Concordium IDApp SDK v1.2 Integration Guide

Introduction

Concordium IDApp SDK — a TypeScript based SDK built for wallet providers who want to easily bring Concordium blockchain and identity features into their apps. With simple, intuitive APIs, this SDK makes integration fast, smooth, and hassle-free — so your wallet can go Concordium-ready in no time.

Accounts and identities are strongly linked on the Concordium platform. To separate concerns, the Concordium ID App can handle the identity-related processes, which allows third-party wallets to simplify their Concordium integration (and focus on signing crypto transactions). This technical specification is intended for third-party wallet developers and focuses on high-level guidance for integrating Concordium account creation and recovery into their wallets.

> **Note:** For clarity, we will refer to these third-party wallets as **Account Wallets** throughout this document, to maintain a clear distinction between account management and identity provisioning.

Understanding the Concordium blockchain platform

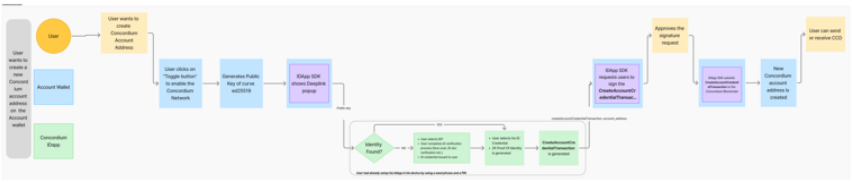
Before diving into implementation details, it's important to understand the fundamentals of the Concordium blockchain and its account creation process.

- **Concordium Blockchain:** Concordium is a public and permissionless blockchain platform designed for both business applications and public use. It combines privacy-preserving features with regulatory compliance, fast transaction processing and cost-effective operations.
- **Concordium Identity Layer:** Concordium provides a new solution to blockchain privacy through its identity layer. When users create an account, they first verify their identity with an authorized identity provider. On the blockchain, their transactions remain private, but their identity can be disclosed through a due legal process.
- **Concordium Account:** To be able to hold, send, or receive CCD or Protocol Level Tokens (PLT), a user must have account address on the Concordium blockchain. Accounts and identities are *strongly linked* on the Concordium Platform. To create an account, user must have a user identity certificate issued by an authorized identity provider on the platform to make sure users are compliant from day one.
- **Concordium ID App:** A standalone mobile application that allows users to create and manage their identities through supported identity providers. The ID App simplifies the integration process by offloading all identity-related responsibilities from wallet developers, enabling them to focus solely on account management—which is their primary area of expertise—without having to handle the complexities of identity verification and issuance.

Account creation user journey

The account creation flow begins when a user initiates the process to generate a new Concordium account address through an Account Wallet. The user enables the Concordium network by clicking a toggle button, prompting the IDApp SDK to display a deep link or QR code. Choosing the relevant option initiates a connection with the Concordium IDApp, passing the user's public key.

Note: If the account wallet and IDApp are on different devices, the user chooses that option, which opens a QR code for the user to scan, and takes them to the IDApp



User journey - Account Creation Flow

If the IDApp is already set up and contains an ID credential, the user has the option to either choose the existing ID credential or create a new one by going through the identity verification process with the identity provider (IDP). Once the user selects the ID credential to use, a zero-knowledge proof of identity is generated, which is then used to create a `CreateAccountCredential` transaction.

The user signs this transaction using the Account Wallet, and the IDApp SDK submits it to the Concordium blockchain. Once the transaction is confirmed, the Concordium account is successfully created, enabling the user to send or receive CCD.

How to integrate Concordium blockchain to create account

Pre-requisite

Reown Protocol:

Account Wallet must support the ReOwn Protocol (formerly, WalletConnect) for connecting with other wallets and dApps. This same protocol will be used to establish sessions with the IDApp.

Note: It is not the responsibility of the IDApp SDK to manage WalletConnect sessions or connections. Account Wallets are expected to handle session initiation, maintenance, and termination.

