Login:-

const login = async function (params) {

    try {

        let loginResponse;

        // let keyResponse = await fetch(

        //  `${BASE\_URL}/ab-digital-bpm/api/v1/rskakey`,

        //  {

        //      method: "POST",

        //      headers: {},

        //  }

        // );

        // keyResponse = await handleResponse(keyResponse);

        // const keyID = keyResponse.data.kid;

        // let publicKey = await cryptoApi.importKey(

        //  "jwk", //can be "jwk" (public or private), "spki" (public only), or "pkcs8" (private only)

        //  {

        //      // ...keyResponse.data,

        //      kid: keyResponse.data.kid,

        //      kty: keyResponse.data.kty,

        //      e: keyResponse.data.e,

        //      n: keyResponse.data.n,

        //      alg: "RS256",

        //      ext: true,

        //  },

        //  {

        //      //these are the algorithm options

        //      name: "RSASSA-PKCS1-v1\_5", // RSASSA-PKCS1-v1\_5, || RSA-PSS || RSA-OAEP

        //      hash: { name: "SHA-256" }, // can be "SHA-1", "SHA-256", "SHA-384", or "SHA-512"

        //  },

        //  true, //whether the key is extractable (i.e. can be used in exportKey)

        //  ["verify"] //"encrypt" or "wrapKey" for public key import or

        //  //"decrypt" or "unwrapKey" for private key imports

        // );

        // let public\_key\_pem = await crypt.cryptoPublicToPem(publicKey);

        // Encrypt.setPublicKey(public\_key\_pem);

        // let encrypted\_password = Encrypt.encrypt(params.password);

        loginResponse = await fetch(

            `${BASE\_URL}/dbpm/utilities/v1/uam/api/auth/2fa/signin`,

            // `${BASE\_URL}/dbpm/utilities/v1/uam/api/auth/signin?country=${USER\_COUNTRY()}&lang=EN`, //OG dbpm login endpoint

            {

                method: "POST",

                headers: {

                    'accept': '\*/\*',

                    'client\_id': CLIENT\_ID,

                    "x-fapi-interaction-id": INTERACTION\_ID(),

                    'uuid': INTERACTION\_ID(),

                    "x-channel-identifier": 'MB',

                    'lang': "en",

                    "country": 'JO',

                    "Content-Type": "application/json",

                },

                body: JSON.stringify({

                    userName: params.username,

                    password: params.password,

                }),

            }

        );

        let result = await handleResponse(loginResponse);

        document.cookie =

            "a=" + result?.accessToken || result?.data?.accessToken;

        // "a=" + (result?.accessToken || result?.data?.accessToken) + '; path=/';

        document.cookie =

            "a.r=" + result?.refreshToken || result?.data?.refreshToken;

        // "a.r=" + (result?.refreshToken || result?.data?.refreshToken) + '; path=/';

        localStorage.setItem("u", params.username);

        return result;

    } catch (e) {

        console.log(e);

        throw e;

    }

};

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Get user details:-

const getUserDetails = async function (userName) {

    if (!userName || !typeof userName === "String") {

        throw new Error("Invalid params: userName");

    }

    try {

        // let response = await fetch( `${getConfig().resourceUrl}/${UAM\_API\_CONTEXT}/api/user/v1/userdetails`, {

        let response = await fetch(

            `${BASE\_URL}/dbpm/utilities/v1/uam/api/user/v1/${userName}?country=JO&lang=EN`,

            {

                // let response = await fetch( `/api/user-details`, {

                // method: "POST",

                method: "GET",

                headers: {

                    client\_id: CLIENT\_ID,

                    "x-service-identifier": "",

                    "x-jws-signature": "",

                    "x-fapi-interaction-id": INTERACTION\_ID(),

                    uuid: INTERACTION\_ID(),

                    "x-channel-identifier": CHANNEL\_NAME,

                }

            }

        );

        let result = handleResponse(response);

        return result;

    } catch (e) {

        throw e;

    }

};

