Contents

[📁 webpack.config.js — Remote App React 2](#_Toc201417039)

[📁 webpack.config.js — Host App React 6](#_Toc201417040)

[✅ How Remote Components Are Shared into WordPress (Manual Web Pack config) 9](#_Toc201417041)

[✅ How Remote React Components Are Shared into TYPO3 (Manual Webpack Federation) 11](#_Toc201417042)

[✅ Remote Auth Microfrontend (React) 14](#_Toc201417043)

[🔐 Wordpress Auth Flow 20](#_Toc201417044)

[✅ How Navigation Works in Remote Header 23](#_Toc201417045)

[✅ How the API Testing button is disabled WP 26](#_Toc201417046)

[🧭 Typo3 Auth Flow 28](#_Toc201417047)

[✅ How the Remote Header Works with the Host (TYPO3) 32](#_Toc201417048)

[✅ How the Profile Page Is Protected in TYPO3 Host 34](#_Toc201417049)

[✅ How the "Test API" Button Is Disabled Without JWT 35](#_Toc201417050)

## 📁 webpack.config.js — Remote App React

**📦 Imports**

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

const HtmlWebpackPlugin = require("html-webpack-plugin");

* **ModuleFederationPlugin**: Enables sharing or consuming modules between different frontends at runtime. This is the core of Microfrontend architecture using Webpack 5.
* **HtmlWebpackPlugin**: Injects output scripts (like the federated module) into index.html.

**🔧 Configuration Object**

module.exports = {

mode: "development",

entry: "./src/index.js",

* mode: "development": Enables faster builds, readable output, and development-focused optimizations.
* entry: Main entry file for Webpack (starting point of the app logic).

**🖥️ Dev Server**

devServer: {

port: 3001,

hot: true,

liveReload: true,

headers: {

"Access-Control-Allow-Origin": "\*",

"Access-Control-Allow-Methods": "GET, OPTIONS",

"Access-Control-Allow-Headers": "\*",

},

allowedHosts: "all",

},

* port: 3001: The app runs on localhost:3001.
* hot: Enables hot module replacement (no full reload).
* liveReload: Browser reloads when files change.
* headers: CORS setup — allows any host to fetch from this remote app (critical for host-remote communication).
* allowedHosts: "all": Ensures external access to the app (useful in dev containers or tunnels like ngrok).

**📤 Output**

output: {

publicPath: "http://localhost:3001/",

},

* publicPath: Ensures that assets and federated modules are loaded from this absolute URL. Required for module federation to find the remote entry file correctly.

**📄 Loaders**

module: {

rules: [

{

test: /\.jsx?$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/preset-react", "@babel/preset-env"],

},

},

},

],

},

* **Transpilation Setup**:
  + Targets .js and .jsx files.
  + Uses babel-loader to transpile:
    - JSX to JavaScript (@babel/preset-react)
    - Modern JS (ES6+) to ES5 (@babel/preset-env)
* **Excludes** node\_modules for performance and compatibility.

**🔌 Plugins**

plugins: [

new ModuleFederationPlugin({

name: "remoteCounter",

filename: "remoteEntry.js",

exposes: {

"./SyncedCounter": "./src/SyncedCounter",

"./UnsyncedCounter": "./src/UnsyncedCounter",

"./AuthProvider": "./src/AuthContext",

"./LoginComponent": "./src/LoginComponent",

"./Header": "./src/Header",

},

**➕ Module Federation Settings:**

* name: Name of the remote app (used by host app to refer to this remote).
* filename: This builds a file remoteEntry.js — the manifest of all exposed modules.
* exposes: Maps internal modules that can be shared with the host. Examples:
  + ./Header: Makes the Header component accessible to hosts.
  + ./AuthProvider: Exposes a global auth context provider.
  + ./SyncedCounter: An example feature module.

shared: {

react: {

singleton: true,

requiredVersion: false,

eager: false

},

"react-dom": {

singleton: true,

requiredVersion: false,

eager: false

},

},

}),

**📦 Shared Libraries:**

* singleton: true: Ensures only one instance of React or ReactDOM is used, even if different apps use it.
* requiredVersion: false: Accepts any version of React — no strict match needed.
* eager: false: Lazy loads these modules — they are fetched when actually used.

