ChatGPT

# Extended Descope Lab Plan – Advanced Identity Scenarios

*Continuing from the initial 15-day (45-scenario) Descope lab plan, we will now cover advanced use cases in Days 16–21. Each day includes three hands-on scenarios using React (front-end) and Node.js (back-end) with Descope's SDKs. We'll explore topics like IdP-initiated SSO, SCIM user provisioning, multi-factor SSO flows, bot detection, test automation, Terraform DevOps, custom theming, user migration, passkeys, and more. As before, each scenario provides a brief conceptual background followed by step-by-step instructions (including any external account setup). Relevant Descope documentation and identity standard references are included for further reading.*

## Day 16: Advanced SSO and Identity Federation

### Scenario 16.1: IdP-Initiated SSO with Okta

**Concept:** In IdP-initiated SSO, the login flow starts from the Identity Provider (IdP) instead of the application. The user first authenticates into the IdP (e.g. Okta) and then launches the target application from the IdP's dashboard [1]. The IdP sends a SAML assertion or OIDC token to Descope (the Service Provider in this context), which then logs the user into your app. This contrasts with SP-initiated SSO, where the user begins at the app and is redirected to the IdP [2]. IdP-initiated flows provide a seamless "dashboard" experience for users who can access multiple apps from one portal (e.g. Okta home page) [3]. We will configure an Okta SAML integration for Descope and test IdP-initiated login from Okta.

**Steps:**

1. **Create an Okta Developer Account:** Sign up for a free Okta Developer tenant if you don't have one. Log in to the Okta admin console.

2. **Set Up a SAML Application in Okta:** In Okta, navigate to **Applications** and click **Create App Integration**. Choose **SAML 2.0** as the sign-in method. Give the app a name (e.g. "Descope React App") and proceed to SAML settings.

3. **Configure SAML Settings:** In Okta's SAML configuration, you'll need to supply values from your Descope project's SSO configuration (since Descope will act as the Service Provider):

4. **Single Sign-On URL (ACS URL):** This is the Assertion Consumer Service endpoint from Descope where Okta will post the SAML response. You can find it in your Descope Console under the tenant's SSO settings (the "SSO URL") [4]. It will look like `https://api.descope.com/v1/auth/saml/assertion/...` for your project and tenant.
5. **Audience URI (SP Entity ID):** Use Descope's SAML Entity ID (also in the SSO configuration settings) [4]. This is a URI that identifies your Descope application (often a URL or unique ID).
6. **Name ID Format:** Set to `EmailAddress` and configure the Name ID to use the user's email. Okta will send the user's email as the SAML NameID.

7. **Attributes (optional):** You can map user attributes if needed (for example, firstName, lastName) so Descope receives them. For basic setup, you may skip custom attributes.

8. **Finalize Okta App Setup:** Finish the Okta app creation. Assign some test users to this Okta application (under the **Assignments** tab) so that they can use it for SSO. Ensure the test user's email matches the Descope user if you have one, or be prepared for Just-In-Time creation of a new user on Descope side.

9. **Configure IdP in Descope:** In your Descope Console, make sure an SSO configuration (SAML) exists for the tenant corresponding to this Okta integration. You likely have already set this up in earlier labs for SP-initiated SSO. Verify that the **IdP SAML settings** in Descope (found under **Tenant > Authentication Methods > SSO**) are correctly filled:

10. **IdP Metadata:** Import Okta's metadata URL or certificate into Descope. (In Okta, under the SAML app's **Sign-On** tab, find the metadata URL or certificate for the IdP. Copy this into Descope's IdP configuration. Descope's SSO setup may allow you to upload the XML or provide the IdP details manually [5].)
11. **Domains:** Ensure the tenant's SSO domain includes the domain of your test user's email (e.g. if your Okta user is alice@acme.com, add `acme.com` as an allowed domain in the Descope tenant SSO settings). This tells Descope to route logins for that email domain to the configured SSO [6].

12. **Just-In-Time (JIT) Provisioning:** Enable JIT for this tenant if you want Descope to auto-create a user on first SSO login [7]. (If SCIM will be used, JIT can be off to avoid conflicts [8].)

13. **Test IdP-Initiated Login:** Log in to Okta as the test user. On the Okta user dashboard, you should see the newly created SAML app (Descope React App). Click the app icon to initiate login. Okta (IdP) will authenticate the user (if not already logged in) and then send a SAML Response to Descope's ACS URL with the user's info [9]. If everything is configured properly, Descope will consume this assertion and log the user into your React app:

14. The user's browser is redirected by Okta to Descope, and then your application. If using Descope's hosted flow or widget, the user will seamlessly be authenticated.
15. **Result:** The user lands in your app already logged in (Descope issues its session/JWT). You should see the user's profile via Descope SDK (e.g., `useUser()` hook) populated with data from Okta.

16. Descope will create a new user entry if it didn't exist (JIT provisioning), or update an existing user if a matching login ID is found and merging is enabled (see Day 17) [10] [11].

17. **Verification:** In Descope Console's **Users** or **Audit Trail**, verify that the login was recorded. The login method should indicate SAML SSO via the Okta IdP. If the user was created JIT, it will have a Descope user ID and profile linked to the tenant.

**References:** IdP-initiated SSO flow summary [1], differences from SP-initiated [2]. (Okta-specific setup is also documented on Okta's site and Descope's SSO docs.)

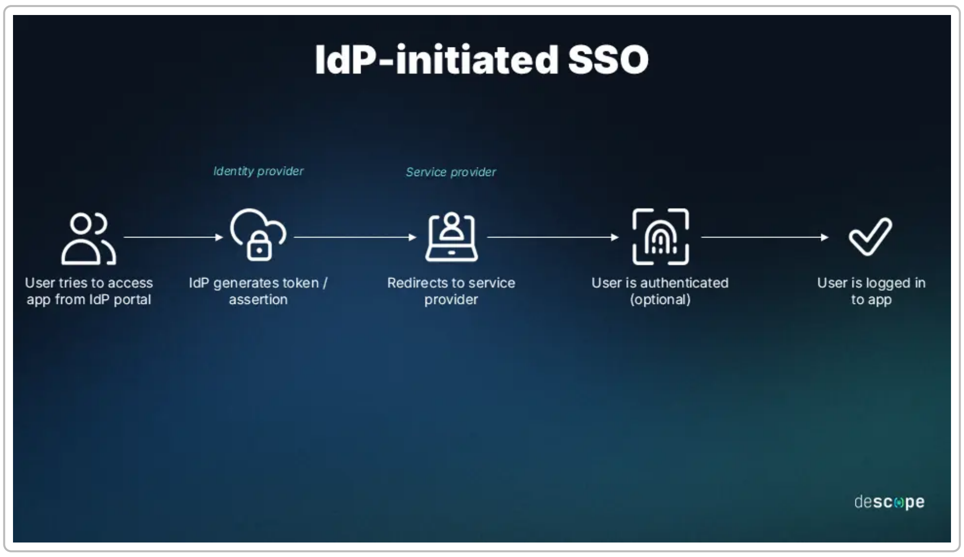
IdP-initiated SSO
Identity provider
Service provider
User tries to access app from IdP portal
IdP generates token / assertion
Redirects to service provider
User is authenticated (optional)
User is logged in to app
descope

*Fig: IdP-initiated SSO – the IdP (Okta) directly posts a token/assertion to the SP (Descope), logging the user in without them starting at the app* [1] .

### Scenario 16.2: Combining SSO with OTP (Multi-Factor SSO)

**Concept:** Even if users authenticate via SSO, there are cases where an additional factor is desirable. For example, an enterprise might want users to verify a one-time passcode if certain risk conditions are met (like logging in from a new device or after hours), or if the IdP's authentication wasn't MFA-enforced. Descope allows combining authentication methods in a single flow. Here we'll augment an SSO login with an OTP step, requiring the user to prove possession of their email or phone after SSO login. This essentially implements **step-up MFA** on top of SSO [12] . We will design a custom flow in Descope that first performs SSO, then triggers an OTP verification.

**Steps:**

1. **Plan the Flow:** The combined flow will work as follows: (a) User initiates SSO login (SP-initiated from your app or IdP-initiated as above) and successfully authenticates via the IdP. (b) Upon return to Descope, instead of finalizing the session immediately, the user is presented with an OTP challenge. (c) The user enters the OTP sent to their email or phone to complete login. We assume the user's email or phone is known (from the SSO assertion or stored in their profile).

2. **Create a Custom Flow in Descope:** In the Descope Console, go to **Flows** and create a new flow (e.g., "sso-with-otp"). Using the visual flow builder:

3. Drag an **SSO** action as the first step. Configure it to use your SSO integration (select the appropriate tenant or IdP configuration). This screen will redirect the user to the IdP if they aren't already authenticated.
4. Next, add a **Condition** node to decide if OTP is required. For simplicity, we will always require OTP in this scenario (in a real case, you could add logic here to check risk signals or user group). You can skip the condition and directly chain OTP for a mandatory step.
5. Add an **OTP Verify** action (One-Time Password) after the SSO step. Set it to deliver via email or SMS. For instance, choose "Send OTP to Email" and use the user's email (which should be available from the SSO step output as `authInfo.user.email`).

6. Finally, connect the OTP step to a **Finalize/Complete** node (or to whatever post-login screen or token issuance step is needed). Ensure the flow ends with issuing a session JWT once OTP is verified.

The flow might visually branch if you included a risk-based condition (e.g., if user IP is in risky range then OTP, else skip). Descope conditions can check things like `geo.country` or device fingerprint risk score [13] [14]. For this lab, we proceed unconditionally.

1. **Implement in React App:** Use the Descope React SDK to invoke this new flow. For example, in place of the normal SSO login button, you might trigger this custom flow:

   ```
   <Descope
     flowId="sso-with-otp"
     onSuccess={(e) => console.log('Login success', e.detail.user)}
     onError={(e) => console.error('Login error', e.detail.error)}
   />
   ```

   This will start the combined flow. Alternatively, you can have a separate route or component for step-up that the user is redirected to after SSO. Descope's hosted flow will handle the screens sequentially: first the IdP redirect, then upon return, an OTP input screen.

2. **Test the Combined Flow:** In a browser, initiate the "SSO with OTP" flow (e.g., by rendering the `<Descope flowId="sso-with-otp" />` component or redirecting to the hosted URL for that flow). Walk through:

3. The app (Descope SDK) redirects you to the IdP (Okta, Azure, etc.). Authenticate there.
4. Upon returning to the Descope flow, you should see an OTP challenge screen rather than being logged in immediately. Descope will have sent a 6-digit OTP to your email or phone (depending on how you configured the OTP step).

5. Enter the OTP code. If correct, the flow completes and you receive a Descope session/token, logging you into the app.

6. **Verify Multi-Factor Behavior:** Test using a valid code vs. invalid code to ensure only the correct OTP lets you in. Observe in the Descope Audit Trail that two authentication steps occurred: one SSO login and one OTP verification (both steps will be logged as part of the flow).

7. **Optional – Conditional MFA:** In practice, you might not want OTP for every SSO login. You can refine the flow using **Conditions**. For example, add a condition that checks if the user's device or location is new or untrusted (Descope provides `authInfo.firstSeen` or device fingerprint data to use in conditions [15] [12]). Only branch to OTP if the condition is met. If not, bypass OTP to complete login seamlessly for low-risk logins. This demonstrates **adaptive MFA** – step-up authentication only when risk is detected.

8. **Cleanup:** If you created this flow for demonstration, you can disable or delete it in the Descope Console when done. In a real app, ensure the flow is properly integrated into your login logic (perhaps as a policy for certain groups or an admin-only option for extra security).

**References:** Descope conditional flows and adaptive MFA [14] [12].

### Scenario 16.3: Descope as an Identity Provider (SAML Federation for External Apps)

**Concept:** Descope can not only consume identities from other providers – it can *be* an Identity Provider for third-party services. This is useful if you have a central user store in Descope and want your users to single-sign-on into other SaaS applications (for example, logging into an internal Docs portal, support system, or dashboard using their Descope credentials). In SAML terms, Descope acts as the **IdP**, and the external application is the **Service Provider (SP)** [16] [17]. We'll configure a federated SAML application in Descope (simulating a docs portal like Mintlify, which supports SSO) so that logging in via Descope grants access to that app.

