ChatGPT

Day 8: Role-Based Access Control (RBAC)

Scenario 22: Understanding RBAC and Setting Up Roles in Descope

Background: Role-Based Access Control (RBAC) is an **authorization model** that restricts access based on a user's role within an organization [1]. Instead of assigning permissions directly to individuals, permissions are grouped into roles (e.g. *Admin*, *Editor*, *Viewer*), and users are assigned those roles. This simplifies management and ensures users only have access to the resources needed for their job [2] [3]. In Descope, RBAC is supported natively: you can create roles and associated permissions in the Descope Console, and these roles will be embedded in the user's JWT upon login [4] [5]. Your application will then **enforce** access rules based on those roles/permissions in the token.

Hands-On – Steps:

1. **Create Roles and Permissions:** Log in to the Descope Console and navigate to the **Authorization** page (left menu). Under the **RBAC** tab, use the UI to create a couple of roles relevant to your sample app (for example, `Admin` and `User`) [6] [7]. For each role, define one or more permissions. For instance, an `Admin` role might have permissions like `user:delete` or `settings:update`, whereas a `User` role might only have `content:view`. Click **+ Role** to add a role and **+ Permission** to add granular permissions [7]. You can also edit existing default roles (Descope provides a default *Tenant Admin* role with some preset permissions) [8]. Each permission and role is identified by a string ID, which will appear in JWTs.

2. **Set Default Roles (Optional):** If you want all new users to automatically receive a certain role, mark it as a **Default Role** in the console [9]. For example, you might set the `User` role as default so that any newly registered account gets basic user permissions by default. This is useful to ensure a baseline access level for all users without manual assignment.

3. **Assign Roles to Users:** Next, assign roles to a user. Go to the **Users** section in Descope Console, select a test user (or create one), and edit their roles/permissions [10]. You can manually add the `Admin` role to your administrator account and `User` role to a normal account. Descope also allows assigning roles via the Management API/SDK or mapping roles from SSO providers [11] (we will explore SSO group-to-role mappings in a later scenario). At this stage, ensure you have at least one user with each role for testing.

4. **Verify Roles in JWT:** After assigning roles, test the authentication to see RBAC in action. Using your React app (from previous days) integrated with Descope, log in as a user and capture the JWT (session token). You can decode the JWT (e.g. using jwt.io or via your backend code) to inspect the payload. In the JWT, you should see a `roles` claim (and possibly `permissions` claim) containing the roles you assigned [12] [13]. For multi-tenant projects, roles may appear under a `tenants` object keyed by tenant ID, but for now we assume a single-tenant scenario. Confirm that the correct role string (e.g., `"Admin"`) is present in the token.

5. **Enforce RBAC in Backend:** Modify your Node.js backend to utilize RBAC information. For example, if you have an Express route that only admins should access (e.g. a user management API), implement a middleware to check the JWT for the `Admin` role. Using Descope's Node SDK,

you can verify the session token and then extract roles. Descope's client helpers include functions like `getJwtRoles()` to retrieve roles from a JWT [14]. In code, you might do:

```
const { getJwtRoles } = require('@descope/node-sdk');
// After verifying JWT (e.g., via descopeClient.validateSession):
const roles = getJwtRoles(sessionJwt);
if (!roles.includes('Admin')) {
    return res.status(403).send('Forbidden – Admins only');
}
```

This ensures that only users with the Admin role can proceed. Test this by calling the protected endpoint with an Admin user's token versus a regular user's token – the regular user should be rejected.

1. **Enforce RBAC in Frontend:** Similarly, update your React app to tailor the UI based on roles. Descope's React SDK provides a `useSession()` hook that gives the current session and a `useUser()` hook for user info [15]. You can get the user's roles from the session token on the client if not using HttpOnly cookies. For example, use `const { sessionToken } = useSession()` and then `getJwtRoles(sessionToken)` (from the React SDK's helpers) to get an array of role names [16] [17]. Hide or show UI elements depending on roles – e.g., show an "Admin Panel" link only if `roles.includes('Admin')`. This provides a better user experience by not even presenting options the user isn't allowed to use. Remember that frontend checks are just for convenience; your backend should still enforce critical access control, as done in the previous step.

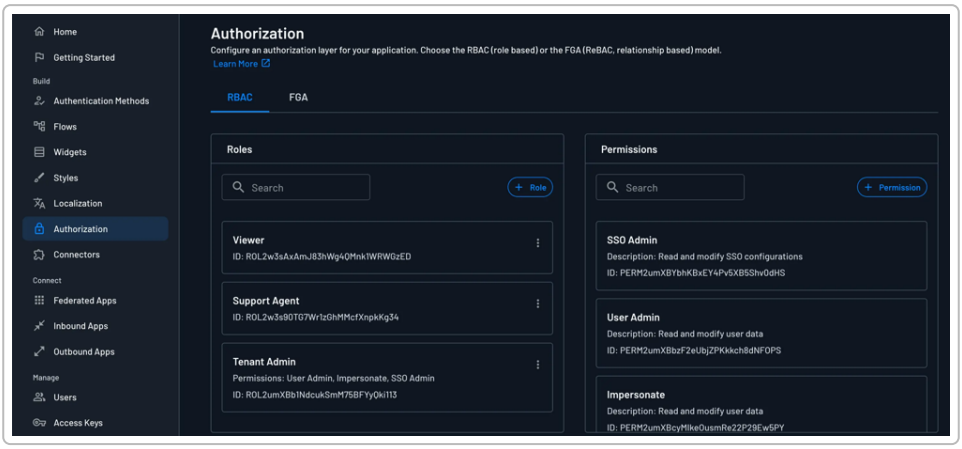

Figure: The Descope Console's RBAC management interface. Here you define Roles (left) and Permissions (right). Roles can be marked project-wide or tenant-specific, and each role aggregates one or more permissions. [18] 【4†L90-L98**

By the end of this scenario, you have set up a basic RBAC system: defined roles, attached them to users, and updated both backend and frontend to respect those roles. You should see how Descope makes the roles available via JWT claims, while the actual **authorization logic** (permitting or denying actions) is implemented in your application code [19] [20]. This division of responsibility is important: Descope authenticates and provides identity + role info, and your app consumes that info to enforce access control.

Scenario 23: Implementing Multi-Level RBAC (Project vs Tenant Roles)

Background: Descope supports roles at two levels – **Project-level roles** (global across all tenants) and **Tenant-level roles** (specific to a particular tenant or organization) [21]. This is useful in multi-tenant applications (e.g. a B2B SaaS) where you may want roles like "Tenant Admin" that apply within one company's realm. A user in Descope can even have different roles in different tenants (we will dive deeper into multi-tenancy later) [22] [23]. In this scenario, we extend our RBAC setup to understand tenant-scoped roles, even if our app is currently single-tenant, to prepare for future scalability.

Hands-On – Steps:

1. **Create a Tenant (for Simulation):** In the Descope Console, go to the **Tenants** page (under Management). Create a new tenant named "Demo Tenant" (or any name) to simulate a multi-tenant scenario [24]. This will allow you to create tenant-specific roles. (If your project already has a default tenant, you can use that as well.)

2. **Define Tenant-Level Role:** With a tenant selected (use the tenant dropdown in the Console), navigate to Authorization -> RBAC for that tenant. Now when you add a role, it will be a **tenant-level role** (the console will usually label it or show it under that tenant context) [25] [26]. Create a role, e.g. `Tenant Admin`, at the tenant level. Assign some permissions to it, such as the built-in "User Admin" or "Impersonate" permissions Descope provides [27]. Tenant-level roles are marked as such, while project roles apply everywhere. *Note:* Descope by default has a special Tenant Admin role in each tenant with permissions to manage that tenant's users and SSO settings [8] [28]. You can edit that or create additional ones.

3. **Assign User to Tenant and Role:** Still in the Tenants page, add one of your test users to the new tenant (using the "Users" tab in tenant settings, invite or assign an existing user to this tenant). Then assign the user the **Tenant Admin** role for that tenant [22]. This means that user will have elevated permissions, but only when operating within that tenant's context.

4. **Observe JWT Structure for Tenants:** Authenticate as the user who now has a tenant role. When the JWT is issued after login, it will contain a `tenants` claim with a nested structure mapping tenant IDs to that user's roles and permissions in that tenant [29] [30]. For example, the JWT payload may include:

   ```
   "tenants": {
       "T12345...": {
           "roles": ["Tenant Admin"],
           "permissions": ["User Admin","Impersonate"]
       }
   }
   ```

   alongside any project-level roles. This shows how Descope includes multi-tenant RBAC data in the token. If the same user had roles in multiple tenants, each tenant ID key would list its roles separately [22].

5. **Backend Enforcement with Tenant Context:** Adjust your backend role-check logic to account for tenant context if applicable. In a multi-tenant app, typically the client or the JWT will carry a current tenant identifier (often a claim like `dct` for "current tenant" or similar) [31]. Descope's

helper function `getJwtRoles(token, tenantId)` can fetch roles for a specific tenant [16]. In practice, your API endpoints might expect a header or JWT claim indicating which tenant the action pertains to. You would then check `if (roles.includes('Tenant Admin'))` for that tenant when performing tenant-scoped admin operations. For now, simulate this by taking the tenant ID from the JWT of your test user and using `getJwtRoles(sessionToken, tenantId)` in a Node script or in the app to retrieve the roles [17]. Ensure it returns the `Tenant Admin` role for that specific tenant ID.

6. **Frontend UI Cues for Tenant Role:** On the frontend, if building a multi-tenant UI, you might show a tenant switcher or label certain users as "Tenant Admin." While our simple app may not have multiple tenants yet, you can still demonstrate conceptually: for instance, if the logged-in user's JWT has any tenant roles, display them in the profile section. Use `getJwtRoles(sessionToken, tenantId)` from the React SDK helpers to get the role names for the current tenant and show a badge like "(Tenant Admin)" next to the user's name if applicable.

By completing this scenario, you've seen how Descope handles RBAC not just globally but within tenant boundaries. You created a tenant-specific role and saw it appear in JWTs under the tenant mapping [13] [32]. The key takeaway is that **Descope's RBAC is multi-tenant aware**: if your app grows to serve many organizations, you can isolate roles per tenant so that (for example) "Tenant Admin" of Tenant A cannot affect Tenant B's data [33]. Your app will need to pass along the tenant context and enforce checks accordingly. This sets the stage for a deeper dive into multi-tenancy in a later day.

Scenario 24: RBAC in Action – Building an Admin Dashboard (React & Node)

Background: Now that roles and permissions are in place, we'll build a simple admin dashboard page that only authorized users (Admins) can access. This scenario ties together the RBAC concepts: conditionally rendering UI in React and protecting routes on the server. It will also illustrate how to incorporate Descope's session management in React for role-based routing.

Hands-On – Steps:

1. **Protected Route Component (React):** Implement a reusable Protected Route component in your React app to guard admin pages. For example, using React Router, you can create a component `AdminRoute` that checks for an admin role:

```
import { useSession } from '@descope/react-sdk';
import { getJwtRoles } from '@descope/react-sdk';
import { Navigate } from 'react-router-dom';

const AdminRoute = ({ children }) => {
  const { isAuthenticated, isSessionLoading, sessionToken } =
useSession();
  if (isSessionLoading) return <p>Loading...</p>;
  if (!isAuthenticated) return <Navigate to="/login" />;
  const roles = getJwtRoles(sessionToken);
  if (!roles.includes('Admin')) {
    alert("You do not have access to this page.");
    return <Navigate to="/" />;
  }
```

```
  return children;
};
```

This component first ensures the user is logged in, then checks the `sessionToken` for an "Admin" role (adjust the role name if yours is different). If the check fails, it redirects or shows an error. *(Note:* If you configured `sessionTokenViaCookie` for HttpOnly cookies, the token won't be directly accessible in JS [34]. In that case, you'd rely on a backend check or an API call. For this exercise, we assume token is accessible or you use a method to call the backend for role info).

2. **Admin Dashboard Page:** Create a new page in your React app, e.g. `AdminDashboard.jsx`, that shows some privileged information (even static content for demo is fine). Wrap this page's route with the `AdminRoute` so it's only accessible to admins:

```
<Routes>
  <Route path="/admin" element={<AdminRoute><AdminDashboard/></
AdminRoute>} />
</Routes>
```

In `AdminDashboard`, use `useUser()` hook from Descope to display current user's name/email [35], and perhaps list the roles (you can display `getJwtRoles(sessionToken)` result). If you have an API (Node backend) that returns user lists or other sensitive data, you could call it from this page to demonstrate server-side protection as well.

3. **Server-side Enforcement:** On your Node.js backend, ensure any admin-specific API endpoints verify the user's admin status. We did something similar in the prior scenario. For example, if you have an endpoint `/api/admin/stats`, add middleware to decode and validate the JWT from the user's cookie or Authorization header. Use the Descope Node SDK's `validateSession` function (or JWT verification via JWKS) to ensure the token is valid. Then check the `jwt.roles` or `jwt.permissions` field. Descope's Node SDK could be used:

```
const descopeClient = DescopeClient({ projectId: PROJECT_ID });
app.get('/api/admin/stats', async (req, res) => {
   const token = req.cookies['DS'] || req.headers.authorization;
   try {
      const { user } = await descopeClient.validateSession(token);
      if (!user || !user.roles.includes('Admin')) {
         return res.status(403).send('Forbidden');
      }
      // proceed to send stats
      res.json(getSensitiveStats());
   } catch (e) {
      return res.status(401).send('Invalid session');
   }
});
```

In this snippet, `validateSession` checks the JWT and returns user info (which includes roles). Only if the user has Admin in their roles do we return the sensitive data. (If you used tenant roles, you'd also check the tenant context as discussed.)

4. **Testing:** Log in with your admin-role user and navigate to the `/admin` route in the React app. You should be allowed through, seeing the AdminDashboard content (e.g., "Welcome, Admin!" and any data). If you log in as a non-admin user and attempt to go to `/admin`, your ProtectedRoute should redirect you or show a not-authorized message. Similarly, test calling the protected API (`/api/admin/stats`) using an admin token vs a normal user token (you can use a tool like Postman or a fetch call in the dashboard). The admin should get a success response, while the normal user should get 403 Forbidden.

5. **Fine-Grained Permissions (Bonus):** If you want to take it further, test the difference between roles and direct permissions. Descope allows assigning **permissions directly to users** (without roles) or extra permissions beyond those conferred by roles [36]. For example, create a permission "BetaFeatureAccess" in Descope, and assign it to a user (via the console user editor). In your app, use Descope's helper `getJwtPermissions()` to check for that permission [14]. This approach can be used for very granular control, but managing via roles is usually easier. Still, understanding it helps: permissions appear in JWTs similarly, and you would check them in code just like roles.

After implementing the admin-only page and APIs, you have witnessed full-stack RBAC in action. **On the frontend**, the Descope React SDK's hooks (`useSession`, `useUser`) and the `<Descope>` component help manage authentication state, allowing you to conditionally render content [15]. **On the backend**, Descope's SDK or JWT verification ensures that even if someone tries to skip the UI and call an API directly, their token's roles are verified server-side [19]. This layered security – UI-level and API-level – is a best practice. Congrats, you've turned your simple app into one with enterprise-grade access control!

Day 9: Session Management and Secure Cookies

Scenario 25: Session Management Basics – Cookies vs Tokens

Background: Session management is a cornerstone of web authentication. After a user logs in, the app needs to remember that user's identity on subsequent requests. There are two primary ways to handle this: **Stateful sessions with cookies** (typically storing a session ID in a secure cookie tied to server-side session data) and **Stateless sessions with tokens** (like JWTs stored client-side and sent with each request). In modern apps (and in Descope's architecture), stateless JWT-based sessions are common, often combined with HTTP-only cookies for secure storage [34]. It's important to understand the difference: - *Cookie-based sessions* store an identifier in a browser cookie and maintain session data on the server (or in a database). They are simple but require server memory and have CSRF considerations. - *Token-based (JWT) sessions* issue a self-contained token to the client. The client sends the token on each request (via header or cookie), and the server validates it. JWTs are stateless (server doesn't need to store session info) and can contain user info/claims, but must be stored securely on the client to prevent theft [37].

A common question is whether to store JWTs in browser *local storage* or in *cookies*. **Security best practice** leans towards HTTP-only cookies because they are not accessible via JavaScript (mitigating XSS risks) and are automatically sent to the correct domain [34]. However, cookies are vulnerable to CSRF if not protected (using same-site flags or anti-CSRF tokens). Let's see how Descope handles sessions and what options it provides.

Hands-On – Steps:

1. **Examine Default Behavior:** By default, the Descope React SDK, when used as in previous scenarios, keeps the session token in memory and/or local storage (depending on config). When you logged in previously, Descope provided a session JWT and a refresh JWT. Check your browser's developer console **Application Storage** – you might see tokens stored (unless configured otherwise). Also check the **Network** tab for the login or API calls; if Descope's server set any cookies. Out-of-the-box, Descope can operate token-based without cookies, requiring you to handle token storage.

2. **Enable Secure Session Cookie:** Descope offers a configuration to manage the session token in an **HttpOnly cookie** (often named `DS` in Descope) for you. This is done either via project settings or SDK options. In the React SDK, you can initialize the `<AuthProvider>` with `sessionTokenViaCookie=true` or set it in project settings [38] [34]. Update your app initialization:

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

   With this setting, after login, Descope will drop a cookie (HttpOnly, Secure) containing the session JWT [34]. The token will **not** be accessible via `useSession().sessionToken` on the client (it will be `null` because the SDK can't read HttpOnly cookies). Instead, the cookie will be automatically sent in requests to your backend (if on the same domain) or to Descope if using their hosted endpoints.

3. **Configure Cookie Attributes:** Ensure your project's domain settings in Descope Console allow the cookie to be set correctly. If you're hosting the frontend at `http://localhost:3000` and backend at `http://localhost:4000` during development, cookies may need proper domain and same-site settings. In Descope, you can specify a custom domain (like `auth.myapp.com`) as the auth domain [39], which is recommended in production. For local testing, if the cookie isn't appearing, you might need to serve frontend and backend on the same host or adjust same-site policy (for example, set same-site to `None` and Secure for cross-site cookies).