To connect with the IDApp, Account Wallets should include the Concordium Proposal Namespace when initiating a connection via the WalletConnect protocol:

```
{
  concordium: {
    methods: ["create_account"],
    chains: [ConcordiumIDAppSDK.chainId.Mainnet, ConcordiumIDAppSDK.chainId.Testnet],
  }
}
```

Account Tracking in the Wallet:

It is assumed that the Account Wallet maintains a record of user-created accounts across supported networks, especially for networks where accounts are not enabled by default and require explicit user activation. This includes tracking the associated public key, account address, network, and account index.

Account wallet installs IDApp SDK

The Concordium IDApp SDK is currently available as a TypeScript package via NPM. Native SDKs for Android and iOS will be made available in future releases. To install it, follow these steps:

1. Install using `yarn` or `npm`

```
yarn add @concordium/id-app-sdk
# or
```

```
npm install @concordium/id-app-sdk
```

2. You are ready to import the SDK in your project

```
import {
  ConcordiumIDAppSDK,
  type CreateAccountCreationRequestMessage,
  IDAppSdkWallectConnectMethods,
  type CreateAccountCreationResponse,
  type CreateAccountResponseMsgType,
  type SignedCredentialDeploymentTransaction,
  type RecoverAccountCreationRequestMessage,
  type RecoverAccountResponse,
  type RecoverAccountMsgType,
} from "id-app-sdk";
```

Account Wallet generates seed and key pairs

The Account Wallet must generate an **ed25519** key pair (publicKey/signingKey). This can be done using its existing key generation methods. We recommend the account wallet to use the `generateAccountWithSeedPhrase()` provided by the IDApp SDK, which derives the key pair from a seed phrase:

- Using SLIP-0010, similar to the BIP-32/39 key derivation.
- Using the following **derivation path** format: `m/44'/919'/0'/0'/accountIndex'`

```
const account:CCDAccountKeyPair =
ConcordiumIDAppSDK.generateAccountWithSeedPhrase(seed: string, network:
Network, accountIndex: number = 0)
```

The account is of type `CCDAccountKeyPair`

```
export interface CCDAccountKeyPair {
    publicKey: string;
    signingKey: string;
}
export type Network = 'Testnet' | 'Mainnet';
```

`accountIndex` should be `0` initially and then sequentially incremented to generate new public_key/signing_key pair from same seed.

Account wallet prepares new account creation request

Account wallet calls `getCreateAccountCreationRequest()` of the IdApp SDK to create appropriate connection request which accepts two parameters:

1. `publicKey` : Public key of the user
2. `reason` : [Optional] Provide an appropriate reason of this request

```
// Prepare the request
const new_account_request: CreateAccountCreationRequestMessage =
    ConcordiumIDAppSDK.getCreateAccountCreationRequest(public_key,"I
want to create a new Concordium account");
```

`CreateAccountCreationRequestMessage` format is as follow:

```
export interface CreateAccountCreationRequestMessage {
    publicKey: string;
    reason: string;
}
```

Account wallet sends the account creation request to IdApp

Once the `CreateAccountCreationRequestMessage` is formed, account wallet can then make a wallet connect request to IdApp by passing as message.

```
const chainId = ConcordiumIDAppSDK.chainId.Mainnet // Or, ConcordiumIDAppSDK.chainId.Testnet
const create_acc_resp: CreateAccountCreationResponse =  await this.wc_client.request({
    topic: this.session.topic,
    chainId,
    request: {
      method,
      params:{ message: new_account_request } ,
    },
});
```

The IdApp processes this request and responds with a `CreateAccountCreationResponse` object.

```
export interface IDAppError {
    code: IDAppErrorCode;
    details?: string;
}

export declare enum Status {
    SUCCESS = "success",
    ERROR = "error"
}

export interface CreateAccountResponseMsgType {
    serializedCredentialDeploymentTransaction: SerializedCredentialDeploymentDetails;
    accountAddress: string;
}

// Response type you get from IdApp when creating an account
export interface CreateAccountCreationResponse {
    status: Status;
    message: CreateAccountResponseMsgType | IDAppError;
}
```

The `CreateAccountCreationResponse` has following values:

1. `status` : Indicates whether the account creation was successful ("Success") or not ("Fail").
2. `message` :
   a. If success, contains details like `CreateAccountResponseMsgType.accountAddress` , `CreateAccountResponseMsgType.serializedCredentialDeploymentTransaction`
   b. If error, contains details like `IDAppError.code` and `IDAppError.details`

`serializedCredentialDeploymentTransaction` : A serialised **CreateAccountCredentialTx** unsigned transaction that must be signed and submitted to the Concordium blockchain to activate the account.