A screenshot of a computer program

Description automatically generated

Fetch user details from uam

Get all entitlements user is exposed to :

const getEntitleMent = async function (username) {

    try {

        let result = await fetch(

            `${BASE\_URL}/dbpm/utilities/v1/ent/entitlements/activities?lang=EN`,

            {

                method: "GET",

                headers: {

                    ...commomHeaders,

                    userName: username,

                    //"country": USER\_COUNTRY()

                },

            }

        ).then(handleResponse);

        return result

    } catch (e) {

        throw e;

    }

};

A screenshot of a computer

Description automatically generated

In a typical web application, the tokens received after a successful sign-in are used for authentication and authorization in subsequent API requests. Here's how these tokens work:

**1. Access Token**

* **Purpose**: The access token (accessToken) is used to authenticate requests to the server. It usually has a limited lifespan (in your case, until the expiration time indicated in the exp claim).
* **Usage**:
  + Include this token in the Authorization header of your HTTP requests to secure endpoints.
  + Example: http Authorization: Bearer <accessToken>
  + The server validates this token on every request to check if the user is authenticated and has the necessary permissions to access the requested resource.

**2. Refresh Token**

* **Purpose**: The refresh token (refreshToken) is used to obtain a new access token when the current one expires without requiring the user to log in again.
* **Usage**:
  + When you detect that the access token is expired (e.g., based on the expiration timestamp or a 401 Unauthorized response from the server), you can use the refresh token to request a new access token.
  + The request to refresh the token generally looks like this:

POST /auth/refresh

Content-Type: application/json

{

"refreshToken": "<refreshToken>"

}

* + If the refresh token is valid, the server responds with a new access token (and possibly a new refresh token).

**3. 2FA and Security**

* In your case, since is2FARequired is false, you don't need to implement two-factor authentication right now.
* If it were true, you would need to handle additional steps to verify the user’s second factor (like a code sent via SMS or email).

**4. Storage of Tokens**

* Tokens are usually stored in cookies (as you mentioned), local storage, or session storage in the browser.
* Cookies can be set with the HttpOnly flag to prevent JavaScript access, making them more secure against XSS (Cross-Site Scripting) attacks.

**5. Logout Flow**

* When a user logs out, you should invalidate both tokens by server-side logic—either manually deleting them from the database or adding them to a blacklist (in case refresh tokens can still be valid based on their lifespan).

**6. Implications for Client & Server Communication**

* On the client side, you need to implement logic for storing the tokens correctly, refreshing them as needed, and sending them with each API request.
* The server must have middleware to handle token validation, ensuring that users can only access resources based on the permissions defined in the token scopes and authorities.

In summary, once you receive the tokens after a successful sign-in, you will use the access token for API calls that require authentication, refresh it using the refresh token when needed, and manage them securely in your application.

Cboss---------------

If production:

The function sends a POST request to a server endpoint to generate RSA keys.

The response is passed to handleResponse, which presumably processes the server's response.

* **Extracting Key ID:**

const keyID = keyResponse.data.kid;

This retrieves the key ID from the server's response, which is likely used for tracking or identification purposes.

let publicKey = await cryptoApi.importKey(

"jwk",

{

kid: keyResponse.data.kid,

kty: keyResponse.data.kty,

e: keyResponse.data.e,

n: keyResponse.data.n,

alg: "RS256",

ext: true,

},

{

name: "RSASSA-PKCS1-v1\_5",

hash: { name: "SHA-256" },

},

true,

["verify"]

);

* The public key received from the server is converted into a format that can be used for cryptographic operations (like verifying signatures).
* It specifies the key type (kty), public exponent (e), modulus (n), and the algorithm for encryption (RSASSA-PKCS1-v1\_5).

**Converting Public Key to PEM Format:**

let public\_key\_pem = await crypt.cryptoPublicToPem(publicKey);

This converts the imported public key into the PEM format, which is a common encoding for cryptographic keys.

1. **Setting Public Key for Encryption:**

Encrypt.setPublicKey(public\_key\_pem);

The public key is set in the Encrypt module, making it available for encrypting passwords.

1. **Encrypting Password:**

let encrypted\_password = Encrypt.encrypt(params.password);

Here, the user's password is encrypted using the public key, ensuring that the plaintext password is not sent over the network.