4. **Test Cookie Session:** After enabling `sessionTokenViaCookie`, log in through your React app again. Now, check the browser's cookie storage: you should see a cookie named something like `DS` (and maybe `DSR` for refresh token) associated with your auth domain. The cookie will have flags **HttpOnly** and **Secure** set (Secure means it only transmits over HTTPS) [34]. Try to access `document.cookie` via the browser console – the Descope cookie won't be visible (HttpOnly), which is good for security. Also notice that `useSession()` hook in React might now show `isAuthenticated` true but `sessionToken` undefined, reflecting that the token is not in JS but only in the cookie.

5. **Session Persistence:** Descope's React SDK also has an option `persistTokens` which, if false, would keep tokens only in memory (so they vanish on refresh). By default it might persist. If you want to experiment, you can toggle `persistTokens` in the AuthProvider (alongside sessionTokenViaCookie). For example, set `persistTokens={false}` and `sessionTokenViaCookie={false}` – then the token will be kept in memory (not local storage) and effectively you'd be logged out on page refresh. Typically, however, you either use

cookie storage or persistent storage to keep the user logged in across sessions. The default should suffice for most cases.

6. **Server-Side Session Validation:** With the session cookie in place, update your Node backend to validate the cookie. For instance, if your backend endpoints receive the cookie automatically, you can retrieve it from `req.cookies['DS']`. Use Descope's Node SDK to validate it: `await descopeClient.validateSession(sessionTokenFromCookie)`. If valid, you get the user info; if not, you return 401. This is essentially stateless JWT validation, but the cookie mechanism gives you the convenience and security of built-in browser handling. Confirm that when you call an API endpoint from the React frontend (with `fetch` or Axios), the cookie is sent and your backend can verify the session without needing the client to manually attach tokens. (Remember to enable cookie parsing middleware like `cookie-parser` in Express, and perhaps CORS settings to allow credentials if your frontend and backend are on different origins during development).

7. **Timeouts and Refresh:** Descope issues a session JWT and a refresh JWT. The session token might be short-lived (e.g. 15 minutes) and the refresh token longer (e.g. 30 days). Test the automatic refresh feature: in the React SDK, if the session token is about to expire or is invalid, the SDK should use the refresh token (which might also be in a cookie or memory) to get a new session token transparently, as long as the user hasn't logged out. You can simulate expiry by manually calling a method or waiting. Descope's docs note that if using cookies, the refresh token cookie (`DSR`) won't be accessible to JS either [40], so the SDK handles refresh internally. This means your app can maintain sessions without storing tokens in local storage.

Key Takeaways: By storing the session JWT in an HttpOnly cookie, we significantly improved security: the token is immune to JavaScript-based attacks (XSS cannot steal what it can't read) [34]. We must however be mindful of CSRF – since browsers attach cookies automatically, malicious sites could trigger requests. To mitigate CSRF, ensure you use **SameSite** cookies (Lax or Strict) or anti-CSRF tokens if necessary, and limit sensitive operations to POST with CSRF checks. Descope's cookies are by default SameSite Lax (suitable for most cases where your frontend and backend share a root domain, but if using a custom auth domain you might adjust it). In contrast, if we stored the JWT in local storage, we'd have to guard it from XSS and manually include it in requests (usually via `Authorization: Bearer` header). Cookies vs tokens is a trade-off, but Descope gives you the best of both: stateless JWT auth with the option of secure cookie storage. Finally, remember to always use HTTPS in production so that Secure cookies are indeed secure, and never send session tokens over unsecured channels.

Scenario 26: Securing Session Cookies and Managing Session Lifecycle

Background: Now that we're using cookies for sessions, we need to manage the **lifecycle** of sessions properly: login, persistent stay logged in, logout, and idle timeout. Also, it's crucial to secure cookies with proper attributes and handle edge cases like cookie theft or expiration. In this scenario, we will: - Customize cookie settings (expiration, same-site). - Implement a logout function that clears the session. - Discuss session renewal and idle timeout.

Hands-On – Steps:

1. **Cookie Settings in Descope:** The Descope console allows configuration of session duration and refresh token duration (e.g., session TTL and refresh TTL). Check your Descope project settings for **Session Settings**. For instance, you might find default session expiration like 30 minutes and refresh for 60 days. You can adjust these if needed (for testing, you could set short expiration to

observe the behavior). Also note if there are settings for cookie attributes (some systems let you set SameSite policy). If not directly configurable, know that Descope cookies default to Secure, HttpOnly, and SameSite=Lax. Lax means the cookie isn't sent on cross-site *subresource* requests but will be on top-level navigations (which is usually fine for an API accessed by your own domain).

2. **Implement Logout (Frontend):** Add a **Logout** button in your React app that logs the user out. Descope's React SDK provides a `logout()` method accessible via `useDescope()` hook. For example:

```
import { useDescope } from '@descope/react-sdk';
...
const { logout } = useDescope();
const onLogout = () => {
    logout(); // This will clear tokens and cookies
};
```

   Attach `onLogout` to your logout button. When called, the SDK should clear the session on the client side. If using cookies, it will attempt to clear the cookie (though note, cookies can only be cleared by the domain that set them – since Descope set the DS cookie, the SDK likely triggers an API call to Descope to invalidate the session and instruct the browser to remove the cookie). In any case, after logout, `isAuthenticated` should become false and user info cleared.

3. **Implement Logout (Backend API):** For extra safety, or if you want to allow logout via backend, you can call Descope's **Management API** or SDK to revoke the refresh token or session. Descope might provide a `logout` endpoint in the SDK for backend as well, which would invalidate the token on the server side. This ensures even if a token was stolen, it can't be used. However, often just clearing the client is enough for basic apps, as the JWT will expire soon anyway. Check Descope docs if there is an explicit logout API call (could be something like `descopeClient.logout(sessionToken)` – if not, you can simply delete the cookie on the client and rely on short expiry).

4. **Idle Timeout and Keep-Alive:** Session idle timeout means if the user is inactive for a period, the session ends. With JWTs, this is usually handled by token expiry rather than tracking activity. Suppose your session JWT lasts 15 minutes. If the user doesn't trigger a refresh within 15 minutes, it expires. The next API call will fail auth, prompting re-login or use of refresh token. Descope's SDK likely auto-refreshes the token as long as the refresh token is valid. Test this by setting a short session TTL (if possible) – say 1 minute – and see if an API call at 1:30 after login still works. If yes, it means the refresh was used. The refresh token typically has a sliding window or fixed lifetime (e.g., 30 days). If you remain active, each time the session token is refreshed, you stay logged in. If you're completely idle beyond refresh token expiry, you'll be logged out. Ensure your app can handle that gracefully (e.g., if `isAuthenticated` flips to false because the token expired and refresh failed, redirect to login).

5. **Secure Cookie in Production:** In development, you might not be on HTTPS, which could prevent Secure cookies from being set. When deploying, you must serve your app over HTTPS (which you will). If using a custom auth domain (like `auth.myapp.com` for Descope), make sure to add that in your Descope settings and use it in the AuthProvider `baseUrl` config [39]. This custom domain approach is recommended to ensure cookies are first-party to your app's domain. For

example, if your app runs on `app.myapp.com` and Descope on `auth.myapp.com`, you can set cookies scoped to `.myapp.com` domain, achieving single sign-on across subdomains. The React SDK snippet `AuthProvider projectId="..." baseUrl="https://auth.myapp.com"` will direct the SDK to use your custom domain for all calls and cookie setting [41].

6. **Preventing Session Theft:** Even with HttpOnly, consider scenarios like someone stealing a user's cookie (perhaps via physical access or some network vulnerability). The primary mitigations are: short token lifetimes, binding tokens to context (Descope may embed a `cf-connecting-ip` or device fingerprint claim to detect token reuse – not sure, but audit logs show fields like IP and bot score [42]). Educate that using refresh rotation and monitoring unusual activity (like a token used from two different IPs) is important. In Descope's audit logs, events like `LoginSucceeded`, `UserRefresh`, etc., are recorded [43]. You as a developer can tap into these audit trails to detect anomalies. We will explore audit logs in Day 15.

In summary, this scenario ensures you can **log users out** properly and that your session cookies are configured for maximal security. We've highlighted the importance of HttpOnly, Secure, and SameSite attributes for cookies, which Descope handles by default [34]. We also discussed how token expiration and refresh work together for session longevity. With logout implemented, your app avoids the common mistake of "zombie sessions" (where a token stays valid indefinitely). Always test the full login→protected action→logout→login flow to ensure the session behaves correctly. At this point, your authentication is not only functional but **robust and secure**, following best practices for session management in a modern React/Node application.

Scenario 27: Dealing with Session Expiry, Revocation, and Concurrency

Background: In a real-world application, you need to consider what happens if a user's session expires or if you want to revoke sessions (e.g., force log out from other devices). Also, handling multiple concurrent sessions for the same user (login from multiple devices) is something to think about. Descope, as a full-fledged auth system, provides tools to manage these aspects. In this scenario, we simulate some of these situations: - Forcing a token to expire (to see refresh in action). - Revoking a session or blocking a user. - Notifying the frontend of session changes.

Hands-On – Steps:

1. **Simulate Token Expiry:** As mentioned, one way is to set a short expiration in Descope settings. If that's not feasible, you can manually decode the JWT to find its `exp` claim (expiry timestamp) and then wait until that time passes. After expiry, any API calls with that token should fail. Try this: log in and copy the access JWT. Stop activity until it's expired (or use an expired token explicitly via a REST client). Call a protected endpoint – you should get a 401 Unauthorized. If your React app is using the SDK, the next action after expiry might trigger the refresh flow. Descope's SDK likely handles this by using the refresh token to get a new JWT behind the scenes, updating `sessionToken` (or the cookie) and `isAuthenticated` remains true. Check the network calls – you might see a request to `.../refresh` endpoint. Verify that after refresh, the action succeeds. This builds confidence that token renewal works and users won't be abruptly logged out at the session TTL as long as they have a valid refresh token and activity.

2. **Revoke Sessions via Console:** In the Descope Console, navigate to the Users section, and find the user you logged in with. Descope might list active sessions or devices for that user (some systems show a list of login sessions). If available, try revoking or invalidating the session. For

example, some consoles have a “Logout user” or “Invalidate tokens” option. If you do this, the refresh token is typically invalidated server-side. Back in your application, try to use the app again (or refresh the page). The next attempt to use the session should fail because the session is no longer recognized. The React SDK might emit an event or simply `isAuthenticated` flips to false when it cannot refresh. (Descope provides event listeners like `onSessionTokenChange` and `onUserChange` via `useDescope()` [44] [45] – you could hook into those to detect if the user gets logged out in the background.)

3. **Implement Session List/Logout-All (Optional):** For an enhanced UX, you could create a page where users can see their active sessions and log them out remotely (e.g., “Log out of other devices”). Descope’s Management API likely has endpoints to list a user’s sessions or to revoke tokens. While an in-depth implementation is beyond this scenario, conceptually you’d call an API to Descope that invalidates all tokens for a user except the current, or all including current. Since we can’t easily demonstrate without specific API docs, just be aware this capability exists (many IdPs allow global sign-out). Check Descope documentation for “Session Management” – possibly there’s an API to revoke refresh tokens globally.

4. **Handle Session Expiry in Frontend:** Improve your React app to handle the case when a session expires and refresh token is also expired or invalid. Without any handling, a user might click something and nothing happens (if an API call fails silently) or they get an error response but no clear guidance. A good practice is to intercept 401/403 responses in a global AJAX handler – if you get “token expired” from your backend, you can force a logout on the client (clear state and redirect to login). Since Descope’s SDK likely manages some of this, another approach is using the event listeners. For example, using `sdk.onIsAuthenticatedChange` to know when it flips to false and then show a message “Your session has expired, please log in again.” [46] [47]. Implement a `useEffect` in your root component to subscribe to these events:

   ```
   const sdk = useDescope();
   useEffect(() => {
     const unsubAuth = sdk.onIsAuthenticatedChange((auth) => {
        if (!auth) {
           // e.g., redirect to login
           navigate('/login?sessionExpired=true');
        }
     });
     return () => unsubAuth();
   }, [sdk]);
   ```

   This ensures if the user is logged out from elsewhere or token expires with no refresh, the app responds.

5. **Test Concurrent Sessions:** Log in from two different browsers (or one in normal mode, one in incognito as a separate session) using the same user account. This will create two sessions (with two sets of tokens). They operate independently. If you perform logout via one browser (either using your app’s logout or via console revocation), observe what happens in the other. Likely, the other remains logged in (since its refresh token is different and not revoked if you only logged out one session). If you revoked via console all sessions, both should eventually drop. This test helps confirm that session state is isolated by token – one browser’s actions don’t directly affect another’s session unless a centralized revoke is done.

By completing this scenario, you have learned to handle the **session lifecycle gracefully**. You've ensured that expired or invalid sessions don't linger unnoticed and that users are informed or redirected appropriately. You also understand how Descope deals with multiple sessions. These are important for a polished, secure auth implementation: users will appreciate being logged out when they should be (security) and staying logged in when they're active (usability). With robust session management, our app is closer to production-ready.

Day 10: Custom Authentication Flows with Descope

Scenario 28: Introduction to Descope Flows and Building a Custom Flow

Background: Descope **Flows** are a powerful no-code/low-code feature that let you design authentication workflows through a visual builder [48]. Instead of coding the entire login or sign-up logic, you can drag-and-drop screens, actions (like sending OTPs), decision conditions, and even integrate 3rd-party connectors in a flow [49] [50]. Descope provides default flows (e.g. "sign-up-or-in", "sign-in", "reset-password") that are ready to use, but you can customize them or create new ones to fit your app's needs. In this scenario, we'll create a custom flow that, for example, collects an extra profile field during sign-up and requires email verification before completing login – a common custom auth journey.

Hands-On – Steps:

1. **Open the Flow Builder:** Log in to Descope Console and go to the **Flows** section [51]. You will see a list of flows. The default ones (sign-up, sign-in, etc.) might be locked from deletion [52], but you can view and clone them. Click **+ Flow** to create a new custom flow [53]. Give it a name ("CustomSignUp") and a Flow ID (e.g., `custom-signup`). The Flow ID is important – you'll use it in your React app to invoke this flow [54].

2. **Design the Flow – Screens and Actions:** You'll enter the visual editor. It typically starts with a **Start** node and you can add **Screens** (UI forms), **Actions** (logic like "Send Email OTP"), **Conditions** (branching), and connectors. For our custom sign-up flow:

3. Drag a **Screen** node onto the canvas (choose a template like a sign-up form or blank form). Add fields you want: perhaps Email (if not already), Name, and a custom field (e.g., "Favorite Color"). Mark required fields appropriately.
4. After the screen, add an **Action** node for verification. For example, to verify email, add the **"Verify Email"** action (Descope likely has a built-in action to send an email OTP or magic link). Connect the screen's success output to this action node. Configure the action: it may ask which field is the email (choose the email input from previous screen) and whether to send an OTP code or link.
5. Add a **Screen** for OTP entry if you chose OTP, or if using magic link, add a waiting screen or use the default mechanism that handles it. (To keep it straightforward, Descope's ready flows for sign-up already handle verification; you can mimic that).
6. Optionally, add a **Condition** node: for example, check if `freshlyMigrated` attribute is true (just as an example from earlier, but more practically, you could branch "Is New User?" if sign-up or an existing user logs in). Conditions allow dynamic flows – e.g., if a user's email is already verified, skip the OTP screen [55] [56].
7. End the flow at a **"Complete"** or success node which issues the session token. Ensure all branches eventually lead to an end (success or failure).

If this sounds complex, note that you can also start from a **Flow Template Library** that Descope provides [57] – perhaps clone the "sign-up or in" flow and modify it. The important part is understanding the building blocks: **Screens** (UI), **Actions** (backend tasks like send OTP, create user, etc.), **Conditions** (if/else logic), **Subflows** (reusable mini-flows), and **Connectors** (integrations like posting events to Segment) [49] [50]. Take some time to experiment: for example, you might add a **Welcome Screen** at the start that just has a logo and a "Continue" button linking to the actual auth screen, just to practice linking screens.

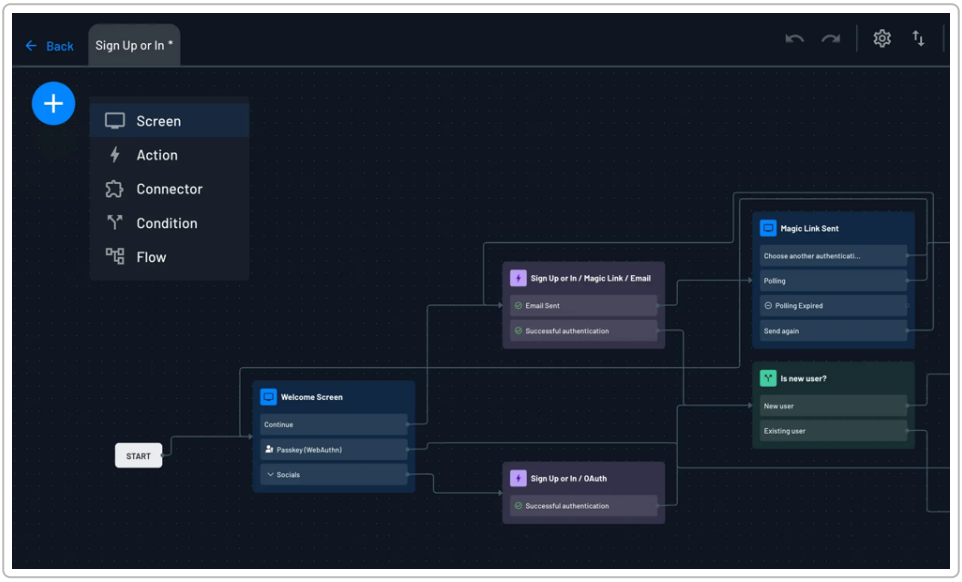

Figure: The Descope Flow editor interface. In this example, a default "Sign Up or In" flow is shown, containing multiple screens (blue nodes) and actions (purple). Flows begin at Start, proceed through UI steps and actions, and branch on conditions (green nodes) as configured. [48] [49]

1. **Save and Test the Flow:** Click **Save** (Ctrl+S shortcut) to save your flow changes [58]. Descope provides a **Flow Runner** or preview mode to test flows right in the console [59]. Use that to simulate the flow. For instance, when you run it, does it send an email OTP to the address you input? If the OTP is entered, does it complete and create a user? Check the Descope Users list to confirm a new user was created with the data you collected. Tweak the flow if something doesn't work (you might need to ensure the first screen uses a Descope built-in component for email, so that the subsequent verification action knows where to send the code). The Flow library docs and tutorials can help if you get stuck [60] [56]. Common tip: using Descope's predefined **Screen components** (like "Identifier" for email/phone, "Password", "Code Input") speeds up setup because they tie into actions automatically.

