# Aptos Keyless Integration Guide

**Keyless Account Scoping**

Use of the ***Aptos Keyless Integration Guide*** will allow for the integration of keyless accounts directly into your application. This means that blockchain accounts are scoped to your application's domain (logging in with your Google account on dApp A and logging in with your Google account on dApp B will create separate accounts). Stay tuned for more to come on Aptos' plan to allow Keyless accounts to be used portably across applications.

To provide feedback, get support, or be a design partner as we enhance Aptos Keyless, join us here: https://t.me/+h5CN-W35yUFiYzkx

At a high level, there are three steps to follow in order to integrate Keyless Accounts.

1. **Configure your OpenID integration with your IdP.** In this step, the dApp will register with the IdP of choice (e.g. Google) and receive a `client_id`
2. **Install the Aptos TypeScript SDK.**
3. **Integrate Keyless Account support in your application client**
    1. Set up the `"Sign In with [Idp]"` flow for your user.
    2. Instantiate the user's `KeylessAccount`
    3. Sign and submit transactions via the `KeylessAccount`.

## Example Implementaion

You can find an example app demonstrating basic Keyless integration with Google in the aptos-keyless-example repository. Follow the directions in the README to start with the example. For more detailed instructions on keyless, please read the rest of this integration guide.

## Step 1. Configure your OpenID integration with your IdP

The first step is to setup the configuration with your IdP(s).

Follow the intructions here

## Step 2. Install the Aptos TypeScript SDK

```
# Keyless is supported in version 1.18.1 and above
pnpm install @aptos-labs/ts-sdk
```

## Step 3. Client Integration Steps

Below are the default steps for a client to integrate Keyless Accounts

## 1. Present the user with a “Sign In with [IdP]” button on the UI

1. In the background, we create an ephemeral key pair. Store this in local storage.

```
import {EphemeralKeyPair} from '@aptos-labs/ts-sdk';

const ephemeralKeyPair = EphemeralKeyPair.generate();
```

2. Save the `EphemeralKeyPair` in local storage, keyed by its `nonce`.

```
// This saves the EphemeralKeyPair in local storage
storeEphemeralKeyPair(ephemeralKeyPair);
```

> Example implementation for `storeEphemeralKeyPair`

3. Prepare the URL params of the login URL. Set the `redirect_uri` and `client_id` to your configured values with the IdP. Set the `nonce` to the nonce of the `EphemeralKeyPair` from step 1.1.

```
const redirectUri = 'https://.../login/callback'
const clientId = env.IDP_CLIENT_ID
// Get the nonce associated with ephemeralKeyPair
const nonce = ephemeralKeyPair.nonce
```

5

4. Construct the login URL for the user to authenticate with the IdP. Make sure the `openid` scope is set. Other scopes such as `email` and `profile` can be set based on your app's needs.

```
const loginUrl = `https://accounts.google.com/o/oauth2/v2/auth?respo
```

5. When the user clicks the login button, redirect the user to the `loginUrl` that was created in step 1.4.

## 2. Handle the callback by parsing the token and create a Keyless account for the user

1. Once the user completes the login flow, they will be redirected to the `redirect_uri` set in step 1. The JWT will be set in the URL as a search parameter in a URL fragment, keyed by `id_token`. Extract the JWT from the `window` by doing the following:

```
const parseJWTFromURL = (url: string): string | null => {
  const urlObject = new URL(url);
  const fragment = urlObject.hash.substring(1);
  const params = new URLSearchParams(fragment);
  return params.get('id_token');
};

// window.location.href = https://.../login/google/callback#id_token
const jwt = parseJWTFromURL(window.location.href)
```

2. Decode the JWT and get the extract the nonce value from the payload.

```
import { jwtDecode } from 'jwt-decode';

const payload = jwtDecode<{ nonce: string }>(jwt);
const jwtNonce = payload.nonce
```

3. Fetch the `EphemeralKeyPair` stored in step 1.2. Make sure to validate the nonce matches the decoded nonce and that the `EphemeralKeyPair` is not expired.

```
const ekp = getLocalEphemeralKeyPair();

// Validate the EphemeralKeyPair
if (!ekp || ekp.nonce !== jwtNonce || ekp.isExpired() ) {
  throw new Error("Ephemeral key pair not found or expired");
}
```

4. Instantiate the user's `KeylessAccount`

   Depending on the type of Keyless you are using, follow the instructions below:

   1. Normal Keyless

```
import {Aptos, AptosConfig, Network} from '@aptos-labs/ts-sdk';

const aptos = new Aptos(new AptosConfig({ network: Network.DEVNET }))
const keylessAccount = await aptos.deriveKeylessAccount({
    jwt,
    ephemeralKeyPair,
});
```

   2. Federated Keyless

```
import {Aptos, AptosConfig, Network} from '@aptos-labs/ts-sdk';

const aptos = new Aptos(new AptosConfig({ network: Network.DEVNET })
const keylessAccount = await aptos.deriveKeylessAccount({
    jwt,
    ephemeralKeyPair,
    jwkAddress: jwkOwner.accountAddress
});
```

## 3. Store the KeylessAccount in local storage (Optional)

1. After the account has been derived, store the `KeylessAccount` in local storage. This allows the user to return to the application without having to re-authenticate.

keyless.ts

```
export const storeKeylessAccount = (account: KeylessAccount): void =
  localStorage.setItem("@aptos/account", encodeKeylessAccount(accoun

export const encodeKeylessAccount = (account: KeylessAccount): strin
  JSON.stringify(account, (_, e) => {
    if (typeof e === "bigint") return { __type: "bigint", value: e.t
    if (e instanceof Uint8Array)
      return { __type: "Uint8Array", value: Array.from(e) };
    if (e instanceof KeylessAccount)
      return { __type: "KeylessAccount", data: e.bcsToBytes() };
    return e;
  });
```

2. Whenever the user returns back to the application, retrieve the `KeylessAccount` from local storage and use it to sign transactions.

keyless.ts

```
export const getLocalKeylessAccount = (): KeylessAccount | undefined
  try {
    const encodedAccount = localStorage.getItem("@aptos/account");
    return encodedAccount ? decodeKeylessAccount(encodedAccount) : u
```

```
    } catch (error) {
      console.warn(
        "Failed to decode account from localStorage",
        error
      );
      return undefined;
    }
  };

  export const decodeKeylessAccount = (encodedAccount: string): Keyles
    JSON.parse(encodedAccount, (_, e) => {
      if (e && e.__type === "bigint") return BigInt(e.value);
      if (e && e.__type === "Uint8Array") return new Uint8Array(e.valu
      if (e && e.__type === "KeylessAccount")
        return KeylessAccount.fromBytes(e.data);
      return e;
    });
```

## 4. Submit transactions to the Aptos blockchain

1. Create the transaction you want to submit. Below is a simple coin transfer transaction for example:

```
import {Account} from '@aptos-labs/ts-sdk';

const bob = Account.generate();
const transaction = await aptos.transferCoinTransaction({
    sender: keylessAccount.accountAddress,
    recipient: bob.accountAddress,
    amount: 100,
});
```

2. Sign and submit the transaction to the chain.

```
const committedTxn = await aptos.signAndSubmitTransaction({ signer:
```

3. Wait for the transaction to be processed on-chain

```
const committedTransactionResponse = await aptos.waitForTransaction(
```

Last updated on May 10, 2025