**Steps:**

1. **Create a SAML Federated App in Descope:** In the Descope Console, navigate to **Applications -> Federated Apps** (this section might be available if your plan supports it) [18]. Click **+ Application** and choose **Generic SAML Application** (or a template if Descope provides one for common apps). Give it a name, e.g. "Docs Portal", and optionally an ID and description [19]. This represents the external SP in Descope.

2. **Obtain SP (Service Provider) Details:** From the external application (the docs portal or any SAML-capable app), collect its SAML configuration info. Typically you need:

3. **SP ACS URL (Assertion Consumer Service URL):** where the SP expects SAML assertions. For example, Mintlify would provide this in their SSO setup docs [20] [21].
4. **SP Entity ID (Audience URI):** a unique identifier for the SP (often a URL or URN).

5. **SP Certificate (if any):** some SPs provide a certificate to encrypt assertions or for signing. If the app (SP) gives you a **metadata URL or file**, you can use that instead for convenience.

6. **Configure Descope as IdP:** In the new Federated App settings in Descope, fill in the **Service Provider** details:

7. If available, enter the SP's Metadata URL to import settings automatically [22]. Otherwise, choose manual and input the ACS URL and Entity ID provided by the SP [23]. You can also upload the SP's certificate if required (or check "sign responses" if needed).

8. Take note of the **Descope IdP details** that are now generated for this app: Descope will show its **IdP Entity ID**, **SSO URL**, and a **Descope Certificate** for this application [4] [5]. These are critical for the SP side configuration.

9. **Configure the External Application (SP):** In the docs portal's SSO configuration page (e.g., Mintlify Enterprise SSO settings [24] or a generic Service Provider configuration):

10. **IdP SSO URL:** Enter Descope's SSO URL (the endpoint Descope will use to log users in) [25].
11. **IdP Entity ID:** Enter Descope's Entity ID from the app config.
12. **IdP Certificate:** Download the Descope certificate and upload it to the SP (so the SP trusts Descope's signatures).
13. **Attributes/Claims:** If the SP expects certain SAML attributes (like email, first name), ensure Descope will send them. In Descope's app config, you can map Descope user attributes to SAML attributes under **Attribute Mapping** [26]. For instance, map Descope's `user.email` to the SP's expected "Email" attribute.

14. Save the configuration on the SP side. In Mintlify's example, you might send the metadata to their support for them to enable it [27] [28].

15. **Test SP-Initiated Login:** Go to the external app's regular login URL and find the SSO option (e.g., "Login with SAML" or your company name). When initiated from the SP, it should redirect to Descope:

16. The browser will hit the SP, which redirects to Descope's IdP SSO URL (from step 3).
17. Descope will show the appropriate flow for authentication. For example, it might present a Descope-hosted login screen (you can design this flow in Descope – often you'd use a simple email/password or passkey, or any available method).
18. Authenticate as a Descope user. Upon success, Descope creates a SAML Response and posts it back to the SP's ACS URL.
19. The SP verifies the SAML Response (using the certificate) and logs the user in to the app.

If everything is configured correctly, you should be logged in to the external app with your Descope user. This effectively federates your Descope identity into the third-party application.

1. **Test IdP-Initiated (optional):** Descope also provides an **IdP-initiated URL** for each federated app [29]. This URL (usually listed in the app config screen) can be used to start the login from Descope's side. If you navigate to that URL, Descope will authenticate the user (if not already) and then immediately post a SAML assertion to the SP, achieving the same result. This is useful if you want to build an app launcher in your product.

2. **Tenant and User Considerations:** When Descope acts as IdP, the SAML assertion will include the user's identifying info. By default, Descope uses the user's Descope User ID as the NameID, but you can configure it to use email or another attribute if the SP expects it [30]. Also, roles can be mapped to SAML roles/groups if needed [26]. Ensure the user you test with exists in Descope and has any necessary attributes/roles that the SP might require.

3. **Wrap-Up:** We have successfully used Descope as the central Identity Provider for another service. This allows for SSO into external apps without those apps needing to manage user passwords – they trust Descope. In a real scenario, you might configure multiple such federated apps (e.g., one for an internal tool, one for a third-party SaaS). Descope's flexibility means you can be IdP, SP, or even a broker in between [31] [32].

**References:** Descope SAML IdP configuration options [4] [5], Federated Applications overview [16] [17].

# Day 17: User Provisioning and Account Management

## Scenario 17.1: SCIM User Provisioning from Okta to Descope

**Concept:** SCIM (System for Cross-domain Identity Management) is an open standard for automating exchange of user identity information between systems [33]. With SCIM, an enterprise IdP like Okta can **provision** and **de-provision** users into Descope automatically, ensuring Descope's user directory stays in sync with the company's directory [33] [34]. This is proactive provisioning, complementing or replacing Just-In-Time. In this scenario, we'll configure Okta to provision users and groups to Descope via SCIM. Whenever a user is created, updated, or removed in Okta (or assigned to the Descope app), Okta will call Descope's SCIM API to create/update or deactivate that user in Descope [35] [36].

**Steps:**

1. **Prerequisites:** Ensure you have SSO set up between Okta and Descope for your tenant (the Okta SAML app from Day 16) – SCIM in Okta requires an SSO app to be in place [37] . Also, in Descope create a dedicated **Access Key** with the "Tenant Admin" role for SCIM, scoped to the relevant tenant (this key will be used by Okta to authenticate to Descope's SCIM API) [36] .

2. **Enable SCIM in Okta App:** In Okta, go to the Descope application integration that you set up. Under the **General** tab, find the option **"SCIM provisioning"**. Check the box **Enable SCIM provisioning** and Save [38] . This will reveal a **Provisioning** tab for the app.

3. **Configure SCIM Connection in Okta:** Navigate to the **Provisioning** tab and click **Edit** in the SCIM section. Enter the following:

4. **SCIM Connector Base URL:** `https://api.descope.com/scim/v2` (the base endpoint for Descope's SCIM API) [39] .
5. **Unique Identifier for Users:** `email` (we use email as the unique key for matching users) [39] .
6. **Supported Actions:** Check **Push New Users**, **Push Profile Updates**, **Push Groups** (if you want to sync groups/roles) [39] .
7. **Authentication Mode:** HTTP Header.
8. **Authorization Header:** Here input a Bearer token: `Bearer <ProjectID>:<AccessKey>`. For example, if your Descope Project ID is `proj_123` and the Tenant Admin access key is `key_abc`, then: `Bearer proj_123:key_abc` [39] . This gives Okta the credentials to call Descope's SCIM API.

9. Click **Test Connector Configuration**. If everything is correct, Okta will show a successful connection test (meaning it can reach Descope's SCIM endpoint with the token) [40] . Save the settings.

10. **Enable Provisioning Actions:** Still in Okta's Provisioning tab, under **To App**, click **Edit** and enable the desired actions:

11. Check **Create Users**, **Update User Attributes**, and **Deactivate Users** [41] . These allow Okta to push adds/updates/deletes. (If you want to manage group assignments, you could also enable group push.)

12. Save these settings [41] .

13. **Push Groups (Optional Role Sync):** If you want Okta groups to map to Descope roles, you can set that up:

14. In Okta, go to the **Push Groups** section of the app. You can select specific groups to push to Descope.
15. For each pushed group, Okta will create or update a corresponding role in Descope. Descope interprets SCIM groups as roles by default [42] . (Ensure the names don't conflict with existing roles in Descope, or use a mapping strategy.)

16. In Descope, you can configure an **SSO Group Mapping** if needed to control how group names map to role names [42] – but by default, Okta group name = Descope role name.

17. **Test SCIM Provisioning:** Now, in Okta, assign a new user to the Descope app or create a new user in Okta and give them access:

18. When you assign a user to the app, Okta will trigger a SCIM **create user** call to Descope. The user should appear in the Descope Console under the tenant's users.
19. Edit an existing Okta user's profile (e.g., change their name) and watch Okta push an **update** via SCIM to Descope (the changes should reflect in the Descope user's attributes).
20. Deassign or deactivate a user in Okta and see Descope mark them as disabled (SCIM sets `active = false`). The user's `status` in Descope should become Inactive/Deleted as appropriate.

21. If group push is enabled, assign a group and ensure the user's Descope roles update accordingly.

22. **Validation on Descope:** Check the Descope tenant's user list. You should see users created with an "External" or "SCIM" indicator, possibly with their email, name, etc., filled in as Okta sent. The user entries may note they are managed by SCIM (Descope may restrict editing such users directly, since Okta is source of truth). Also check the **Logs/Audit** in Descope for SCIM events.

23. **Disable JIT (if using SCIM exclusively):** If SCIM is fully provisioning users, it's recommended to **disable JIT** for that tenant in Descope to avoid duplicates or attribute conflicts [43] [8]. (JIT was an alternative path – we typically use one or the other to manage users.) In Tenant Settings, ensure "Just-In-Time Provisioning" is off for that tenant.

24. **Result:** Going forward, any changes in Okta propagate to Descope automatically. This reduces manual work – when an employee joins or leaves, you handle it in Okta, and Descope updates behind the scenes [44] [33]. Okta runs provisioning on a schedule (e.g., every 40 minutes) for bulk updates, but direct assignments via the UI may sync immediately. You can also force a sync in Okta ("Provision Now" option) to test changes on demand.

**References:** Descope SCIM setup for Okta [39] [45], SCIM standard overview [33].

## Scenario 17.2: SCIM Provisioning with Azure AD (Microsoft Entra ID)

**Concept:** Many enterprises use Azure AD (now called Microsoft Entra ID) as their IdP. Azure AD can likewise provision users into Descope via SCIM. The setup is similar to Okta's but with Azure's interface. One key difference: Azure AD's SCIM client typically runs periodically (approx. every 40 minutes) and uses a "Secret Token" for authentication rather than a raw header. We will configure Azure AD to automatically create, update, and deactivate users in Descope using SCIM [46] [47].

**Steps:**

1. **Prerequisites:** As before, ensure SSO integration exists for Azure AD in Descope (i.e., you have a SAML/OIDC config for Azure AD logins) [48]. Also have a Descope Tenant Admin access key ready. In Azure, you should have created an **Enterprise Application** for your Descope integration (the SAML app). Users and/or groups should be assigned to this enterprise app in Azure AD.

2. **Open Provisioning Settings in Azure AD:** In the Azure Portal, go to **Azure AD > Enterprise Applications** and select your Descope app integration. In the left sidebar, click **Provisioning**. Then click **Get Started** if not already set up [49].

3. **SCIM Endpoint Configuration:** Set **Provisioning Mode** to **Automatic** [50]. Then enter:

4. **Tenant URL:** `https://api.descope.com/scim/v2` (same SCIM base endpoint) [51].
5. **Secret Token:** Here, Azure expects a token to connect. Provide the `ProjectID:AccessKey` (with Tenant Admin role) as a single string. For example, `proj_123:key_abc` (Azure will store this securely). This is effectively the basic auth for SCIM, analogous to the header Okta used [51].

6. Click **Test Connection**. Azure will verify it can reach Descope's SCIM endpoint with the token [52]. If successful, **Save** the configuration.

7. **Attribute Mappings:** After saving, you'll see **Mappings** for Users and Groups:

8. Azure automatically maps standard attributes like `userPrincipalName` to `userName`, `givenName` to first name, `surname` to last name, etc., based on SCIM schema [53]. Review these under **Provisioning > Mappings > Synchronize Azure Active Directory Users to CustomApp**. Adjust if needed (for example, ensure the `mail` attribute is mapped to emails in Descope).
9. For **Group mappings** (if you want to sync groups to Descope), Azure can send group display names. Azure's defaults might not include group -> role by default; you may need to enable it under the **Synchronize Azure AD Groups** mapping. Ensure the "Group object" mapping is enabled and configured to send group name or ID to Descope's groups [54].

10. If you want to filter which users or groups sync, Azure allows scoping by attribute or assigned-only. By default, it will provision all users assigned to the enterprise app.

11. **Start Provisioning:** Back in the **Provisioning** blade, set **Provisioning Status: On** and Save. Azure will begin the initial cycle. You can also click **Provision on demand** to test a specific user [55].

12. When provisioning starts, Azure AD will create any new users (that are assigned to the app) in Descope. It will also push group memberships if configured.

