typescript-demo)

Typescript SDK

CCL IBC SDK for typescript developers

Typescript Demo

See a fullstack Next.js typescript demohere (using Osmosis Mainnet). Code availablehere .

Dependencies

For encryption, we recommend using@solar-republic/neutrino which has usefulchacha20poly1305 related functionalities and additional primitives for generating ephemereal lightweight wallets. Installation:

```

Copy npminstall--save@solar-republic/neutrino

```

For signing, encoding and other cryptographic needs in the Cosmos ecosystem, it is common to use the suite of@cosmjs packages. You can install the following:

```

Copy npminstall--save@cosmjs/crypto@cosmjs/amino@cosmjs/encoding

```

If you are developing in the browser environment or connecting to a public network you might also need

```

Copy npminstall--save@cosmjs/stargate

```

or

npminstall--save@cosmjs/cosmwasm-stargate

```

Note: You can also use any Typescript / Javascript package managers and runtimes e,g, bun , yarn , pnpm etc.

Generating Wallets

Forchacha20poly1305 , we need to use a crypthographic keypair and it's advised to use one that isn't the same as the user's wallet. The SDK provides a method for generating a new wallet that can be used for encryption purposes. For our purposes, we just need a private / public keys ofSecp256k1 type and there are various ways to generate them.

@cosmjs/crypto

```

Copy import{ Slip10Curve,Random,Bip39,Slip10,stringToPath,Secp256k1 }from"@cosmjs/crypto"

```
constseed=awaitBip39.mnemonicToSeed(Bip39.encode(Random.getBytes(16)));
const{privateKey}=Slip10.derivePath(Slip10Curve.Secp256k1,seed,stringToPath("m/44'/1'/0'/0"));
constpair=awaitSecp256k1.makeKeypair(privateKey); // must be compressed to 33 bytes from 65
constpublicKey=Secp256k1.compressPubkey(pair.pubkey);
```

@solar-republic/neutrino

```
Copy import{ gen_sk,sk_to_pk }from"@solar-republic/neutrino"

constprivateKey=gen_sk(); constpublicKey=sk_to_pk(privateKey);
```

@secretjs

```
Copy import{ Wallet }from"secretjs"; const{privateKey,publicKey}=newWallet()
```

Query Client

Before proceeding to encryption, you might want to create a quering client that will be used for querying the state and contract of the Secret Network. At the very least, it is requited for fetching the public key of a gateway contract for deriving a shared key used later for encryption.

To perform a simple query on a secret contract we can use methods from@solar-republic/neutrino :

```
Copy import{ SecretContract }from"@solar-republic/neutrino"

// create a contract instantse constcontract=awaitSecretContract( secretNodeEndpoint, secretContractAddress )

// query example: // get encryption key from a gateway contract constqueryRes=awaitcontract.query({ encryption_key:{} })

// extract res value constgatewayPublicKey=queryRes[2]
```

For more persistent use-cases you can usesecretjs:

```
Copy import{ SecretNetworkClient }from"secretjs"

// create a client: constclient=newSecretNetworkClient({ chainId, url// endpoint URL of the node });

// query the contact and get the value directly constgatewayPublicKey=awaitclient.query.compute.queryContract({ contract_address code_hash,// optionally { encryption_key:{} }// query msg });
```

Signatures

To make sure that malicious applications aren't tricking the user into signing an actual blockchain transaction, it is discouraged to sign arbitrary blobs of data. To address the situation, there are various standards that inject additional data to the message before signing it. The most used in the Cosmos ecosystem is defined inADR 036 which is also used in the SDK.

Browser Wallets

Most of the Cosmos wallets provide a method for signing arbitrary messages following the mentioned specification.

Here is a definition taken from documentation ofKeplr wallet :

```
```

Copy // window.keplr.signArbi.... signArbitrary(chainId: string,signer: string,data: string|Uint8Array) :Promise

```

Although the API method requires achainId, it is set to empty string before signing the message

Cosmology

Cosmologydefines signArbitrary method as part of the interface for their wallet client and provides implementation / integration for every popular Cosmos wallet out there

CosmJS

The logic of the method has already been implemented and proposed as an addition to the library however it has been hanging in a unmerged PR for a while. You can find the full implementation with examples and tests[PR] Here

Manually

Getting Signer and Signer Address

Firstly we need to get an amino signer that will be used for generating the signature.@cosmjs has a defined interfaceOfflineAminoSigner withsignAmino andgetAccounts methods and any other signer that implements it can be used for the purpose.

```

Copy // In browser environment we can get it from a wallet extension. E.g with Keplr: constsigner=window.keplr.getOfflineSigner(chainId);

// ...

// In Node environment we can useSecp256k1Wallet class that also implementsOfflineAminoSigner interface import{ Secp256k1Wallet }from"@cosmjs/amino"

// see examples of generating a random above but in this case you will probably be using a persistent one from a .env file // you can also pass extra options like prefix as the second argument constsigner=awaitSecp256k1HdWallet.fromMnemonic(userMnemonic)

// ...

