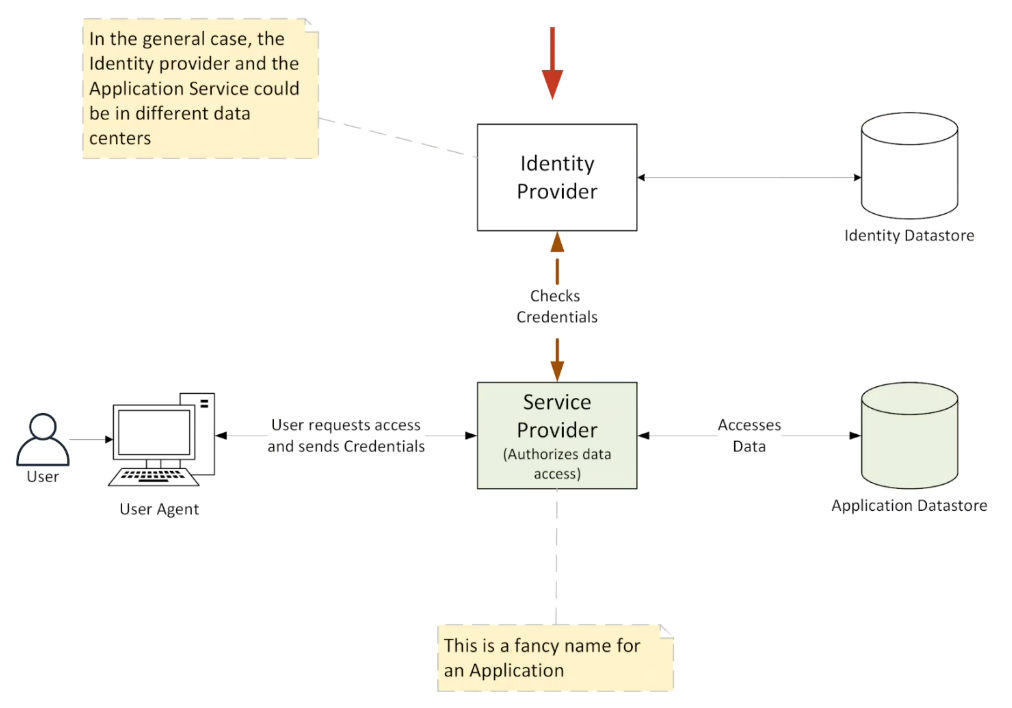
# Authentication and Authorization



In modern authentication and authorization systems, several key components work together to allow users to securely access applications. These components include the **Identity Provider (IdP)**, **Identity Store**, **Service Provider (SP)** (or application), **Application Data Store**, and the **Client/User Agent**. Here is a simplified explanation of how they fit together and the overall process.

**Key Components:**

1. **Client/User Agent**: This refers to the device or app (e.g., web browser, mobile app) that the user interacts with. The client sends requests on behalf of the user to the application they want to access.
2. **Identity Provider (IdP)**: This is a service (e.g., Keycloak, Okta, Microsoft ADFS) that handles authentication. The IdP holds the user’s credentials (passwords, certificates, etc.) and verifies their identity. It also issues tokens (like SAML or JWT) that confirm the user’s identity after they log in.
3. **Identity Store**: This is a database or directory (e.g., LDAP, Active Directory) where user information, like usernames and passwords, is stored. The IdP uses this to validate the user’s credentials.
4. **Service Provider (SP)** or **Application**: This is the application or service that the user wants to access (e.g., a company’s internal app, or a cloud-based service like Udemy). It relies on the IdP to authenticate users and determine what they are allowed to do in the app.
5. **Application Data Store**: This holds the application’s actual data (e.g., user profiles, content, transactions). The SP decides what data the user can access based on their authorization level.

**Simplified Process of Authentication and Authorization:**

**1. User Requests Access to the Application:**

* The user uses a web browser or mobile app (the client) to try to access the application (SP).
* The SP detects that the user has not logged in yet and does not know who they are.

**2. Redirect to Identity Provider (Authentication):**

* The SP sends the user to the **Identity Provider (IdP)** to log in. This redirection happens in the background. (browser redirection)
* The IdP asks the user for their login credentials (username and password) and checks them against the **Identity Store** (e.g., LDAP, Active Directory).