1. **Sending Login Request:**

loginResponse = await fetch(

`${getConfig().resourceUrl}/${UAM\_API\_CONTEXT}/api/auth/signin2`,

{

method: "POST",

headers: {

"Content-Type": "application/json",

uuid: keyID,

},

body: JSON.stringify({

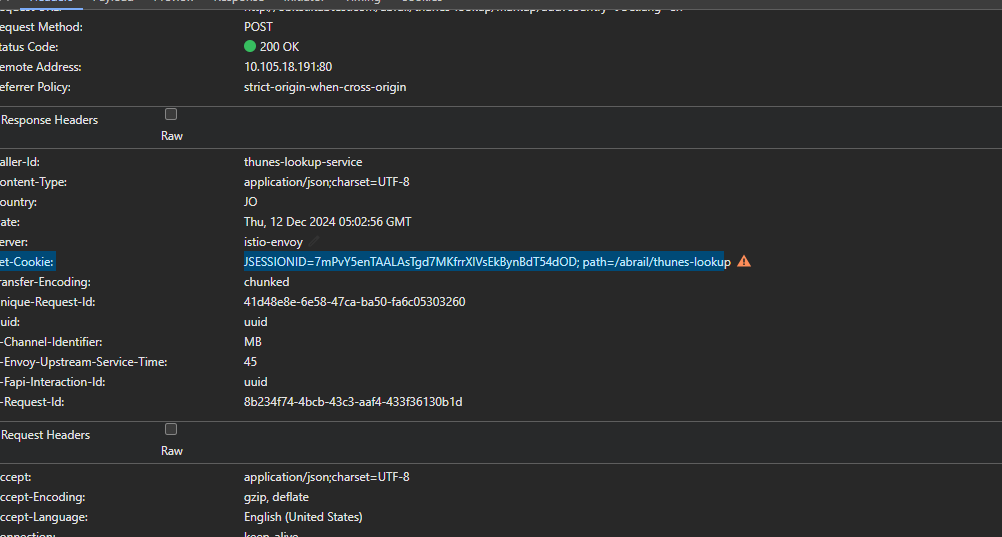
username: params.username,

password: encrypted\_password,

}),

}

);

A POST request is sent to the authentication endpoint with the username and the encrypted password. The header includes the uuid (the key ID) for identification.  
JSESSIONID:-  


No jwt token is send , session based authentication is happening for services

In the provided code snippet, the editAccount function is responsible for sending an HTTP POST request to update account details. The interesting aspect of this setup is the handling of authentication and authorization, particularly through the use of cookies.

**How Authentication and Authorization Work**

1. **Session Management via JSESSIONID**:

* The JSESSIONID is a cookie commonly used by Java-based web applications (like those running on a Java servlet container, e.g., Apache Tomcat) to manage user sessions.
* When a user successfully logs in to the application (via a preceding login request that is not shown in your provided code), the server generates a JSESSIONID and sends it back to the client in the response header (as a Set-Cookie).

1. **Cookie Persistence**:

* Once the client (typically a browser) receives the JSESSIONID cookie, it stores it. For subsequent requests (like the editAccount request), the browser automatically includes this cookie in the request headers.
* This means that while you do not see the JSESSIONID in the headers() function within the editAccount call, the browser includes it automatically if it is configured to do so for the domain of the request.

1. **Authorization**:

* On the server side, the application retrieves the JSESSIONID from the incoming request when handling the editAccount call.
* It then checks the session associated with this ID to verify that the user is authenticated and has the appropriate permissions to perform the action (in this case, editing the account).

**Summary of the Process**

1. **User Login**: The user logs in, and the server generates a JSESSIONID.
2. **Cookie Storage**: The JSESSIONID is stored in the client's cookie storage.
3. **Subsequent Requests**: During subsequent requests, the cookie is automatically sent in the request headers.
4. **Server Validation**: The server validates the JSESSIONID to authenticate the user and determine if they have the rights to perform the requested action.

**Conclusion**

In your example, despite not explicitly passing a token in the header of your editAccount function, the authentication is handled through the use of the JSESSIONID cookie, which the browser automatically manages. This is a common pattern in applications that require user authentication and can provide a seamless user experience.