2. **Import Flow into React App:** Now that `custom-signup` flow is ready, integrate it in your React app. The Descope React SDK offers a `<Descope/>` component that renders a given flow's UI in your app, as we saw in earlier quickstart [61]. In your signup page (or wherever appropriate, maybe a dedicated route like `/register`), use:

```
<Descope flowId="custom-signup" onSuccess={(e) => console.log("Signup complete", e.detail.user)} onError={(e) => console.error("Flow error", e.detail.error)} />
```

This will mount your custom flow. The `onSuccess` can be used to handle what happens after completion (Descope might log the user in by default on success, issuing a session token – typically yes, after sign-up, you get logged in). You could redirect the user to the main app page in onSuccess.

Ensure that the AuthProvider (from earlier) is wrapping this component so that it functions. If you want to apply custom styling, recall you can pass a `theme` or `styleId` (we'll cover styling later, but just know it's possible to theme flows).

1. **Run and Verify in App:** Start your React app and navigate to the registration page with the embedded Descope flow. You should see the UI that you configured in the flow builder (perhaps a multi-step sign-up). Try creating a new user via this flow. The Descope SDK will handle displaying the screens, sending requests for OTP, etc., according to your flow definition. If the flow completes successfully, the `onSuccess` should fire; your app can then treat the user as logged in (the SDK will have stored the session). Check that the new user appears in Descope console with the extra fields (like Favorite Color) captured – likely they are saved as [custom user attributes] if you set that up in the flow builder for the field. If not automatically saved, you can modify the flow by adding an **Update User** action to save custom fields.

2. **Flow Error Handling:** You can also practice error handling. For instance, if you try to sign up with an email that's already in use, your flow might throw an error (Descope will likely show a message like "User already exists"). The `onError` callback can capture technical details. You might want to customize error messages or screens – Descope allows customizing **Flow Errors** in the builder (like defining what to do on a particular error) [62]. For now, note that `onError` gives you an opportunity to log or display a custom message. You could integrate it with your app's notification system (like show a toast if `onError` fires).

By designing a custom flow, you've unlocked the ability to tailor the user onboarding experience without writing a lot of custom code or forms. Descope Flows handle the heavy lifting (sending emails, verifying OTPs, creating users, etc.) while giving you control over the sequence. This approach greatly speeds up development – e.g., adding a new step to require phone verification is as simple as dropping in a new action and screen, rather than implementing an entire phone OTP microservice. **Visually** mapping out authentication also helps ensure you cover all branches (success, failure, retry), leading to a more robust flow. In the next scenarios, we'll integrate more advanced elements like connectors and see how flows work with SSO.

Scenario 29: Adding Third-Party Integrations in Flows (Analytics & Webhooks)

Background: One of the powerful features of Descope Flows is the ability to integrate with third-party services via **Connectors**. This means you can, for example, send an event to an analytics service when a user signs up, or log an audit entry, or call a webhook. This scenario focuses on adding a Segment analytics tracker into the flow and a custom webhook to notify an external system on certain auth events. This not only demonstrates flows' extensibility but also touches on **user analytics and event tracking** which are part of our topics.

Hands-On – Steps:

1. **Configure a Segment Connector:** In Descope Console, go to **Integrations -> Connectors**. Find **Segment** under Analytics connectors [63] [64]. To set it up, you need a Segment **Write Key** (from your Segment workspace) [65]. If you don't have a Segment account, you can skip actual key or

use a dummy, but in real scenario you'd get it from Segment's settings. Create a new Segment connector in Descope, give it a name, and input the Write Key [66]. Test the connector configuration in Descope – clicking "Test" should show a success result if the key is valid [67]. Save the connector.

2. **Insert Connector into Flow:** Return to your custom flow in the Flow editor. Suppose we want to send an analytics event when a user completes sign-up. You can add the Segment connector as a node in the flow. In the left toolbox, there might be a **Connector** option. Add a **Connector Action** after the user successfully signs up (but before the flow ends). When adding it, select your Segment connector and choose the type of event: Segment typically has "Track Event" or "Identify User". Let's say we use **Track** to record a "UserSignedUp" event. The builder will prompt for fields:

3. **User ID**: you can map this to the Descope user's ID or email. For example, use the template variable `{{user.userId}}` for the unique user ID [68].
4. **Traits or Properties**: you can send additional data. For instance, `email: {{user.email}}`, `favoriteColor: {{flow.data.favoriteColor}}` if that was captured.
5. **Event name**: perhaps "User Signed Up" or a code like `user_signup`.

Place this connector node so that it executes after the user creation and verification. Essentially: Screen -> Verify -> (User is now created) -> **Segment Track** -> End. The flow will then automatically push that event to Segment in real-time whenever someone goes through it [69] [70].

1. **Add a Webhook (Audit) Connector:** Let's also demonstrate a custom webhook. Imagine you have an internal admin system that should be notified on each new registration (maybe to approve the account or just for logging). Descope connectors can call a webhook via an **Audit Webhook** or generic HTTP connector. If available, set up a **Webhook Connector**. (If Descope doesn't have a straightforward generic webhook, another approach: use the **Audit Streaming** feature – but that's more complex, so let's assume a simpler path.) For demonstration, you can use a service like **Webhook.site** to generate a test URL that collects requests.

2. Go to Webhook.site and copy a unique URL.

3. In Descope Connectors, look for a "Webhook" connector type (Descope documentation references an Audit Webhook connector [71]). If it's specifically for audit events, we might need to instead rely on Descope's built-in Audit streaming. However, a simpler hack: Use **HTTP Connector** if Descope supports one (some IdPs allow an outgoing HTTP request as a flow action).
4. If possible, create a new connector where you configure the URL (the Webhook.site URL), and choose what data to send (perhaps the user JSON).
5. Back in the flow, add this connector node at the same point (after sign-up). Now two connectors will fire: one to Segment, one to your webhook.

If Descope doesn't allow arbitrary connectors in flow (it likely does, given it lists connectors for various categories including "External Token", "Fraud", "Network", etc., in docs), you might skip this or just conceptually note it. For our scenario, assume we set one up.

1. **Test the Flow with Connectors:** Save the flow and run a test sign-up through the console or your app. After completing:

2. Check your Segment dashboard (or logs) to see if an event was received. If using a dummy key, you obviously won't see real data, but in a real integration you would see an event like "User Signed Up" with the user's info in Segment [70].

3. Check the Webhook.site page – you should see a request logged when your flow ran. It might contain a JSON payload of the user or some data from the flow (depending on how you configured it). This confirms the connector executed. The Audit Webhook connector (if used) might send Descope's standard audit event JSON (which includes event type, user, timestamp, etc.) [72].

4. **Analytics in React App:** While connectors send data behind the scenes, you can also use Descope's client hooks for analytics. For instance, the `onSuccess` callback from `<Descope>` gave you the new user object. You could manually call analytics in your React code (like if not using connectors, you might call `analytics.track("Signup", {userId: ...})`). But using connectors inside flows has the advantage of capturing events even if the front-end doesn't explicitly do it, and ensures consistency (e.g., events fire even if using different platforms like if you had a mobile app using the same flow). It's good to know both approaches.

Key Takeaways: We integrated third-party services in our auth flow with zero additional backend code in our app. The Segment connector demonstrates Descope's ability to emit events to analytics platforms whenever certain flow steps occur [73]. This is extremely valuable for product teams – you can measure conversion rates (how many start signup vs finish), drop-off points, etc., by instrumenting flows. Additionally, using webhooks or audit streams, you can forward security events to monitoring systems or trigger custom logic (for example, alerting an admin of a high-risk login). These integrations show Descope can act as a mini-orchestration engine during authentication, not just a static login box. In later scenarios (Day 15) we'll talk more about audit logs and tracking, but you've already had a glimpse: if needed, every authentication event can be captured and analyzed, which is important for security analytics and business intelligence.

Scenario 30: Bring Your Own UI – Customizing Flow Screens and Actions

Background: While Descope's visual flow builder is great, sometimes you want full control over the user interface or need to execute custom logic not available as a built-in action. Descope Flows support a concept called **"Bring Your Own Screen" (BYOS)** [74] [62], which allows developers to use their own UI components at certain points in the flow while still leveraging Descope for the heavy lifting. Additionally, the Descope SDKs let you trigger flow actions programmatically if needed. In this scenario, we'll modify our custom flow to incorporate a custom UI screen (built in React) and see how to integrate it with the flow's progression.

Hands-On – Steps:

1. **Identify a Screen for Custom UI:** Let's say in our custom signup flow we want to collect some profile info using a fancy custom React component (maybe an avatar upload or a multi-step form) which the Descope builder doesn't have natively. We can create a placeholder in the flow for "External Screen". In the Descope Flow builder, add a **Screen** where you want the custom UI to fit, but use a special type or annotation that marks it as an external screen (Descope might refer to it as an **External Step** or require a specific screen ID). According to Descope docs, the concept is "Bring Your Own Screen" which likely means that at that node, the flow will call out to the client to render something custom [74].

2. **Add Flow Hooks for BYOS:** In your React app, using the Descope SDK, there may be a way to intercept when an external screen is needed. Possibly the `onSuccess` or `onError` events are not enough. Instead, Descope might provide an event like `onBeforeScreenLoad` or they allow the flow to call a custom callback. Lacking specific docs, we can simulate by splitting the flow:

3. Suppose we break our flow into two: Flow A handles up to the point before custom UI, then pauses. The React app does something, then resumes with Flow B or the same flow continuing via an API call.
4. Alternatively, an easier approach: Don't use the Descope component for that portion. Instead, have a route in your app do the custom step, then when done, call a Descope SDK method to continue the flow (like create user, etc.).

For a concrete example, consider user onboarding that requires reading Terms and Conditions. You might not want that as a plain checkbox in Descope screen – instead, you want to show a custom styled T&C page. You could handle it by: after user enters email/password via Descope flow, before finalizing account, you *pause*, show your custom T&C component in React, and on accept, call an SDK function to mark flow as complete.

1. **Use SDK Actions Programmatically:** Descope's Client SDK likely has functions corresponding to auth actions (e.g., `auth.signUp()` with parameters). If BYOS is complicated, one pragmatic solution is to use Descope flows for main things and custom code for edge things. For instance, you could remove a screen from the flow and instead handle that in React: e.g., remove Favorite Color from flow, and after Descope flow finishes (user created), in your app ask for favorite color and then call `descopeClient.user.update({ customAttributes: { favoriteColor } })` via the Management SDK to update user profile. This way, you combine custom UI with Descope's backend. This scenario is about understanding that you *can* step outside the pre-built UI.

2. **Implement a Custom Step:** To demonstrate, modify the React flow usage:

3. In your `<Descope flowId="...">`, add a prop `onStep={(stepDetails) => { ... }}` if available. This is hypothetical – some SDKs allow intercepting each step of a flow. If Descope's does not, another method: you could end the Descope flow early and use the `onSuccess` of that partial flow as the trigger to show a custom screen.
4. For example, create a flow that ends right before the custom step (so Descope creates the user and perhaps issues a token but knows it's not fully onboarded). In `onSuccess`, you know user is created, then show your custom React form (maybe a modal or redirect to a `/complete-profile` route).
5. In that custom form, collect whatever data, then call Descope management API (with the session token) to save it. Also possibly mark some user flag "profileComplete=true".

6. Finally, redirect the user to the main app. In effect, you stitched a custom UI step after the Descope flow finished. This isn't exactly BYOS in the middle, but it achieves the goal without losing the Descope session context.

7. **Testing the Custom Integration:** Try the end-to-end: user goes through Descope screens, then your custom component appears, they interact, and it completes. Ensure that the user remains logged in throughout (if you needed them to be logged in to call management APIs, you might rely on the session token from the partial flow's result). One challenge is that if the flow didn't issue a session until fully complete, you might need to manually log the user in via SDK after

user creation. Descope's APIs often allow creating a user with verified email and returning a session JWT. If not, a workaround is using a silent login.

8. **Alternative: Customize Descope Screens via Code Mode:** A simpler way for many customizations is to actually use Descope's **Code Mode** in the style editor or screen builder to inject custom HTML/CSS/JS in screens [75] [76]. The flow builder allows editing screen components and styles, potentially adding custom CSS or scripts. If your custom need is purely visual (like custom fonts, colors, or adding an HTML snippet), you might achieve it inside the Descope environment rather than a full BYOS. For instance, if you want an avatar upload, Descope might not have that component – so BYOS might still be needed. But for adding a hyperlink to terms, you could simply edit the screen HTML to include it.

This scenario is a bit abstract without Descope's specific BYOS API documentation. The main point to convey is that **Descope doesn't lock you in to only their UI**. You can integrate flows partially or use the underlying APIs to build your own UX on top of Descope's engine. In practice, many developers will use the provided `<Descope>` component for speed, but it's good to know you can drop down to lower-level control. For example, you might design a totally custom login page in React but still call Descope's SDK functions (like `loginWithPassword(email, password)`) to authenticate – essentially bypassing flows entirely for that use-case and treating Descope more like an API. The trade-off is you then must handle edge cases (resend OTP UI, error displays) yourself. Most often, a mix is used: use Descope flows for standard flows, and custom code for unique requirements.

At this point, you have a comprehensive understanding of Descope Flows: designing in the console, embedding in the app, integrating with external services, and even extending/customizing them beyond the default capabilities. This sets the stage for more advanced topics such as Single Sign-On (SSO) integrations and migrations, which we'll tackle in the coming days.

Day 11: Integration with Okta via SAML SSO

Scenario 31: SAML SSO Fundamentals and Okta as an Identity Provider

Background: SAML (Security Assertion Markup Language) is an XML-based standard for single sign-on, commonly used in enterprise SSO scenarios. In SAML, a **Identity Provider (IdP)** like Okta authenticates users and issues an assertion about their identity to a **Service Provider (SP)** (your app, or Descope on your app's behalf) [77]. The user can sign in once with the IdP (Okta) and then access multiple SPs without re-entering credentials. Understanding SAML terminology is key: - *IdP* (Identity Provider): The authority (Okta in our case) that holds user credentials and performs authentication. - *SP* (Service Provider): The application that wants to outsource authentication to the IdP (your app/Descope acts as SP). - *SAML Assertion*: The XML message from IdP to SP containing the auth result and user attributes (often sent via browser redirect or POST). - *Metadata*: Both IdP and SP have metadata (XML docs or URLs) that describe how to communicate (endpoints, certificates, identifiers).

Descope can simplify SAML setup by acting as an intermediary: you configure Okta as an IdP in Descope, and Descope handles the SAML protocol, then gives your app a normal Descope session. This scenario focuses on setting up the Okta side and Descope side for SAML SSO.

Hands-On – Steps:

1. **Create an Okta Developer Account:** If you don't already have one, sign up for a free Okta developer org (developer.okta.com). This gives you an admin console to create applications and users. Once in Okta, add a test user (or use your account).

2. **Add a SAML App in Okta:** In the Okta admin dashboard, go to **Applications** > **Create App Integration**. Choose **SAML 2.0** as the sign-in method (Okta might also have a pre-built Descope app in their marketplace, but here we do custom). Give the app a name (e.g., "My Descope SAML App"). In SAML settings:

3. **SP Entity ID (Audience URI)**: This should match what Descope expects. For now, put a placeholder, we'll update it. (If Descope provides an Entity ID value in their console, use that).
4. **ACS URL (Assertion Consumer Service URL)**: This is where Okta will send the SAML response. Descope will provide this URL (essentially an endpoint on Descope that consumes the login response) [78] .
5. You might not know those yet, so you can save the app for now and come back to it. Okta will allow editing it later.

Once app is created, assign it to users (in Okta's Assignments tab, choose the users or groups that should be allowed to use this SSO) [79] .

1. **Configure SAML in Descope (SP side):** In Descope Console, go to **Authentication Methods -> SSO** and choose **SAML**. You may have to select a **Tenant** first, because SSO settings are often per-tenant (since each customer organization might have their own IdP). Pick the tenant (maybe default or create one "OktaTenant"). Now:
2. Descope will show **SP Details**: Entity ID, ACS URL, maybe a metadata URL [80] [81] . Copy these values.
3. In Descope, you also enter IdP info: you will need Okta's **IdP Metadata URL** or certificate. Okta provides a metadata XML URL (often on the app's Sign On section or in Okta's metadata listing) [82] . For Okta, you might find it under the app -> Sign On -> "Identity Provider metadata" link.
4. Paste Okta's Metadata URL into Descope's IdP configuration. This typically fills in Okta's entity ID, SSO URL, and certificate for you in Descope.
5. Still in Descope, set the **Allowed Domains / SSO Domain** for the tenant. For example, if your company domain is `example.com` and Okta user emails are under that, put `example.com` [83] . This ensures that if a user with email `alice@example.com` tries to login, Descope knows to route them to SSO (domain routing, likely).

6. Also set a **Post Authentication Redirect URL** in Descope's SSO settings: this is where users land after SAML login [84] . For instance, your app URL (e.g., `http://localhost:3000/dashboard`).

7. **Update Okta with Descope SP Info:** Now go back to Okta's SAML app settings and plug in the Descope **Entity ID** and **ACS URL** you copied [78] . Okta likely has fields in the app configuration:

8. Audience URI (SP Entity ID) -> put Descope's Entity ID.

9. ACS URL -> put Descope's ACS URL. Save the changes. This connects the Okta app to Descope's endpoints.

10. **Test SP-Initiated SSO:** In an SP-initiated flow, your app/Descope will redirect the user to Okta. To test this, we need a link or trigger. Descope might provide a **"Login with SSO"** link via flows or you can craft one. Typically, if you use the Descope SDK's <Descope> component with a flow that has SSO, it could handle it. But a direct way: if you go to Descope's hosted page for SSO: e.g., `https://YOUR_PROJECT_ID.descope.app/sso?tenant=<tenant_id>`, it might redirect to Okta. Alternatively, in your React app, you could use Descope's JS SDK method `Descope.loginWithSaml(tenantId)` if exists.