13. Azure's interface shows the sync status and any errors for each user. Check the **Audit Logs** in Azure AD for provisioning events to confirm success.

14. **Verify in Descope:** The users from Azure should now exist in Descope's tenant:

15. Check the **Users** section for new entries. The usernames will likely be the Azure user principal names or emails. Attributes like name and email should be populated as per mappings.
16. If group sync was on, check Descope **Roles** or tenant roles – you should see roles corresponding to Azure AD groups, and users associated with those roles (Descope sees SCIM groups as roles) [42] [54].

17. If a user was updated or removed, verify those changes reflect. (Azure might mark the Descope user as inactive on deletion.)

18. **Test End-to-End:** Now that Azure AD is pushing users, test an SSO login from Azure AD into Descope (if you have not already). The difference you'll notice with SCIM is that the user likely pre-exists in Descope before their first login (thanks to provisioning). During SSO login, since the user is already there (and JIT is likely off to avoid duplication [8]), Descope will simply map the incoming SAML assertion to the existing user and update any mapped attributes. The login

should succeed as usual, but now the user's life cycle (creation, attribute updates, deactivation) is controlled by Azure AD.

19. **Deprovisioning and Sync Cycles:** Azure AD runs SCIM provisioning periodically (around every 40 minutes). If you remove a user from the app or disable them in Azure AD, on the next cycle Azure will send an update to mark that user as inactive in Descope (or delete, depending on how Descope handles `active=false`). Verify that removing an Azure assignment indeed prevents the user from logging in (Descope should reject if the user is marked inactive). Note that immediate removals might not happen until the sync occurs (Azure's on-demand provision can remove individually if needed).

20. **Troubleshooting:** If some attributes aren't syncing, check mapping configuration. Azure's SCIM implementation might require the SCIM server to support certain filtering or PATCH operations – Descope's SCIM should handle basic CRUD. The Descope SCIM API base URL and token must be correct; if test connection fails, regenerate the access key or check formatting.

**References:** Azure SCIM setup in Descope docs [56] [57], prerequisites and JIT note [47] [8].

## Scenario 17.3: User Identity Merging and Conflict Handling

**Concept:** When the same person can authenticate via multiple methods (e.g., an enterprise SSO and a personal email login), there's potential for duplicate accounts. *User merging* refers to linking or unifying such accounts. For example, if *alice@acme.com* already has a Descope account (from email/password or OTP signup) and later Acme configures SSO, Alice might end up with a separate SSO account unless merging is handled. By default, Descope **does not automatically merge SSO identities with non-SSO identities sharing the same email** [58]. This is to preserve security – SSO accounts are usually tied to enterprise controls, and letting a user bypass SSO via a personal login can introduce risks [59] [60]. However, in some scenarios you may want to intentionally merge or allow a backup login. Here we discuss how to manage merging and the trade-offs.

**Steps/Strategies:**

1. **Understand Default Behavior:** If a user logs in with SSO and also with a non-SSO method (like OTP or password) using the same email, Descope will treat them as separate users unless you explicitly link them [61] [58]. Each will have its own user ID. This means things like roles or profile data won't automatically transfer between these two entries. The reason is primarily security and clarity: SSO user sessions must obey IdP revocation and policies, which could be undermined if the user could switch to a local login [59] [60].

2. **Option 1 – Enforce SSO (No Merge):** The simplest approach in enterprise scenarios is to *not allow any non-SSO logins for those users*. In Descope, you can **Enforce SSO** for a tenant, meaning users with that domain must use SSO only and cannot use other methods [62]. This avoids duplicate accounts because the user will never create a personal login in the first place. This is most secure and is the default recommended approach [63] [64]. In this mode, merging isn't needed – the user sticks to their SSO identity managed by the IdP.

3. **Option 2 – Automatic Merging by Email:** Descope provides a setting (in project or tenant config) to **merge users by login ID** for SSO logins [65]. If enabled, when an SSO assertion comes in, Descope will check if a user with the same email (login ID) already exists in that tenant. If yes, instead of creating a new user, it will attach the SSO identity to the existing user record [10] [66].

This way, "Alice" remains one user in Descope, with potentially multiple login methods. You can enable this in code or Terraform by setting `authentication.sso.merge_users = true` for the project [65]. **Use with caution:** Merging blindly by email can be risky if, for example, a personal email coincidentally matches a corporate email or if the account was compromised. Only enable if you're confident that the pre-existing account truly belongs to the same person coming via SSO.

4. **Manual Linking via Management API:** Another approach is to link identities explicitly:

5. Descope supports adding additional login IDs to a user (e.g., you can add a secondary email or phone, or even a federated ID, to an existing user via the Management SDK).
6. For example, if Alice has a local account and now SSO is available, you could programmatically update Alice's Descope user to include an SSO federated ID or tenant association so that a login via SSO maps to her account. This usually involves using Descope's Management API to attach SSO identity info to the user or vice versa.

7. This method is complex and typically not needed unless migrating (see Day 21). It's better to use the merge setting or avoid multiple methods.

8. **Exercise – Toggle Merge Setting:** In your Descope project settings (or via Terraform), try enabling the **Merge Users** option:

9. Navigate to **Project Settings > Authentication > SSO** and look for an option like "Merge SSO Users with existing users by login ID" (the wording may vary). If the console UI doesn't expose it, assume the default is off and we enable via API/Terraform.

10. Using Terraform for instance:

```
resource "descope_project" "proj" {
  // ... other config ...
  authentication = {
    sso = {
      merge_users = true
      # other SSO settings...
    }
  }
}
```

Apply this change [65].

11. Now, simulate the scenario: Have a user that exists via a non-SSO method (e.g., create user jane.doe@example.com with a password). Then have the same person attempt SSO login (for example, via Google SSO if that domain is configured).

12. With merge enabled, Descope will recognize "jane.doe@example.com" already exists and will **not create a new user**. Instead, it will log her into the existing account and mark that account as having logged in via SSO. You can verify this by checking that the user's Descope User ID remains the same; no duplicate entry is added.

13. **Observe Security Implications:** After merging, note that Jane Doe can now log in either via the corporate SSO or the original method. This might be convenient, but as highlighted:

14. If her password/email got hacked, an attacker could bypass SSO restrictions (MFA, etc.) [67].
15. If the IT team disables her in the IdP, but she still has a local password set, she could potentially still log in unless you also disable or monitor that route [60].

16. Audit logs might show logins from two methods for one user, which could be confusing for compliance [68]. In regulated environments, this is usually avoided. However, for *certain admin users*, you might allow a backup login path (e.g., if the IdP is down, an admin can use a break-glass account) [62]. Descope conditions can enforce SSO normally but allow an alternate login if SSO is unreachable or for specific roles [62] [69].

17. **Communicate & Document:** If you choose to merge identities, ensure it's documented and users are aware of how they can log in. Also, keep an eye on audit trails. Descope will log when a user's identities are merged or when one user ID has multiple login methods.

18. **Reversing Merge:** If needed, you cannot truly "unmerge" in Descope (since it was a single user all along). But you could remove one of the login methods (e.g., delete the password credential) to re-separate the access. It's simpler to decide the strategy up-front (either keep separate or merge) to avoid this.

**Conclusion:** For most enterprise apps, **not merging** and enforcing a single IdP per user is safest [63] [64]. But if merging is required, Descope provides the tools to do it thoughtfully. Always ensure that merging doesn't undermine security controls from the IdP.

**References:** Descope's stance on SSO vs non-SSO identities [61] [58], risks of mixing methods [59] [60], merge_users config flag [65].

## Day 18: Security Features and Access Controls

### Scenario 18.1: Bot Detection and Risk-Based Authentication Flows

**Concept:** Not all login attempts are equal – some may be malicious (bots, scripts) while others are legitimate users. Descope's flow builder lets you incorporate **risk signals** and **bot detection** to adapt your auth flow in real-time [70] [71]. For example, you can integrate device fingerprinting to identify headless browsers or known fraud patterns, then enforce additional checks (like CAPTCHA or MFA) for high-risk logins [71] [72]. Descope has built-in **Conditions** and supports third-party risk feeds (e.g., FingerprintJS, Forter, reCAPTCHA) to help implement **adaptive authentication**. In this scenario, we'll enhance a login flow to detect bot-like behavior and trigger a step-up challenge only for suspicious attempts.

**Steps:**

1. **Integrate Device Intelligence (Optional):** For advanced bot detection, you can use **Descope's FingerprintJS connector**. FingerprintJS provides a persistent device ID and bot score. Descope's connector can capture a visitorID from the client and fetch risk signals (like "automation tool detected", "VPN or Tor usage", etc.) [73] [70]. Setting this up:
2. Sign up for a free FingerprintJS account (fingerprint.com) and get an API key.

3. Add the FingerprintJS snippet to your React app (this typically generates a `visitorId`). Ensure you capture this ID before the user begins login.
4. Pass the visitorId to Descope: One way is to include it as a **context variable** when starting a Descope flow. For instance, Descope's flow initiation API or SDK might allow passing custom parameters (check Descope docs for how to attach the fingerprint ID).
5. On the Descope Flow side, use a **Fingerprint / Get Event** action (Descope provides a template flow action for Fingerprint) [74] which takes the visitorId and retrieves the detailed device info and risk signals.
6. **Add Conditions in the Flow:** In the Descope flow builder for your login/signup:
7. Drag a **Condition** node after the initial authentication step (e.g., after "Sign In" action, but before issuing tokens). This condition will decide whether to treat the session as low-risk or high-risk.
8. Configure the condition rules: e.g., use `event.riskScore` from Fingerprint's data (if available). Say Fingerprint provides a bot score from 0-1; you might set: **If riskScore > 0.8 (high risk)** or if **device is headless == true**, etc. Descope's dynamic values would have those fields if the Fingerprint event is attached.
9. Alternatively, without Fingerprint, use Descope's own available signals: For example, a condition on `geo.country` (if you want to challenge logins from abroad) [75] , or `authInfo.firstSeen` (to challenge new devices) [15] , or check if the IP is in a blacklist (Descope has an IP address condition) [76] .
10. The condition node will have two branches: **True** (risk detected) and **False** (no risk).
11. **Branch the Flow:** Connect the **False** branch (no risk) directly to the end of the flow (issue token, login complete). For the **True** branch (high risk or bot suspected), insert additional verification steps:
12. For example, add a **reCAPTCHA challenge**: Descope can integrate Google reCAPTCHA Enterprise or other CAPTCHA providers [77] . You might have an action or screen that presents a CAPTCHA to the user. Only if they pass, proceed.
13. Or, require **MFA** for high-risk cases: e.g., add an **OTP verification** or **TOTP** step on the True branch. This becomes a step-up MFA invoked only if the risk condition is met [72] .
14. You could even route to a denial: e.g., if you are certain it's a bot, route to a **"Access Denied" screen** and end the flow there (blocking the login entirely) [78] .
15. **Implement Bot Defense in React:** Ensure your front-end is supplying needed data:
16. If using reCAPTCHA, you might need to include a CAPTCHA widget that gets solved and then validated by Descope (Descope flows can include a CAPTCHA screen).
17. If using FingerprintJS, as mentioned, pass the visitorId into the flow. For Descope React SDK, this could mean initiating the flow via an API call with the ID. Alternatively, you might send the ID to your backend and call Descope's Flow Execution API including that context.
18. **Test Low vs High Risk:** Simulate two login attempts:
19. **Legitimate user scenario:** Use a normal browser in a typical environment. The device likely isn't flagged. Descope's condition should fall through to the False branch (no extra steps). The user logs in without friction.

20. **Bot scenario:** Simulate by using a headless browser or a tool. For a quick test, you could open a browser dev console and set the user agent to a headless one, or use an automated script via Playwright/Puppeteer to attempt login. If FingerprintJS is integrated, it will likely flag this (e.g., `botd` signal). Descope condition then takes the True branch. You should see the flow present the CAPTCHA or OTP step. Since it's a script, it cannot easily proceed, effectively stopping the bot. If it's a real user just in an odd situation, they still have a way through by completing the extra step (MFA or CAPTCHA).