In summary, if you see JSESSIONID in the Set-Cookie of previous responses, it will be included automatically by the client's browser in subsequent requests, including your editAccount POST request, allowing the server to authenticate the user accordingly.

In the code segment provided, the authentication and authorization process likely relies on session management using cookies, specifically the JSESSIONID. Here’s a breakdown of how this typically works:

**Authentication and Authorization Mechanism**

1. **Session Management with JSESSIONID:**

* When a user logs in to the application, the server usually validates the user's credentials (username, password, etc.). If the credentials are correct, the server creates a new session.
* During this session creation, the server generates a unique identifier (in your case, JSESSIONID), which is a way to track the user's session. This identifier is sent back to the client's browser as a cookie.
* Every subsequent HTTP request made by the client will include this cookie, allowing the server to identify the user and determine their permissions or roles.

1. **No Token in Headers:**

* In your code, you observed that no authentication token (such as a JWT) is included in the request headers. Instead, the cookies (where the JSESSIONID is stored) are automatically sent with the request, which allows the server to authenticate the user without needing to provide a token explicitly in the headers.

1. **Cookie Handling:**

* The server sets the JSESSIONID as a cookie when the user logs in, which the client's browser stores.
* This cookie gets sent along with each subsequent request to the server, including the editAccount function when it calls the update endpoint.

1. **Authorization:**

* After authentication (i.e., confirming the user’s identity through the session), the server checks whether the user has the necessary permissions to carry out the requested operation (like editing an account).
* This step typically uses role-based access control (RBAC) or similar mechanisms, allowing or denying actions based on the user's role.

**Cookie Expiration**

* **Expiration and Refreshing of JSESSIONID:**
* The expiration of JSESSIONID is usually determined by server-side settings in the application server configuration, which can specify how long the session should remain valid.
* When the session is created, it may have an associated timeout (e.g., 30 minutes of inactivity). After this period, the session becomes invalid, and the JSESSIONID cookie may be deleted by the browser.
* In many implementations, if the user is actively using the app (sending requests), the server can refresh the session's expiration time on each request, extending the validity of the session.

**Summary**

In conclusion, the JSESSIONID is a key component of session management and is used for both authentication and authorization by linking requests to the user's session without needing explicit tokens passed in headers. The server manages the lifecycle of the session and its expiration, typically based on activity and predefined timeout settings. If you're interested in specific handling or mechanisms in the application you're working with, consulting the server-side code or configuration would provide more detailed insights into how these aspects are handled.

In this scenario, it looks like the service for editing the account is using session-based authentication through the use of a JSESSIONID. Here’s how authentication and authorization typically works in such a case:

**1. Session-based Authentication**

In session-based authentication, the server creates a session for the user when they log in successfully. This session is typically identified by a unique session ID (like JSESSIONID), which is sent to the client as a cookie. The cookie allows the server to recognize subsequent requests from the same client.

* **Login Process**: When the user logs in, the server validates the credentials and creates a session. The server then sends back a Set-Cookie header in the response, which contains the JSESSIONID. The browser stores this cookie.
* **Subsequent Requests**: For every subsequent request made by the client (e.g., calling the editAccount function), the browser automatically includes the JSESSIONID cookie in the request headers. This allows the server to identify the user's session and determine if they are authenticated.

**2. Authorization**

Once the user is authenticated (i.e., they have a valid session), the server can determine what the user is authorized to do based on their role or permissions associated with the session.

* When the editAccount function is called, it sends the request with the saved JSESSIONID in the cookie. The server checks the session associated with that JSESSIONID and verifies whether the user has the necessary permissions to perform the edit account operation.

**3. Token-less Authentication in Headers**

In the case you provided, aside from the JSESSIONID, no explicit token is passed in the headers for authentication. This is common in session-based authentication, where the session ID managed via cookies suffices for authenticating requests.

**4. Handling of JSESSIONID in Cookies**

* If you delete the JSESSIONID from your cookies and call the service, the server will not recognize the session, and typically it would either return an unauthorized response (like a 401 status) or redirect to a login page depending on server configuration.
* When the server responds to your request without an existing session (with a new JSESSIONID), it would create a new session and send the JSESSIONID back in a Set-Cookie header. This behavior ensures that any user who accesses the service will go through a login or authentication process first, creating a new session if necessary.