**3. Authentication by Identity Provider:**

* The IdP verifies the user’s credentials by checking the **Identity Store**.
* If the login is successful, the IdP creates a secure authentication token (e.g., a **JWT** or a **SAML assertion**) that proves the user’s identity.

**4. Redirect Back to the Service Provider (Authorization):**

* The IdP sends the user back to the application (SP) with the **authentication token**.
* The SP checks the token to confirm the user’s identity.

**5. Authorization Decision by Service Provider:**

* Once the SP knows who the user is, it checks what the user is allowed to do by looking at their **roles/permissions**.
* These permissions might be included in the token from the IdP or retrieved from the Identity Store.

**6. Access Granted:**

* If the user is authorized, the SP grants them access to the requested resources (like a profile page, transaction history, or specific data in the app).
* The user can now interact with the application and its **Application Data Store**, depending on their permissions.

**Common Protocols Used:**

1. **OAuth 2.0**: This is mainly used for **authorization**. It issues tokens that allow the client to access resources (like data or actions) on behalf of the user.
2. **OpenID Connect (OIDC)**: This is built on top of OAuth 2.0 and is used for **authentication**. It allows the application to get user profile information from the Identity Provider.
3. **SAML (Security Assertion Markup Language)**: This is a protocol used for **Single Sign-On (SSO)** between the IdP and the SP, commonly used in enterprise environments.

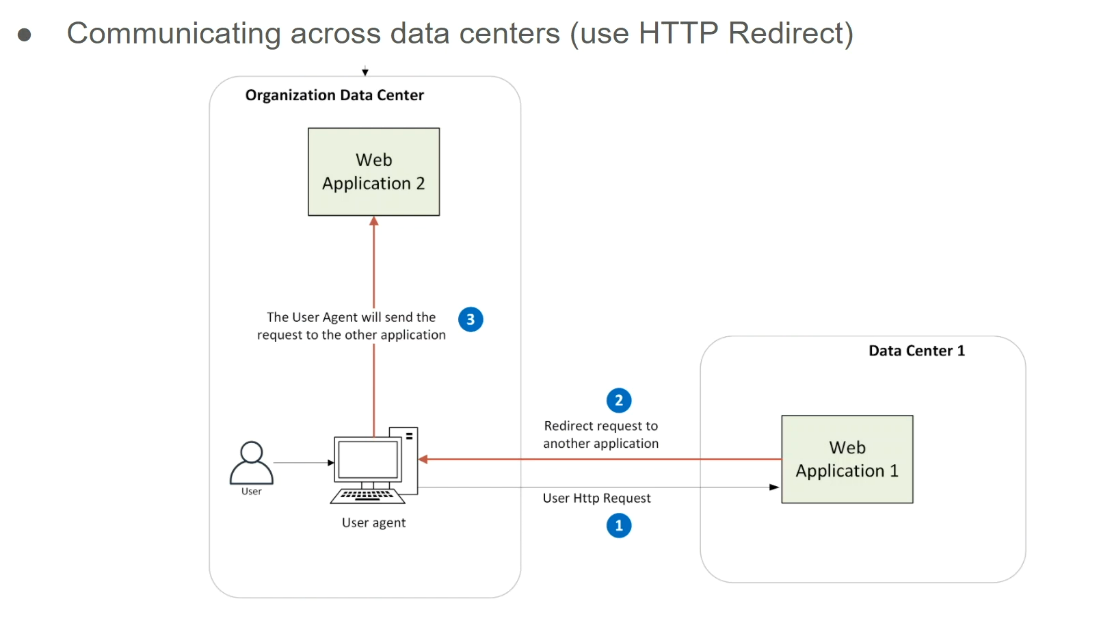
**Example Flow (Simplified):**

* A user tries to access an application (Service Provider).
* The SP sends the user to an Identity Provider (like Google or Okta) to log in.
* The Identity Provider checks the user’s credentials (e.g., against LDAP) and issues an authentication token (like a JWT).
* The user is redirected back to the SP with a SAML token.
* The SP validates the token, checks the user’s permissions, and allows access to certain data or actions based on those permissions.

**Why This is Important:**

This system provides a **secure** and **scalable** way for users to log in once (via Single Sign-On, or SSO) and access multiple applications without needing to enter credentials each time. It is commonly used in both corporate environments and across cloud-based services. The use of tokens ensures that the user’s identity is confirmed, and their data access is properly controlled by the application.