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

],

};

* Generates an index.html and injects script tags (like the federated remoteEntry.js) automatically.

This config enables the **Remote app** to:

* Be served on localhost:3001
* Expose multiple components (counter, header, login, auth)
* Share dependencies with other apps using **Module Federation**

## 📁 webpack.config.js — Host App React

**📦 Imports**

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

const HtmlWebpackPlugin = require("html-webpack-plugin");

* Same as the remote app — used to dynamically import remote modules and inject the HTML template.

**🔧 Core Configuration**

module.exports = {

mode: "development",

entry: "./src/index.js",

* mode: development: Optimized for fast build and detailed output.
* entry: Starting point of the host app.

**🖥️ Dev Server**

devServer: {

port: 3000,

hot: true,

liveReload: true,

headers: {

"Access-Control-Allow-Origin": "\*",

},

historyApiFallback: true,

},

* Runs on localhost:3000.
* CORS headers to allow loading federated modules.
* historyApiFallback: true: Ensures routing works on refresh (important for SPAs).

**🧪 Loaders**

module: {

rules: [

{

test: /\.jsx?$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

presets: ["@babel/preset-react", "@babel/preset-env"],

},

},

},

],

},

* Transpiles JSX and modern JS syntax for React.

**🔌 Plugins**

plugins: [

new ModuleFederationPlugin({

name: "host",

remotes: {

remoteCounter: "remoteCounter@http://localhost:3001/remoteEntry.js",

},

**🧩 Module Federation:**

* name: "host": Identifier for this app.
* remotes: Defines a **remote name** and its federated entry URL.
  + remoteCounter: Is the alias used to import components.
  + @http://localhost:3001/remoteEntry.js: Remote’s manifest.

shared: {

react: { singleton: true },

"react-dom": { singleton: true },

},

* **Singleton sharing** of core React libraries to avoid version conflicts and duplication.

}),

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

],

};

* Generates HTML file and includes built scripts.

**🔁 Host-Remote Interaction Explained**

The Host app (localhost:3000) can now **dynamically load** components from the Remote app (localhost:3001) via:

import Header from "remoteCounter/Header";

This works because:

* Remote app exposes the module (Header)
* Host app declares the remote URL and loads its remoteEntry.js on runtime
* Webpack connects both apps, and ensures React and ReactDOM are **shared as singletons**

## ✅ How Remote Components Are Shared into WordPress (Manual Web Pack config)

Doing **manual module federation from vanilla JavaScript + PHP**

**📦 Step-by-Step Explanation**

**1. Expose React/ReactDOM to Global Scope in WordPress**

<script src="https://unpkg.com/react@18/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

window.React = React;

window.ReactDOM = ReactDOM;

* This makes React and ReactDOM globally available to the remote app.
* The remote expects React as a shared module (singleton) via Module Federation.

**2. Manually Set the Webpack Share Scope**

window.\_\_webpack\_share\_scopes\_\_ = window.\_\_webpack\_share\_scopes\_\_ || {};

window.\_\_webpack\_share\_scopes\_\_.default = window.\_\_webpack\_share\_scopes\_\_.default || {};

window.\_\_webpack\_share\_scopes\_\_.default.react = {

"18.0.0": {

get: () => () => React,

loaded: true

}

};

* This fakes the share scope used by Webpack Module Federation runtime.
* It tricks the remote module loader into thinking "React is already available" on the page.
* Ensures React is not loaded **again** by the remote app.

**3. Dynamically Load the Remote App**

const script = document.createElement('script');

script.src = 'http://localhost:3001/remoteEntry.js';

* Loads the remote app’s remoteEntry.js, which was configured using ModuleFederationPlugin.

**4. Initialize the Remote App Container**

await container.init(window.\_\_webpack\_share\_scopes\_\_.default);

* Initializes the remote container with the shared scope you manually defined.
* This allows the remote app to **use WordPress’s React**, not its own bundle.