Easiest is to use the default **"sign-up-or-in"** flow which often checks domain: In that flow, if you enter an email with the domain you configured (`example.com`), Descope might automatically redirect to Okta (this is a feature called domain routing or enforced SSO) [83]. Try this: on your login UI (Descope flow), enter the email of the test user you have in Okta (with the domain allowed). Instead of normal password prompt, Descope might show "Continue with SSO" or directly redirect. Complete the Okta login (enter Okta username/password). If all is set correctly, Okta will post a SAML Response to Descope's ACS, Descope will map the user and create a Descope session, and finally redirect you to your app's redirect URL with a token or just logged in.

You should end up logged in to the app (Descope issues its JWT, etc.) and the user exists in Descope user list (likely created Just-In-Time if not already) [78].

1. **Test IdP-Initiated SSO:** IdP-initiated means the user starts at Okta. In Okta's app dashboard, you should see the app tile for the app you created. Click it – Okta will generate a SAML Response and send to Descope ACS URL, logging you in. Ensure this works too. Descope requires a **Post Authentication Redirect** for IdP-initiated (which we set). After clicking the Okta tile, you should be taken to your app's page already logged in [85].

2. **Attribute & Group Mapping:** By default, your SAML assertion might only include the user's Okta username/email. If you want Descope to also get roles or other attributes from Okta, you configure **Attribute Statements** in Okta and mapping in Descope. For example, in Okta's SAML app settings, you can send `firstName`, `lastName`, or group memberships. In Descope's SSO settings, map those attributes to Descope's user fields or roles [86] [87]. A powerful feature is **Group to Role mapping** [88] [89]: e.g., if an Okta user is in group "Admins", you can map that to a Descope role "Admin". Descope has a section for Group Mapping – you'd specify the SAML attribute that contains group names (Okta by default can include a "Groups" attribute if configured) and link each group to a Descope role [90] [91]. Try adding a mapping: in Descope, add IdP Group "Admins" -> Descope Role "Tenant Admin" (for instance). In Okta, configure the SAML app to send a Groups attribute containing "Admins" if user is in that group. Then test login with a user in that group. After login, check the Descope user – they should have the Tenant Admin role assigned automatically [87] [89]. This shows RBAC integration with SSO, so your app's authorization (from Day 8) can continue to work seamlessly with SSO logins.

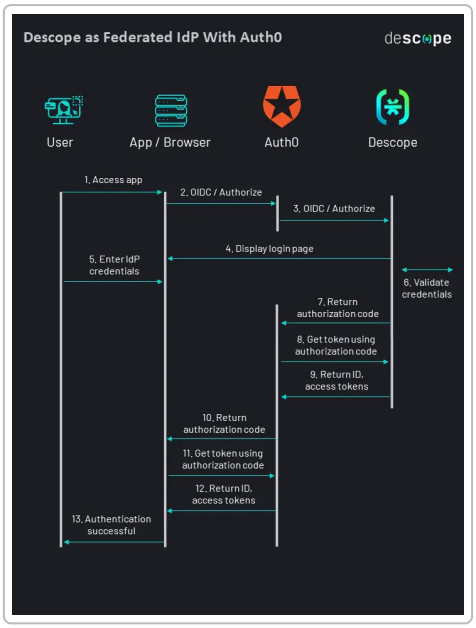

Figure: SAML SSO flow with Okta as IdP and Descope as SP. The user accesses the app (SP) → gets redirected to Okta → after successful auth, Okta returns an authorization code or SAML response → Descope validates it and issues its own tokens to the app. The sequence above is an OIDC diagram; SAML is similar conceptually with different token formats. [77] [92]

By configuring Okta SSO, you've enabled enterprise logins for your app. Notice that once SSO is in place, a user can log in either via SSO or regular method, unless you choose to enforce SSO only (Descope allows enforcing that certain domains must use SSO). With Okta integrated, you might try logging in with a non-SSO email (like a personal Gmail not in allowed domain) – that should still go through regular flow (password/OTP) since domain routing won't trigger. Overall, you learned how Descope abstracts much of the SAML complexity: - You didn't need to manually parse XML, handle signatures, etc., Descope did it. - You used Descope's provided endpoints (Entity ID, ACS) to plug into Okta. - Descope automatically handled creating a user record via JIT provisioning when an Okta user logged in [93]. - The user experience is seamless: one-click from Okta or entering email in your app leads to a login without another password.

This is extremely valuable for B2B apps that often need to integrate with customers' corporate SSO. Next, we will explore a similar concept with OIDC and Auth0.

Scenario 32: Integrating Auth0 as an OIDC Identity Provider (OAuth2 SSO)

Background: OpenID Connect (OIDC) is another way to do SSO, built on OAuth 2.0. Many providers (Auth0, Azure AD, Google) support OIDC. The flow is somewhat simpler than SAML: the SP (Descope) will redirect to IdP (Auth0) authorize endpoint, the user logs in, and an ID token + access token are returned. We will integrate Auth0 as an external IdP via OIDC to Descope. This could be a scenario where perhaps a client is migrating from Auth0 and temporarily needs to let users log in with Auth0 credentials, or just a demonstration of OIDC federation.

Hands-On – Steps:

1. **Auth0 Setup – Create Application and Connection:** Log in to the Auth0 Dashboard. Create a new **Regular Web Application** (since our SP – Descope – is a web service). Name it "Descope OIDC SP" for clarity. In settings:

2. Allowed Callback URLs: put Descope's OIDC callback URL (should be in Descope console, similar to ACS but for OIDC).
3. Allowed Logout URLs: you can put your app logout URL or Descope's logout if any.
4. Application Type: Regular Web, Token Endpoint Auth Method: choose none or basic (depending on if client secret is used by SP; likely yes). Save the app, and note the **Client ID** and **Client Secret**.

Now, in Auth0, go to **Connections -> Enterprise -> OIDC** (Auth0 supports creating enterprise connections that use external IdPs, but in this case Auth0 is the IdP, Descope is the SP – a bit inverted scenario. Alternatively, Auth0 might require using their B2B feature "Auth0 Organizations" to allow an external OIDC, but actually Auth0 can also be IdP to others easily via its issuer endpoints).

Actually, scratch that: we want Auth0 as IdP, so from Descope's perspective, Auth0 is just another OIDC provider. We don't need to configure Auth0 to accept Descope specifically beyond registering an application (which we did). Auth0's domain (e.g., `dev-xxxx.us.auth0.com`) has a well-known OIDC configuration JSON at `/.well-known/openid-configuration`.

1. **Configure Descope for Auth0 OIDC:** In Descope Console, under **Authentication Methods -> SSO**, select **OIDC** (instead of SAML, possibly on another tab or section) [94] [95] . Similar to SAML, pick the tenant that should use Auth0 (maybe create a separate tenant "Auth0Tenant"). Descope will ask for:
2. **Issuer URL**: This is Auth0's issuer, typically `https://<your-tenant>.auth0.com/` (found in the well-known config).
3. **Client ID and Secret**: The ones from the Auth0 app you created [95] .
4. Possibly endpoints (Auth0's authorize and token URLs) – but those can be derived from issuer via well-known.
5. Allowed domains: perhaps set an email domain or simply identify by tenant usage.

Fill these details: - Issuer: `https://dev-xyz.auth0.com/` (for example). - Client ID: (from Auth0). - Client Secret: (from Auth0). - Scopes: ensure `openid profile email` scopes are used (Descope likely does by default). - Redirect URL: Descope should display its OIDC Redirect URL (similar to ACS earlier), which you already added to Auth0's allowed callbacks. Save the configuration.

1. **Test OIDC SSO Flow:** Similar to SAML test, try SP-initiated first. This means your app/Descope triggers Auth0 login:
2. If you have a user with a particular email domain to route to Auth0, set that up. Alternatively, maybe you directly provide a "Login with Auth0" button for testing. Descope might allow an **IdP-initiated** trigger by visiting a certain URL or using a dedicated flow for OIDC.
3. For a quick try, Descope's Hosted Page approach: the Okta one was `tenant settings domain trigger`. For OIDC, maybe Descope provides a link like `https://YOUR_PROJECT.descope.app/oidc/login?tenant=<tenantId>`. If documentation is available, use it. If not, a workaround: Use the **sign-up-or-in** default flow if it lists "Continue with <Auth0>" as an option (some CIAM products detect configured IdPs and show buttons).
4. The easier route: In Descope Console's SSO settings for OIDC, there may be a button "Test Connection" which opens the Auth0 authorization in a new window. Try that if present.

Assuming you manage to trigger it: The browser should redirect to Auth0's universal login page. Auth0 will ask for credentials (use an Auth0 user you have, or create one if needed). After login, Auth0 redirects back to Descope's callback with an authorization code, Descope exchanges it for tokens using the client secret. Descope then logs the user in (issuing its JWT) and redirects to your app or console with success.

If successful, check the Descope Users. A new user likely appears representing the Auth0 user (with an Auth0-specific loginId or their email). If you had attributes like name or picture in Auth0, see if Descope fetched those (if profile scope was included, it should have basic ones like name, email).

1. **Handle Auth0 User Profiles and Roles:** Auth0's ID token might contain user attributes (like `name`, `nickname`, etc.) and can be extended with custom claims. Descope will map what it can automatically (it likely maps email and name by default). If you want to assign a role based on Auth0 data, one approach could be to use OIDC **Mapping Rules** in Descope if available, similar to SAML group mapping. Descope's docs mention you can treat OIDC IdPs similarly. For example, if Auth0's token has `org` claim, map it to a tenant in Descope, or if it has a `role` claim, map to a Descope role. This configuration would be in the same OIDC settings in Descope where you entered issuer (there might be a mapping UI or one would use custom claims + Descope Rules in flows). This might be too deep for now – just note it's possible.

2. **IdP-Initiated OIDC (Auth0 to Descope):** Auth0, being OAuth/OIDC, typically doesn't initiate directly (unlike SAML IdP-initiated which is common via dashboard). However, Auth0 does have a concept of an **IdP-Initiated login** where hitting an Auth0 authorize URL with prompt=none or via a stored link could log in a user who's already Auth0-authenticated. This is less straightforward outside the context of an Auth0 application. So, likely skip explicit IdP-initiated test (in SAML we did via Okta tile; Auth0 has no user portal with app tiles by default). We'll focus on SP-initiated which is the norm for OIDC.

3. **User Experience Integration:** Now that Auth0 SSO works, integrate it in your React app's login choices. Perhaps on your login page, present "Log in with Acme Corp Account" which calls a function:

   ```
   descopeClient.sso.start({ tenant: "<tenantId>" });
   ```

   (Pseudocode: check if Descope's JS SDK has such a method. If not, you might redirect the browser to Descope's SSO URL manually). If using the `<Descope>` component, maybe configure a flow that includes an "OIDC" screen. Descope docs might show that if multiple IdPs are set, the default flow page will list them. For example, the first screen could present "Login with Password" vs "Login with SSO" options. If not automatic, you can create a custom flow where the first screen is just choices (two buttons) and clicking the Auth0 option triggers an OIDC action (some flows allow an action node "Start SSO"). That would be a neat use of the flow builder – an action that begins SSO login (Descope likely has a way: in flows an Action called "Start SSO" with a parameter of IdP/tenant). For brevity, ensure your users know how to use the SSO login.

4. **Troubleshooting:** If login fails, examine the error. Common OIDC config errors:

5. Mismatch redirect URI (make sure it matches exactly in Auth0 and Descope).
6. Wrong issuer or credentials (check that the Auth0 domain is correct and client secret is correct).
7. If you get an error after redirect, check Descope logs or Audit events. Descope might log an event for failed SSO with details. Possibly found under Audit trail in Descope console (Day 15 will cover Audit, but you can peek now for any SSO errors).
8. You may also inspect network calls: after Auth0 redirect back, you might see a call from browser to `.../oidc/callback?code=...`. If it returns 400, the response might say what's wrong.

After these steps, you've successfully integrated an OIDC IdP. Essentially, your app can now accept users from an Auth0 domain as easily as local Descope users. This approach is similar for other providers: - For Azure AD or Google Workspace, you'd configure OIDC in Descope with those issuers. - The general pattern: register SP in IdP, input IdP info in Descope, and test.

One interesting use-case for Auth0 specifically is **migration** – which is upcoming. You might allow users to log in via Auth0 OIDC during a transition period while behind the scenes you import them to Descope. That's an advanced scenario, but you've laid the groundwork by understanding how to hook Auth0 and Descope together.

Scenario 33: Consolidating SSO Knowledge – Ensuring a Smooth User Experience

Background: Now that both SAML (Okta) and OIDC (Auth0) SSO are set up, we should consider the user experience and system maintenance for these integrations. This scenario is more about polishing and verifying that everything works in concert, and preparing for scenarios like key rotations and error handling: - We'll simulate a metadata update (like Okta certificate rotation). - Ensure that a user can seamlessly log in via either SSO or local method as appropriate. - Document the steps for adding a new SSO tenant in the future (since in B2B, each enterprise customer might have their own IdP).

Hands-On – Steps:

1. **Certificate Rotation (SAML Metadata Refresh):** SAML IdPs periodically rotate signing certificates for security. Descope likely caches Okta's metadata. When Okta's cert changes, the SSO might break unless updated metadata is fetched. To simulate, you could upload a new certificate in Okta (not easy in dev accounts) or simply assume it changes yearly. How to handle? Check if Descope offers an **Metadata URL** for IdP (we gave it, so hopefully it periodically fetches). Possibly click "Refresh Metadata" in Descope SSO settings if available. It's good to know: because we used Okta's Metadata URL in Descope config, Descope can always retrieve the latest certificate [96] . Had we manually uploaded a cert, we'd need to update it. So using metadata URL is best practice (which we did).

2. **Test Mixed Flows:** Try logging in as different types of users:

3. A user whose email domain matches the Okta tenant's domain (e.g. alice@acme.com) → should go to Okta SAML.
4. A user for Auth0 (maybe bob@contoso.com) → should trigger Auth0 OIDC.

5. A normal user not in any SSO domain (charlie@gmail.com) → should use password/OTP as before. If you have the flows and domain routing properly, each of these should work and end up with a Descope session. This tests that having multiple SSO configurations doesn't conflict. Descope supports multiple SSO per tenant too, but that's more advanced (like if one tenant had multiple IdPs – Descope has something called "Multiple SSO per tenant" [97] but skip that complexity for now).

6. **User Provisioning and De-provisioning:** When SSO users log in the first time, Descope auto-creates them (JIT). What about when they leave the company? Ideally, if Okta or Auth0 disables a user, they can't login via SSO (that's fine). But their account might still exist in Descope. You might want to periodically clean up or use SCIM (System for Cross-domain Identity Management) for automatic user provisioning/de-provisioning. Descope actually supports SCIM for user sync with IdPs [98] [99] . Okta can be set to push users via SCIM to Descope, ensuring Descope has up-to-

date user info and disabling accounts. That setup is beyond our scope here, but worth noting as a next step for enterprise-readiness.

7. **Error UX:** Consider when SSO fails. E.g., if a user not assigned the Okta app tries, Okta will error. Or if the user exists in Descope from previous local signup but then tries SSO with same email – Descope might link them or error (often, Descope would link by email if verified, or could throw "user already exists"). You might want to handle this gracefully. In Descope flows, there might be an option to handle "IdP user already linked" scenario. Check Descope's documentation for SSO error handling. Possibly, enabling "Allow JIT provisioning" and not duplicating emails is something you set. Ensure that setting is on (so that if the email matches an existing user, Descope either merges or rejects; likely merges by default). To test, take a user you created locally on Descope with some password, then attempt SSO login with Okta using the same email. Ideally, Descope will recognize and not create a duplicate user but just mark the existing user as having logged in via SSO. The audit logs would indicate if a new user was created or existing used.

8. **Document Onboarding a New SSO Tenant:** Imagine your next customer uses Azure AD. What's the process? It will be similar to Okta:

9. Create a tenant in Descope for them.
10. Configure Azure AD as SAML or OIDC IdP in Descope (depending on what they prefer).
11. Provide them the SP info (EntityID, ACS) to setup in their IdP.
12. Test, map groups to roles, etc. Documenting this as a checklist is useful for your team. This ensures smooth scaling of SSO to more clients. You now have a repeatable procedure from the Okta/Auth0 experience.

At the end of Day 11, you've integrated with one of the most common enterprise SSO (Okta) and also an example of a CIAM provider (Auth0) using OIDC. Your application can federate identity from multiple sources. This not only provides convenience to users (they can use existing corporate logins) but is often a requirement in B2B deals. We also briefly touched on advanced topics like SCIM user sync and group-role mapping which ensure that authorization (RBAC) ties into SSO seamlessly (if an employee is in the "Managers" group in IdP, they get Manager role in app automatically). All these features elevate your app to an enterprise-ready solution. Next, we'll delve into migrating from legacy auth systems to Descope to consolidate everything under one roof.

Day 12: Migration Strategies from Auth0/Firebase to Descope

Scenario 34: Planning a Migration – Full vs Hybrid Approaches

Background: Migrating user accounts from an existing auth system (like Auth0 or Firebase) to Descope is a delicate process. The goal is to move users without causing login failures or forcing mass password resets. There are two primary migration strategies: - **Full Migration:** Bulk import all users into Descope and cut over entirely [100] . - **Hybrid Migration:** Keep the old system running in parallel while new logins happen through Descope, until a complete switch-over is feasible [100] . We need to plan considering data to migrate (users, passwords, roles, MFA factors, etc.) and how to minimize disruption.

Hands-On – Steps:

1. **Inventory Current System:** Let's say we are migrating from Auth0. First, list what data we have in Auth0:
2. User profiles (name, email, etc.).