# SAML and Single Sign On



This write-up explains how **SAML (Security Assertion Markup Language)** addresses two key problems of **LDAP** when used for authentication, especially in a cloud environment, and how it enables **Single Sign-On (SSO)**.

**Problem 1: LDAP Requires Being in the Same Data Centre**

* **LDAP** (like Active Directory) stores user credentials, but it needs to be in the same **data centre** as the application that uses it for authentication.
* This becomes a problem in **cloud applications** because the cloud and on-premise data centres are often separate, making it difficult to use LDAP for cloud-based authentication.

**How SAML Solves Problem 1:**

* **SAML** uses a technique called an **HTTP redirect** to perform authentication **across different data centres**.
* Imagine two web applications: **Web App 1** (in a cloud data centre) and **Web App 2** (in the organization's data centre).
* **Direct communication** between the two apps is not possible, but they use the **browser** as a middleman. Here is how:
  1. The user sends a request to **Web App 1**.
  2. **Web App 1** needs to communicate with **Web App 2** for authentication. Instead of sending data directly, it tells the **browser** (user agent) to **redirect** the request to **Web App 2**.
  3. **Web App 2** processes the request and might redirect it further within its system if needed.
  4. Once done, **Web App 2** tells the **browser** to send the final response back to **Web App 1**.
* This **series of redirects** allows the two applications to communicate across data centres without having a direct connection.

**Problem 2: Users Must Manually Enter Credentials**

* In the traditional **LDAP setup**, users need to enter their credentials (username and password) every time they access an application, which can be cumbersome and raise security concerns.

**How SAML Solves Problem 2:**

* SAML leverages the fact that in many enterprises, users are already **logged into the corporate network** (e.g., when using a company laptop).
* Instead of asking the user to **manually enter** their credentials again, the **SAML Identity Provider** (IdP) recognizes that the user is already logged into the corporate network.
* The **SAML IdP** then authenticates the user **without needing credentials again** and passes the necessary user information to the application.

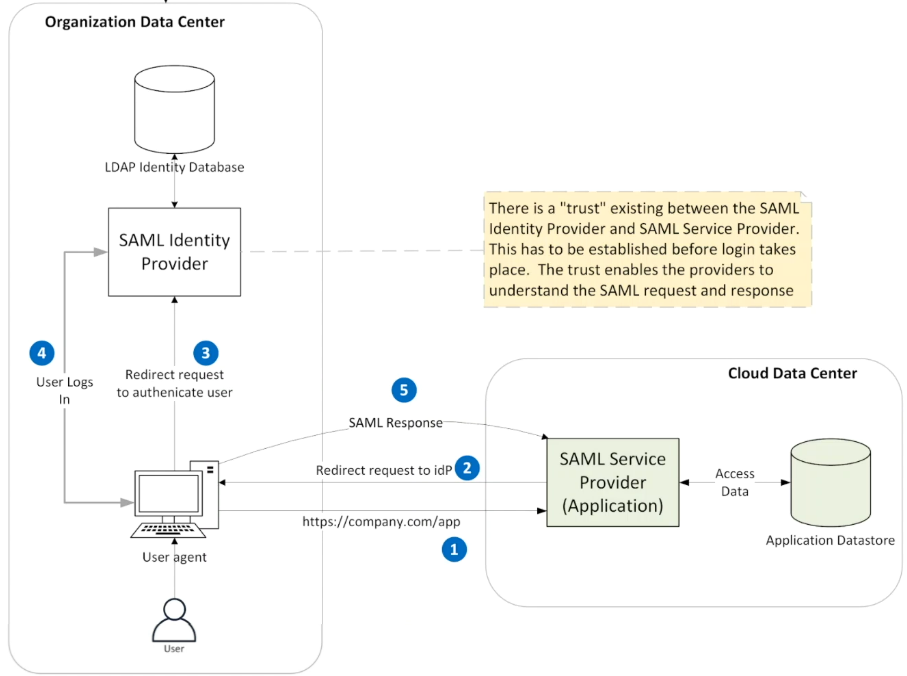
**Why This Is Helpful:**

* Users do not have to **repeatedly enter passwords**, which is convenient and more secure.
* **Single Sign-On (SSO)** is achieved, meaning the user logs in once and can access multiple applications without re-entering credentials.