**5. Load and Render Remote Components**

const Header = (await container.get('./Header')).default;

const headerRoot = ReactDOM.createRoot(shadowContainer);

headerRoot.render(React.createElement(Header, { ... }));

* Dynamically loads the Header component from the remote app.
* Renders it into a Shadow DOM within the WordPress page.

Absolutely! Here's a detailed breakdown **for your TYPO3 setup**—just like the WordPress one you referenced.

## ✅ How Remote React Components Are Shared into TYPO3 (Manual Webpack Federation)

✅ Using plain JavaScript and <script> tags inside TYPO3 templates to dynamically integrate React microfrontends built with **Webpack Module Federation**.

**📦 Step-by-Step Explanation**

**1. Expose React/ReactDOM to Global Scope in TYPO3**

<script src="https://unpkg.com/react@18/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>

window.React = React;

window.ReactDOM = ReactDOM;

✅ Purpose:

* Makes React and ReactDOM globally available before loading the remote bundle.
* Remote app expects React as singleton via Webpack MF.
* Prevents double loading of React, avoids version conflicts.

**2. Manually Set the Webpack Share Scope**

window.\_\_webpack\_share\_scopes\_\_ = window.\_\_webpack\_share\_scopes\_\_ || {};

window.\_\_webpack\_share\_scopes\_\_.default = window.\_\_webpack\_share\_scopes\_\_.default || {};

window.\_\_webpack\_share\_scopes\_\_.default.react = {

"18.0.0": { get: () => () => React, loaded: true }

};

window.\_\_webpack\_share\_scopes\_\_.default["react-dom"] = {

"18.0.0": { get: () => () => ReactDOM, loaded: true }

};

window.\_\_webpack\_init\_sharing\_\_ = window.\_\_webpack\_init\_sharing\_\_ || (() => Promise.resolve());

✅ Purpose:

* Fakes Webpack’s shared scope so the remote module doesn’t re-bundle or re-fetch react and react-dom.
* Essential for singleton dependencies like React.

**3. Dynamically Load the Remote App’s Entry File**

const script = document.createElement('script');

script.src = 'http://typo3.local:3001/remoteEntry.js';

script.onload = () => { /\* Proceed to init \*/ };

document.head.appendChild(script);

✅ Purpose:

* Pulls in the remoteEntry.js bundle built with ModuleFederationPlugin.
* This file exposes your remote modules (like ./Header, ./LoginComponent, etc.).

**4. Initialize the Remote Container**

await window.remoteCounter.init(window.\_\_webpack\_share\_scopes\_\_.default);

✅ Purpose:

* Connects the remote container (remoteCounter) with TYPO3’s manually defined share scope.
* Required before calling get() to fetch a remote component.

**5. Load and Render Remote Components**

const headerFactory = await window.remoteCounter.get('./Header');

const RemoteHeader = headerFactory().default;

const headerRoot = ReactDOM.createRoot(document.getElementById('typo3-remote-header-container'));

headerRoot.render(React.createElement(RemoteHeader, {

currentPage: 'dashboard',

onNavigate: (page) => window.location.href = '/some-path'

}));

✅ Purpose:

* Dynamically imports exposed components from the remote app.
* Renders them into TYPO3-managed DOM nodes using React’s runtime.

## ✅ Remote Auth Microfrontend (React)

**📦 Overview**

A microfrontend built with React, exposed via **Webpack Module Federation**. It handles:

* User login/logout
* JWT management
* Automatic token refresh
* Secure token sharing with host apps (via postMessage)
* Exposes React components (AuthProvider, LoginComponent, etc.)