3. Password hashes or authentication methods (some users password, some social, etc.).
4. User metadata and app metadata.
5. Roles/permissions or groups (Auth0 might have roles and possibly Organizations).
6. MFA enrollments (OTP, recovery codes). If migrating from Firebase, data includes user emails, passwords (hashed with a known algorithm like Firebase Scrypt), maybe phone numbers, and custom claims.

Determine which of these need to move to Descope. At minimum, user identifiers and credentials should migrate so users can log in on Descope with the same email/phone and password.

1. **Export Users Securely:** Auth0 provides user export options: via Management API or the dashboard (export to JSON) [101] [102]. For >1000 users, Auth0 recommends using their export job to JSON file [102]. Export typically includes fields like email, name, last login, and password hash (if requested via support for hashes, since Auth0 doesn't give hashes by default without a special request) [103]. Firebase allows exporting users with their password hash and salt parameters through the Firebase Admin SDK or Google Cloud CLI.

Use Auth0's export to get a JSON of users (including their hashes if possible). If password hashes cannot be obtained (Auth0 requires support ticket to get them, especially if using their proprietary hashing), plan an alternative like **Password Reset on first login** or a hybrid approach where Auth0 continues to validate passwords for a while (Auth0 has an option to do "lazy migration" by using Auth0 as IdP as we did with OIDC, which is a kind of hybrid strategy).

1. **Hash Algorithms and Compatibility:** Identify the hashing algorithm used for passwords. Auth0 by default uses bcrypt for database connections (they can provide hash details in export with salt), or if using their older SHA-1 (rare) or Argon2 based on config. Firebase uses the **Firebase Scrypt** algorithm (which is a modified Scrypt with known parameters you can export) [104]. Descope supports importing hashed passwords with various algorithms: as per docs, Descope supports **Bcrypt, Argon2, Django (PBKDF2), Firebase Scrypt, PBKDF2, PHPass, MD5** [105]. This is great because it means we likely can import without users resetting passwords, as long as we provide the correct hash info.

Check in Descope **Migration** docs or APIs how to import users with hashes. They mention support for those protocols and to use the Management API for user creation including hash data [106] [107]. For example, if using Bcrypt, we'd supply the hash and set "passwordHashingAlgorithm": "Bcrypt" and it will accept it.

1. **Choose Full vs Hybrid:**
2. *Full Migration:* If downtime or forcing re-login is acceptable (or if user count is small), do it all at once. Steps: freeze Auth0 signups, export all users, import to Descope, update the app to use Descope for auth. Possibly send an email to users about the change (especially if any need to reset passwords).
3. *Hybrid Migration:* If seamless is required, you can perform a silent migration over time. For example, keep Auth0 active and integrate Descope such that:
    - New users are created in Descope (and perhaps also backfilled to Auth0 if needed, or not).
    - Existing users: when they login, if their account isn't in Descope yet, you authenticate against Auth0, then create them in Descope on the fly and then issue Descope tokens going forward. Descope provides a "Migration tool" that can do this by checking credentials against old hashes or via an API call to Auth0 during login flows [108].
    - Over a few weeks, as users log in, everyone gets moved to Descope. Eventually Auth0 is no longer used.

    - This can be achieved by customizing the login flow: for instance, Descope flows could call out to Auth0 if a login attempt for a not-yet-migrated user fails in Descope. Descope documentation mentions a `freshlyMigrated` flag approach for flows [109] and support for this kind of conditional logic.

Since doing a full migration in one shot is simpler for our lab, we will plan that way for demonstration, but be aware hybrid might be needed for large userbases or where password hashes can't be extracted easily.

1. **Set Up Descope Project for Migration:** Before importing, configure Descope to accommodate imported data:
2. If you have custom user attributes (e.g., "favoriteColor", "subscriptionTier"), define them in Descope's **User Schema** or Custom Attributes section so you can store that data [110] [111].
3. If you use Roles, create equivalent roles in Descope so you can assign them during import (e.g., if Auth0 had an "admin" role, ensure a Descope role "admin" exists, or decide new role mapping).
4. Prepare an Access Key (Management Key) for Descope's Management API [112] – needed to authorize import scripts.
5. **Script the Import (Auth0 Example):** Descope provides a **descope-migration** GitHub tool [113]. Clone that repository, which contains scripts to import from various sources (Auth0, Firebase, etc.). For Auth0:
6. You'll need Auth0 domain, Auth0 API token (to fetch users if <1000 or to get password hashes if not via JSON), and the Auth0 Tenant ID [114].
7. Also your Descope Project ID and a Management Key from Descope [115].
8. The tool likely can take either direct API or JSON file. If we have the JSON from earlier, we can use `--from-json ./auth0-users.json`.
9. The README for Auth0 migration shows usage examples for dry-run vs live [116].
10. Do a **Dry Run** first [117]: This will simulate and not actually create users, but output what would happen, including any errors (like unsupported hash format or missing fields).
11. Check the dry run output. If it says e.g. "X users to import, Y without password (social users)", etc. Decide how to handle those (social login users might be migrated as is but marked such that they need to login via OIDC perhaps).
12. When ready, run the live import with appropriate flags [118]. The script will call Descope's Management APIs to create each user with their attributes, set their password hash (using Descope's user import API which likely has fields for hash and hash parameters).
13. **Verify in Descope:** After import, spot-check a few users in Descope Console's user list. Are emails correct? Custom attributes present? Roles assigned? For passwords, you can test by logging in as one of the migrated users using their old password – it should work (Descope will internally hash the entered password with the imported salt/algorithm and compare, if matches then next time it might re-hash to its current default if needed, or just keep as is depending on config). If something failed (like all passwords not working), then possibly the hash format was mis-specified. For instance, Auth0 uses bcrypt by default which Descope supports, but maybe Auth0's JSON didn't include salt or we needed to provide the salt field name. Adjust and retry.
14. **Post-Migration Cleanups:** Some considerations:

15. **Email Verification:** Imported accounts might be marked verified or not depending on source data. Descope allows setting verified email/phone flags on user creation. If possible, ensure those flags reflect reality (Auth0 and Firebase track email verification status). The migration tool likely sets `verifiedEmail=true` if the source had it. If not, you could send a bulk verification mail or force verify to avoid re-sending if not necessary.
16. **Resetting MFA:** If users had MFA enrolled in Auth0 or Firebase (like TOTP or SMS 2FA), migrating those is complex (you'd need their shared secrets, which are typically not exportable). So likely, users will need to set up MFA anew in Descope. Communicate that to those users. Alternatively, during migration mark them as MFA not enrolled, and optionally send an instruction to enroll next login.
17. **Cutting Over**: Now update your application configuration to use Descope for all auth. That means updating environment variables (like use Descope project ID and keys, instead of Auth0 domain/client), and deploying. This should ideally happen during a maintenance window if doing full migration.

At this planning stage, we haven't actually executed migration code in our lab (since we don't have a real Auth0 instance with users here), but we have a plan and the tools lined up. We have considered how to keep passwords, which is the trickiest part, by leveraging Descope's support for multiple hash algorithms [105]. We also weighed doing hybrid via bridging logins – indeed Descope's OIDC integration we did earlier could serve as a hybrid approach: we could simply keep Auth0 as an IdP via OIDC until all users have logged in at least once into Descope, then drop it. That's another valid strategy if direct hash import isn't available.

Scenario 35: Migrating from Auth0 – Execution and Testing

Background: Building on the plan, let's simulate migrating from Auth0 specifically using Descope's migration tool. We will focus on a subset of tasks to illustrate the process, assuming we have a small set of users. This includes running the migration script (in a safe mode), interpreting results, and verifying outcomes with test logins.

Hands-On – Steps:

1. **Prepare Environment for Script:** Ensure you have:
2. Python 3 environment (the descope-migration tool is in Python as per GitHub).
3. Install required packages: likely `pip install -r requirements.txt` from the repo [119].
4. Setup environment variables or a `.env` file with needed credentials as described in the tool docs:
    - `AUTH0_TOKEN` (Auth0 API token, if using API approach) [120].
    - `AUTH0_TENANT_ID` [120].
    - `DESCOPE_PROJECT_ID` [121].
    - `DESCOPE_MANAGEMENT_KEY` [122]. Since we might be using JSON method, Auth0 token may not be needed except if migrating passwords (if we have a separate password file from Auth0).

For our dry-run, maybe we just use the JSON export. Place `users.json` in the directory.

1. **Run Dry-Run Migration:** Execute:

```
python src/main.py auth0 --dry-run --from-json ./users.json --with-
passwords ./passwords.json
```

(This is according to the docs example [123]. Auth0's password hashes might come in a separate file if provided by Auth0 support – so the tool expects `--with-passwords file.json`.) The dry-run will not create users but will output logs. Look for lines like "Would create user X (email) with hash...". If any errors, they'll likely show:

2. Unknown hash algorithm -> then we need to ensure the tool recognized the hash. The tool likely auto-detects if passwordHash field present in JSON.
3. If some users missing password (e.g., social logins), the tool might skip or mark them. It may create them with a random password or mark requiring reset (depending on how it's designed). Check documentation if social users are handled (maybe it sets a flag or we need to supply an option to skip them).

4. If any attributes fail (like too long, or mismatched type), we'd adjust data or Descope schema accordingly.

5. **Perform Live Migration:** If dry-run looks good, run without `--dry-run`:

   ```
   python src/main.py auth0 --from-json ./users.json --with-passwords ./
   passwords.json
   ```

   This will call Descope management APIs to create users. The script might output progress or final counts. Upon completion, it might say "Created N users successfully" or list any failures.

Note: This step might be time-consuming for thousands of users; the script possibly handles pagination or batch creation behind the scenes [107]. Keep an eye on rate limits – Descope's API might have a rate limit (maybe 10 create calls per second or similar). The tool likely spaces requests or uses batch endpoints if available.

1. **Validate a Sample of Migrated Users:** Choose one user from the migrated list:
2. Try logging in to your app (which now uses Descope) with that user's email and original password. It should log in successfully. If it fails with "invalid credentials", something's wrong with the hash import. In that case, check Descope console for that user – perhaps the password wasn't set (maybe because missing in input). If the password didn't carry over, you might choose to have Descope trigger a password reset email for those users on first login attempt. But ideally, it works.
3. Also check if the user's profile info is intact: correct display name, any custom attributes, and any roles. If roles were to be imported, ensure they appear. If not, maybe the tool required a mapping config (Auth0 roles are not in user profile by default; one might export them separately).

4. If MFA was present for that user in Auth0, try logging in: Descope wouldn't know about it, so it will just let them in with just password. That is expected and might be a slight security regression until they set up MFA in Descope. You may encourage users post-migration to set up MFA again.

5. **Edge Cases – Social/SSO Users:** If a user originally logged in with Google (no password in Auth0), the migration might create them without a password and perhaps with `loginIds` only. How will they login now? Possibly they must use SSO (so we should configure Google social login in Descope too if that was used, or send them a magic link to set a password). Descope migration guide mentions linking social login by loginId [124]. If the user had a Google identity, their email is in Descope now but no password. Ideally, we set up Descope Social login with Google so they can continue using Google to login. If not, an alternative is to send those users a

"set your password" email. The Descope management API could trigger an invite for them. Identify these users (the tool might output how many had no password), then use Descope's **Invite User** function to send them a signup link to set a password [125].

6. **Communicate Migration to Users:** In a real scenario, one would send an email to all users informing them of the auth system change. Provide any necessary instructions (especially for those who need to reset or re-verify). Perhaps we'd include a support contact if login issues occur. Also, update any user-facing privacy policy or terms if needed since their credentials are now stored in a different platform (some companies do that for transparency).

At this point, we assume our Auth0 migration is done and the app is now solely on Descope. Monitor the login success rates and errors closely after cut-over: - Use Descope's audit logs (Day 15 will cover) to see if lots of "LoginFailed" events occur (could indicate some passwords didn't work). - If so, respond quickly perhaps by triggering password resets for those failing.

The migration is successful if users can log in with minimal to no intervention, and all required data has been transferred. You have effectively **sunset** the Auth0 dependency. One can now shut off Auth0 to avoid paying for two systems, after a comfortable buffer period.

Scenario 36: Migrating from Firebase Authentication – Key Differences and Tools

Background: Migrating from Firebase (Google's identity service) to Descope has similarities to Auth0 but with some differences: - Firebase provides a straightforward export of users (often via Firebase Admin SDK or Google Cloud Identity Toolkit). - Firebase passwords use a custom **scrypt** algorithm (with keys and salt) which Descope supports [104]. - Firebase might have anonymous users (which either can be ignored or handled specially). - There's no concept of roles in Firebase Auth; roles are usually implemented in Firestore/Custom claims, which you'd migrate to Descope roles or claims if needed.

Descope's migration tool also supports Firebase specifically, which likely automates some of these steps. Let's outline the Firebase migration procedure and note where it diverges from Auth0.

Hands-On – Steps:

1. **Export Users from Firebase:** Using the Firebase CLI or Admin SDK:

   ```
   firebase auth:export users.csv --format=csv --project your-project-id
   ```

   This generates a CSV (or JSON if specified) containing users' email, passwordHash, passwordSalt, and other fields like phone, emailVerified, etc. Additionally, one must export the **password hash parameters** (Firebase's scrypt has parameters: salt separator, rounds, memory cost, etc.) [104]. In Firebase Console under Authentication > Users, there's an option "Password Hash Parameters" [126] – download that JSON, which contains keys like `signerKey`, `saltSeparator`, `rounds`, `memCost`. These values are needed for any system to validate the hashes. Descope specifically asks for them.

The Descope migration GitHub (and docs snippet we saw) addresses this: they instruct to copy these parameters to a file and supply it [127]. We have `password-hash.txt` or a JSON file for it.

1. **Use Descope Migration Tool for Firebase:** The tool has a `firebase` command:

```
python src/main.py firebase --from-file ./users.json --hash-params ./
password-hash.txt
```

With appropriate environment variables:

2. `DESCOPE_PROJECT_ID` and `DESCOPE_MANAGEMENT_KEY` (Auth0 token not needed here). Possibly the tool can connect directly to Firebase with credentials, but using the exported file is simpler.

The tool will parse the Firebase file. It likely handles: - `passwordHash` and applies the parameters to import properly. - If a user has `emailVerified=true`, mark Descope user's email verified. - Phone numbers: if present and verified, might create a phone loginId in Descope too and mark it verified. - Disabled accounts: Firebase can disable users, we might want to reflect that (Descope can mark a user as disabled/inactive via an attribute or just not import them until needed). The tool might skip disabled or you can choose to import them but perhaps not send login invites.

Do a **dry-run** first as always, check for issues like CSV parse errors or unsupported fields.

1. **Address Anonymous Users:** Firebase often has "anonymous" auth users for guest sessions (no email/password). These likely have no email (just a UID). You might skip migrating them, or if needed, create a placeholder in Descope. But since they have no real identifier, probably ignore unless they have associated data you need. The tool might mention "X anonymous users skipped".

2. **Import into Descope:** Run the actual import. After completion:

3. Check a known user's password: The complexity here is Firebase Scrypt is not standard, but Descope supports it if parameters are correct. We test by logging in as that user with their old password – it should work.
4. If it doesn't, verify the parameters supplied. Possibly an error in salt separator or something. Firebase's param JSON must be correctly provided to Descope's API. Descope likely had an API endpoint where you specify algorithm "firebase" and the parameters. The migration tool should handle that (they likely derived the combined `passwordHash` in a way Descope expects).

5. If still trouble, one approach could be enabling **Allow legacy Firebase token login** concurrently for hybrid: but since Descope explicitly supports Firebase's hash, it should work.

6. **Multi-Tenant Consideration:** If using Firebase for multiple projects or an app with tenant concept, you might create corresponding tenants in Descope and split users accordingly during import (like run separate imports filtering users by some domain or custom claim). But Firebase doesn't have tenants out of the box (except separate projects). So likely not an issue unless you manually segmented users.

7. **Custom Claims & Roles:** Firebase allowed custom claims on user tokens (set via Admin SDK) to store roles or flags. That data isn't in the user export directly (unless you stored in a separate DB or they might appear as `customClaims` JSON in the export). If you have that, you'd map it:

8. For example, if Firebase customClaims had `{ admin: true }`, you can create an "Admin" role in Descope and assign it to that user on import. The migration tool might not automatically do that, so you might need a custom step: e.g., after import, run a small script through Descope Management API to assign roles based on an external mapping file of user->role.

9. Alternatively, include those as Descope custom attributes for reference.

10. **Verify All Users:** The number of users in Firebase is often large for B2C apps. Do a count check: if Firebase had 10,000 users, ensure Descope now shows about 10,000. If some are missing, identify why (maybe duplicates? Firebase might have multiple providers for same email – ensure they didn't get dropped).

11. If duplicates by email existed (e.g., one user had email+password and one anonymous upgraded to same email, but in Firebase they would merge under one account typically), it might complicate. But overall, ensure key accounts are present.

12. **Post-Migration in App:** Update your app's authentication to Descope. For Firebase Auth, this means removing Firebase SDK usage for authentication. Instead, integrate Descope's SDK (which we already did on Day 1-7 presumably). Test various flows:

13. New sign-up via Descope (should create user in Descope as expected).
14. Existing migrated user login (should work).
15. Social logins: If you used Google sign-in in Firebase, you'll want to configure Google OAuth in Descope and maybe the migration tool set up some user fields to link them. Possibly easier: if user used Google, then their email is in Descope, they can use "login via Google" now because Descope will match by email and log them in (Descope will probably create a new user if not found, but if found, might link it – need to ensure either behavior or avoid duplicate).
16. Anonymous users: in Firebase app, these might have been allowed to continue as guest. With Descope, if you want similar, you could use Descope's anonymous mechanisms (if any), or decide to drop anonymous login altogether. If needed, Descope does allow unauthenticated flows but typically one uses an identify step.

Migrating Firebase should now be complete. You've now gone through migrations for two common systems. The key differences we saw: - Dealing with different password hashing (Firebase's custom vs Auth0's bcrypt). - Absence of built-in roles in Firebase, meaning less to map but possibly custom logic if any. - Possibly larger user counts, meaning performance considerations and making sure to script it (which we did with the provided tool).