> Please note if `chainId` from Account Wallet is different than `chainId` (selected network) set in IdApp, then the IdApp *switches the network* automatically and let user use ID(s) on the requested network.

Note: If `chainId` from Account Wallet is different than `chainId` (selected network) set in IdApp, then the IdApp switches the network automatically on the requested network and let user use ID(s) on that network.

Account wallet submits the CreateAccountCredentialTx on the blockchain

The most critical value in the response is `serializedCredentialDeploymentTransaction`, an encoded transaction that proves ownership of the identity. This transaction must be signed and submitted to the Concordium blockchain.

Generate signature:

```
const signedCredentialDeploymentTx:SignedCredentialDeploymentTransaction = await ConcordiumIDAppSDK.signCredentialTransaction(createAccountCredentialTx, secret_key);
```

The response of `signCredentialTransaction()` if of type `SignedCredentialDeploymentTransaction`

```
export type SignedCredentialDeploymentTransaction = {
    credentialDeploymentTransaction: CredentialDeploymentTransaction;
    signature: HexString;
}
```

Which has **CreateAccountCredentialTx** and a signature string.

Submit the transaction:

```
const txHash: string = await ConcordiumIDAppSDK.submitCCDTransaction(signedCredentialDeploymentTx.credentialDeploymentTransaction, signedCredentialDeploymentTx.signature, 'Testnet')
```

> **Notes:**
>
> - *The `accountIndex` index must be incremented after each request, regardless of transaction success or failure.*
> - *The Account Wallet should prompt the user to sign the **CreateAccountCredentialTx**.*
> - *This is a **gasless transaction**, so there is no need to worry about transaction fees.*

Account Wallet stores keys and newly generated wallet address

Once the transaction is submitted, it is the responsibility of the Account Wallet to securely store the following:

- The **key pair** (sk, pk)
- The associated **Concordium account address**
- The `accountIndex` index counter

Account recovery user journey

The Concordium account recovery flow begins when a user wishes to restore access to an existing Concordium account within their account wallet. The user first completes the wallet's standard recovery process. After recovery, the account wallet can then recover wallet address using IdApp SDK by passing user's public key.

The account wallet then retrieves the latest transaction sequence number for that account from the blockchain and stores all necessary information such as the account address and sequence number. With the account data fully restored, the user regains access and can now send or receive CCD tokens.

How to integrate Concordium blockchain to recover an account

The account recovery process is straightforward too.

Account wallet retrieves user's public key

The Account Wallet starts by retrieving the public key of the account to be recovered. If the key was originally generated using the IDApp SDK, this process involves recovering the seed phrase and `account_index`, then deriving the public key using `generateAccountWithSeedPhrase()` of the IDApp SDK.

> **Note:** The Account Wallet is expected to manage `account_index` counter, so that it knows how many public_key/signing_key pairs have been created from a given seed phrase.

Account Wallet recovers the account

The Account Wallet then calls the `getKeyAccounts()` method provided by the IdApp SDK. This method connects with Concordium blockchain and looks for associated accounts with the user's public key of type `KeyAccount`:

```
const keyAccounts: Array<KeyAccount> =
        ConcordiumIDAppSDK.getKeyAccounts(public_key, network);
```

`KeyAccount` format:

```

export interface KeyAccountPublicKey {
  schemeId: string;
  verifyKey: string; // public key
}

export interface KeyAccount {
  address: string;  // wallet address
  credential_index: number;
  is_simple_account: boolean;
  public_key: KeyAccountPublicKey;
}
```

> **Note:** In future versions, there will be support for batch recovery of account addresses.

Account Wallet stores keys and recovered account address

After a successful recovery, the Account Wallet must securely store the following:

- The **key pair** (sk, pk)
- The associated **Concordium account address**
- The `accountIndex` index counter

Account Wallet can query account status from the blockchain

Account Wallet may use getAccountInfo() of Concordium Web-SDK to retrieve information about an account from Concordium blockchain by providing `accountAddress` as parameter.