**🧠 Architecture Overview**

**Main File:** AuthContext.js  
**UI File:** LoginComponent.js  
**Expose in Module Federation:** remoteEntry.js via Webpack  
**Shared State:** React Context API  
**Communication:** window.postMessage & sessionStorage

**🔐 Authentication Context (AuthContext.js)**

**📌 React Context**

const AuthContext = createContext();

export const useAuth = () => {

const context = useContext(AuthContext);

if (!context) throw new Error('useAuth must be used within an AuthProvider');

return context;

};

**🔐 Login Function**

const login = async (username, password) => {

const response = await fetch(`${API\_BASE\_URL}/auth/login`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ username, password }),

});

const data = await response.json();

if (response.ok) {

setUser(data.user);

setAccessToken(data.accessToken);

setRefreshToken(data.refreshToken);

shareTokenWithHost(data.accessToken, data.user);

localStorage.setItem('refreshToken', data.refreshToken);

return { success: true, user: data.user };

} else {

return { success: false, error: data.error };

}

};

**🚪 Logout Function**

const logout = async () => {

if (refreshToken) {

await fetch(`${API\_BASE\_URL}/auth/logout`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ refreshToken }),

});

}

setUser(null);

setAccessToken(null);

setRefreshToken(null);

clearStorage();

// Notify host

window.postMessage({ type: 'AUTH\_LOGOUT', payload: { timestamp: Date.now() } }, '\*');

};

**🔄 Refresh Token (Auto & Manual)**

const refreshAccessToken = async () => {

const storedRefreshToken = localStorage.getItem('refreshToken');

if (!storedRefreshToken) return false;

const response = await fetch(`${API\_BASE\_URL}/auth/refresh`, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ refreshToken: storedRefreshToken }),

});

const data = await response.json();

if (response.ok) {

setAccessToken(data.accessToken);

localStorage.setItem('refreshToken', data.refreshToken);

shareTokenWithHost(data.accessToken, user);

return true;

} else {

logout();

return false;

}

};

**⏰ Auto Token Refresh**

useEffect(() => {

if (accessToken) {

const refreshInterval = setInterval(() => {

refreshAccessToken();

}, 14 \* 60 \* 1000); // Every 14 minutes

return () => clearInterval(refreshInterval);

}

}, [accessToken]);

**🔄 Sharing Token with Host App**

const shareTokenWithHost = (token, user) => {

window.postMessage({

type: 'AUTH\_TOKEN\_UPDATE',

payload: {

accessToken: token,

user,

timestamp: Date.now(),

source: 'remoteCounter',

},

}, '\*');

sessionStorage.setItem('jwt\_token', token);

sessionStorage.setItem('user\_data', JSON.stringify(user));

};

**🧏 Token Request Listener**

useEffect(() => {

const handleMessage = (event) => {

if (event.data.type === 'REQUEST\_AUTH\_TOKEN' && accessToken) {

shareTokenWithHost(accessToken, user);

}

};

window.addEventListener('message', handleMessage);

return () => window.removeEventListener('message', handleMessage);

}, [accessToken, user]);

**🧾 UI – LoginComponent.js**

**🔐 Login Form**

<form onSubmit={handleSubmit}>

<input name="username" value={credentials.username} onChange={handleChange} />

<input type="password" name="password" value={credentials.password} onChange={handleChange} />

<button type="submit">Login</button>

</form>

**👋 Post-login UI**

{isAuthenticated && (

<div>

<h3>✅ Logged in as {user.username}</h3>

<div>JWT Token: {accessToken?.substring(0, 30)}...</div>

<button onClick={logout}>Logout</button>

</div>

)}

**🛠️ Webpack – Module Federation Setup**

new ModuleFederationPlugin({

name: "remoteCounter",

filename: "remoteEntry.js",

exposes: {

"./AuthProvider": "./src/AuthContext",

"./LoginComponent": "./src/LoginComponent",

"./Header": "./src/Header",

},

shared: {

react: { singleton: true, eager: false },

"react-dom": { singleton: true, eager: false },

},

}),

**🧠 Storage Responsibilities**

| **Storage Type** | **Purpose** |
| --- | --- |
| sessionStorage | JWT token + user info |
| localStorage | Refresh token (private) |

**🔁 Communication Events**

| **Event Type** | **Trigger Point** | **Description** |
| --- | --- | --- |
| AUTH\_TOKEN\_UPDATE | login, refresh, host req | Sends token/user to host app |
| AUTH\_LOGOUT | manual or auto logout | Notifies host app of logout |
| REQUEST\_AUTH\_TOKEN | host app → remote | Remote sends current token back |

**📌 Notes: JWT Communication Between Remote App and Host WordPress App**

## 🔐 Wordpress Auth Flow

JWT authentication flow between a **remote React microfrontend** (served via Module Federation) and a **WordPress host site** using postMessage and shared storage (sessionStorage, localStorage).

