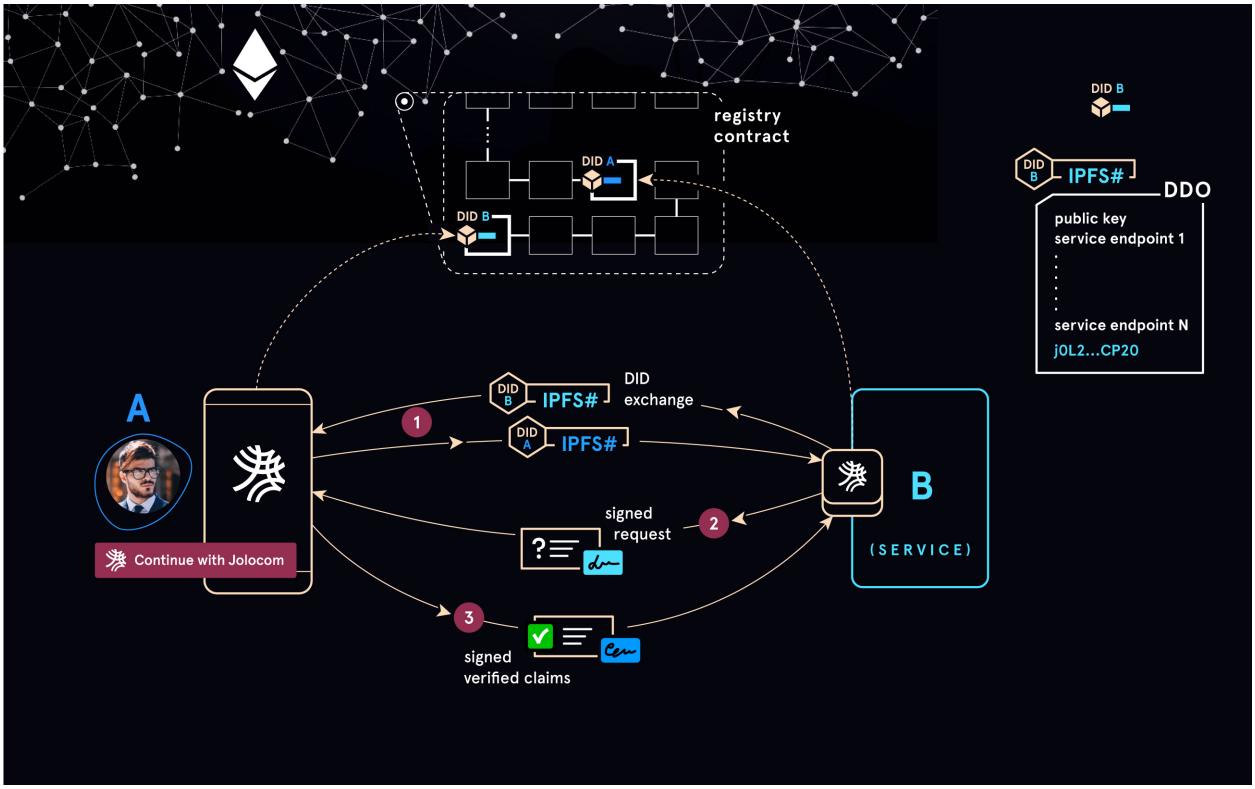
Jolocom-Lib Documentation

Jolocom

Mar 10, 2021

Where to go next?



The Jolocom identity solution aims to be a universal, lightweight, open source protocol for decentralized digital identity and access right management.

The Jolocom stack aggregates a number of existing specifications, largely adopted within the SSI community, and aims to simplify the creation / management of digital identities (including the associated data such as Verifiable Credentials), as well as enable for more complex interactions between the aforementioned identities.

The core specifications on which the Jolocom protocols build are:

- Decentralized Identifiers (DIDs) - every entity making use of the Jolocom stack needs a DID if it wishes to issue / receive Verifiable Credentials, or initiate / participate in interactions with other Jolocom identities. The DID specification allows each agent in the network to generate a decentralized identifier and register / anchor it (the implementation of this operation is normally defined in the corresponding DID Method specification). Once an identity has been "anchored", the corresponding DID will become resolvable by other network participants, and can be used as part of various supported interactions (e.g. for issuing and receiving Verifiable Credentials).
- Verifiable Credentials - this specification allows us to model / express statements made by one identity / DID (in this context - issuer) about another one (in this context - subject). Any identity can ensure that a Verifiable Credential they are presented with has been issued by the listed issuer, and has not been tampered with / modified (e.g. by a malicious holder, or other parties involved in the interaction) due to the inclusion of digital signatures (generated by the signing keys associated with the issuer's DID).

The Jolocom protocol can be used to:

- Create self-sovereign identities (e.g. for humans, organizations, smart agents).
- Associate information with the aforementioned identities in the form of Verifiable Credentials.
- Undertake interactions between identities in order to easily share / receive verifiable information.

CHAPTER 1

Getting Started

Warning: Please be aware that the Jolocom library is still undergoing active development. All identities are currently anchored on the Rinkeby testnet. Please do not transfer any real ether to your Jolocom identity.