// Here we are getting the first accounts from the signer // accessing the address and renaming it tosignerAddress const[{ address :signerAddress}]=awaitsigner.getAccounts();

// ...

```

Generating the message and StdSignDoc

To usesignAmino, we need to generate aStdSignDoc object that will be used for signing the message to pass as an argument.

CosmJS provides a function for this:

```

Copy function makeSignDoc(msgs: AminoMsg[], fee: StdFee, chainId: string, memo: string | undefined, accountNumber: number | string, sequence: number | string, timeout_height?: bigint): StdSignDoc;

```

The 036 standard requires the message fields to AminoMsg to be:

```

Copy typeAminoMsg={ // static type url type:; value:{ // signer address signer:string; // plaintext or base64 encoded message data:string; } }

```

As for the rest of the fields, they can be set to an empty string or 0. The final example will look like this:

```

Copy constdata="my message";

constsignDoc=makeSignDoc([{// list of amino messages type:"sign/MsgSignData", value:{ signer:signerAddress, data:data } }], { gas:"0",amount:[] },// StdFee "",// chainId "",// memo 0,// accountNumber 0// sequence // timeout_height)

```

After getting the document we can sign it with the signer:

```

Copy constsignRes=awaitsigner.signAmino(signerAddress,signDoc);

```

Encryption

After getting a public key of a gateway contract you can use it to derive a shared key like this:

```

Copy import{ sha256,Random }from"@cosmjs/crypto" import{ fromBase64,toBase64,toAscii }from"@cosmjs/encoding"; import{ chacha20_poly1305_seal,ecdh }from"@solar-republic/neutrino" // this is a dependency of @solar-republic/neutrino so consider importing these methos import{ concat,json_to_bytes }from"@blake.regalia/belt";

// ... // define // // clientPrivateKey // gatewayPublicKey // signerAddress // // like described above // ...

constsharedKey=sha256(ecdh(clientPrivateKey,gatewayPublicKey))

// We also need to generate a one-time nonce, which can be done like this: constnonce=Random.getBytes(12)

// Prepare a message for the action you want to take on the contract constmsg=ExecuteMsg {...}

/// Defining payload structure idential to the Rust data structure constpayload:EncryptedPayload={ // e.g, cosmos1... user_address:signerAddress, // uint8array -> base64 (-> Binary) user_pubkey:toBase64(signerPubkey), // e.g. "cosmos" from "cosmos1..." hrp:signerAddress.split("1")[0], // or to toBinary(msg) from @cosmjs/cosmwasm-stargate msg:toBase64(json_to_bytes(msg)) }

/// getting the payload ciphertext constciphertext=concat(chacha20_poly1305_seal(sharedKey, nonce, // or toUtf8(JSON.stringify(payload)) json_to_bytes(payload)));

// finally the payload_hash is sha256 of the ciphertext constciphertextHash=sha256(ciphertext);

// ...

```

Encrypting + Signing

Produced digest of hashing the ciphertext can be used as our message that we want to sign according to the 036 standard. The final message will look like this:

```

Copy // calling makeSignDoc with nullifed fields like described earlier constsignDoc=getArb36SignDoc(signerAddress,ciphertextHash); // signing the message constsignRes=awaitsigner.signAmino(signerAddress,signDoc);

```

After this we are getting all the required fields for creating anEncryptedPayload message or anExecuteMsg::Encrypted { ... }

```

Copy constencrypted={ // uint8array -> base64 (-> Binary) nonce:toBase64(nonce), // public key of a pair that was used in deriving a shread key user_key:toBase64(clientPublicKey), // ciphertext of with the user data and actual message payload:toBase64(ciphertext), // sha256 hash of the ciphertext payload_hash:toBase64(ciphertextHash), // signatire over sha256(getArb36SignDoc(payload_hash)) already in base64 payload_signature:signRes.signature.signature, }

```

Broadcasting the message
```

The encrypted message is safe to broadcast over public blockchain and other infrastructure.A common use-case in context of Cosmos account might be broadcasting it over IBC originating from a chain other than the Secret Network.

The potential use-case might involve broadcasting the message by initiating an IBC message directly and attaching the message as a payload (IBC-Hook) or passing the message to a smart contract on a remote chain to process and bridge it to the Secret Network.

Since Cosmwasm is quite flexible with defining messages due to supporting JSON serialization, it is possible the process is very similar in both cases so we only going to cover IBC-hooks for simplicity:

```

Copy import{ MsgTransfer }from"cosmjs-types/ibc/applications/transfer/v1/tx"; import{ SigningStargateClient,MsgTransferEncodeObject }from"@cosmjs/stargate";

/// creating a client constclient=awaitSigningStargateClient( "https://rpc.cosmoshub.io"// endpoint of the remote network signer,// offlineSigner )

// defining the IBC transfer message constmsg:MsgTransferEncodeObject={ typeUrl:"/ibc.applications.transfer.v1.MsgTransfer", value:MsgTransfer.fromPartial({ sender:signerAddress, receiver:secretGatewayContractAddress, sourceChannel:"channel-0", sourcePort:"transfer", timeoutTimestamp:BigInt( // 5 minutes from now | ms -> ns Math.floor(Date.now()+300_000)*1_000_000 ), // IBC Hook memo msg memo:JSON.stringify({ wasm:{ // must be same as receiver contract:secretGatewayContractAddress, // encrypted message defined above msg:encrypted } }) }

// signing and broadcasting the message constres=awaitclient.signAndBroadcast(signerAddress,[msg])

```
Previous Functions, Methods, and Data Structures Next IBC-Hooks Last updated2 months ago