**Key Takeaways:**

* **SAML** enables secure **cross-data-centre** authentication using browser-based **HTTP redirects**.
* It improves the user experience by eliminating the need for users to enter their credentials repeatedly, especially in corporate environments where they are already authenticated by **LDAP** or **Active Directory**.

# SAML Single Sign-On Flow



This write-up explains how **SAML (Security Assertion Markup Language)** works to enable **Single Sign-On (SSO)**, particularly when a **Service Provider (SP)** (like an application) is hosted in a cloud data centre, while the **SAML Identity Provider (IdP)** (like ADFS from Microsoft) and the user’s credentials are in the organization's corporate data centre. The communication happens through a secure redirection using the user’s browser, rather than a direct connection.

**Key Terms:**

* **SAML Identity Provider (IdP)**: This is the entity that authenticates the user, such as **ADFS** (Active Directory Federation Services) from Microsoft. It verifies the user's identity.
* **SAML Service Provider (SP)**: This is the application the user is trying to access, which is hosted in the cloud.
* **SAML Token**: A secure package containing user information (called **claims**) that is passed between the IdP and the SP.

**The Flow of SAML SSO:**

1. **User Requests the Application (SP)**
   * The user opens their browser and navigates to the Service Provider application (e.g., company.com/app).
   * The application does not recognize the user as logged in, so it needs to verify their identity.
2. **Application Redirects to Identity Provider (IdP)**
   * Since the application cannot talk directly to the SAML Identity Provider (IdP) due to being in separate data centres, it **redirects the user’s browser** to the IdP.
   * The redirect also contains a **SAML Request**, asking the IdP to authenticate the user.
3. **IdP Authenticates the User**
   * When the user’s browser is redirected to the IdP, the IdP recognizes that the user is already **logged into the corporate network** (e.g., via LDAP or Active Directory).
   * Since the user is already logged in, the IdP **automatically authenticates** them without asking for credentials.
4. **IdP Sends SAML Response with Claims**
   * The IdP creates a **SAML Response**, which contains a **SAML Token** with **user information** (called **SAML Claims**). These claims include details like the user’s name, email, and group memberships.
   * This SAML Response is **encrypted and signed** by the IdP to ensure security.
   * The IdP then **redirects** the browser back to the Service Provider (SP) with the **SAML Response**.
5. **Service Provider Validates the Response**
   * The SP receives the SAML Response via the user's browser.
   * To ensure that the response is valid, the SP uses the **certificates** from the IdP (which were exchanged earlier during the **trust setup**). The certificates allow the SP to:
     + **Decrypt** the SAML Response.
     + **Verify** the IdP’s signature.
   * Once the SP verifies everything, it uses the **SAML Claims** to **authorize** the user (e.g., deciding what actions they can perform in the application based on their role).

**Trust Between the IdP and SP:**

* For this entire process to work, there must be a **trust relationship** between the SAML Identity Provider (IdP) and the SAML Service Provider (SP).
* This trust is established by exchanging **SAML metadata**, which is basically an **XML file** containing important details like the certificates used for encryption and signature verification.
* Both the IdP and SP know each other's details through this metadata exchange, allowing them to securely communicate via the user’s browser.

**SAML Token and Claims:**

* The **SAML Token** contains the **claims**, which are pieces of information about the user (e.g., first name, last name, email, and group memberships).
* These claims help the Service Provider (SP) authorize the user within the application.

**Key Benefits:**

* **No need to re-enter credentials**: The user is already authenticated in the corporate network (LDAP/Active Directory), so they do not have to log in again.
* **Federated Identity**: The user’s identity is managed in one place (corporate LDAP/AD), but they can use the same identity to access different applications across multiple data centers.
* **Secure Communication**: The exchange of user information is encrypted and signed, ensuring that it’s secure.