21. **Adjust and Tune:** You can refine the risk logic. Descope's **Conditions** support combining multiple checks (they evaluate a list of conditions) [79] [15]. For example, *if (new device OR high IP risk) AND user has high-privilege role, then enforce MFA*. You can get creative to minimize user annoyance but maximize security. Also consider maintaining a "trust" cookie or device binding: once a user passes MFA on a device, you might mark it trusted (set a flag in their user metadata) and have the condition skip MFA for that device in the future.

22. **Log and Monitor:** All these conditional decisions can be logged. Use Descope's logs to monitor how often logins are marked risky vs not. Fine-tune thresholds if needed (maybe too many false positives requiring MFA).

In summary, this scenario shows how to **bolster your app's defenses against bots and fraud** by injecting risk-based steps in your auth flow. The user experience adapts on the fly: good users sail through, suspicious ones get challenged or blocked [72] [80].

**References:** Descope + Fingerprint integration (device intelligence) [70] [71], using conditions for adaptive flows [81] [14].

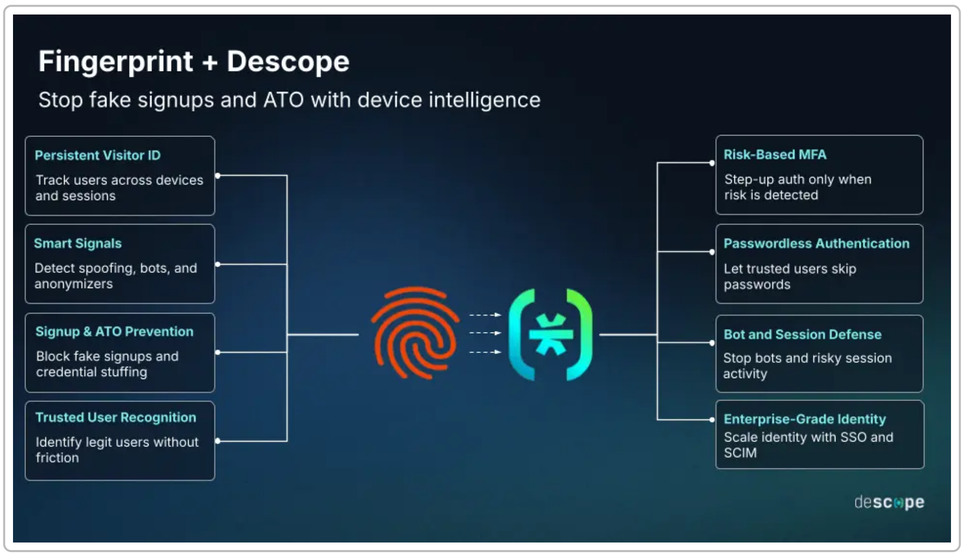
Fingerprint + Descope
Stop fake signups and ATO with device intelligence
Persistent Visitor ID
Track users across devices and sessions
Smart Signals
Detect spoofing, bots, and anonymizers
Signup & ATO Prevention
Block fake signups and credential stuffing
Trusted User Recognition
Identify legit users without friction
Risk-Based MFA
Step-up auth only when risk is detected
Passwordless Authentication
Let trusted users skip passwords
Bot and Session Defense
Stop bots and risky session activity
Enterprise-Grade Identity
Scale identity with SSO and SCIM
descope

*Fig: Device intelligence integration example – FingerprintJS gathers signals (left) and Descope uses them to drive risk-based outcomes like step-up MFA or bot defense (right)* [71] [77].

## Scenario 18.2: Role-Based Access Control (RBAC) in JWTs and Backend Enforcement

**Concept:** Descope allows you to assign **roles** and **permissions** to users and include these in the issued JWT (JSON Web Token) for the session [82]. This means the token your React app or Node.js backend receives can carry the user's roles (and tenant roles), which you can use to enforce authorization. In this scenario, we explore how to include RBAC info in JWTs and, importantly, how to safely enforce it on the backend. We also discuss limitations: since JWTs are client-visible, they can be decoded by the user, so you must always validate them and be cautious about long-lived permissions in tokens.

**Steps:**

1. **Define Roles and Permissions in Descope:** If not already set up, create some roles in Descope. For example, a "BasicUser" role and an "Admin" role. You can do this in **Authorization** settings or via Terraform:

    ```
    authorization = {
      permissions = [
        { name = "read_reports", description = "Can read reports" },
        { name = "edit_reports", description = "Can edit reports" }
      ]
      roles = [
        { name = "BasicUser", description = "Regular user", permissions =
    ["read_reports"] },
        { name = "Admin", description = "Administrator", permissions =
    ["read_reports", "edit_reports"] }
      ]
    }
    ```

    (This is an example; adjust to your app's needs.) [83] [84] . Apply these changes so Descope knows about these roles/permissions.

2. **Assign Roles to Users:** Through the Descope Console or Management API, assign the appropriate role to each user. For instance, mark your test user as "Admin". If you use tenants, roles can be tenant-specific; assign roles within the tenant as needed. You could also have gotten roles via SCIM (e.g., Okta groups mapping to roles, from Day 17) – those roles would be attached to the user already [42] .

3. **JWT Configuration:** By default, Descope's user JWT includes the roles and permissions in its claims. The default format might have a `roles` claim with a list of project roles, and if multi-tenant, a `tenants` claim that includes roles per tenant [82] . You can confirm this:

4. In Descope Console, under **Project Settings > JWT Templates**, check the default user JWT template. It likely has "Default Descope JWT" structure which includes roles in the payload [85] .

5. If needed, you can customize it (for example, to exclude some data or change claim names), but for now we accept the default.

6. **Front-End Usage (Limited):** While you *could* use the roles in the JWT on the front-end to conditionally render UI (for example, show admin menu if JWT contains "Admin"), remember that the front-end is not fully trustable. A user could inspect or even swap their JWT if they manage to compromise storage. It's okay for convenience (non-sensitive UI cues), but never trust it for serious checks purely on the client.

7. **Back-End Enforcement:** This is crucial. We will use Node.js to enforce that only an Admin can access certain API endpoints:

8. In your Node.js server, when a request comes to an admin-only route (e.g., `/api/reports/edit`), extract the JWT from the request (likely from the `Authorization: Bearer <token>` header or from a cookie).
9. Use the Descope Node SDK to **validate the session token**. For example:

```
const descopeClient = DescopeClient({ projectId: DESCOPE_PROJECT_ID });
// In an express middleware:
const token = req.headers.authorization.split(" ")[1];
try {
  const authInfo = await descopeClient.validateSession(token);
  // authInfo.user contains user data, authInfo.token contains JWT
claims
  console.log("Valid token for user:", authInfo.user.userId);
} catch (e) {
  return res.status(401).send("Invalid or expired token");
}
```

   This ensures the token is genuine (signature valid and not expired) [86] .
10. After validation, check the user's roles. Suppose `authInfo.user` has a `roles` array or similar. Or if roles are inside `authInfo.token` claims (e.g., `authInfo.token["urn:descope:roles"]` or simply `authInfo.token.roles` depending on format). Locate the roles.

```
const roles = authInfo.token.roles || [];
if (!roles.includes("Admin")) {
  return res.status(403).send("Forbidden: Admins only");
}
// else proceed with handling the request
```

11. Now your backend gate is in place. Even if someone fiddled with the client, they can't fake the role past the signature check (since they'd need your private key to sign a bogus token). The Node SDK's `validateSession` ensures the token was issued by Descope [86] . If the user changed a role in the JWT without re-signing, validation would fail.

12. **Test Authorization Logic:** Log in as the Admin user, get a valid token (the React SDK stores it; you can retrieve with `getSessionToken()` client-side [87] , or call a test endpoint that prints `authInfo`). Call the protected API – it should succeed (200 OK). Then try as a non-admin user (or modify the Admin's token to remove the role and resign it – not trivial without the key, so instead perhaps change the user's role in Descope to BasicUser and get a new token by logging in). The non-admin call should be blocked with 403 Forbidden.

13. **Observe JWT Contents (Limitations):** Decode one of the JWTs (they are base64; you can use jwt.io or a library) to see the claims. You will see something like:

```
{
  "sub": "descope-user-id",
  "aud": "descope-project-id",
  "roles": ["Admin"],
  "perms": ["read_reports", "edit_reports"],
  "iss": "https://api.descope.com/",
  "exp": 1694567890,
  ...
}
```

The exact keys might vary (Descope could namespace them, e.g., `permissions` claim). The presence of `roles` here is what we used. Note that this token's expiration (exp) might be, say, 1 hour or 24 hours depending on your settings. **Limitation:** If you change a user's roles in Descope, that change will not reflect in tokens already issued until they log in again or the token is refreshed. For instance, if you remove "Admin" role from Alice in Descope, her existing JWT still says "Admin" until it expires. Mitigation: keep token life short (e.g., 15 minutes) and use refresh tokens so that changes apply on next refresh. Descope allows setting the session JWT lifetime in project settings (and refresh token lifetime) [88] [89] .

14. **Additional Security:** Because JWTs are powerful (they grant access), ensure you handle them securely:

15. Always validate on backend (we did that).
16. Use HTTPS so tokens aren't intercepted.
17. Consider storing JWTs in **HttpOnly cookies** for web apps (Descope SDK can do sessionTokenViaCookie which stores the refresh token cookie scoped to your domain [90] [91] ). This prevents XSS from stealing tokens.
18. If using cookies across subdomains, set proper domain attribute (Descope custom domain config handles that) [92] [93] .
19. Log out/ revoke sessions when needed. Descope's management API can invalidate a user's sessions if, say, their role is revoked and you need immediate effect.

By embedding RBAC in JWTs, our app can make authZ decisions quickly. We just must treat the token as **untrusted until verified** and ephemeral. The approach scales: your Node backend doesn't need to query Descope for the user's role on every request – it's in the token, which is nice. But always balance convenience with caution.

**References:** Default JWT content includes roles (Descope default claims) [85] . Code example for validating session token on backend [86] . Front-end retrieval of session token [94] .

### Scenario 18.3: User Impersonation by Admins (with User Consent)

**Concept: User impersonation** allows an admin or support agent to "step into" a user's account for troubleshooting or support purposes [95] . For example, a customer support rep can impersonate a user to see what they see, without knowing the user's password. Descope supports building impersonation flows where an admin with the proper permission can generate an impersonation session, ideally after the end-user has consented to be impersonated [96] [97] . Crucially, impersonation should be done ethically and securely: it requires user consent, is limited in time, and is audited [98] [99] . We will implement a basic impersonation setup: - Add an "Impersonate" role for support admins. - Create a flow where a support admin can log in as another user (after that user has granted permission).

**Steps:**

1. **Create Impersonation Role:** In Descope, define a role (via the Authorization settings) named "Impersonator" or "SupportAdmin". Grant this role the special **Impersonate permission** [100]. (Descope's system has a built-in permission or flag for impersonation – ensure your role includes that.) Assign this role to the admin users who should be allowed to impersonate others.

2. **User Consent Flow:** It's best practice to obtain user consent before impersonating [96] [101]. We'll create a flow that the end-user triggers (or is triggered when support requests):

3. In Descope Flows, create a flow called "impersonation-consent". This flow's purpose is to prompt the user "Support would like to view your account to assist you. Do you allow this?"
4. Flow design: perhaps first require the user to re-authenticate (step-up verify) for security, then use the **"Update User / Impersonation Consent"** action [102]. This action, when executed, marks the user's record with a consent flag for a certain duration.
5. Configure the Impersonation Consent action's **Consent Expiration In Hours** – e.g., 1 hour of validity [102]. That means within the next 1 hour, an admin can impersonate this user. After that, consent lapses.
6. The end of this flow could simply show a "Consent granted" message to the user.

Deploy this flow in your app – for instance, under the user's profile settings, have a button "Allow Support Access" which triggers this Descope flow for consent.

1. **Impersonation Flow for Admin:** Next, create another flow "impersonate-user" that is used by support admins:
2. Start with an **authentication** step for the admin themselves (ensure the admin is logged in with the Impersonator role; you might skip this if you only let already logged-in admins access it).
3. Then present a screen with an **input** for "Impersonated User Login ID" (Descope offers an "Impersonated User" input component specifically) [103] 【45†】. The admin will enter the identifier (email/ID) of the user they want to impersonate.
4. Add the **"User / Impersonate"** action after that [104]. This action will attempt to create an impersonation session for the target user. It will **fail** if the target user hasn't given consent (or consent expired) unless you choose to bypass consent (there's a setting "Validate user consent" which you should leave enabled in production) [104].
5. If successful, this action returns a session token (JWT) or triggers a session switch to the impersonated user. The flow should then redirect the admin to the application but now as the impersonated user.

