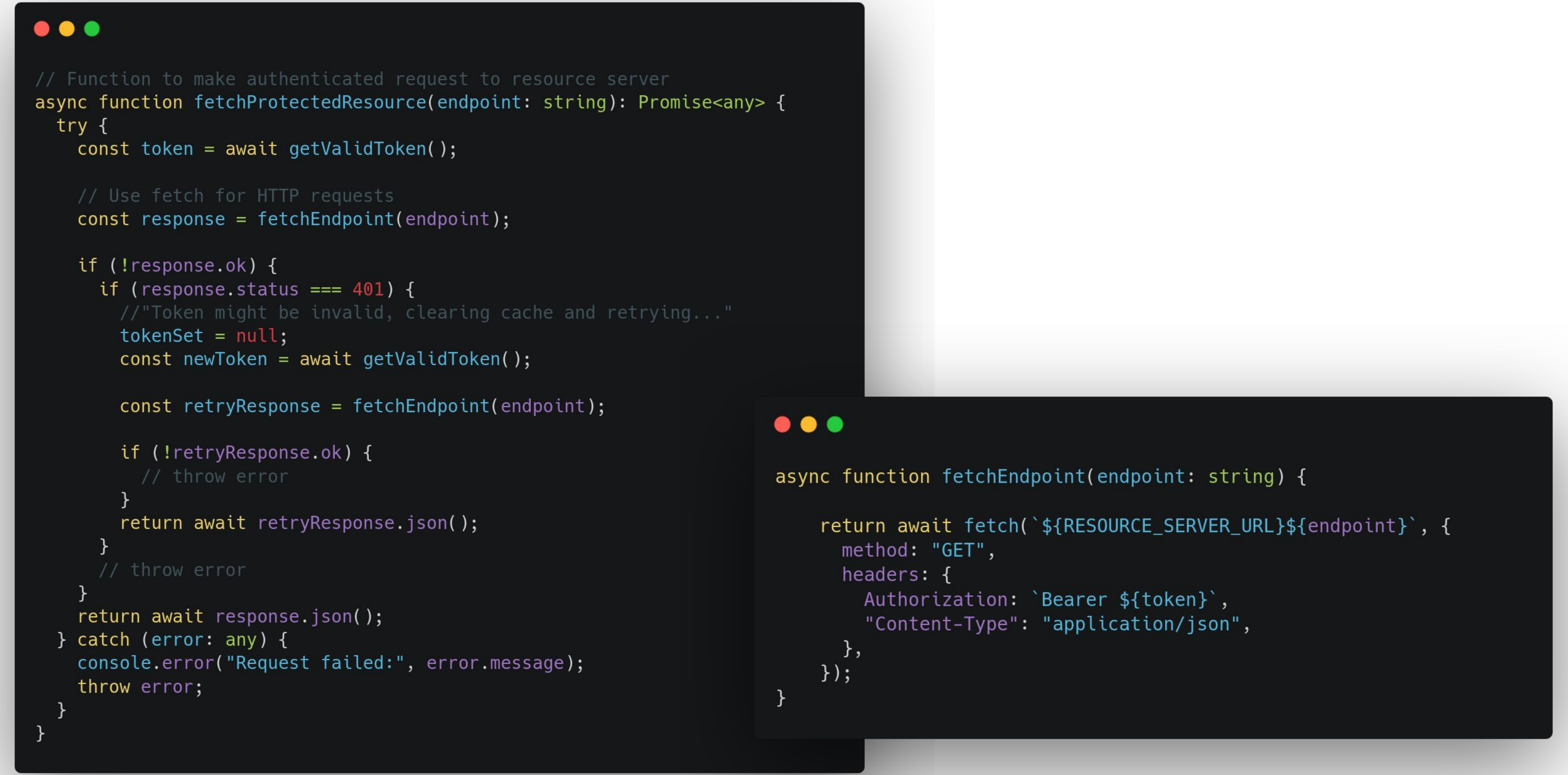


```
// Function to get access token using Client Credentials Flow
async function getAccessToken(): Promise<string> {
  try {
    console.log("Requesting new access token using client credentials...");

    // Use openid-client for client credentials grant
    tokenSet = await client.grant({
      grant_type: "client_credentials",
      scope: "openid", // Add any required scopes
    });

    return tokenSet.access_token!;
  } catch (error) {
    console.error("Failed to get access token:", error);
    throw new Error("Token acquisition failed");
  }
}
```

```
// Function to get valid access token (refresh if needed)
async function getValidToken(): Promise<string> {
  // tokenSet is a global
  if (!tokenSet || tokenSet.expired()) {
    console.log("Token is missing or expired, requesting new token...");
    return await getAccessToken();
  }

  return tokenSet.access_token!;
}
```

file: ../client/src/index.ts

Client Application (2 / 2)