**Simplified Summary:**

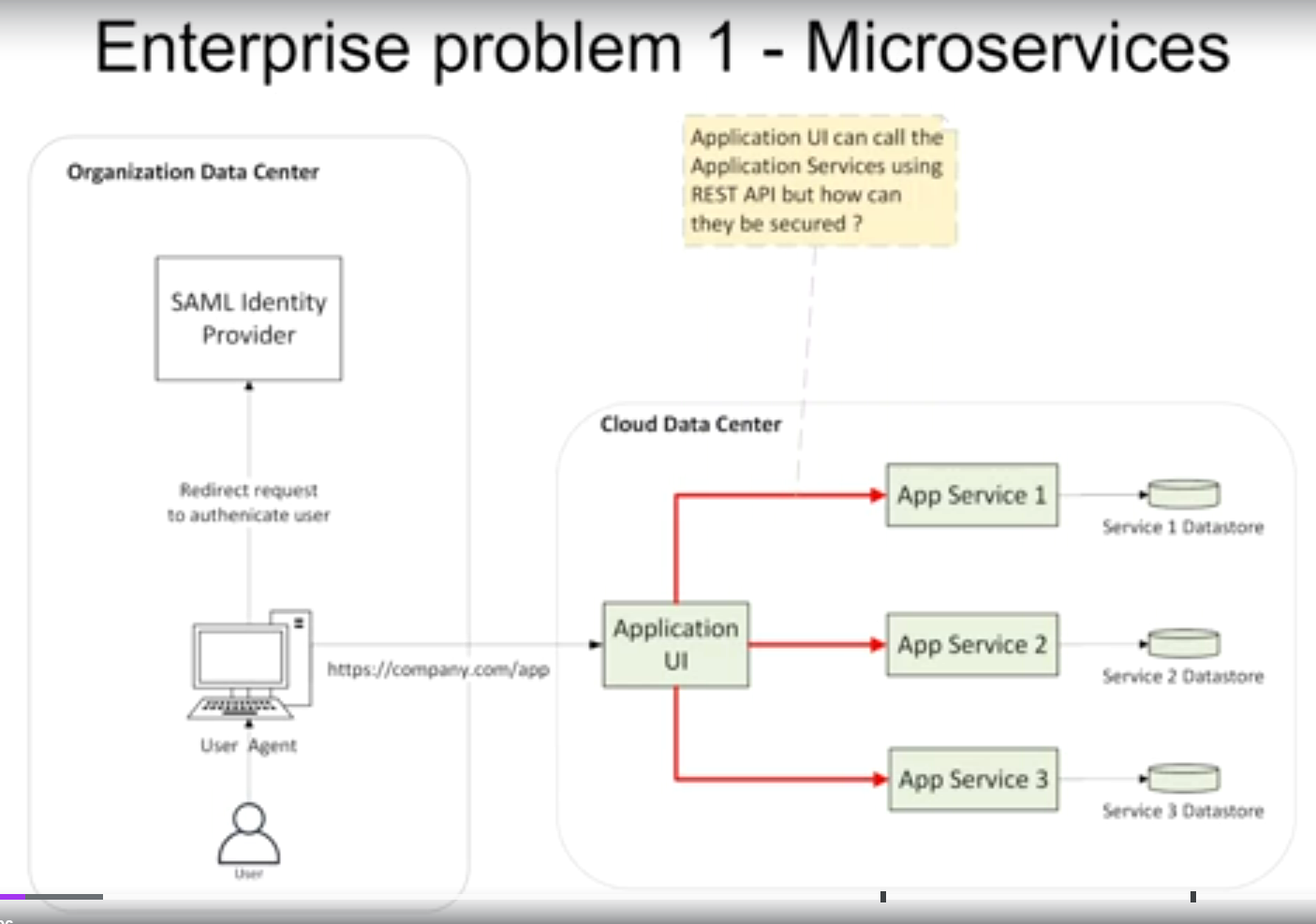
In SAML SSO, when a user tries to access an application in the cloud (the Service Provider), the application redirects the user’s browser to the organization's Identity Provider (like ADFS) to authenticate the user. Once authenticated, the Identity Provider sends the user information (SAML claims) back to the application using a secure, encrypted token. This process allows the user to access the application without entering their credentials again, thanks to the trust established between the Identity Provider and Service Provider. The user’s identity is stored centrally in the organization's LDAP, and this information can be reused across various applications in different locations.

# Enterprise Application Security and Problem Use cases

**What is SAML good at?**

SAML is designed for **Single Sign-On (SSO)**. It allows users to log in once and access multiple applications without logging in again. This is especially useful in environments where users access applications that are hosted in different locations (like a company's data centre or the cloud). SAML works well with web applications where a user logs in via their browser.