**🔄 Authentication Data Flow**

**✅ Remote to Host (on Login):**

1. Remote app (React) issues a JWT after login.
2. It uses postMessage to notify the WordPress host:

window.parent.postMessage({

type: 'AUTH\_TOKEN\_UPDATE',

payload: {

accessToken: 'JWT\_TOKEN\_HERE',

user: { username: 'auria', role: 'admin' },

source: 'remoteApp'

}

}, '\*');

1. WordPress host receives the message:

window.addEventListener('message', (event) => {

if (event.data.type === 'AUTH\_TOKEN\_UPDATE') {

const { accessToken, user } = event.data.payload;

wpAuthState.isAuthenticated = true;

wpAuthState.user = user;

wpAuthState.accessToken = accessToken;

saveWpAuthState(); // Persist in localStorage

updateWpAuthStatus();

}

});

**🚪 Remote to Host (on Logout):**

window.parent.postMessage({

type: 'AUTH\_LOGOUT'

}, '\*');

Host handles it:

if (event.data.type === 'AUTH\_LOGOUT') {

logoutWp();

}

**🔁 Host to Remote (State Sync):**

After restoring or changing auth state, the host proactively notifies remote components:

window.postMessage({

type: 'WP\_AUTH\_STATE\_UPDATE',

payload: {

isAuthenticated: wpAuthState.isAuthenticated,

user: wpAuthState.user,

accessToken: wpAuthState.accessToken

}

}, '\*');

**🧠 Storage Structure Summary**

| **Storage** | **Content** | **Purpose** |
| --- | --- | --- |
| sessionStorage | JWT Token + user info | For session-limited security |
| localStorage | Refresh token (private) | For persistent login across tabs/sessions |

**📡 On Token Expiration:**

* Host checks if JWT is expired during API call:

if (!stateToken || !wpAuthState.isAuthenticated) {

logoutWp();

return;

}

* If server returns 401/403, host triggers aggressive logout.

**🧪 Test API Call with JWT**

const response = await fetch('http://localhost:3002/api/user/profile', {

headers: {

'Authorization': `Bearer ${stateToken}`

}

});

**📦 Module Federation Bootstrap**

const container = window.remoteCounter;

await container.init(window.\_\_webpack\_share\_scopes\_\_.default);

const headerFactory = await container.get('./Header');

**📨 Summary of postMessage Types**

| **Type** | **Direction** | **Purpose** |
| --- | --- | --- |
| AUTH\_TOKEN\_UPDATE | Remote ➝ Host | Deliver JWT and user info |
| AUTH\_LOGOUT | Remote ➝ Host | Logout trigger |
| WP\_AUTH\_STATE\_UPDATE | Host ➝ Remote | Sync restored login state |
| NAVIGATE\_TO | Remote ➝ Host | Navigation command |

## ✅ How Navigation Works in Remote Header

Great question! Here's a breakdown of how **navigation** is handled via the **remote header** in your WordPress host integration, how the **URL changes**, and how **protected pages** work.

**✅ 1. How Navigation Works in Remote Header**

Remote React header component (from the MFE app) supports navigation. It sends navigation requests to the WordPress host via window.postMessage.

**🔁 Remote Header Code (onNavigate)**

onNavigate: (destination) => {

if (destination === 'dashboard') {

window.location.href = '<?php echo home\_url('/remote-component'); ?>';

} else if (destination === 'profile') {

window.location.href = '<?php echo home\_url('/profile'); ?>';

}

}

This means when you click something like "Profile" or "Dashboard" in the **remote header**, it sends a signal to the **host**, which updates the browser URL.