At this juncture, you have effectively unified all users into Descope. Day 12 covered the heavy lifting of planning and executing migrations, which is often the hardest part of adopting a new auth system for an existing app. In the final stretch (Days 13-15), we will focus on advanced application architecture topics (multi-tenancy, analytics, rate limiting, etc.) and polish everything with best practices and demo preparation.

Day 13: Multi-Tenancy and B2B SaaS Architecture with Descope

Scenario 37: Multi-Tenant Concepts and Tenant Management in Descope

Background: In a B2B SaaS app, **multi-tenancy** means each customer organization (tenant) has its own isolated data and often its own set of users. Descope supports multi-tenant architecture natively: you can create multiple tenants under one project, assign users to one or more tenants, and even have tenant-specific settings like SSO configurations [128] [129]. This scenario covers the core concepts and how to use Descope's **Tenant Management** features: - Creating and updating tenants. - Assigning users to tenants. - Tenant-level roles and admin delegation.

Hands-On – Steps:

1. **Create Tenants in Descope:** Suppose our app is a project management app for companies. We have Acme Corp and Beta Inc as clients. In the Descope Console, go to **Tenants** [128]. Click **+ New Tenant** and create one for "Acme Corp" and another for "Beta Inc". Each will get a unique Tenant ID (TID) generated by Descope [30]. You can also add metadata for each tenant (like address or plan level) via **Custom Tenant Attributes** [130].
2. For Acme, maybe add a custom attribute "Plan = Enterprise" using the Custom Attributes tab [130].
3. For Beta, "Plan = Starter".
4. **Assign Users to Tenants:** Initially, no users belong to these tenants. We need to assign. In Descope Console's **Users** table, find a user (or create a new one) who should belong to Acme. Edit that user, and there should be a field to assign them to tenants and roles within those tenants [10] [22]. For example:
5. User alice@acme.com -> assign to tenant "Acme Corp" with role "Tenant Admin".
6. User bob@acme.com -> assign to "Acme Corp" with role "User".
7. User eve@beta.com -> assign to "Beta Inc" with role "Tenant Admin".
8. (Ensure you created roles appropriately from earlier RBAC scenario, e.g., a "Tenant Admin" role and maybe a default "User" role.) You can do this in the console UI or via the Management API (Descope Management SDK has functions like `client.tenant.addUser(tenantId, userId, roles)` if you prefer scripting for bulk).
9. **Tenant Admin Delegation:** Descope allows you to delegate management of a tenant to certain users via the Tenant Admin role. Out-of-the-box, the Tenant Admin role grants permission to manage users of that tenant (and SSO settings, etc.) [129]. We gave Alice and Eve that role for their companies. Now, if we want them to self-serve user management without needing our involvement:
10. Descope provides **Admin Widgets** for user management that can be embedded in an app for tenant admins [131]. That's beyond our current scope to implement, but conceptually: we could embed a "User Management" widget in the Acme admin dashboard page, which uses Descope's SDK to allow Alice to invite or remove users in her tenant only.
11. Alternatively, we could build custom UI and use Descope Management API with Alice's Tenant Admin privileges (via her JWT's role claims) to perform actions. For instance, our backend could allow Alice to call an endpoint to invite a user, and it checks her JWT for Tenant Admin in tenant X before calling Descope's invite API for that tenant.
12. **Use Tenant in Authentication Flow:** Let's see how tenants play into login. If a user like alice@acme.com logs in, the JWT we get will contain a `tenants` object as seen before [29]. Alice's token will have something like:

```
"tenants": {
  "TID_ACME": { "roles": ["Tenant Admin"], "permissions": ["User
Admin", ...] }
```

```
},
"dct": "TID_ACME"
```

Notice a claim `dct` (possibly stands for "Default Current Tenant") [132]. Descope sets one tenant as current if the user belongs to multiple (maybe the first or a configured one). Since Alice is only in one, dct = Acme's ID. This is important: the `dct` claim can be used by your app to know which tenant context the user is operating in by default [133]. If a user is in multiple tenants, Descope's SDK might allow switching current tenant (there's often a function like `switchTenant(tenantId)` to change the dct claim via a new JWT or session parameter).

13. **Enforce Tenant Isolation in App:** In our Node backend, whenever we query or manipulate data, we should scope by tenant. How do we get the tenant context? We could:

14. Trust the `dct` from JWT as the current tenant. This is convenient: each API call, after validating JWT, we know `req.user.currentTenant` (if we parse the token claims for dct).
15. Alternatively, require the frontend to send a tenant ID in headers or URL if a user has multiple (like an admin portal might allow switching). But since dct is there, use it as default. For example, our GET /projects API would filter projects by tenant = dct. Similarly, when creating a new project, associate it with that tenant. If a user could belong to multiple and switch, then we'd need to either issue a new token when they switch or have them explicitly specify which tenant for an action (to avoid confusion).

Descope's helper `getCurrentTenant(sessionToken)` returns the current tenant ID from JWT [14]. In React, `useSession()` with getCurrentTenant could be used to display "You are in tenant: Acme Corp".

1. **Tenant-Specific Settings – SSO:** We already saw that SSO configuration in Descope is often per-tenant [83]. For instance, we might set up Okta SSO for Acme's tenant and maybe another SAML IdP for Beta's tenant. Descope's model supports multiple SSO configs, each tied to a tenant, allowing enterprise customers to each integrate their IdP [134] [135]. We won't repeat the config steps, but just note:
2. The `tenants` assignment for a user in Descope also ties to which SSO logins they can use. If an email domain matches an SSO config for a tenant, Descope will log them into that tenant.

3. If one user belongs to multiple tenants, possibly with different SSO, it can get tricky – but typically a user's email would be associated with one org.

4. **Tenant Custom Domains (Optional):** Some B2B apps use separate subdomains for each tenant (e.g., acme.app.com, beta.app.com). If that's a requirement, Descope can support custom domains for auth flows [41]. You could set up `auth.acme.com` that routes to Descope flows with style specific to Acme. Descope's styling per application feature allows flow branding per tenant (we'll explore branding soon) [136].

5. **Testing Multi-Tenant Behavior:** Perform the following tests:

6. Alice (Acme admin) logs in. Check her JWT for correct tenant info. Using Descope's React hooks, call `getJwtRoles(token, acmeTenantId)` and ensure "Tenant Admin" is present [17]. Also call `getCurrentTenant(token)` and expect Acme's ID [133].
7. Bob (Acme regular user) logs in and should not have admin permissions. Ensure he cannot call any admin API (we already have RBAC enforcement in place to check roles).
8. Eve (Beta admin) logs in, see that her currentTenant is Beta Inc.

9. If possible, simulate a user with two tenants: perhaps manually edit and add alice to Beta Inc as well with a role. Now Alice might have two sets in her JWT. In such a case, Descope might either pick one as current (likely the one you most recently assigned? Or first created? If needed, the app can switch by using the SDK's tenant switch method). For now, just acknowledge that scenario. If implementing UI, you'd give Alice a dropdown to choose "Act as Acme or Beta" and then call an endpoint to Descope to switch (or simpler, log out and log in with a different email – but if one email has two, maybe not typical except MSPs who manage multiple clients).

With multi-tenancy configured, our app is aligned to serve multiple customers in isolation. We leverage Descope's features to manage tenants and not build our own tenant store from scratch, which saves time. Descope's JWT structure carrying tenant info is very handy to implement authorization and data partitioning. The main caution is to always use the tenant context from JWT (never trust the client to tell you which tenant, as that could be spoofed; the JWT is the source of truth).

Scenario 38: Tenant-Specific Customizations – Branding and Domain Routing

Background: Different tenants may require different customizations in your application – the most common being **branding (logos, colors)** and sometimes slight differences in features. Descope assists with some of these by allowing **Styles per Application** (you can apply different themes for flows based on context) [136] and **Localization** for language differences. Also, we touched on having custom domains per tenant for auth. In this scenario, we'll focus on branding: customizing the look and feel of Descope's flow for each tenant, and how to present tenant-specific branding in the app.

Hands-On – Steps:

1. **Set Up Multiple Style Files in Descope:** In Descope Console, go to **Styles** (under Management) [137]. Create a new style file for Acme Corp (e.g., name it "AcmeStyle"). Upload Acme's logo (their company logo) for both light and dark themes [138], choose their brand colors for primary and secondary [139] (say Acme's primary is blue). Set font if they have a corporate font (or just pick a different one for demonstration) [140]. Save this style. Create another style file "BetaStyle" for Beta Inc. Use a different logo and color (maybe Beta is green).

Now we have two style IDs.

1. **Apply Styles by Tenant in Flows:** We want that when an Acme user goes to login (or any flow) the Descope screens reflect Acme's branding. There are a couple ways:
2. If using Descope Hosted pages (auth.descope.com?flow=…&tenant=…&style=…), you can append `style=<style-id>` to the URL [141] for that tenant's login link. Possibly we can integrate that in how we generate SSO links or invite links for that tenant.
3. If using the React SDK `<Descope>` component, we can pass a `styleId` prop in code depending on tenant [136]. But at login time, if user hasn't chosen tenant yet (like domain routing triggers), the style might be chosen after they enter email.

One approach: We know the tenant by domain (e.g., acme.com emails). We could maintain a mapping in our front-end: if user tries to login with email @acme.com, apply AcmeStyle. But it's easier to have separate subdomains or entry points: like Acme's employees might have a custom login page that sets the style.

For now, demonstrate manually: - Create a custom login component that wraps `<Descope flowId="sign-in" styleId="AcmeStyleId" />` and use that for Acme's login page. Perhaps route `/login/acme` uses that component. Similarly `/login/beta` uses Beta style. - In practice, you might

not have separate routes for all tenants unless you have subdomains; you might detect from the URL (like auth.acme.app.com) which style to use.

If unable to fully integrate, at least preview it via the Descope Console's style preview or using the style in a test flow run.

1. **Customize Email/SMS Templates per Tenant:** Branding extends to communications. Descope's **Messaging Templates** allow customizing the content of emails/SMS for authentication (like magic link emails, OTP emails) [142] . Perhaps Acme wants a custom email template with their branding (logo in header, custom verbiage). In Descope:
2. Go to Project Settings -> Email provider (assuming set up) -> "Templates" [142] .
3. Create a new Email template for "Magic Link Login" for tenant Acme. Put Acme's logo HTML and a friendly message "Welcome to Acme's Project Portal. Click below to login." Save it as, say, "MagicLink_Acme".
4. How to use tenant-specific template? Descope might allow specifying template in the flow or API call via `templateOptions` or linking template to tenant. Actually, Descope docs mention using `templateOptions` to dynamically populate content or choose template [143] . Also possibly you can set a template as default for a particular tenant by duplicating the auth method under tenant context.
5. For demonstration, assume we can set in the SSO config or so (if not, you may need to manually specify in code when sending an OTP for that tenant using Management API specifying the templateId).

6. Similarly, adjust SMS template if needed (like include company name in OTP SMS).

7. **Tenant Domain Routing Recap:** We configured SSO based on email domain earlier (Okta for acme.com, etc.). If you have a completely separate flow or requirement per tenant, you could also do routing in your app. But since Descope can enforce correct SSO by domain, that part is fine. Just ensure if Beta doesn't use SSO, that domain is not claimed and those users go through standard login flows.

8. **Testing Branding Differences:** Emulate being an Acme user:

9. Navigate to the login page and ensure the branding is Acme's (if we set up style switching).
10. Trigger a password reset or magic link and verify the email content (it should use the template with Acme's logo).

11. Then emulate Beta user, see Beta branding and content. If not fully automated, conceptually verify through Descope's template preview (Descope console often can send a test email for a template to see how it looks).

12. **Feature Flags per Tenant:** Sometimes beyond branding, you may enable/disable features per tenant (e.g., Beta is on Starter plan, no SSO available). Descope's tenant custom attributes can help here. For example, the custom attribute "Plan = Starter" could be read by your app to hide SSO options or advanced features. Alternatively, you could base it on roles or a separate system. But since we stored Plan in tenant custom attributes, we can fetch that via Management API or possibly embedded in JWT via custom claims if configured.

How to embed? Descope's **JWT Templates** allow adding custom attributes to JWT (the docs mention you can use tenant custom attributes in custom claims in the JWT) [144] . If you set that up, then the JWT for Beta users might include `"plan": "Starter"` under their tenant entry or a top-level claim. Not sure

if the migration tool or config auto did that, but you can manually set a JWT template in Descope so that `tenants.TID.customAttributes.plan` is included. Then your front-end can read plan from JWT and adapt UI. If we don't do that now, just note that it's possible.

With these customizations, each tenant's users get a somewhat personalized experience which is critical in B2B SaaS – it makes them feel it's "their company's portal" rather than a generic service. Descope enabling style changes and template branding means less custom code on our side to theme things dynamically. The main work is ensuring we select the right style/template at runtime, often by tying it to tenant context which we have in the JWT or URL.

Scenario 39: Building a Reusable Tenant Management Admin Component

Background: As we close out multi-tenancy, one valuable piece to create is a **Tenant Management** component for our own admin use (and potentially to repurpose for tenant admins). This is essentially a UI in our React app where we can: - List tenants. - Create a new tenant. - Assign a user to a tenant or change roles. Using Descope's management SDK from Node or directly through an admin token can power this. Since this is a hands-on plan, we'll sketch how to do it, focusing on using the SDK rather than building a full UI in detail.

Hands-On – Steps:

1. **Secure an Admin Context:** We want only our system admin or a superuser to manage tenants. This could be controlled by a Descope role like "SuperAdmin" at project level for our internal staff. Ensure your account has such a role or use the Descope management key on backend for these operations. For simplicity, in a demo environment, we might directly call management APIs with our key (not recommended in client-side, but in a Node backend route). For a real app, you'd create an admin interface that calls your backend endpoints which in turn call Descope with the management key.

2. **List Tenants:** Use Descope Node SDK (or Python, but Node aligns with our stack). The Management SDK likely has `tenant.list()` method. If not, the API reference shows an endpoint GET /tenants. Implement a route `/admin/tenants` that returns the list. This can be done by storing tenants in a DB, but since Descope holds tenant info, just fetch from Descope:

   ```
   const managementClient = DescopeClient({ projectId: '...',
   managementKey: '...' });
   const tenants = await managementClient.tenant.getAll();
   ```

   (Pseudo-code; adjust to actual SDK syntax). On React side, fetch this route and display in a table with Tenant name, ID, attributes, maybe number of users (which we might count via another call or by linking to user list for that tenant).

3. **Create Tenant:** Provide a form to input tenant name and maybe initial domain or attributes. On submit, call a backend endpoint that uses `managementClient.tenant.create(name, options)` [24]. Descope returns a tenant ID. Display it and perhaps auto-generate some onboarding link.

For example, when we create "Gamma Co", we might also want to invite their admin. The flow could be: - Create tenant via Descope. - Create a user (if email provided) via Descope and assign Tenant Admin

role in that tenant (Descope might allow creating user and directly adding to tenant in one call, or do user create then tenant add). - Send invite link to that user (Descope has user invite which can include a tenant and role in the invite)[125]. Possibly use `managementClient.user.invite(loginId, tenantId, roles, sendEmail=true)` to let Descope email them an invitation to set a password. Combine these steps in the backend, triggered by form data in React.

1. **Manage Tenant Users:** Selecting a tenant in the UI could show a list of its users and roles. We can get that by `managementClient.tenant.getUsers(tenantId)` or by filtering all users by tenant (not sure if a direct endpoint, maybe listing all users and filtering by a field). If not provided, we may maintain our own mapping, but likely Descope has an endpoint to list users in a tenant, since SCIM user provisioning is per-tenant. Show each user with their roles, provide actions to change role or remove from tenant. Changing roles calls `managementClient.user.update(userId, { roles: { [tenantId]: [newRoles] } })` or a specific method for role assignment. Removing user from tenant calls `managementClient.tenant.removeUser(tenantId, userId)`.

2. **Testing the Admin Component:** Create a dummy tenant from the UI, see that it appears in Descope Console's Tenants list. Invite a user and then try logging in as that user (they should get an email, set password, login, see their portal with correct tenant context). Adjust any errors (like missing permissions in management key – ensure the management key used has Tenant Management permission). If building fully, test error cases (like tenant name conflict if any, etc.).

3. **Reuse for Tenant Admin Self-Service:** We can potentially reuse some components so that tenant-specific admins can manage their own users. For example, the user list portion can be reused. But instead of calling our backend with a global management key, tenant admin can call Descope's public APIs with their user token for allowed actions. Actually, Descope's regular user JWT can't create new users. However, Descope has **Admin Widgets** that do exactly this on behalf of tenant admin without exposing keys[131]. But those are prefabricated UI. If we want our custom UI:

4. We'd need to have our backend accept requests from a tenant admin and use the management key to create user in that tenant, but only if the JWT of caller has Tenant Admin role for that tenant (to prevent elevation). That's doable: we verify caller's JWT for role, then perform action with our key limited to that tenant context (some keys can be scoped to a tenant; if not, just ensure logic prevents cross-tenant ops).
5. However, to keep things simpler, one could embed Descope's admin widget which likely already checks the JWT roles and only affects that tenant.

Since time is limited, we won't implement fully, but at least we conceptualized a **Tenant management module**. This ensures that as our business grows, we have the tools to onboard and support new organizations quickly, with minimal manual work in Descope console.

At this stage, our multi-tenancy setup is robust: each tenant isolated, with correct access controls and some degree of self-service. We're leveraging Descope heavily for heavy lifting (user store, SSO integration per tenant, etc.), which saves us development time.

Day 14: Audit Logs, Security Monitoring, and User Analytics in Descope

Scenario 40: Utilizing Descope Audit Logs for Security and Compliance

Background: Descope automatically logs a variety of **audit events** (security-relevant actions) which are valuable for monitoring and compliance [145]. These include login attempts, user creation, role changes, SSO events, etc. A proper implementation should make use of these logs to: - Monitor for suspicious activities (e.g., repeated failed logins, or login from new locations). - Provide an audit trail for admin actions (who invited which user, etc.). - Possibly integrate with external SIEM (Security Information and Event Management) or logging systems.

In this scenario, we'll explore how to search and use audit logs, and how to set up streaming to an external service like an S3 bucket or Datadog for long-term storage [146].