Error Handling

In the event of a failure, the IDApp responds with a status of `Error`, along with a specific error code and corresponding details to help identify the cause.

The available error codes are defined in the `IDAppErrorCode` enum as follows:

```
export enum IDAppErrorCode {
  AccountNotFound = 1,                // The specified account does not exist
  AccountCreationFailed = 2,          // Failed to create the account
  DuplicateAccountCreationRequest = 7, // An account with the publicKey already exist
  UnknownError = 99,                  // An unexpected or unknown error occurred
}
```

IDAppSDK Popups

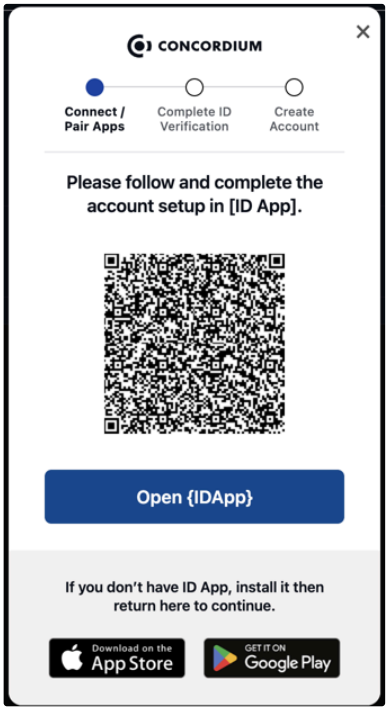
The `ConcordiumIDAppPoup` module from the `id-app-sdk` enables account wallets to display optional popups for interacting with the Concordium ID App. These popups provide a smooth and embedded user experience for flows like opening the ID App, creating an account, or recovering an account — directly within the wallet interface. To use these popups, wallets must define handler functions (`onIdAppPopup` , `onCreateAccount`) which encapsulate the logic for each action.

Open IDApp via Deeplink Popup

To open the ID App through a deeplink popup, use the `invokeIdAppDeepLinkPopup` method.

```
import { ConcordiumIDAppPoup } from "id-app-sdk";

ConcordiumIDAppPoup.invokeIdAppDeepLinkPopup();
```



This popup can be embedded behind a button in the UI, such as:

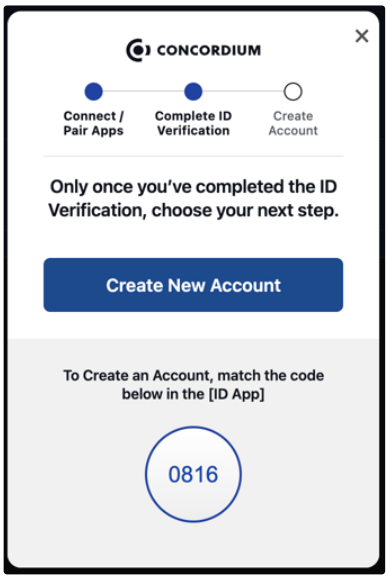
```
<button onClick={() => ConcordiumIDAppPoup.invokeIdAppDeepLinkPopup()}>
  Open ID App
</button>
```

Open IDApp Actions Popup (Account Creation)

To support account creation and recovery flows via the popup, use the `invokeIdAppActionsPopup` method. Provide `onCreateAccount` handler to define what should happen in each case.

```
import { ConcordiumIDAppPoup } from "id-app-sdk";

ConcordiumIDAppPoup.invokeIdAppActionsPopup({
  onCreateAccount,
  walletConnectSessionTopic
})
```



Secure Authentication

The account creation flow uses a secure authentication mechanism between the user's Account Wallet and the ID App to guard against identity theft, ensuring that requests can only be delivered to the legitimate Account Wallet rather than a malicious third-party. When the Account Wallet triggers the "createAccount" action via `invokeIdAppActionsPopup()`, the Account Wallet must include the `walletConnectSessionTopic` — a unique session identifier negotiated over WalletConnect—to validate and bind that request to the correct session.

> **Note:** These popup integrations are **optional**. Wallets that choose to use them must implement the handler functions with the desired logic.