Essentially, upon success, the admin's context is now that of the user. Descope likely manages this by issuing a JWT for the target user, tagged as an impersonation. It might also store who impersonated whom.

1. **Using Impersonation in the App:** How you handle the switch in the app can vary:
2. If using Descope's hosted flows, after the impersonation flow, the admin might be directly logged in as the user (their SDK state now holds the user's JWT).
3. Alternatively, the flow might provide the JWT to your backend which you then set in the admin's browser context. Check Descope docs for the exact mechanism. Often, impersonation flows can redirect with a token.

4. For simplicity, assume the admin's front-end will now treat them as logged in as the user. You might show a banner like "You are now impersonating Alice (alice@example.com)" to make it clear.

5. **Test Consent and Impersonation:**

6. As a normal user, run the **Consent flow**. Authenticate (if required) and grant consent. In Descope Console > Users, see that this user now has an "Impersonation Consent until [time]" listed [105] .
7. As an admin, run the **Impersonation flow**. Enter the user's email. If done within the consent window, it should succeed and log you in as that user.
8. Verify you can now navigate the app exactly as that user (e.g., see their data, perform actions as them). This is powerful – be careful with actions that trigger emails or notifications (some systems will note "performed by admin on behalf of user" if possible).

9. When done, provide a way for the admin to "exit impersonation" – essentially log out the impersonated session and return to their own account. This could mean storing the admin's original session somewhere safe (or simply logging them out entirely and letting them log back in as admin).

10. **Audit Trail:** Check Descope's Audit Trail. It will log an event like `LoginSucceed (Impersonate)` showing that Admin X impersonated User Y [99] . These logs include both user IDs so you have traceability. This is important for compliance – you always want a record of who accessed which account by impersonation.

11. **Security & Ethics:** Emphasize that impersonation should be used sparingly:

12. Only admins with the Impersonate role can do it (least privilege principle) [100] .
13. Always get user consent except perhaps in emergency cases. Descope's system is designed with user awareness in mind [101] .
14. The impersonation token should have a limited lifespan (maybe it inherits regular session TTL but note the consent expiration ensures new impersonations can't start beyond that).
15. Ensure users understand what it means (maybe put in your terms of service or show a pop-up when they grant consent).

16. From a technical POV, Descope prevents the admin from seeing the user's password – they are logged in without needing credentials. And when the admin impersonates, any action they take could be logged or attributed accordingly.

17. **Clean Up:** For testing, you can remove the consent after. In Descope, you might revoke consent by setting it to expire immediately (or just wait it out). Also, if you created flows you don't intend to use in production, disable them. But in a real scenario, these flows (consent and impersonation) would be integrated as features in your app.

By implementing impersonation, support teams can effectively help users while maintaining security barriers (no password sharing, tracked access, and user control) [98] [99] .

**References:** Descope user impersonation overview and best practices [95] [98] , consent flow and action [102] , impersonate action settings [104] .

## Day 19: Testing, DevOps, and Continuous Deployment

### Scenario 19.1: Test Automation of Descope Flows with Playwright

**Concept:** After implementing authentication flows, it's important to automate tests for them. End-to-end tests (using tools like **Playwright** or **Cypress**) can simulate a user signing up or logging in. This is tricky because flows involve OTPs, emails, or external SSO redirects. Descope provides features to facilitate testing: - **Test Users:** You can flag users as test users and use the Management API to generate OTP codes or magic links for them without needing real email/SMS delivery [106] [107]. - **Static Credentials for Automation:** For example, using an OTP code that is retrievable via API, or using a predefined password. - **Tenant isolation:** Perhaps use a separate Descope project or test tenant to not interfere with real data.

In this scenario, we will set up a test user and write a Playwright script that logs in via OTP, intercepting the OTP through the API instead of relying on actual email.

**Steps:**

1. **Mark a User as Test User:** In Descope Console or via API, create a user and mark them as a **Test User** [108] [109]. You can do this through the Management SDK:

```
// Using Node SDK for example
await descopeClient.management.user.createTestUser(
    loginId = "testuser@example.com",
    email = "testuser@example.com",
    phone = "+15555550001", // optional, if testing SMS OTP
    ...other fields...
);
```

   Setting a user as a test user typically means Descope will not actually send out communications (emails/SMS) for them, and allows using special API endpoints to fetch OTP codes [106] [107]. Confirm in the Console that the user is marked as Test (there's often a label).

2. **Prepare the Test Environment:** Write a Playwright test that will:

3. Launch a headless browser.
4. Navigate to the login page of your React app.

5. Interact with the Descope widget or form to initiate login for the test user.

6. **Simulate OTP Login Flow:**

7. The test will input the login identifier (email) of the test user and click "Login" (assuming an OTP by email flow).
8. Normally, Descope would send an email with the OTP code. But because this is a test user, we can use the Management API to fetch the code directly. We'll do this in the test script.
9. Playwright can execute an HTTP request to your server or Descope's API. However, for security, Descope doesn't expose OTP codes via public API, but it has a Management SDK function specifically for test users: `generateOTPForTestUser` [110] [107].
10. Use Node (or Python) in the test to call this function right after triggering the OTP send in the UI. This will return the actual OTP code for that test user [111]. Example using Node:

```
const resp = await
descopeClient.management.user.generateOTPForTestUser("email",
"testuser@example.com");
```

```
const code = resp.data.code;
console.log("Test OTP code:", code);
```

11. Now you have the 6-digit code.

12. **Complete OTP in Playwright:**

13. In the test script, after fetching the code, have Playwright fill the OTP input fields on the web page with this code (e.g., `page.fill('input[otp-digit]', code)` or similar, depending on your UI).

14. Submit the form or let the widget auto-submit.

15. **Verify Login Success:**

16. Wait for navigation or a specific element that indicates the user is logged in (e.g., the presence of a logout button or the user's name on dashboard).
17. If using Descope's React SDK, you might check that `isAuthenticated` becomes true or that a protected content is visible.

18. In Playwright, use assertions: `await expect(page.locator('text=Welcome, Test User')).toBeVisible()` for instance.

19. **Test SSO flows (Optional):** Automating SSO is harder because it involves redirecting to an external IdP (Okta, etc.). One approach is using **OIDC credentials grant** directly for tests, but that's complex. Alternatively, you can use a **mock IdP**:

20. Descope's docs mention using `mocksaml.com` for testing SSO [112]. This is a fake IdP that will accept any login @example.com and succeed. For testing, you might configure your tenant's SAML to use mocksaml metadata [113]. Then Playwright can click "Login with SSO", get redirected to mocksaml (which might just prompt for an email), enter an `example.com` email, and upon "login" it will redirect back immediately as a success [114]. This can automate SSO without a real Okta.

21. Given time, we focus on OTP example which is straightforward.

22. **Tear Down:** After tests, you might want to remove or reset test users. Descope offers deletion of test users via API [115] [116], or you can keep reusing the same test account with static login.

23. **Running in CI:** Integrate this test into your CI pipeline. Because you used Descope's test user and API, the test won't depend on external emails or phone lines, making it reliable and fast. Ensure your Descope Management Key (for the test project) is available to the test (maybe via an environment variable in CI, but keep it secure!). The test user's OTP generation is authorized by that key, so restrict its scope to test tenant only for safety.

This scenario demonstrates that even tricky auth flows can be automated by leveraging Descope's testing utilities. We simulated an email OTP login entirely within the test by retrieving the OTP code programmatically [107] [111]. Similarly, Magic Links and Enchanted Links have test methods too (the API can return the magic link URL or token for a test user [117] [118]). Using these, you can validate all your signup/login scenarios in an automated fashion.

**References:** Creating test user and generating OTP via management SDK [106] [107]. Mocksaml for SSO testing [112] [113].

### Scenario 19.2: CI/CD and Environment Management with Descope's Terraform Provider

**Concept:** As your application grows, you'll have multiple environments (development, staging, production) and many auth configurations (flows, roles, screens). Managing these via the web console can be error-prone. Descope provides a **Terraform provider** so you can treat auth config as code and integrate it into CI/CD [119] [120]. Using Terraform, you can script the creation of roles, user attributes, authentication methods, etc., and easily clone or promote config from dev to prod. Here, we will set up a simple Terraform script to manage part of our Descope config and describe how to use it in a pipeline.

**Steps:**

1. **Install Terraform and Provider:** Make sure you have Terraform installed locally [121]. Add the Descope provider to your Terraform configuration:

```
terraform {
  required_providers {
    descope = {
      source = "descope/descope"
      version = "1.0.0"  # or latest version
    }
  }
}
provider "descope" {
  project_id     = var.descope_project_id     # Your Descope Project ID
  management_key = var.descope_management_key # A Management Key with
proper permissions
}
```

   Variables `descope_project_id` and `descope_management_key` will be passed in (for security, don't hard-code keys; use env vars or a CI secret store). The management key must allow config changes (Descope requires a Pro plan for Terraform use) [122].

2. **Manage Project Settings as Code:** In a Terraform file (e.g., `descope.tf`), declare resources for what you want to configure. Example:

```
resource "descope_project" "this" {
  name = "MyApp Dev Project"
  # Basic settings
  project_settings = {
    refresh_token_expiration = "7 days"
    enable_inactivity = true
    inactivity_time = "30 minutes"
  }
  # Authentication settings (enable/disable methods)
  authentication = {
    magic_link = { expiration_time = "1 hour" }
    password = {
      enabled = false
    }
    sso = {
      redirect_url = "https://myapp.dev/callback"
      # (Assume SSO domains etc. configured via console or separate API
- TF v1 may not cover all nested SSO configs)
    }
  }
  # Authorization: define roles/permissions as in Day 18
  authorization = {
    permissions = [
      { name = "view_dashboard", description = "View dashboard" }
    ]
    roles = [
      { name =
"Viewer", description = "Read-only access", permissions =
["view_dashboard"] }
    ]
  }
  # Custom Attributes (if any)
  attributes = {
    user = [
      { name = "Department", type = "string" }
    ]
  }
}
```

This describes the desired state: one permission, one role, a custom user attribute, etc. Terraform will compare against actual project and apply changes [123] [83].

3. **Initialize and Apply Terraform:** Run `terraform init` to fetch the Descope provider plugin [124]. Then run `terraform plan` to see what changes will be made (it might create roles, etc.). Finally, run `terraform apply`. If the credentials are correct and your management key has rights, Terraform will configure your Descope project:

4. E.g., the “Viewer” role will be created, with the specified permission [83] [84].
5. The custom attribute “Department” will be added to the project’s user schema [125].
6. Password auth will be disabled (so users can’t set passwords) because we set `enabled = false`.

Terraform’s output will tell you resources created/updated.

1. **Use Terraform for Multi-Environment:** Typically, you’d have separate Descope projects for dev, test, prod (since user pools might be separate per environment). With Terraform, you can maintain one config and apply it to different projects by switching `project_id` and keys. For example, you might have a pipeline where:
2. On a git tag for release, you apply Terraform to the prod project (with prod credentials).
3. Meanwhile, developers regularly apply TF to the dev project.

This ensures consistency: same roles, same flows, etc., across environments. You avoid manually clicking in the console for each environment.

1. **State and Secrets:** The Descope provider does not manage user data (users, tenants) via Terraform – those are dynamic and generally out of scope for config-as-code [126]. It focuses on configuration. So your TF state will contain project settings, not thousands of users. This is good. Just keep the state secure (if using a remote state backend).

2. **CI Integration:** In a CI pipeline (GitHub Actions, GitLab CI, etc.), you can incorporate Terraform:

3. Use the official Terraform CLI Docker or install it in the runner.
4. Store `DESCOPE_PROJECT_ID` and `DESCOPE_MANAGEMENT_KEY` as secrets in your CI.
5. Have the pipeline run `terraform apply -auto-approve` when appropriate (maybe on the main branch for staging, and on a protected trigger for production).
6. Descope also offers a GitHub Actions template for migrating config between projects [127]. They have a GitHub repo that can automatically export from one project and import to another, helping with environment promotion.

7. Using Terraform is often simpler: you version control the `descope.tf` file in your code repo, so changes to auth config are reviewed just like code.