**Conclusion**

In summary, the authentication and authorization in this service revolve around session management using JSESSIONID, which is managed through cookies. Each request is authenticated via the session tied to that cookie, meaning the server can easily keep track of who the user is and their permissions without requiring a separate token in the request headers.

When JSESSIONID is deleted and run the response:

The behavior you're observing is related to how session management works in web applications, particularly those that utilize Java Servlet technology or similar frameworks.

1. **JSESSIONID**: This is a cookie that is automatically created by the Java web application when a session is started. It serves as a unique identifier for each user session. The server tracks the session state and associates it with this session ID.
2. **No Token in Header**: In your case, the request does not include an authentication token because the server identifies the session through the JSESSIONID cookie. This is a common approach for maintaining user sessions in web applications. The server checks for the presence of the cookie to determine if a session exists and whether the user is authenticated. If the session is valid, it allows access to the requested resource.
3. **Behavior on Deleting the Cookie**: When you deleted the JSESSIONID cookie, the server identified that there was no valid session associated with your request. However, if you make a request to the server without a session, it typically interprets this as a new session request, resulting in the server creating a new JSESSIONID and sending it back to the client as a response. This means that even if you delete the cookie, you can still interact with the server; the server will just treat you as a new user.
4. **Flow**:
   * **First Request**: If you access the application, the server creates a new session, generates a JSESSIONID, and sends it as a cookie to the client.
   * **Subsequent Requests**: When the client sends requests, the JSESSIONID cookie is included, allowing the server to recognize the session and manage user state.
   * **Cookie Deletion**: If you delete the cookie and then make a request, the server detects the absence of the session ID and creates a new session, resulting in a new JSESSIONID.
5. **Security Considerations**: Using session cookies for authentication can be less secure compared to token-based authentication methods (like JWT), especially if the cookies are not configured with secure attributes. It's essential to ensure that session management techniques are implemented correctly to mitigate risks such as session fixation or hijacking.

In conclusion, the absence of a token in the header does not signify a lack of security or state management; instead, it implies reliance on session cookies for maintaining the session state between the client and server.

The behavior you're observing with the `JSESSIONID` cookie being set and sent in the response headers is typical in web applications that use session management. Here's a breakdown of what's happening:

1. \*\*Session Management\*\*: When you make a request to a server (in this case, the `editAccount` function), the server may create a session for the user if one does not already exist. This session is often identified by a unique session ID, which is stored in a cookie (like `JSESSIONID`).

2. \*\*Setting the Cookie\*\*: When the server processes your request, it may respond with a `Set-Cookie` header, which instructs the client's browser to store the cookie. In your case, the server is setting the `JSESSIONID` cookie with a specific value and a path.

```http

Set-Cookie: JSESSIONID=txRDSO6MtEMvjCsUgcWkeG04rMI9x0qZYxDROXyV; path=/abrail/thunes-lookup

```

This means that the cookie will be sent with any subsequent requests to the specified path.

3. \*\*Cookie in Subsequent Requests\*\*: Once the cookie is set, it will automatically be included in the headers of any future requests made to the same domain and path. This is how the server can recognize the session associated with the user without needing to pass a token explicitly in the headers.

4. \*\*Why No Token?\*\*: In your case, it seems that the server is relying on the session cookie (`JSESSIONID`) for authentication and session management instead of requiring a token to be passed in the headers. This is a common approach in web applications, especially those that use Java-based backends (as indicated by the `JSESSIONID` naming convention).

5. \*\*Security Considerations\*\*: While using cookies for session management is standard, it's important to ensure that cookies are secured properly (e.g., using the `Secure` and `HttpOnly` flags) to prevent issues like session hijacking.

### Summary

- The `JSESSIONID` cookie is being set by the server to manage the user's session.

- The cookie is automatically included in subsequent requests to the server, allowing it to recognize the session without needing an explicit token in the headers.

- This is a common practice in web applications for session management.

Global transfer:-

authorization:

Bearer vXteaOdDZOyj8DW9ex\_yWD8Ad4Q