Use case -1

  
**The Problem with SAML**

SAML was not designed to handle modern architectures like **microservices** or **REST APIs**, which are widely used today. In these architectures, you have:

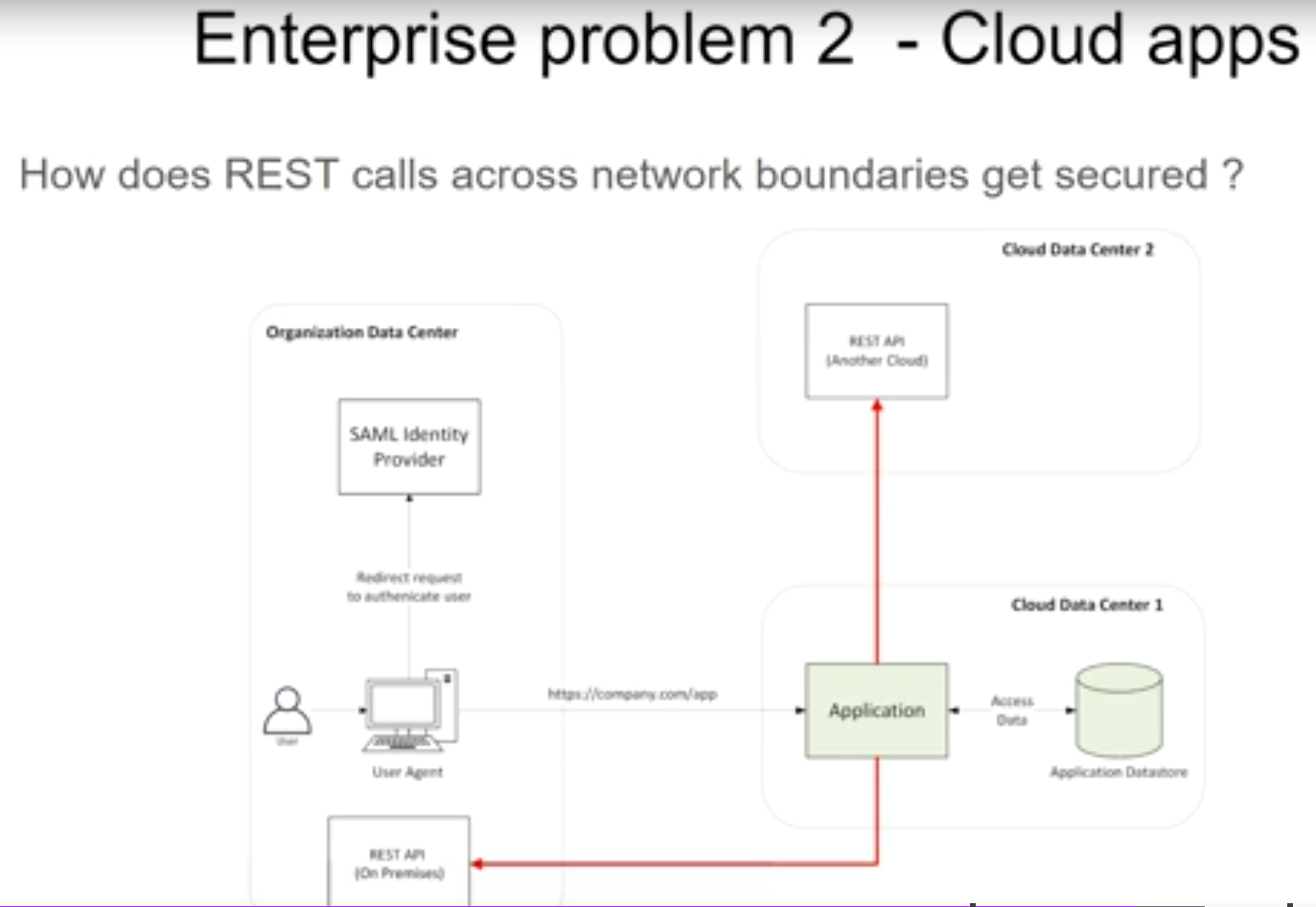
* **Multiple services (app service 1, 2, 3)** that need to talk to each other.
* **APIs** (Application Programming Interfaces) used by applications to request and send data between these services.

SAML can manage the login for the **main application (UI)**, but it struggles when:

1. **The application UI needs to call a service like app service 1, 2, or 3.**
   * The main application (UI) knows who the user is because it has authenticated them using SAML. But app service 1 does not know who the user is, and there is no way to verify the user’s identity when the application UI requests something from app service 1.
   * The **SAML token** (which confirms who the user is) cannot easily be passed around between services, because it is large and would slow down the system.
2. **Single Page Applications (SPA)**
   * If your app is a Single Page Application (built using JavaScript in the browser), then the browser itself is making calls to services (app service 1, 2, and 3). These services need a way to verify that the request is coming from the right user, and not someone trying to spoof the request.