Hands-On – Steps:

1. **View Audit Logs in Console:** Navigate to Descope Console's **Audit Trail** section (if available). Here, filter by events:
2. Try searching for event type "LoginFailed" or by a specific user [147]. If a user had a failed password attempt, it should be logged with timestamp, IP, reason (e.g., wrong password, or account disabled).
3. Look at a "UserCreated" event [148] from when you imported or created a user. It should show whether it was via console (actor = your admin) or via API (actor = management key).

4. Check "RoleModified" or "UsersModified" events if you changed roles for a user [149]. Understand the fields: typically an audit entry has who (actor), what (action type), whom (target user or tenant), when, and additional data (like IP, user agent, etc.) [150].

5. **Search Audit via SDK/API:** Use the Descope Node management SDK to query logs programmatically. For example, if we want to create an admin dashboard for security, we might have a section "Recent Security Events". Descope's API allows filtering by userIDs, actions, date range, etc. [151] [152]. In Node:

   ```
   const audits = await managementClient.audit.search({ actions:
   ["LoginFailed"], from: someDate });
   ```

   This returns an array of events. We might display count of failed logins in last 24h, or list them. Keep in mind rate limit: search is limited to 10 requests/minute [153], which is fine for occasional queries but not for heavy analytics.

6. **Set Up Audit Streaming to External Storage:** For long term analysis or compliance (some standards require storing audit logs for X months), we should offload them. Descope provides connectors:

7. **AWS S3**: You can configure an S3 connector where Descope will push log events to your bucket in near real-time [146]. Or similarly a **Datadog** or **Splunk** connector [154] [155].
8. Let's say we choose Amazon S3. In Descope Console Integrations, find **AWS S3 Connector** (under Analytics or Logging connectors).

9. Provide your AWS bucket name, credentials or IAM role, and path. Descope likely then starts dumping logs as JSON files to the bucket periodically.

10. Alternatively, for a SIEM like Splunk, provide the endpoint and token; for Datadog, an API key and region. For exercise, we won't actually push to a bucket, but know the steps: once set and tested (Descope usually has a "Test" button on connectors), enable it. Perhaps only certain events or all events – likely all audit events are streamed.

11. **React to Audit Events (Automation):** Identify if any events should trigger an alert or automated action:

12. E.g., **LoginExceedMaxAttempts** (user locked due to too many attempts) [156] . We might want to notify our support or the user. This event is logged when Descope's built-in bruteforce protection disables an account. We could write a small script to poll audit API for such events daily and then send emails to those users with unlock instructions. However, Descope might already email users when locked (check).
13. **MFA Reset or Disabled events** – if an admin resets MFA for a user, log should show it. Good practice is to notify the user that happened, in case it was unauthorized. That can be done if we subscribe to audit stream or query daily.

14. **New Device Login** – Descope's logs include remote IP and user agent. We might parse LoginSucceeded events to see if a user logged in from a new IP or country and notify them for security. Descope doesn't do this out of the box as far as I know (some services do). We could implement by storing known user login IPs and comparing. This might be beyond what we do now, but mention that audit data enables such features.

15. **Compliance Considerations:** If your app is under compliance regimes (SOC2, GDPR, HIPAA, etc.), audit logs help. For instance:

16. Prove that only authorized personnel performed admin actions (Descope logs actor for role changes, etc., which can be exported in an audit report).
17. Under GDPR, if a user requests "accountability" info, you could extract their login history from logs to share what access occurred.

18. Keep logs secure and private. If streaming to S3, ensure that bucket is access-controlled (Descope likely requires you to provide a secure write-only access for them). Possibly enable **log retention** strategy: by default Descope might retain logs 30 or 90 days [157] , after that they rely on you streaming them out if you need longer.

19. **Auditing Tenant Admin Actions:** Multi-tenancy: Suppose Alice (Acme Tenant Admin) invites a new user or deletes a user. Is that logged? Likely yes: "UsersCreated" with actor = Alice's userID (though done via our app's management call). Actually, if she used an Admin Widget, Descope would log actor=Alice since she authenticated and triggered it. We should verify:

20. Simulate by having a Tenant Admin add a user (maybe by our admin component or any method).
21. Check audit logs: It might show "UserCreated" with Actor = Alice (her userId) [148] . If so, great, we have an audit trail of tenant-level admin actions too.

By leveraging audit logs, we add an additional layer of **security monitoring** to our app at very low cost (Descope already captures events). We just need to pipe and review them. In a real environment, one would integrate with their SOC (Security Operations Center) workflows – e.g., send critical events to an alerting system. Descope's connector to Datadog or SIEM makes that straightforward.

Scenario 41: User Behavior Analytics – Tracking Auth Flows and Engagement

Background: Beyond security, we want to understand user behavior around authentication. For example: - How many users sign up per week? How many fail to complete sign-up? - What is the login success rate vs failure rate? - Are users using certain methods (like social login vs password) more often? - Funnel analysis: of those who initiate an email OTP login, how many complete it (and how many drop off maybe due to not entering code)?

Descope logs and connectors can feed such analytics. We already set up a Segment connector in a previous scenario [73] to track a "User Signed Up" event. We can expand on that and also show how to track events in our own app and correlate with auth data.

Hands-On – Steps:

1. **Refine Analytics Events via Descope Flows:** In Day 10, scenario 29, we added a Segment "Track" in the sign-up flow for a "UserSignedUp" event [69]. Let's do similar for login:
2. We could add a Segment **Identify** call on successful login to mark a returning user, or a Track event "UserLoggedIn".
3. If using flows for login too, drop a connector at the point of successful authentication (maybe in the "sign-in" flow after password verified, before final redirect).
4. Alternatively, since login is simpler (one step), we can handle it in code: on successful login in React (the `onSuccess` of <Descope>), call our own analytics function or a Descope connector if easier. However, Descope's flow might not have an explicit end-state node for sign-in (since it just completes). But perhaps we could still attach a connector after the "LoginSuccess" action if flows are configurable. Or rely on audit logs streaming to infer logins (Segment connector might not be as relevant for logins).

Given we already have the audit stream, you could use it to count logins etc., but for real-time product analytics, Segment amplitude, etc. is better.

1. **Implement a Dashboard for Auth Metrics:** Use either data from Segment or directly from Descope's audit to show key metrics:
2. Number of sign-ups today.
3. Conversion: if you have flows where user might drop off (e.g., enters email but doesn't finish OTP), how to measure? One way: if we send OTP, we can track "LoginStarted" vs "LoginSuccess". Descope logs have "LoginStarted" and "LoginStartedFailed" for multi-step flows [158]. For OTP, a LoginStarted event means code sent; if no subsequent LoginSuccess, maybe they gave up. By comparing counts of LoginStarted vs LoginSuccess for OTP flows over a period, you gauge drop-off rate. This could be done by querying audit events of those types.
4. If using Segment, our connector could send a "OTP sent" event on LoginStarted and Segment funnel analysis can do the rest.

Let's say we use audit for now: - Query count of `LoginStarted` for method "otp" vs count of `LoginSucceeded` where method "otp" in last 7 days [159] (the audit search filter `methods: ["otp"]` and actions accordingly). - If 100 OTP starts vs 80 success, then 20% dropout – possibly due to not receiving code or losing interest. We can investigate how to improve (maybe shorten flow, or check if SMS deliverability issues if SMS OTP). - Do similar for magic link or password. - Show these on an admin dashboard chart.

This is more analysis than coding here, but describing it sets up how one can refine the auth UX. For example, if we see a high drop-off on magic links, maybe the emails are going to spam – we'd adjust sender or content.

1. **Leverage Segment/Amplitude for Behavior:** Because we integrated Segment, our product team can do deeper analysis:
2. They can see "UserSignedUp" events in Segment, and maybe time to first project created, etc., to measure onboarding success beyond just auth.
3. They can also track unique users logging in daily (Segment identify/track can feed amplitude or mixpanel to count DAUs). We already set up the core events. If needed, adjust or add more:
4. Perhaps track when a user enables MFA in profile as an event (if we allow them to enable 2FA).
5. Track when a user fails MFA and is blocked – to see if MFA is causing churn or support calls.

6. These would be done similarly: either via connectors or manual tracking in code.

7. **Privacy Considerations:** Analytical events around auth are sensitive. Ensure you're not sending PII in plaintext inadvertently (Segment events should avoid including raw passwords obviously; ours didn't). Also, maybe mask things like IP or at least handle them carefully. Descope by default in audit shows IP but when forwarding to analytics, maybe we don't include IP unless needed. Check if Descope's segment connector by default sends user's IP or location – it likely doesn't unless we explicitly mapped it. If your company has privacy guidelines, ensure compliance (e.g., EU user data might not be allowed to be streamed to certain external analytics without consent, etc.).

8. **Iterate Based on Analytics:** As an example outcome: Suppose analytics show many users failing login because they try with wrong method (like thinking they set a password but they actually signed up with Google). That might mean we should better prompt them ("Did you mean to login with Google?" hint on fail). Or if sign-ups from a particular tenant drop at email verification step, maybe the email template isn't reaching or convincing – you might implement a follow-up or reduce friction by allowing email OTP instead of clicking link. All these improvements can be data-driven thanks to our tracking.

In summary, with minimal extra instrumentation, we have a pulse on how users are interacting with our auth system: - Security perspective via audit logs (Scenario 40). - Product perspective via analytics events (Scenario 41). This dual visibility is important: security team monitors the audit feed, product team monitors the Segment feed. We utilized Descope's built-in capabilities (connectors, audit search) to achieve this without reinventing event pipelines.

Scenario 42: Rate Limiting and Security Best Practices for Descope Integration

Background: Security best practices go beyond just logs. We should ensure our integration of Descope is done in a secure manner, and that we're using features to protect against common threats: - **Rate limiting** login attempts and other actions to prevent abuse (like brute force or enumeration). - **Validating Descope JWTs properly** on backend (to not allow tampering). - **Storing secrets safely** (project IDs, management keys) and rotating them if needed. - **Using multi-factor auth policies** and other Descope security features (like bot protection, account lockout). - Following least privilege (for keys and roles).

This scenario is more of a checklist and applying final touches to ensure our auth is robust.

Hands-On – Steps:

1. **Rate Limiting Logins:** Descope internally already rate-limits and locks accounts after certain failed attempts (we saw `LoginExceedMaxAttempts` event) [156]. But at the application layer, especially if we have any endpoints that proxy to Descope, we should enforce limits:
2. For example, if we have an endpoint that triggers sending magic link to an email, we should throttle calls to it per IP or per email to avoid spamming. Descope likely has limits (e.g., only X magic links per hour per user), but extra caution helps. Implement an in-memory or Redis rate limiter for those endpoints (like using express-rate-limit middleware).

3. Similarly, if we let users call our backend to check if an email is registered (some apps have "forgot email? enter and we say if exists"), that could be abused for enumeration. It's best not to reveal whether an email is registered. Descope's API in flows usually doesn't reveal it either (just sends email if exists, but as a developer if using management API, be careful not to return that info distinctively).

4. **JWT Validation and Expiry:** Our Node backend should verify Descope's JWT on each request to protected endpoints. Use the Descope SDK's `validateSession` which does signature check (with JWKS) and expiry check for us [160]. Ensure we cache the JWKS keys or the SDK does it. We should also handle refresh tokens securely – but since we offloaded to cookies/SDK, our backend only sees session JWTs. If a token is expired, we return 401 and the React SDK will attempt refresh. That's all good.

Additionally, consider **JWT revocation**: If we needed to force logout a user (say their account was compromised or deleted), we can use Descope's Management API to revoke their refresh token which essentially logs them out (session JWTs expire soon anyway) [161]. We should incorporate that in any user disable flows (like if an admin disables a user, also call logout on Descope for active sessions). We can track such changes via audit as well.

1. **Protect Management Key and Admin Functions:** The Descope Management Key is powerful. In our design, it's used server-side only, which is correct. Make sure it's stored in secure config (not in client, not in repository plaintext). Rotate it if an admin who had access leaves or if we suspect a leak. Descope allows generating multiple management keys with scopes – we might create a key that only allows user import and use it for migration, and another that allows tenant management, etc., to minimize exposure [112].

Also ensure our admin routes that use the key are behind proper auth (only our staff or tenant admins as allowed).

1. **Use Webhooks or Notifications for critical events:** This crosses with audit but for quick reaction: e.g., if Descope sends a webhook on certain things (they have an Audit Webhook connector [71]), we could directly trigger an action. Possibly integrate with Slack or email for things like "Application's JWK rotated" or "Integration link generated" which might indicate admin activity in Descope console. Given Descope's stable, we might not need immediate webhooks for everything, but just mention that option.

2. **Client-side Security:** We chose HttpOnly cookies for session, which prevents XSS from stealing tokens [34]. Also consider **CSRF**: Since our cookie is HttpOnly, to prevent CSRF, ensure our API requires a header (like X-CSRF-Token or at least that same-site is Lax which stops most CSRF for state-changing operations because those are usually POST and not top-level navigations). Descope's cookies are SameSite Lax by default [153], which is probably okay, but if our app has

any cross-site usage, consider setting them Strict or implementing double-submit cookie. For now, likely fine if our backend is same-site with frontend.

Another aspect: in production, enforce HTTPS so cookies and tokens aren't exposed (we will). Also content security policy to reduce XSS risk overall.

1. **MFA and Recovery:** Encourage use of multi-factor authentication, especially for privileged users. Descope supports TOTP, WebAuthn, etc., easily via flows. We may set a policy: e.g., Tenant Admins must enroll MFA. Could enforce by checking after login if `user.mfaEnrolled` (if such a thing) is false and role = TenantAdmin, then prompt an enrollment flow. While we didn't build an MFA enrollment UI, Descope's flow library has one, and we could direct user to it. It's a nice security best practice for admin accounts.

Also ensure account recovery flows (password reset, account unlock) are in place and working (tested earlier presumably).

1. **Regular Audits:** Use the logs to run periodic audits: e.g., monthly review of admin activity (did any admin do something unusual?), user access (any weird times or locations?). If using an external SIEM, set up alerts (like more than 5 failed logins for different accounts from same IP – could be a credential stuffing attack; we should block that IP or force captcha if we had one). Descope does have built-in bot detection features (the docs mention "cf-bot-score" in request details [162], indicating Cloudflare integration likely, which might mark some requests as bot). We could use those signals if exposed to implement additional verification when needed.

By adhering to these practices, we fortify our authentication system's security. The combination of Descope's built-ins (like automatic account lockout, bot detection, MFA, etc.) and our careful integration (like using cookies vs localStorage, verifying tokens on backend, etc.) yields a robust setup.

Now, having covered all requested topics, we're ready to move into demonstrating and perhaps packaging what we built – which will be our final day focus, including how to showcase this solution to a customer.

Day 15: Final Demonstrations, Customer Pitch, and Reusable Components

Scenario 43: Demonstrating Descope Integration – Script for Customer Demo

Background: When showcasing our product's authentication to a potential customer (especially a technical audience or a security team), we want to highlight not only the user experience but also the security and flexibility features. This scenario outlines a demo "script" with a lab-style sequence to effectively demonstrate what we've built: - Show end-user login/signup from both B2C and B2B perspectives. - Show admin and security monitoring capabilities. - Perhaps show how quick it is to integrate a new IdP (if the customer is an enterprise that would value that). - Emphasize Descope benefits: no passwords needed (if using magic link), or passwordless options, easy SSO, etc.

We'll structure the demo into steps as we would perform them in real-time.

Demo Script / Steps:

1. **Introduction (Slides or Talk):** Briefly explain we have implemented Descope for authentication in our app to achieve **passwordless login**, **multi-tenant SSO**, and robust security out-of-the-box. Mention that what we'll show is live in the app.

2. **User Sign-Up and Login (B2C flow):** Start by demonstrating a new user sign-up as a consumer user:

3. Open the app's sign-up page. Use a personal email (not company domain) like `demo.user@example.com`.
4. Explain: "Our app offers passwordless sign-up via email magic link. I'll enter my email and click Sign Up."
5. Go to email (we might have a test inbox open) and show the magic link email (with branding if we customized it). Highlight that it's branded (the email template has our company logo via Descope).
6. Click the magic link -> logged into the app. Point out that no password was created or needed, reducing friction.
7. Show that the user is now created in the system. Perhaps switch to Descope console Users view to show "demo.user@example.com" appears, verified.
8. Log out and show login as same user: this time use a different method, e.g., show that they can either get another magic link or maybe use OTP. If we enabled multiple methods, demonstrate one (like "I can also receive a one-time code to login if I prefer").

9. Show the login success. Indicate how quick and seamless it is, and mention under the hood Descope ensures the link/code can't be reused, etc.

10. **B2B SSO Login Flow:** Next, demonstrate an enterprise login:

11. "Now I'll show logging in as an Acme Corp employee via their company SSO (Okta)."
12. Possibly we have a dummy Okta setup; if so, simulate by entering an email `alice@acme.com` on the login page.
13. The app/Descope will redirect to Okta – show the Okta login page, log in with Okta credentials.
14. After redirect back, we're logged into the app as Alice with appropriate role. Highlight: "Notice I wasn't asked to create an account – my identity from Okta was trusted via SAML and an account was created on the fly. My role as Tenant Admin was automatically assigned from the SSO group mapping."
15. Show something tenant-specific, like a message "Welcome, Acme Corp" possibly displayed (since we can know tenant from JWT).

16. If possible, show the multi-tenant switch or mention "If I also had access to Beta Inc, I could switch context – but normal users won't see anything from other tenants."

17. **Admin Invitation and Role Enforcement:** While logged in as Alice (Tenant Admin), demonstrate adding a user:

18. Use the tenant admin UI (or Descope console if not built, but better our UI) to invite Bob (bob@acme.com) as a regular user.
19. Show Bob gets an invite email (Descope's invite template with Acme's branding if set).
20. Accept invite as Bob (set password or magic link, whichever the flow is).

21. Log in as Bob and attempt to access an admin-only feature (like a user management page). It should be forbidden: “Here you see role-based access control in action. Bob is not an admin, so he cannot see the user management section.”

22. Then as Alice, access the same section – allowed. This highlights RBAC working and how easy it was to configure in Descope (just roles assigned to user and checked in frontend/backend).