8. **Example: Adding a New Role via CI:** Let's say you need a new "Editor" role. Instead of clicking in the console on each environment, you edit the Terraform file to add:

```
{
  name = "Editor", description = "Edit access", permissions =
["view_dashboard", "edit_reports"]
}
```

   under roles. You commit the change. The CI pipeline runs, and Terraform updates Descope: it creates the Editor role and attaches the appropriate permission. Now all envs are in sync. No manual error, and you have an audit trail (the git commit) of who made the change.

9. **Terraform Limitations:** As of writing, not every Descope feature may be covered by Terraform (e.g., fine-grained SSO settings or flows might not be fully configurable via TF yet). You might still export/import flows manually or via CLI. Descope's provider is evolving, and they encourage Terraform for the stable parts like roles, attributes, templates [126] [128]. For flows and screens, Descope offers export as JSON (you could store those JSON in git and use their CLI or API to import to prod). In any case, the principle is to avoid clicking around for reproducible config.

**References:** Terraform provider usage example [124] [129], project resource schema (roles, attributes, etc.) [83] [125], note that Terraform is mainly for config not dynamic elements [126].

### Scenario 19.3: Session Management – Validating Descope Sessions in API Gateway/Backend

**Concept:** After authentication, the front-end typically has a **session token (JWT)** for the user. In a React + Node setup, your React app might call protected APIs on your Node.js server. The server must validate the user's session on each request to enforce security (you should not trust anything coming from the client without verification). Descope facilitates this by providing the session JWT and a method to validate it, as we saw in Day 18. We'll expand on that by focusing on **cookie-based session tokens** and how to perform session validation in an API gateway or backend service for each call.

**Steps:**

1. **Use Secure Cookies for Sessions:** The Descope React SDK by default stores the session token (JWT) in memory and optionally localStorage. But it also supports storing refresh tokens in HttpOnly cookies for security [90]. In a production environment, you might configure:

   ```
   <AuthProvider projectId="YOUR_PROJECT_ID" sessionTokenViaCookie={true}>
   ```

   This causes the SDK to use a cookie (usually named `DSR`) for the refresh token, and possibly `DS` cookie for session, scoped to your domain [90]. The advantage is that the browser will automatically include this cookie with requests to your backend (if same domain), so the JWT is sent securely (HttpOnly, not accessible to JS).

2. **Backend Middleware Setup:** In Node/Express or whatever server:

3. If using cookies, use a middleware to parse cookies from requests.
4. The refresh token cookie (DSR) might not be needed for each API call; the session token (DS or the Authorization header) is what we want. However, if you opted to store session token in a cookie too, it will come along.
5. Grab the session token. If using Authorization header (Bearer token) approach, the client can send `Authorization: Bearer <session_jwt>` (the React SDK snippet shows how to set that header on fetch calls [94]). If using cookies, the session JWT might be in a cookie, or you might need to call `getSessionToken()` on client and manually set auth header.

6. In short, ensure each request includes the token somehow. Let's assume Authorization header for clarity.

7. **Validate Session on Backend:** Integrate Descope Node SDK or use a JWKS method:

8. Using SDK: as in scenario 18.2, call `descopeClient.validateSession(sessionToken)` on each request (or use a caching strategy to avoid decoding repeatedly, but since JWT validation is usually just a signature check and cheap, it's fine).
9. This returns `authInfo` with user details if token is valid [86]. If invalid or expired, it throws an error – in which case respond with 401 Unauthorized.

10. Consider token expiration: If using refresh tokens via cookies, when a session JWT expires, the front-end should automatically refresh (the React SDK handles this behind the scenes using the refresh cookie). So ideally, API calls always have a valid token. If not, the backend can also attempt to exchange a refresh token – but keeping that logic in front-end or a gateway is more common.

11. **Example Express Middleware:**

   ```
   app.use(async (req, res, next) => {
     const authHeader = req.headers['authorization'];
   ```

```
  const token = authHeader && authHeader.split(' ')[1];
  if (!token) {
    return res.status(401).send("No token");
  }
  try {
    req.descopeAuth = await descopeClient.validateSession(token);
    // attach user info to request for handlers to use
    next();
  } catch (e) {
    console.error("Invalid session token:", e);
    return res.status(401).send("Invalid session");
  }
});
```

Now any route after this middleware will have `req.descopeAuth.user` etc. available.

12. **Validate Across Microservices:** If you have an API gateway or multiple services, you might not use the Node SDK everywhere. Instead, you can validate the JWT using the public keys (JWKS) that Descope publishes for your project:

13. Descope's JWKS URL is typically `https://api.descope.com/v1/auth/keys?projectId=<yourProjectId>` [130]. You can retrieve the public signing keys and use a JWT library in any language to verify the signature of the token.

14. This is useful for non-Node backends (Go, Python, etc.) or if you prefer not to depend on the SDK. But the Node SDK simplifies it to one call, as shown.

15. **Cookie Domain Consideration:** If your frontend is at `app.example.com` and backend at `api.example.com`, sharing cookies requires setting the cookie domain to `.example.com` and appropriate CORS. Descope custom domain config allows you to have refresh token cookie on a parent domain [91] [92]. We covered this in Day 20 perhaps, but just be mindful: ensure your cookie domain is set so that the cookie is sent to the backend domain. If that's not feasible, stick to Authorization header approach.

16. **Test Session Validation:** Manually or via automated test:

17. Call a protected API without a token -> expect 401.
18. Call with a valid token -> expect 200 and your logic executes (maybe returns user-specific data).
19. Call with a tampered token (e.g., change one character) -> should 401 (signature fails, `validateSession` throws).

20. Call with an expired token (you can simulate by using an old token or setting a short expiration and waiting) -> should 401, and ideally the client should then use refresh token to get a new one and retry. Descope's SDK might do this automatically if you use their fetch wrapper. If not, you handle 401 by prompting re-login or trying refresh endpoint.

21. **Session Logout:** When a user logs out, ensure you clear the session on client (remove token and cookies). You can also call Descope's **logout API** to invalidate the refresh token (Descope has a method to log out which essentially invalidates the session or refresh token on server side). This prevents the token from being used again. In tests, you might not validate logout via backend except to ensure a subsequent call after logout token is invalid.

In summary, robust session management means verifying the user’s token on each request and not trusting any data blindly from the client. Descope’s validate functions and JWT keys make this straightforward [86] . This scenario reinforces best practices for API security.

**References:** Example code for session validation using Node SDK [131] (which we used). The React example showing sending token in header [94] . Also Descope docs on retrieving JWKS keys [130] for manual verification if needed.

## Day 20: Customization and Multi-Tenant Features

### Scenario 20.1: Custom Styling and Branding per Tenant (White-Labeling)

**Concept:** If you serve multiple customers (tenants), each might want their own branding on the login UI. Descope’s **Styles & Themes** allow you to define custom style files (with logos, colors, fonts) and apply them dynamically [132] [133] . You can either switch styles based on which tenant is logging in or which application/subdomain is being used. We will create two style themes (e.g., “Light Theme” and “Dark Theme” for two different clients) and see how to apply them in the authentication flow.

**Steps:**

1. **Create Style Files in Descope Console:** Go to **Styles** in the Descope Console. Click “+ New styles file” [134] . Create one style named “TenantAStyle” and another “TenantBStyle”. Each will get a unique Style ID. For each style:
2. Upload a different company logo for demonstration (maybe Tenant A’s logo and Tenant B’s logo) [135] .
3. Choose a primary color and secondary color reflecting each brand (Descope will auto-generate a palette) [136] .
4. Optionally pick different font or typography settings per style (e.g., Tenant A uses a serif font, Tenant B uses sans-serif) [137] .
5. Save the styles. You now have, for example, style IDs `style_tenant_a` and `style_tenant_b`.

6. **Apply Style via SDK (Multiple Apps):** If your React app knows which tenant or subdomain it’s serving, it can pass the corresponding style to the Descope components. For example:

```
const styleId = (tenant === 'TenantA') ? 'style_tenant_a' :
'style_tenant_b';
<Descope flowId="sign-up-or-in" styleId={styleId} ... />
```

   This will render the embedded flow using that style theme [138] . The UI will automatically reflect the colors, logo, etc., defined in that style file.

7. **Apply Style via Hosted URL:** If using Descope’s hosted pages (Auth Hosting) rather than embedding, you can append a `style` parameter to the URL:

```
https://YOUR_PROJECT.descope.app?flow=sign-up-or-
in&tenant=TENANT_A_ID&style=style_tenant_a
```

When your user from Tenant A hits this URL, Descope will load the tenant's flow with Tenant A's branding [139]. For Tenant B, use their style ID accordingly. This is helpful if you simply redirect users to Descope's hosted login – you can ensure the look matches their brand.

8. **Subdomain Routing:** In many SaaS apps, each customer might have their own subdomain (e.g., acme.yourapp.com and beta.yourapp.com). You can key off the subdomain to choose style. For instance, in your app initialization:

```
const hostname = window.location.hostname; // e.g., "acme.yourapp.com"
let styleId;
if (hostname.startsWith("acme")) styleId = "style_tenant_a";
else if (hostname.startsWith("beta")) styleId = "style_tenant_b";
<Descope flowId="sign-in" styleId={styleId} />
```

   Now each subdomain sees its respective style. (You can also do a similar thing for language localization using Descope's Localization features, not covered here.)

9. **Dynamic Style Switching (if needed):** Descope's SDK also allows switching the style at runtime if for example a user selects a theme (light/dark mode toggle). The `theme` prop is separate (for dark vs light theme, which automatically inverts colors), but style files are more about brand. Usually, one style per tenant is enough.

10. **Test the Theming:** Try logging in as a user of Tenant A – you should see Tenant A's logo on the login page, and the color scheme matching what you set [140]. Log out, then simulate being on Tenant B's site (or manually force style B in code) – you should now see Tenant B's branding. Ensure that all screens (login, OTP, error screens) follow the style. If you had custom CSS in the style file (Descope allows code mode CSS for fine tweaks), those would also apply.

11. **Advanced – Multiple Applications:** Descope also has a concept of **Applications**, where each can have a default style associated [141]. If you created separate Application IDs in Descope for Tenant A vs Tenant B, you could link a style file to each. Then, if the flow knows which app it's coming from, Descope would apply the style automatically. But passing styleId directly as above is straightforward for most cases.

12. **Reset to Default:** If no styleId is provided, the flow uses the project's default style (the one named "Default" in styles tab). So you always have a fallback. You can also define a nice generic style for when no specific tenant is known.

Brand customization is critical for a white-labeled SaaS experience. With a few style definitions, you can reuse the same auth flow but make it look native to each customer's product [138]. Descope's theming engine even supports adding custom CSS or loading custom fonts for ultimate flexibility [142] [143].

**References:** Creating and configuring style files [144] [145], passing styleId in React SDK [138], using style param in hosted URL [139].

### Scenario 20.2: Using Descope User and Admin Widgets

**Concept:** Descope provides pre-built **Widgets** – embeddable UI components for common user management tasks (profile management, password reset, user invite, etc.) [146]. This allows you to delegate certain screens to Descope instead of building them from scratch. There are **User Widgets** (for end-users to manage their account, e.g., update profile info, manage MFA methods) and **Admin Widgets** (for tenant admins to manage their organization's users, roles, etc.) [147] . We will enable a **User Profile widget** so that logged-in users can update their profile (like name, picture, and even add MFA devices) on their own. We'll also briefly mention an Admin User Management widget for tenant admins.

**Steps:**

1. **Create a Widget in Descope Console:** Go to **Widgets** in the console. Click **+ Create Widget** [148] . You'll see templates. For user self-service, choose **User Profile** template (this might allow updating profile attributes, resetting password, enabling TOTP, etc.). Name it "UserProfileWidget". Similarly, if you want an admin widget, choose **User Management (Admin)** template for tenant admins to manage user list – name it "AdminUserMgmtWidget".

2. **Customize Widget (Optional):** After creating, you can edit the widget's design:

3. Add or remove fields. For instance, you might add the custom user attribute "Department" we made to the profile form (drag a custom attribute component).
4. Mark certain fields read-only if needed (maybe email is read-only if you don't allow change) [149] [150] .