**✅ WordPress Host receives navigation**

Your WordPress host listens to this event:

if (event.data.type === 'NAVIGATE\_TO') {

const destination = event.data.payload.destination;

if (destination === 'dashboard') {

window.location.href = '<?php echo home\_url('/remote-component'); ?>';

} else if (destination === 'profile') {

window.location.href = '<?php echo home\_url('/profile'); ?>';

}

}

It performs **native WordPress page redirection** using window.location.href.

**🔐 2. How Protected Pages Work (Example: /profile)**

The /profile page is protected by **checking authentication status in JavaScript**.

**In wp\_profile\_shortcode():**

if (authState && authState.isAuthenticated && authState.user && authState.accessToken) {

// Show user profile

} else {

// Show unauthorized message and redirect option

}

This is done client-side:

* It checks the saved JWT in localStorage (wp\_auth\_state).
* If not found or invalid, it shows a red “Unauthorized” block.
* If valid, it renders the profile HTML dynamically.

**🔒 3. Navigation + Protection Flow Summary**

1. User clicks "Profile" in the **remote header**.
2. Header calls onNavigate('profile').
3. Host receives postMessage and redirects to /profile.
4. On /profile, JS checks:
   * wpAuthState
   * localStorage.wp\_auth\_state
5. If valid JWT exists, it shows profile; otherwise, shows error and login suggestion.

**🧠 Notes**

* **postMessage type:**
* window.postMessage({
* type: 'NAVIGATE\_TO',
* payload: { destination: 'profile' }
* }, '\*');
* **No routing library is used.** All navigation is done via full window.location.href reloads.
* You can expand this to support hash routing or even SPA behavior with History API, but currently it's simple full-page navigation.

## ✅ How the API Testing button is disabled WP

Inside the <script> block, you have this function:

function updateWpAuthStatus() {

const statusDiv = document.getElementById('wp-auth-status');

const apiButton = document.getElementById('wp-test-api');

// Safety check

if (!statusDiv || !apiButton) {

addWpLog('⚠️ Auth status elements not found, will retry...');

return;

}

// Get current auth state

const hasToken = wpAuthState.accessToken && wpAuthState.accessToken.trim() !== '';

const isAuthenticated = wpAuthState.isAuthenticated;

const hasUser = wpAuthState.user;

// ✅ If authenticated and token exists

if (isAuthenticated && hasToken && hasUser) {

// Enable button

apiButton.disabled = false;

apiButton.style.background = '#28a745';

apiButton.style.cursor = 'pointer';

apiButton.onclick = testWpAPI;

} else {

// ❌ Disable button if not authenticated

apiButton.disabled = true;

apiButton.style.background = '#ccc';

apiButton.style.cursor = 'not-allowed';

apiButton.onclick = null;

}

// Render the auth status info as well

// ...

}

**🔁 When This Function Runs**

The updateWpAuthStatus() function is triggered at several key points:

1. **After login message received via postMessage:**
2. if (event.data.type === 'AUTH\_TOKEN\_UPDATE') {
3. // set wpAuthState...
4. updateWpAuthStatus(); // 👈 called here
5. }
6. **On page load (in initializeWpAuth)**:
7. if (statusDiv) {
8. updateWpAuthStatus();
9. }
10. **After logout is triggered** (to immediately disable UI):
11. logoutWp(); // calls updateWpAuthStatus();

**🔐 Final Behavior Summary**

* ✅ If wpAuthState.accessToken exists and wpAuthState.isAuthenticated === true, the button is **enabled**.
* ❌ Otherwise, it's **disabled** and unclickable.
* CSS styles (gray background and not-allowed cursor) visually indicate its inactive state.

## 🧭 Typo3 Auth Flow

TYPO3 dashboard:

* Loads a remote React header and login module (via Module Federation).
* Shares authentication state (JWT + user info) via postMessage and sessionStorage.
* Reacts to login/logout to protect the API test functionality.