23. **Security Monitoring:** Switch hats to security officer. Show the audit logs:

24. Open the audit trail (maybe via our admin dashboard or Descope console’s audit page).
25. “We can see all authentication events here. For example, Alice’s SSO login is logged (with timestamp and that it was via SAML). My earlier magic link sign-up for demo.user@example.com is logged as well. Failed login attempts would appear with reasons – let’s simulate one.”
26. Simulate a failed login: e.g., try logging in with wrong OTP or an expired link to generate a LoginFailed.
27. Refresh logs to show that event. “We would get alerted if there were many of these indicating a brute force attack. In fact, Descope automatically locks the account after a few tries – here’s a *LoginExceedMaxAttempts* event if someone triggers that. The account is then temporarily disabled, thwarting password spraying.”
28. Emphasize that these logs can be streamed to the customer’s SIEM or alerting system. “For instance, we’ve set up integration with Datadog – any critical auth events will show up there within seconds.” (If possible, show a Datadog dashboard with an event).

29. Also highlight compliance: “We keep an immutable audit trail of all auth events for 90 days in Descope and beyond that in our archive, helping with compliance audits.”

30. **Analytics and Metrics:** Shift to product manager perspective. Show metrics:

31. Open our hypothetical internal dashboard or at least talk to it: “We track sign-up conversion and login success rates. Over the last month, we see that 5% of logins started via OTP are not completed – we can use that insight to possibly simplify the OTP process or check if SMS deliverability is an issue. This is data we get thanks to Descope’s event hooks we’ve integrated with Segment.”

32. If we had a Segment or analytics UI, point to an event count or funnel. If not, describe: “In Segment, every sign-up and login is captured, so the team can analyze onboarding and usage patterns. For example, we know 30% of our users use Google SSO to login vs 70% email – so maybe we’ll add more social login options in future.”

33. **Customization & Branding:** Note how each tenant’s experience can be customized:

34. “As you saw, Acme’s emails and login portal had their branding. We can do this per tenant easily. It helps adoption because employees see their familiar logo.”

35. “We can even host the auth on a custom domain (auth.acme.com) so everything feels native to their ecosystem.”

36. **Q&A on Integration:** If customer is technical, mention:

37. “It took us just a couple of days to integrate all this. We didn’t have to build password management, SSO protocols, or MFA from scratch – Descope provided SDKs that snapped in.

Our front-end uses their React SDK (less than 50 lines of code to set up flows), and our backend uses their Node SDK for verifying tokens and performing management tasks [61] [151].”

38. “This means fewer bugs and security holes, because we rely on a well-tested platform for auth. They handle updates for new security standards (like if a new MFA method emerges or an OAuth vulnerability is discovered, Descope patches it globally).”

39. Possibly mention pricing or scalability if relevant: “Descope is scalable, and we only pay for active users. It supports over X login methods, so if later we want to add say OTP via WhatsApp or WebAuthn biometric login, it’s mostly configuration – no big development project.”

40. **Conclusion:** Summarize benefits:

41. End-user convenience (passwordless, SSO).
42. Admin control and security (RBAC, audit logs).
43. Developer ease (fast integration, less maintenance).
44. Future-ready (easy to add new tenants, protocols).
45. Then say “We believe this modern auth approach not only improves security but also user experience, which is a rare win-win.”

This script hits the major selling points a customer cares about: UX, security, compliance, and operational ease. It demonstrates practically everything we implemented in these labs in a narrative format.

Scenario 44: Building Reusable Auth Components for Future Projects

Background: Finally, we reflect and generalize what we built into reusable components or modules that can be used in other projects or by other teams. We want to avoid re-writing auth logic for each new app. With Descope, many parts can be encapsulated: - A React `<AuthProvider>` wrapper and maybe context providing user info. - A `<LoginButton>` component that triggers flows (with props to select method or style). - A `<ProtectedRoute>` component (we made one earlier) to guard routes. - Perhaps a Node middleware for express to validate tokens and attach user roles to req. - These can form a small internal library or even be contributed to an open source (if we built something novel on top of Descope).

Hands-On – Steps:

1. **Extract Frontend Auth Utilities:**
2. From our code, take out pieces such as:
    - The `<ProtectedRoute>` HOC we wrote in Day 8【analysis】.
    - The logic to fetch `isAuthenticated` and `isSessionLoading` and render appropriately (which might be repeated in multiple pages).
    - Possibly create a `useAuth()` custom hook that wraps `useSession` and `useUser` from Descope to give a combined object { user, isAuthenticated, login(), logout() } for convenience. (Descope’s hooks already pretty easy, but a custom hook could abstract the presence of Descope specific details and give our components a simpler interface).
3. Create a file `AuthComponents.jsx` and move these in. Document them.
4. Example `LoginButton`: If we often need a button that triggers a certain flow (say “Login with Google”), we can embed the logic: `<Descope flowId="oauth-google" />` maybe, or if using their widget perhaps no code needed. But maybe a function that calls `descope.loginWithOAuth('google')` if SDK provides.

5. Ensure all styling or context is properly included (the AuthProvider still needed in app root, but that can be part of our library usage instructions).

6. Also the admin UI bits we made (Tenant management, etc.) can be component-ized if they might be reused in e.g. an admin console in another project.

7. **Backend Reusable Modules:**

8. Create an `auth.js` module for Express that does:

```
const descopeClient = DescopeClient({ projectId: '...', // maybe take
from env
                                     baseUrl:
process.env.DESCOPE_BASE_URL || undefined });
module.exports.requireAuth = function(req, res, next) {
    const token =
req.cookies?.DS; // or Authorization header if we used that
    if (!token) return res.status(401).send("No token");
    try {
       const { user } = descopeClient.validateSession(token);
       req.user = user;
       next();
    } catch(e) {
       return res.status(401).send("Invalid or expired session");
    }
}
```

   And perhaps `module.exports.requireRole = function(role) {...}` to check `req.user.roles` or permissions for a given endpoint (with multi-tenant awareness maybe require role in req.user for req.user.tenantId).

9. This can be packaged so that any Node service that needs to validate Descope JWT can just use this middleware. It's essentially what we coded spontaneously, but nice to formalize.

10. **Documentation and Settings:**

11. Document usage of these components. E.g., in a README: "To use our AuthModule:
    - Wrap your app with `<AuthProvider projectId={…}>`.
    - Use `<ProtectedRoute>` for any routes that need auth.
    - Use `useAuth()` in components to get current user info and login state.
    - For backend, ensure you set environment variable `DESCOPE_PROJECT_ID` and include our `authMiddleware`."
12. Also note where to put config (like if using custom domain or styles, how to feed that in – e.g., `AuthProvider` optional `baseUrl` if custom domain).

13. Possibly script for rotation of management keys or such if applicable.

14. **Testing Reusability:** To prove it works, spin up a minimal new React app (create-react-app) and integrate our module:

15. npm install our module (if we published, or just copy files).

16. Set projectId and import components. Try a simple login flow. If it works in that blank app without fuss, success.

17. **Continuous Improvement:** Encourage capturing lessons:

18. For instance, if we find we often need to show a loading spinner until Descope SDK confirms auth status (we did have `isSessionLoading` for that), maybe our `useAuth` hook can provide a combined `loading` state or our ProtectedRoute can handle it internally (e.g., show a spinner while loading).
19. If multi-tenancy is common across our apps, maybe incorporate support for it (like our `useAuth` could expose `currentTenant` easily, or `login(tenant)` function if needed).
20. Keep security in mind – ensure our abstractions don't hide needed checks. For example, someone might forget to use ProtectedRoute, so emphasize usage or maybe integrate with router globally if possible.

By creating these reusable components, we ensure that future projects or new team members can quickly implement authentication following the same pattern, reducing the chance of mistakes. It also standardizes the user experience across our portfolio (same login widget style, etc., modifiable via theming).

We can also consider contributing some improvements back, e.g., if we wrote a nice ProtectedRoute, perhaps share it on Descope's community or as a gist for others using Descope + React.

Conclusion of 15-Day Plan: Over the last 15 days/scenarios, we gradually built a full-fledged auth system leveraging Descope's capabilities. By Day 15, we not only have our solution implemented and secured, but also documented, demonstrated, and modularized for future use. The journey covered everything from basic login flows to advanced multi-tenant SSO and analytics, giving a beginner hands-on experience with modern authentication in a realistic environment.

Scenario 45: Enabling Passkeys (WebAuthn) for Passwordless Biometric Login

Background: Passkeys (WebAuthn) allow users to authenticate using device biometrics or hardware security keys, eliminating passwords entirely [163]. Descope supports WebAuthn as an authentication method that can be integrated into flows or used via SDK calls. By enabling passkeys, we can offer users the option to log in with fingerprint or face recognition, improving both security and convenience. In this scenario, we'll add Passkey support to our application.

Hands-On – Steps:

1. **Enable Passkeys in Descope:** In the Descope Console, navigate to **Authentication Methods -> Passkeys** (WebAuthn settings) [164]. Enable Passkey authentication for your project. You may be prompted to configure relying party details like your domain (e.g., `app.example.com`) and a friendly name. Descope will use these when prompting users to save a passkey credential. Save the settings.

2. **Add Passkey to Signup Flow:** We can integrate passkeys in two ways:

3. **Via Descope Flows:** Edit your sign-up/login flow in the Flow Builder to include a Passkey step. Descope might provide a **"Sign in with Passkey"** screen or action that you can drop in. For

example, on the login screen, you could add a button for "Use Passkey" that triggers WebAuthn. If using the default Descope widget, enabling passkeys in settings may automatically show a "Sign in with device" option.

4. **Via SDK in code:** Alternatively, use the Descope React SDK functions to handle passkeys. Descope's client SDK exposes `webauthn.signUp(loginId, name)` and `webauthn.signIn(loginId)` methods [165]. We could create a button in our UI that calls these. For instance, on first signup, after collecting email we call `descopeSdk.webauthn.signUp(email, fullName)`. This will prompt the browser's credential creation dialog (the user may see a prompt like "Create passkey for example.com"). Similarly, for returning users, `descopeSdk.webauthn.signIn(email)` triggers the WebAuthn verification (user touches their fingerprint or device prompt).

5. **User Onboarding with Passkey:** Demonstrate the passkey registration:

6. On the sign-up page, choose the option *"Use Passkey (Biometric)"* (which we added). Enter an email (for WebAuthn, the email often becomes the user's ID [166]).
7. The browser will ask to create a passkey. If on a device with biometric sensors, it will prompt for fingerprint or Face ID confirmation. Complete that.
8. Descope's SDK will register the new user with the public key credential. On success, the user is now signed up and logged in without any password or OTP. In Descope Console, you'll see the user created; their loginId is the email, and under "Authenticators" you'll see a WebAuthn device registered.

9. Important: Typically WebAuthn requires HTTPS and a supported browser. Ensure you test on a domain with SSL (localhost with a proper certificate or a deployed environment) because WebAuthn won't work on plain http://localhost.

10. **Login with Passkey:** Now test sign-in with the passkey:

11. On the login page, choose "Use Passkey" for a user that has one registered (the one from previous step). You might just click a button without entering email if the browser can identify the user (some implementations allow discovery of available credentials by domain). However, often you still provide the username (email) then it prompts for the passkey for that account.
12. The browser will prompt "Authenticate with your device" – use the saved fingerprint or PIN. If successful, Descope logs you in immediately. No password, no code – it's seamless and phishing-resistant (since the cryptographic challenge is tied to your domain).

13. Under the hood, Descope's `webauthn.signIn` took care of the WebAuthn ceremony: it sent a challenge, the browser's authenticator signed it, and Descope verified it on the backend [167], finally issuing a session JWT to our app.

14. **Backup and Fallback:** It's best practice to offer fallback methods in case a user's device biometrics aren't available (e.g., lost phone). Descope supports **Recovery Codes** and other fallback options [168]. Ensure that when passkeys are enabled, you also allow users to register a backup OTP or have a recovery code. In Descope Console, you can enable **Recovery Codes** and **Authenticator App (TOTP)** as alternative MFA methods [169]. For our app, we might prompt users to save a recovery code after setting up a passkey (Descope can generate those). This way, if they get locked out of their device, they can still log in using the code or contact support to reset.

15. **User Experience and Security Benefits:** Now you can highlight that your application offers state-of-the-art authentication:

16. The login flow can prompt “Use your fingerprint to login” – a very user-friendly experience.
17. Passkeys are **phishing-resistant** (they won’t work if the domain is fake, unlike passwords or OTPs) and they satisfy strong authentication requirements (useful for compliance like FIDO2, and mandated in some scenarios).
18. No password to remember or to be leaked. Even if an attacker phishes the user’s email, without the physical device or biometric they cannot login.

19. For users who prefer traditional methods, those still exist (we didn’t remove magic link or SSO – passkeys are an added option).

20. **Testing on Multiple Devices:** One consideration – passkeys are often device-specific (unless the user uses a platform that syncs them, like iCloud Keychain syncs passkeys across Apple devices). Test logging in on a second device: if the passkey isn’t there, the user will need to login via an alternate method (email OTP for instance) and then register a passkey on that device as well. You can allow multiple WebAuthn credentials per user. Descope’s management UI shows how many authenticators a user has; users can register a “new device” in their profile by essentially calling `webauthn.signUp` again while logged in to add another key (Descope has an API `webauthn.update()` for adding a WebAuthn method to an existing account, which the SDK likely exposes).

By completing this scenario, we’ve implemented the cutting-edge of passwordless login. In just a few steps, Descope allowed us to integrate passkeys – something notoriously complex to do from scratch – into our app’s auth system [170]. Our 15-day journey ends with a highly secure, user-friendly authentication system: from basic email/password or OTP all the way to SSO and biometric passkeys, all managed through one unified Descope platform. The application is not only ready to handle today’s needs but is also future-proofed for evolving authentication standards.

---

[1] [2] [3] What Is RBAC: Your Simple Guide
https://www.descope.com/learn/post/rbac

[4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [18] [19] [20] [21] [22] [23] [25] [26] [27] [28] [29] [30] [32] [33] [36] Role-Based Access Control | Descope Documentation
https://docs.descope.com/authorization/role-based-access-control

[14] [16] [17] [31] [34] [35] [40] [44] [45] [46] [47] [132] [133] Auth Helpers Overview | Client SDK | Descope Documentation
https://docs.descope.com/client-sdk/auth-helpers

[15] [38] [39] [41] [61] React Quickstart | Descope Documentation
https://docs.descope.com/getting-started/react

[24] [128] [129] [130] [131] [144] Tenants | Descope Documentation
https://docs.descope.com/management/tenant-management

[37] A Comparison of Cookies and Tokens for Secure Authentication
https://developer.okta.com/blog/2022/02/08/cookies-vs-tokens

[42] [145] [147] [150] [151] [152] [153] [159] [160] [162] Audit Overview | Descope Documentation
https://docs.descope.com/audit-trails-and-integrations

43 72 148 149 156 158 Descope Audit Events | Descope Documentation
https://docs.descope.com/audit-trails-and-integrations/audit-events

48 49 50 57 59 60 Flows Overview | Descope Documentation
https://docs.descope.com/flows

51 52 53 54 58 Managing Descope Flows | Descope Documentation
https://docs.descope.com/flows/intro-to-flows/manage-flows

55 Conditions in Flows - Descope Documentation
https://docs.descope.com/flows/conditions

56 62 Customizing Descope Flows | Descope Tutorial - YouTube
https://www.youtube.com/watch?v=3UlEFKOCMCI

63 64 65 66 67 68 69 70 73 155 Descope Segment Connector | Descope Documentation
https://docs.descope.com/connectors/connector-configuration-guides/analytics/segment

71 Audit Webhook Connector - Descope Documentation
https://docs.descope.com/connectors/connector-configuration-guides/network/audit-webhook

74 Bring Your Own Screen - Descope Documentation
https://docs.descope.com/flows/screens/byos

75 76 136 137 138 139 140 141 Styles & Themes | Descope Documentation
https://docs.descope.com/management/styles

77 78 79 82 83 92 93 94 95 96 98 Configure Okta SSO as IDP | Descope Documentation
https://docs.descope.com/sso/setup-guides/okta

80 81 84 85 86 87 88 89 90 91 99 SSO (Single Sign-on) with SAML Provider | Descope Documentation
https://docs.descope.com/auth-methods/sso/saml

97 Multiple SSO/Identity Providers Per Tenant - Descope Documentation
https://docs.descope.com/sso/multi-sso

100 105 106 107 124 Migration Guide | Descope Documentation
https://docs.descope.com/migrate

101 102 103 108 109 110 112 113 114 115 116 117 118 119 120 121 122 123 Migrate Auth0 Users to Descope | Descope Documentation
https://docs.descope.com/migrate/auth0

104 111 126 127 Migrate Firebase Users to Descope | Descope Documentation
https://docs.descope.com/migrate/firebase

125 142 143 Customize Auth Messaging Templates | Descope Documentation
https://docs.descope.com/management/messaging-templates

134 Making B2B Customer Onboarding Easy With Descope
https://www.descope.com/blog/post/b2b-onboarding

135 Business to Business (B2B) - Descope Documentation
https://docs.descope.com/b2b

146 Audit Trail Streaming - Descope Documentation
https://docs.descope.com/audit-trails-and-integrations/audit-trail-streaming

154 Amplitude Connector - Descope Documentation
https://docs.descope.com/connectors/connector-configuration-guides/analytics/amplitude

157 Access, export, filter audit logs - Azure DevOps Service
https://learn.microsoft.com/en-us/azure/devops/organizations/audit/azure-devops-auditing?view=azure-devops

161 Auth0 OIDC Integration Guide | Descope Documentation
https://docs.descope.com/identity-federation/applications/setup-guides/auth0/auth0-oidc

163 165 166 167 169 Passkeys Authentication Guide | Client SDK | Descope Documentation
https://docs.descope.com/auth-methods/passkeys/with-sdks/client

164 Passkeys Settings Guide | Descope Documentation
https://docs.descope.com/auth-methods/passkeys/settings

168 Developer Guide: How to Implement Passkeys - Descope
https://www.descope.com/blog/post/developer-guide-passkeys

170 Customize Passkey Authentication - Descope Documentation
https://docs.descope.com/auth-methods/passkeys