5. Check the flows/actions tied to it: The profile widget typically has actions to update user data. Admin widget has actions to invite user, assign roles, etc. These come pre-wired but you can adjust (this is advanced; by default it's fine).

6. **Embed Widget in React:** Descope's SDK provides React components for widgets. For example, for the user profile widget:

   ```
   import { UserProfile } from '@descope/react-sdk';
   ...
   <UserProfile widgetId="UserProfileWidget" />
   ```

   That's it – this component will display the user profile management UI, automatically using the current logged-in user context (the SDK knows the session) [151] . For an Admin widget:

   ```
   import { UserManagement } from '@descope/react-sdk';
   ...
   <UserManagement widgetId="AdminUserMgmtWidget" tenant={tenantId} />
   ```

   Notice admin widgets often require a `tenant` prop to know which tenant's users to manage [151] . You'd supply the tenantId that the admin is responsible for. Possibly you get that from `user.tenantIds` in the JWT.

7. **Place Widgets in Your App:** Perhaps create a page `/profile` in your React app for the user profile. Only accessible when logged in. On that page, render `<UserProfile ... />` as above. For tenant admins, maybe a page `/admin/users` that shows the `<UserManagement ... />` widget. Only show it if the user has admin role and a tenant context.

8. **Test User Profile Widget:** Log in as a normal user. Navigate to the Profile page. You should see a form with your details (name, email, etc.), possibly an option to set a profile picture, and maybe sections for security like "Add Authenticator App (TOTP)" or "Manage Devices". These capabilities come from Descope:

9. Try updating your display name and saving. Descope will handle saving and you should see the change reflected (the SDK likely updates `user.name`).
10. Try enabling an MFA method (if profile widget includes that). For instance, enable TOTP: it might show a QR code to scan with Google Authenticator – that is all pre-built in the widget. After scanning, it will verify a code and associate a TOTP factor with your account. Next time you login, that factor might be used (if your flow requires MFA).

11. All of this without you writing backend code – Descope's widget and flows handle those updates and calls via the SDK behind the scenes.

12. **Test Admin User Management Widget:** Log in as an admin user (with an appropriate role). On the Admin Users page, the widget should list users in your tenant (Descope SDK calls the management API to list users of that tenant). You might see options like "Invite User" or "Edit User" depending on template:

13. Try adding a user via the widget (enter email, roles, etc.). Descope will send an invite (maybe via email magic link) to that user automatically as defined by the widget's flow.
14. Try changing a user's role via the widget. This uses Descope's management API under the hood to update the user's roles.
15. If there's a "remove user" button, test that – it might deactivate or delete the user record.

16. Essentially, this widget can replace building an entire admin user CRUD UI in your app.

17. **Experience and Theme:** The widget will adopt your project's style like other Descope components. You can even pass a `styleId` or `theme="dark"` prop to the widget component if you want to override style or use dark mode [152] [153]. For instance:

```
<UserManagement widgetId="AdminUserMgmtWidget" tenant={tenantId}
styleId="style_tenant_a" theme="dark" />
```

This would render the admin widget with Tenant A's branding in dark mode.

18. **Security:** Descope widgets respect user permissions. The admin widget will likely only show up if the logged-in user has a role that allows user management. But you should also enforce that on your side (don't even route non-admins to that page). The widget's actions will fail or be hidden if user lacks rights.

By using widgets, you accelerate development of profile and admin features. They are fully integrated with Descope's backend – for example, the profile widget uses Descope's **Management SDK** under the hood, similar to how we manually did in Day 19.1 for test users. This means less custom code for you to maintain [154].

**References:** Widgets overview [146], embedding React widget component (example with props) [151], customizing widget design [149].

### Scenario 20.3: Passkey (WebAuthn) Adoption and Domain-Bound Credentials

**Concept: Passkeys**, based on WebAuthn/FIDO2, allow users to authenticate with a fingerprint or face or a PIN, using cryptographic keys stored on their devices (e.g., Touch ID, Windows Hello, YubiKey) [155]. Descope supports passkeys as a primary auth method. One nuance of WebAuthn is that credentials are **bound to a domain** (relying party ID). If your auth domain changes (e.g., moving from dev domain to prod domain) or if you use multiple domains, the passkey created on one won't work on another [156]. We'll demonstrate enabling passkey login and explain domain-bound behavior and how to manage it.

**Steps:**

1. **Enable Passkeys in Descope:** Ensure Passkey authentication is turned on for your project (Descope Console > Authentication Methods > Passkeys). If it's not already, toggle it on. You can choose to allow Passkeys for signup and login.

2. **Add Passkey to Auth Flow:** If using the default "Sign-up or in" flow, Descope likely already includes a "Sign in with passkey" button if passkeys are enabled. If not, you can add a **Passkey** action in your flow:

3. In Flow editor, after email/phone identification step, drag a **Passkey** action (either "Sign Up or In with Passkey" which handles both registration and login) [157].
4. Alternatively, you might use the ready-made "Sign Up or In (Passkey)" flow template.

5. **Front-End Integration:** With React SDK, using `<Descope flowId="passkey-flow" />` is one way. Or use the provided hook:

6. Descope's React SDK offers `useDescope()` which might include a function like `loginWithPasskey()`. Check docs; often they abstract WebAuthn.
7. But simplest: if you rely on the hosted or embedded flow, the "Use Passkey" button in the flow will handle calling WebAuthn APIs. No extra code needed.

8. **Test Passkey Registration:** In a supported browser (Chrome, Safari, etc.), go through the sign-up process and choose "Sign up with Passkey" if available:

9. The browser will prompt "Create a passkey for this site?" with your device's biometrics or PIN. On approval, a credential is created in the device authenticator and a public key is sent to Descope. Descope links it to your user.
10. If successful, you might not even need a password or OTP – the passkey itself created your account and logged you in.
11. Note: For testing on localhost, some browsers require HTTPS or specific flags for WebAuthn. Also, some older Android or iOS may not fully support cross-device passkeys yet. Use latest devices for best results.

12. **Test Passkey Login:** Log out, then click "Login with Passkey":

13. The browser will likely show a native prompt (e.g., platform authenticator or a prompt to use phone nearby if cross-device passkey via QR).

14. Use your fingerprint/FaceID. It should authenticate and log you in without any password or code. Magic!

15. **Domain-Bound Behavior:** Now the interesting part – the passkey you registered is tied to the domain (relying party ID). If you did this on `localhost` or a custom domain `auth.myapp.dev`, the credential only works there [156]. If you switch to another domain (say deploying to `myapp.com`), the browser won't offer that passkey. In practice:

16. If you open the prod site, it will appear as a new site to WebAuthn. You'd have to register a new passkey for that domain.
17. This can confuse users if they made a passkey on one environment and expect it to work on another. For prod usage, you likely use a stable domain.

18. If you anticipate domain changes (like migrating your auth domain), note that **passkeys cannot be migrated** – users would need to re-enroll on the new domain [158].

19. **Custom Domain for Consistency:** To avoid multiple domain issues, use a single top-level domain for auth. For example, set up `auth.myapp.com` as a custom domain for Descope and use it in dev and prod (point it to dev when testing, then to prod – though that has its own complexities). Alternatively, have separate credentials per environment but inform users accordingly (less ideal). Descope mentions the **Passkey Top-Level Domain** setting – if you set your project's domain to `myapp.com`, then subdomains can share credentials [159] [160]. Descope will use the eTLD+1 (effective top-level domain) as RP ID if configured, so that `app.myapp.com` and `docs.myapp.com` share passkeys [161]. Ensure to configure that in Project Settings if needed.

20. **Multi-Device Passkeys:** Modern ecosystems (Apple iCloud Keychain, Google/Windows) will sync passkeys across your devices for the same domain. So if a user registers on their phone, it might be available on their laptop if same Apple ID, etc. This is great for usability. But cross-platform, the user might rely on a platform-specific mechanism (like scanning a QR to use phone's passkey on a PC). Descope's flows likely handle this via the browser's WebAuthn UI – e.g., "Use a passkey from an iPhone nearby" prompt.

21. **Managing Passkeys:** With Descope's user profile widget or via API, users can view and **remove** their registered WebAuthn credentials. For example, if they want to revoke a stolen device's passkey, they can delete it. Ensuring your profile management (from Scenario 20.2) exposes that is useful. Descope's profile widget does include device management typically.

22. **Backup Methods:** Always encourage users to have a backup login (like OTP or recovery code) in case they lose their passkey device. Descope supports **Recovery Codes** and you can enable those as a fallback.

In summary, passkeys provide phishing-resistant, super convenient auth, but tie closely to domain and device. Plan your domain strategy upfront [156]. Descope's implementation makes it easy to incorporate without dealing with low-level WebAuthn details – a huge benefit since dealing with attestation, challenge signing, etc., can be complex.

**References:** Domain-specific passkeys explanation [156] ("passkeys are tied to the domain where they were created"), custom domain and RP ID notes [159] [160]. Descope passkey integration guide [157]. (We discussed limitations from external source too [158]).

## Day 21: Migration and Federation Advanced Topics

### Scenario 21.1: Just-in-Time User Migration via SSO (Hybrid Migration)

**Concept:** When moving from a legacy auth system to Descope, one approach is **Just-in-Time (JIT) migration** [162] [66]. Instead of bulk importing all users, you configure Descope to create users on the fly the first time they log in via SSO or some delegated check. This is often used when migrating from an existing IdP or auth database without forcing password resets. We'll illustrate JIT migration using Auth0 as an example IdP, where Auth0 remains the IdP temporarily and Descope is gradually populated as users log in.

**Steps:**

1. **Set Up Legacy IdP as SSO in Descope:** Suppose your current users authenticate via Auth0. You can add Auth0 as an **OIDC identity provider** in Descope (or SAML, depending on Auth0 config). Essentially:
2. In Descope, create a new SSO Tenant for the migrating users (or use your default tenant).
3. Under Tenant's SSO settings, add an **OIDC Custom IdP**. Input the Auth0 issuer, client ID, client secret, etc. Alternatively, use SAML: Auth0 can act as a SAML IdP and you'd input metadata accordingly.
4. This means when a user tries to log in, if they select this IdP, Descope will redirect to Auth0 to authenticate.
5. **Enable JIT Provisioning:** In the Descope Tenant settings, ensure **Just-In-Time provisioning** is ON [162]. This allows Descope to create a user record the first time it sees a successful login from Auth0 (if one doesn't exist already).
6. Also, in the SSO mapping, choose how attributes map. For example, map Auth0's `email` claim to Descope email, etc., so that the Descope user gets populated with a name/email from the Auth0 assertion.
7. **User Flow:** A user goes to log in to your app. They either click "Login with Auth0" (if you surface that as an SSO option) or you silently direct certain domains to Auth0 IdP.
8. The user authenticates against Auth0 (which might be using its own DB or social login, doesn't matter).
9. Auth0 returns a SAML/OIDC response to Descope asserting the user's identity (say email `bob@oldcorp.com`).
10. Descope looks for a user `bob@oldcorp.com`. If not found, **JIT creates** a new user in Descope on the fly [10]. This user might have a random Descope userID, and is associated with the SSO tenant.
11. The user is then logged in to your app with a Descope session.
12. **Outcome:** Over time, each user who logs in via Auth0 IdP is migrated into Descope's user store [7]. You can monitor Descope's Users list growing. Eventually, after, say, 90 days, most active users have been migrated (anyone who never logged in remains only in Auth0).

13. **Sunsetting Auth0:** Once you reach a comfortable state, you can encourage or force remaining users to migrate (maybe send them an email with a Descope Magic Link to set up a new login if needed). Then shut down the Auth0 connection.

14. Alternatively, keep Auth0 as a permanent IdP if some users prefer it – but usually the goal is to phase it out.

15. **No Passwords in JIT:** Note that in this flow, users didn't set a new password in Descope. They continue using Auth0 to authenticate, and Descope trusts Auth0's assertion. The Descope accounts might not have a password set at all (they are marked as SSO-only). If later you want to let them login directly with Descope (no Auth0), you might require a password reset or set initial passwords via a bulk update (or allow Magic Link login, etc.).

16. **Audit & Logging:** Descope audit logs will show these JIT user creations via SSO. Ensure you disable JIT once migration period is over if you don't need it further (to avoid accidentally creating accounts for unexpected logins).