**Use Case 1: Problem with Securing Microservices**

* Imagine the user has logged into the application UI using SAML, but now the UI needs to call app service 1.
* SAML does not provide a good way for app service 1 to **securely know who is calling** (because app service 1 does not talk to the SAML Identity Provider directly).
* The proposed idea of sending the SAML token every time a service is called would be inefficient because the token is large, and each service would need to have its own copy of the SAML metadata to verify it—this would create overhead and complexity.

Use case -2  


**Problem with Applications in Different Data Centres**

* Imagine an application is deployed in one data centre, and it needs to call a REST API in a different data centre (or within the same centre, but in a different environment).
* SAML does not provide a good way to secure these REST API calls. Traditionally, a user ID and password might be used, but that is not a scalable or secure solution for REST API communication.

**Cron Jobs and REST API Security**

* Sometimes, applications perform automated tasks using **cron jobs** (scheduled tasks that run at regular intervals). These tasks need to call REST APIs, but no user is involved.
* SAML, being user-centric, is not designed for these scenarios where **no user interaction** occurs. Again, there is no good way to secure the REST API calls made by these automated tasks using SAML.

**Common Theme: Securing REST APIs**

The main limitation of SAML is that it is good for **single sign-on for web applications**, but it **cannot handle securing REST APIs** across multiple services, especially in complex environments with microservices, cloud-based applications, and scheduled tasks.