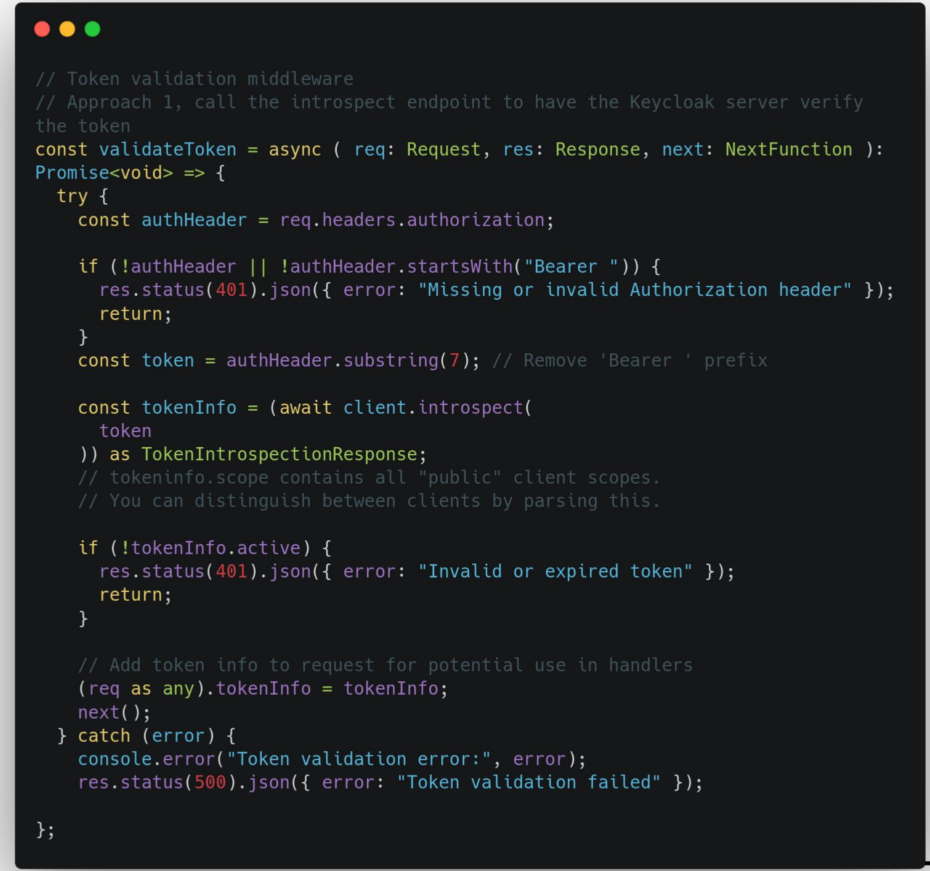


```typescript
// Function to make authenticated request to resource server
async function fetchProtectedResource(endpoint: string): Promise<any> {
  try {
    const token = await getValidToken();

    // Use fetch for HTTP requests
    const response = fetchEndpoint(endpoint);

    if (!response.ok) {
      if (response.status === 401) {
        //"Token might be invalid, clearing cache and retrying..."
        tokenSet = null;
        const newToken = await getValidToken();

        const retryResponse = fetchEndpoint(endpoint);

        if (!retryResponse.ok) {
          // throw error
        }
        return await retryResponse.json();
      }
      // throw error
    }
    return await response.json();
  } catch (error: any) {
    console.error("Request failed:", error.message);
    throw error;
  }
}
```

```typescript
async function fetchEndpoint(endpoint: string) {

    return await fetch(`${RESOURCE_SERVER_URL}${endpoint}`, {
      method: "GET",
      headers: {
        Authorization: `Bearer ${token}`,
        "Content-Type": "application/json",
      },
    });
}
```