17. **Alternate JIT (Custom DB):** If migrating from a custom legacy DB (not SSO), another pattern is a **custom authentication webhook**: e.g., Descope could call your API to verify credentials if user not found, then create the user (this is more custom and not using standard protocols; not covered deeply here as Descope primarily supports SSO JIT or SCIM).

**Benefit:** JIT means no big bang migration – users transition seamlessly by logging in, with no downtime [163] [11]. The drawback is dependency on the old system until everyone logs in once.

**References:** JIT provisioning workflow [7], explanation that IdP assertion triggers new account creation [10]. Auth0 hybrid migration context [164] [165] ("Hybrid Migration – still use Auth0 as IdP while Descope auth via federation").

## Scenario 21.2: Bulk User Migration with Descope APIs (Full Migration)

**Concept:** The other migration approach is **Full Bulk Migration** – export all users from the old system and import them into Descope in one go [165] [166]. Descope provides tools (scripts, APIs) to facilitate this, including handling password hashes if possible. We will outline migrating from Auth0 using Descope's migration script.

**Steps:**

1. **Export from Source:** Using Auth0 as example: Auth0 allows exporting user data either via API (if <1000 users) or via a JSON export for larger userbases [167] [168]. Suppose we exported a JSON file `auth0_users.json` containing users and their attributes (and possibly password hashes if obtained with a support ticket) [169].

2. **Descope Migration Tool:** Descope has an open-source migration tool (on GitHub, e.g., `descope-migration` repository) [170]. This is typically a Python script. Steps would be:

3. Clone the repo and install requirements [170] [171].
4. Set up environment variables for Auth0 and Descope credentials in a `.env` file [172]:

```
AUTH0_TOKEN=<Auth0 Mgmt API Token>
AUTH0_TENANT_ID=<Auth0 Tenant>
DESCOPE_PROJECT_ID=<Descope Project ID>
DESCOPE_MANAGEMENT_KEY=<Descope Mgmt Key (with migrate permissions)>
```

And if using password hashes, the path to that file.

5. Run the script in **dry-run** first to see what would happen:

```
python3 src/main.py auth0 --dry-run --from-json auth0_users.json
```

This lists how many users would be created, any errors, etc. [173] .

6. **Handle Passwords:** If you managed to get password hashes from Auth0 (not always possible; requires requesting hashed passwords under certain conditions) [174] , the tool can import those so users won't have to reset passwords. You'd supply `--with-passwords passwords.json` file which maps user->hash [175] . The tool then uses Descope Management API to create each user with that hash (Descope supports importing password hash for certain algorithms – the tool presumably converts them if needed).

7. **Run Live Migration:** If dry-run looks good, run:

```
python3 src/main.py auth0 --from-json auth0_users.json --with-passwords
passwords.json
```

(omit `--with-passwords` if not applicable). This will iterate through users and call Descope's Management APIs to create them. It likely uses bulk endpoints or sequential calls to create user, set email, phone, custom attributes, roles, etc., and mark if verified [176] [177] . If password provided, it sets the password (perhaps via hashing or directly if using salted hash injection – details depend on support).

8. **Custom Attributes and Roles:** The migration tool also handles creating any custom user attributes (like connection type, a migrated flag) [178] . For example, it might add a custom boolean `freshlyMigrated=true` for each user so you know these accounts came from old system [177] . It also might tag the old Auth0 user ID for reference (so no data lost).

9. **Post-Migration Steps:**

10. Verify random samples of migrated users: check in Descope Console that profiles look correct (name, email, etc. present, maybe test logging in with a known password if migrated).
11. If passwords weren't migrated, decide strategy: e.g., send Magic Link emails to users prompting them to set a password or simply have them do a password reset on first login. Descope's **Enchanted Link** or **Magic Link** flows are useful to onboard users without knowing their old password.
12. Migrate any **SSO connections or tenants**: If the old system had SSO setups for B2B customers, you may need to recreate those in Descope (the migration tool doesn't auto-configure SAML connections – that you do manually or via Terraform).

13. Once done, you can redirect your application to start using Descope for auth exclusively.

14. **Clean Up & Launch:** Turn off the old auth system or put it in read-only. All logins now go to Descope. Users either use their migrated password or complete account activation via a link (depending on what you did about passwords). Monitor logs in early days to catch any login issues.

**Considerations:** Migrating thousands of users in bulk should be done during a maintenance window if possible. But since users often can't log in during the transition, using JIT or phased migration (hybrid) is sometimes preferred unless you absolutely need to cut over immediately. Descope's tool helps ensure minimal errors and provides verbose logs (especially run with `-v` for debug info) [179] .

**References:** Descope Auth0 Migration Guide snippet [165] [167] explaining full vs hybrid. Steps for setting up environment for migration [180] [170] . The example CLI commands for dry run vs live run [173] [181] .

## Scenario 21.3: Self-Service SSO Configuration for Tenants

**Concept:** A very advanced use-case in B2B SaaS is allowing each customer to **set up their own SSO** with your app, without developer intervention. Descope offers a **Self-Service SSO Setup** flow (often called an SSO Setup Suite) [182] [183] . This is essentially a widget/flow combination where a tenant admin can input their IdP details (like upload metadata XML for Okta/Azure) and Descope will create the SSO connection (tenant) on the fly. We will describe how you could integrate this to let, say, an admin from ACME Corp configure SSO for their users.

**Steps:**

1. **Enable SSO Suite Template:** In Descope Console, look for **SSO Setup Suite** tutorial or template [182] . Descope likely provides a ready flow for this. If not visible, it might be enabled on request or via their documentation. The idea: a Descope flow that includes screens for entering IdP info (like a field for IdP Metadata URL, a button to test connection, etc.), and logic to create a new SSO Tenant in your project with those details.

2. **Create an SSO Config Flow:** If a template is available, use it. If building manually:

3. Start a flow (for logged-in admins) that asks for the **Identity Provider Type** (Okta, ADFS, etc.) perhaps as a dropdown.
4. Ask for required info. For SAML: the Metadata URL or file upload, maybe allowed email domains for that tenant. For OIDC: client ID, secret, issuer URL.
5. Use Descope's **Management Action** to create a tenant and SSO configuration. Descope's Management APIs can programmatically create a tenant and set up SAML settings if provided. The flow likely uses an "SDK Management" action behind the scenes for this [182] .

6. Perhaps include a verification step: e.g., generate a test login link to ensure the IdP works.

7. **Expose to Customer Admin:** On your app's admin dashboard, have a section "SSO Integration" with a button "Set up SSO". When clicked, either:

8. Open the Descope SSO setup flow (embedded or a hosted link) for that tenant admin.

9. The flow will run, and if successful, that tenant's SSO is configured in Descope. Possibly the admin might get a confirmation screen with their assigned tenant ID or some info on how users can now login.

10. **Under the Hood:** Suppose ACME's admin enters their Okta metadata and domain `acme.com`. The flow calls Descope's API to:

11. Create a new Tenant (e.g., tenant name "ACME Corp").
12. Set allowed SSO domain to `acme.com` and attach the IdP metadata (certificate, SSO URL, entityID).
13. Possibly set up SCIM at same time if included (maybe not in initial flow).

14. The admin's own user account might be automatically linked to that tenant with an admin role.

15. **Test the New SSO:** After setup, ACME employees should be able to go to your app, enter their email `bob@acme.com`, and Descope will route them to their Okta for login (because domain matches the new tenant's SSO config). This is the ultimate goal: the customer set it up themselves, no dev needed.

16. **Admin UX:** Provide guidance around this flow: e.g., "You will need an Okta (or other IdP) admin account to gather certain information. Follow these steps... Then input here." Possibly link to a guide.

17. **Security:** Only allow this flow for trusted tenant admins. The flow itself should verify the user is allowed to create SSO (maybe check they have some "TenantAdmin" role in your app). The Descope actions themselves might also enforce that only an authorized management key is used to create tenants (the flow might use a constrained key behind scenes).

18. **Benefits:** This drastically reduces your support effort – you don't have to manually configure dozens of SSO connections for each enterprise client; they do it on their own in a guided way [183].

19. **Monitoring:** As a super admin of your app, you should keep an eye on new tenants created. Descope Console will list them. You might want to validate the configs or at least have a process to confirm everything is working (maybe your customer success team can be notified and assist if needed).

**References:** While we don't have the exact flow code, Descope docs note the existence of such a suite [182] [183]. It's an advanced feature often highlighted in Descope's enterprise use-cases.

---

This concludes the extended lab plan. By progressing through Days 16–21, a developer will have tackled advanced SSO scenarios, enterprise user management automation, enhanced security flows, DevOps integration, and migration strategies – all on top of the foundational Descope integration from the first 15 days. With this knowledge, you can implement a full-fledged customer identity solution in your React/Node application, from first login to scaling up enterprise integrations.

---

[1] [2] [3] [9] IdP-initiated SSO vs SP-initiated SSO
https://www.descope.com/blog/post/idp-vs-sp-sso

4 5 18 19 22 23 25 26 29 30 SAML Federated Applications Overview | Descope Documentation
https://docs.descope.com/identity-federation/applications/saml-apps

6 112 113 114 How to Setup a Mock SAML Tenant | Descope Documentation
https://docs.descope.com/management/tenant-management/sso/mock-saml-testing

7 10 11 66 162 163 Just-in-Time (JIT) Provisioning | Descope Documentation
https://docs.descope.com/sso/jit-provisioning

8 46 47 48 49 50 51 52 53 54 55 56 57 SCIM Provisioning with Azure | Descope Documentation
https://docs.descope.com/management/tenant-management/scim/azure-scim

12 13 14 15 75 76 79 81 Conditions in Flows | Descope Documentation
https://docs.descope.com/flows/conditions

16 17 31 32 Overview of SSO Integrations | Descope Documentation
https://docs.descope.com/sso-integrations

20 21 24 27 28 Single sign-on (SSO) - Mintlify
https://mintlify.com/docs/advanced/dashboard/sso

33 34 44 SCIM: What It Is & How It Works in 2025
https://www.descope.com/learn/post/scim

35 36 37 38 39 40 41 42 43 45 SCIM Provisioning with Okta | Descope Documentation
https://docs.descope.com/management/tenant-management/scim/okta-scim

58 59 60 61 62 63 64 67 68 69 Risks in Merging SSO Users with Non-SSO Identities | Descope Documentation
https://docs.descope.com/sso/merging-sso-identities-risk

65 83 84 119 120 121 122 123 124 125 126 128 129 Descope Terraform Provider | Descope Documentation
https://docs.descope.com/managing-environments/terraform

70 71 72 73 74 77 78 80 Device Intelligence & Bot Detection With Fingerprint & Descope
https://www.descope.com/blog/post/fingerprint-connector

82 85 88 89 130 JWT Templates | Descope Documentation
https://docs.descope.com/management/jwt-templates

86 87 90 94 131 React & Node.js Quickstart | Descope Documentation
https://docs.descope.com/getting-started/react/nodejs

91 92 93 159 160 Configuring Custom Domain with CNAME | Descope Documentation
https://docs.descope.com/how-to-deploy-to-production/custom-domain

95 96 97 98 99 100 101 102 103 104 105 Learn how to impersonate users | Descope Documentation
https://docs.descope.com/user-impersonation

106 107 108 109 110 111 115 116 117 118 Test Users in Backend | Descope Documentation
https://docs.descope.com/test-users/sdks

127 Managing Environments | Descope Documentation
https://docs.descope.com/managing-environments

132 133 134 135 136 137 138 139 140 141 142 143 144 145 Styles & Themes | Descope Documentation
https://docs.descope.com/management/styles

146 147 148 149 150 151 152 153 Widgets Overview | Descope Documentation
https://docs.descope.com/widgets

154 descope/descope-js: Descope JavaScript Packages - GitHub
https://github.com/descope/descope-js

155 156 157 161 Customize Passkey Authentication | Descope Documentation
https://docs.descope.com/auth-methods/passkeys

158 Can passkeys be used on other domains without an iframe? - Corbado
https://www.corbado.com/blog/iframe-passkeys-webauthn/passkeys-cross-domain-usage-without-iframe

164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 Migrate Auth0 Users to Descope | Descope Documentation
https://docs.descope.com/migrate/auth0

182 183 SSO and SCIM Tutorials | Descope Documentation
https://docs.descope.com/sso/tutorials