**🔄 2. Authentication Flow Step-by-Step**

**✅ a. Remote Auth Component Loads**

In loadRemoteComponents():

const loginComponentFactory = await window.remoteCounter.get('./LoginComponent');

const LoginComponent = loginComponentFactory().default;

const authRoot = ReactDOM.createRoot(document.getElementById('typo3-auth-component'));

authRoot.render(

React.createElement(AuthProvider, {},

React.createElement(LoginComponent)

)

);

This injects your **React login UI** into the TYPO3 container, mounted into #typo3-auth-component.

**📩 b. Login Emits Message with JWT**

Your LoginComponent (inside the React microfrontend) does this after successful login:

window.parent.postMessage({

type: 'AUTH\_TOKEN\_UPDATE',

payload: {

accessToken: 'jwt-token',

user: { username: 'john' }

}

}, '\*');

**🛡️ c. Host TYPO3 Page Receives This Message**

In the TYPO3 container page, you handle it here:

window.addEventListener('message', (event) => {

if (event.origin !== 'http://typo3.local') return; // ✅ Check origin

if (event.data?.type === 'AUTH\_TOKEN\_UPDATE') {

const { accessToken, user } = event.data.payload;

authState = {

isAuthenticated: true,

user,

token: accessToken

};

updateAuthUI(); // 👈 Enable API button, show user, etc.

}

});

**📦 d. Token Stored in SessionStorage**

Optionally, you may store it (if you want cross-tab/session support):

sessionStorage.setItem('jwt\_token', token);

sessionStorage.setItem('user\_data', JSON.stringify(user));

**🔁 e. UI Updates Based on Token**

In updateAuthUI():

if (authState.isAuthenticated) {

document.getElementById('typo3-test-api').disabled = false;

document.getElementById('typo3-test-api').onclick = testAPI;

} else {

document.getElementById('typo3-test-api').disabled = true;

document.getElementById('typo3-test-api').onclick = null;

}

So the API test button is **disabled until** a token is available and user is authenticated.

**🌐 3. Navigation Inside Remote Header**

const RemoteHeader = HeaderModule.default;

React.createElement(RemoteHeader, {

currentPage: 'dashboard',

onNavigate: (page) => {

if (page === 'profile') {

window.location.href = 'http://typo3.local/typo3-poc/prof';

}

}

});

This allows:

* Remote Header to **emit navigation events** via onNavigate.
* Host container (TYPO3 page) to **control routing** based on those events.

**🧠 4. Logout Flow**

If the user clicks logout in the microfrontend:

window.parent.postMessage({ type: 'AUTH\_LOGOUT' }, '\*');

Host handles it:

if (event.data?.type === 'AUTH\_LOGOUT') {

authState = {

isAuthenticated: false,

user: null,

token: null

};

updateAuthUI(); // ❌ Disables API button

}

Also clears sessionStorage to clean up token.

**🧪 5. Testing Protected API Endpoint**

Button uses authState.token:

fetch('http://localhost:3002/api/user/profile', {

headers: {

Authorization: `Bearer ${authState.token}`

}

});

Only works if a valid token is available.

**✅ Summary**

| **Feature** | **How It Works** |
| --- | --- |
| Remote Login | Rendered inside AuthProvider via Module Federation |
| Token Transfer | Done via postMessage from remote → host |
| Auth State | Stored in authState, synced via sessionStorage |
| API Test Button | Disabled by default, enabled only if authState.isAuthenticated === true |
| Navigation | Header emits onNavigate("profile") → Host redirects |
| Cross-tab Logout | Listens to storage event to detect logout elsewhere |

## ✅ How the Remote Header Works with the Host (TYPO3)

**1. Loaded as a Remote Component via Module Federation**

The host TYPO3 page loads the remoteEntry.js script from the React microfrontend:

const script = document.createElement('script');

script.src = 'http://typo3.local:3001/remoteEntry.js';

document.head.appendChild(script);