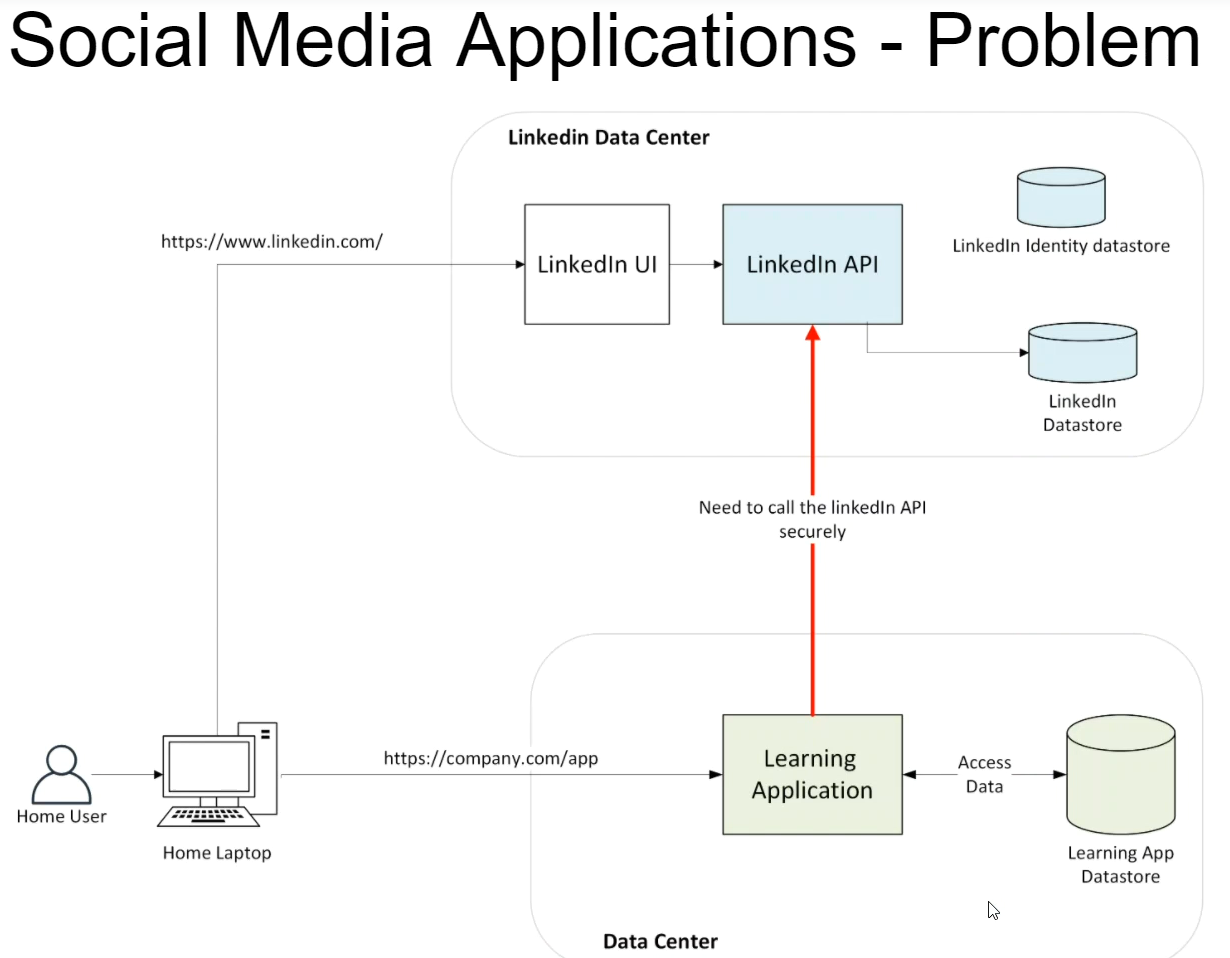
**What is the Solution?**

We need a different security mechanism to manage secure communication between services and REST APIs in such environments. Modern systems often use **OAuth 2.0** or **JWT (JSON Web Tokens)** to solve these problems:

* **OAuth 2.0** is designed to secure REST APIs and works well with modern microservice architectures.
* **JWTs** are lightweight tokens that can be passed between services and verified efficiently.

In summary, while SAML works great for logging into web applications, it struggles with securing REST API calls, and a new approach is needed for modern, distributed, microservice-based systems.

# Social Applications and its problems



This write-up discusses how external applications, like a **Learning Application** (e.g., Udemy), can interact with **social media platforms** (like LinkedIn) and the challenges involved in securely accessing a user's LinkedIn account.

**The Scenario**

1. **LinkedIn's setup**: LinkedIn has its own data centre, which includes:
   * **LinkedIn's API** (which allows external applications to interact with LinkedIn data).
   * **LinkedIn's UI** (what you see when you log in to LinkedIn through [www.linkedin.com](http://www.linkedin.com)).
   * **LinkedIn's Identity database** (where user identities are stored).
2. **User's setup**: The user has accounts on different platforms, like LinkedIn and a Learning Application. Each of these platforms has its own Identity data store (where users' identities and credentials are stored separately).

**The Use Case**

Imagine a user completes a course on a Learning Application and wants to **share the achievement** on their LinkedIn profile. The Learning Application would need to **call LinkedIn's API** to publish the user's certificate.

**The Problem**

The Learning Application **does not have access** to the user's LinkedIn credentials. Here is why:

* The Learning Application is **a third-party app**. From a security standpoint, it cannot and should not have direct access to the user's **LinkedIn credentials**.
* Asking the user to **enter their LinkedIn credentials** directly into the Learning Application is a bad practice because the user should only provide their credentials to LinkedIn itself—not to any third-party app like the Learning Application.

**The Solution: Authorization**

Instead of giving the Learning Application the user's credentials, there needs to be a way for the user to **authorize** the Learning Application to access LinkedIn on their behalf. This is where **OAuth 2.0** comes in.

**Simplified Explanation of OAuth 2.0:**

* **OAuth 2.0** allows users to **authorize** third-party applications (like the Learning Application) to access their data on other platforms (like LinkedIn) without sharing their credentials.
* The user logs into LinkedIn and **grants permission** for the Learning Application to interact with LinkedIn on their behalf.
* LinkedIn then issues a **token** to the Learning Application, which it can use to call LinkedIn's API securely.
* The Learning Application uses this token instead of the user's credentials to publish the course completion certificate to the user's LinkedIn account.

**Why This Matters**

This approach:

1. Ensures the user's credentials remain **private** and are only handled by LinkedIn.
2. **Secures** the connection between the Learning Application and LinkedIn, allowing the Learning Application to publish on the user’s behalf **without needing the user’s password**.

**Key Takeaways**

* Third-party applications (like a Learning Application) need a **secure way** to interact with a social media platform's API (like LinkedIn's API) without having direct access to the user’s credentials.
* OAuth 2.0 is the standard solution to this problem. It allows users to **authorize** third-party applications to perform actions on their behalf by granting them a **token** that represents their permission to access their account securely.