file: ../client/src/index.ts

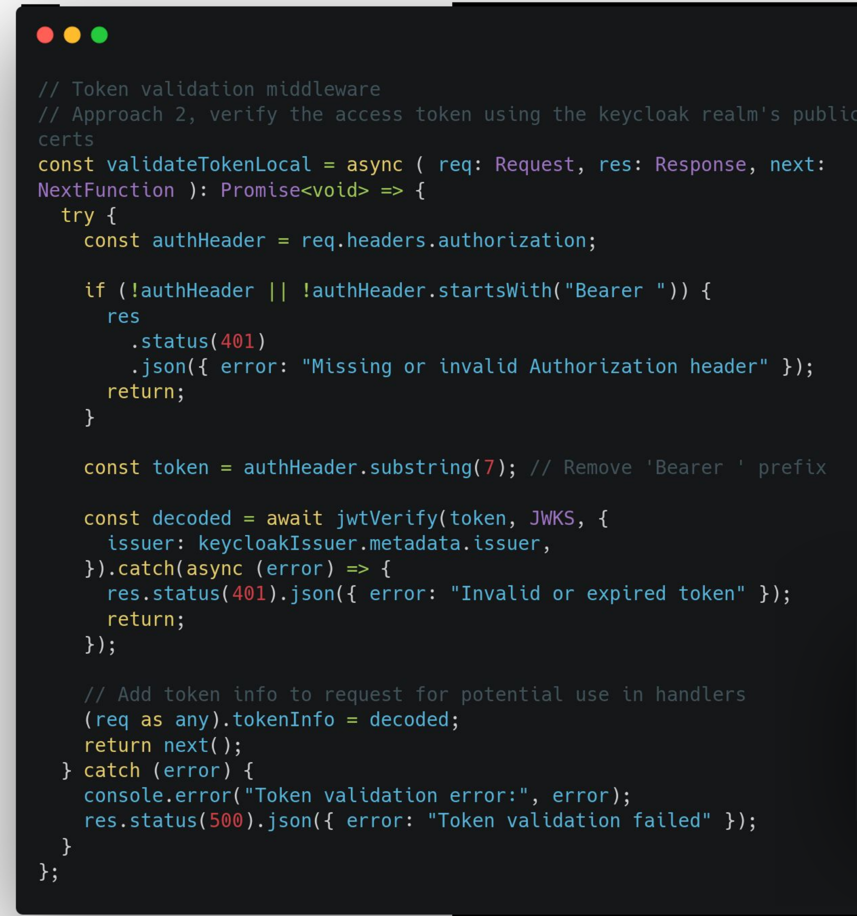
Resource Server (1 / 2) - Calling the Introspection endpoint



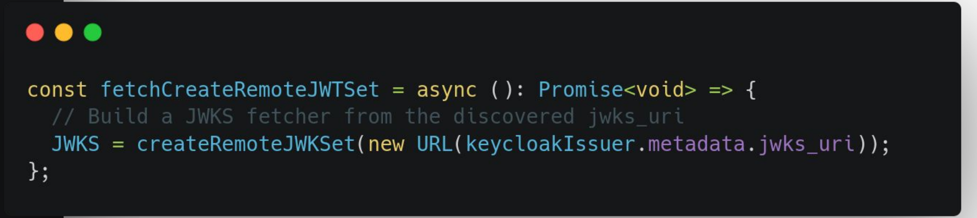
```
// Token validation middleware
// Approach 1, call the introspect endpoint to have the Keycloak server verify
the token
const validateToken = async ( req: Request, res: Response, next: NextFunction ):
Promise<void> => {
  try {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
      res.status(401).json({ error: "Missing or invalid Authorization header" });
      return;
    }
    const token = authHeader.substring(7); // Remove 'Bearer ' prefix

    const tokenInfo = (await client.introspect(
      token
    )) as TokenIntrospectionResponse;
    // tokeninfo.scope contains all "public" client scopes.
    // You can distinguish between clients by parsing this.

    if (!tokenInfo.active) {
      res.status(401).json({ error: "Invalid or expired token" });
      return;
    }

    // Add token info to request for potential use in handlers
    (req as any).tokenInfo = tokenInfo;
    next();
  } catch (error) {
    console.error("Token validation error:", error);
    res.status(500).json({ error: "Token validation failed" });

};
```

file: ../resource-server/src/index.ts

Resource Server (2 / 2) - Local Validation using realm public certs



```
// Token validation middleware
// Approach 2, verify the access token using the keycloak realm's public
certs
const validateTokenLocal = async ( req: Request, res: Response, next:
NextFunction ): Promise<void> => {
  try {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
      res
        .status(401)
        .json({ error: "Missing or invalid Authorization header" });
      return;
    }

    const token = authHeader.substring(7); // Remove 'Bearer ' prefix

    const decoded = await jwtVerify(token, JWKS, {
      issuer: keycloakIssuer.metadata.issuer,
    }).catch(async (error) => {
      res.status(401).json({ error: "Invalid or expired token" });
      return;
    });

    // Add token info to request for potential use in handlers
    (req as any).tokenInfo = decoded;
    return next();
  } catch (error) {
    console.error("Token validation error:", error);
    res.status(500).json({ error: "Token validation failed" });
  }
};
```



```
const fetchCreateRemoteJWTSet = async (): Promise<void> => {
  // Build a JWKS fetcher from the discovered jwks_uri
  JWKS = createRemoteJWKSet(new URL(keycloakIssuer.metadata.jwks_uri));
};
```

file: ../resource-server/src/index.ts

This to consider in production

- reverse proxy setup
- cookie attributes (SameSite, secure attribute)
- refresh token encryption
- config files
- docker config files

Further Reading

- OAuth 2.0 Introduction by 0Auth.
- Keycloak Server Administration Guide.
- node.js openid-client package documentation.

The End

stzagkarak@07/25