Then it uses Webpack Module Federation to load the actual component:

const headerFactory = await window.remoteCounter.get('./Header');

const RemoteHeader = headerFactory().default;

**2. Rendered with React in Host DOM**

const headerRoot = ReactDOM.createRoot(document.getElementById('typo3-remote-header-container'));

headerRoot.render(

React.createElement(RemoteHeader, {

currentPage: 'dashboard',

onNavigate: (page) => {

if (page === 'profile') {

window.location.href = 'http://typo3.local/typo3-poc/prof';

}

}

})

);

* RemoteHeader is a **React component** exported by the microfrontend.
* Host app passes:
  + currentPage to highlight the active tab (dashboard or profile).
  + onNavigate() as a callback to handle navigation **outside** the remote app.

**3. Inside the Remote Header**

The remote React app has buttons/links like this:

<button onClick={() => props.onNavigate('profile')}>Profile</button>

It **does not do navigation itself**. Instead, it *asks* the host:

“Hey, the user clicked Profile, what do you want me to do?”

The host receives 'profile' and redirects using:

window.location.href = 'http://typo3.local/typo3-poc/prof';

**🔁 Summary: How Clicking on "Profile" Works**

| **Step** | **What Happens** |
| --- | --- |
| 1️⃣ | RemoteHeader is mounted in host DOM |
| 2️⃣ | Host passes onNavigate(page) function |
| 3️⃣ | User clicks “Profile” in header |
| 4️⃣ | Remote app calls onNavigate('profile') |
| 5️⃣ | Host redirects using window.location.href |

✅ This keeps navigation logic **centralized in TYPO3**, while the header remains a **pure view-only** component.

## ✅ How the Profile Page Is Protected in TYPO3 Host

**🔐 Protection Logic (Pure JavaScript – no React needed)**

In the TYPO3 prof page, this block checks if the user is authenticated:

function checkAuthOnce() {

const token = sessionStorage.getItem('jwt\_token');

const userData = sessionStorage.getItem('user\_data');

if (token && userData) {

const user = JSON.parse(userData);

renderAuthenticatedProfile(user, token);

} else {

renderUnauthorizedProfile();

}

}

**🔒 What It Does:**

* If **jwt\_token** and **user\_data** are **NOT found** in sessionStorage, it renders:

renderUnauthorizedProfile();

That gives a red "Unauthorized Access" warning, along with “Go to Home” and “Refresh” buttons.

**✅ If token exists:**

* Calls renderAuthenticatedProfile(user, token) and loads actual user data like:
* <p><strong>Username:</strong> johndoe</p>
* <p><strong>Email:</strong> john@example.com</p>

## ✅ How the "Test API" Button Is Disabled Without JWT

This is part of the **Dashboard** page (not profile), in the updateAuthUI() function:

const statusDiv = document.getElementById('typo3-auth-status');

const apiButton = document.getElementById('typo3-test-api');

if (authState.isAuthenticated) {

apiButton.disabled = false;

apiButton.style.background = '#28a745';

apiButton.style.cursor = 'pointer';

apiButton.onclick = testAPI;

} else {

apiButton.disabled = true;

apiButton.style.background = '#ccc';

apiButton.style.cursor = 'not-allowed';

apiButton.onclick = null;

}

**✅ Explanation:**

* This updateAuthUI() function is triggered on:
  + First page load (via initializeAuth())
  + When JWT is received from remote app (AUTH\_TOKEN\_UPDATE)
  + When user logs out (AUTH\_LOGOUT)

**🔄 authState is updated like this:**

authState = {

isAuthenticated: true,

user,

token: accessToken

};

If no token is found:

authState = {

isAuthenticated: false,

user: null,

token: null

};

Then updateAuthUI() disables or enables the API button accordingly.

**🧪 Summary Table**

| **Condition** | **Effect on Profile Page** | **Effect on "Test API" Button** |
| --- | --- | --- |
| ✅ Has JWT | Shows profile content | Button enabled |
| ❌ No JWT | Renders "Unauthorized Access" | Button disabled, greyed out |