1.1 How to install the Jolocom library

To begin using the Jolocom library, first add it as a dependency in your project. You can use `npm` or `yarn` to do so:

```
# using npm
npm install jolocom-lib --save

# using yarn
yarn add jolocom-lib
```

Note: To use the library in a browser or a react native environment additional polyfilling is required. For an example of integrating this library with a react native application, please see the Jolocom SmartWallet metro configuration.

1.2 How to create a self-sovereign identity

In the context of the Jolocom protocol / stack, an SSI is essentially a combination of a DID and a set of signing / encryption / controlling keys. The exact number and type of cryptographic keys required depends on the requirements of the used DID Method.

Before a new identity can be created, a new `SoftwareKeyProvider` instance is required. This class is responsible for managing the cryptographic material associated with an identity. An empty key provider can be instantiated as follows:

```
import { walletUtils } from '@jolocom/native-core'
import { SoftwareKeyProvider } from '@jolocom/vaulted-key-provider'

const password = 'secretpassword'

SoftwareKeyProvider.newEmptyWallet(walletUtils, 'id:', password).then(emptyWallet => {
  console.log(emptyWallet)
})
```

At this point `emptyWallet` is not yet populated with a DID or any signing / encryption / identity management keys. The easiest way to configure the wallet with the required keys is to use the `createIdentityFromKeyProvider` helper provided by the Jolocom library:

```
import { createIdentityFromKeyProvider } from 'jolocom-lib/js/didMethods/utils'
import { JolocomLib } from 'jolocom-lib'

const didJolo = JolocomLib.didMethods.jolo

// The emptyWallet created in the previous example

createIdentityFromKeyProvider(
  emptyWallet,
  password,
  didJolo.registrar
).then(identityWallet => {
  console.log(identityWallet.did)
})
```

The function takes an `emptyWallet` and the corresponding encryption `password` as the first two arguments. The `password` will be used to decrypt the wallet contents before adding new keys / changing the associated DID / storing metadata, etc. Once the wallet state has been updated, the same password is used to encrypt the new state.

Note: Please note that this function mutates the contents of the wallet instance passed as an argument. I.e. if the creation is successful, the `emptyWallet.id` will be set to the new DID, and the *emptyWallet.getPubKeys* method will return all populated keys.

Deriving the required keys, as well as the wallet DID is fully delegated to the `IRegistrar` instance passed as the third argument. Internally, the `registrar` has access to the passed `SoftwareKeyProvider` instance, and can generate and persist all required keys according to the DID method specification. This approach results in greater flexibility when deriving keys (since the behavior is fully encapsulated in the `registrar`), allowing for various approaches to cater to different needs (for instance, the `JoloDidMethod` and the `LocalDidMethod` modules internally rely on specifications such as BIP32 and SLIP0010 respectively for HD key derivation and simpler backups / recovery).

To reiterate, the `registrar` implementation encapsulates the specification(s) employed for deriving keys (including metadata required for derivation, such as paths, indexes, etc.), as well as the process for deriving a DID based on the aforementioned keys.

Provisioning the `SoftwareKeyProvider` with keys and a DID is the first step of the identity creation process. At this point, a DID Document (which indexes the keys and DID we've just created) can be created and "anchored" (the exact operations are DID method specific, and might for example entail creating a record mapping the newly created DID and the DID Document in a verifiable data registry).

Note: For more documentation on the `DidMethod` abstraction, as well as examples of DID methods integrated with

the Jolocom stack, check out the jolo-did-method and the local-did-method repositories.

Please note that the wallet passed to this function is generally expected to be empty (i.e. the `wallet.id` value should not be set to a valid DID, and no keys should be present), with the configuration fully delegated to the specified `registrar`.

The `JoloDidMethod` and `LocalDidMethod` registrars can also create an identity using a correctly populated wallet (i.e. the `id` value is set to a correct DID matching the `registrar's` DID method prefix, and the wallet is populated with the right set of keys, of the right type. In this case, the key / DID generation steps are skipped, and the anchoring operations are fired right away. Whether this functionality is supported or not depends on the `registrar` implementation used.

In case the wallet is not empty, and populated with a DID / set of keys incompatible with the passed registrar, an error is thrown.

Note: Check out the SoftwareKeyProvider documentation for examples on how to manually populate a wallet instance with keys.

1.3 How to reuse a self-sovereign identity

At later points, the identity can be reused if a `SoftwareKeyProvider` provisioned with the corresponding keys is available. The corresponding `SoftwareKeyProvider` can be instantiated in a number of ways (e.g. the wallet's encrypted contents can be persisted to storage, and read / decrypted later, or a BIP39 / SLIP0010 mnemonic can be saved as part of identity creation, and then retrieved / used to derive all required keys).

Given a populated wallet instance, the following alternative to `authAsIdentityFromKeyProvider` can be used to instantiate the identity:

```
import { JolocomLib } from 'jolocom-lib'
import { authAsIdentityFromKeyProvider } from 'jolocom-lib/js/didMethods/utils'

const didJolo = JolocomLib.didMethods.jolo

// E.g. using the previously created / populated SoftwareKeyProvider instance
authAsIdentityFromKeyProvider(
  emptyWallet,
  password,
  didJolo.resolver
).then(identityWallet => console.log(identityWallet.did))
```

The function is similar to the helper we've used to create the identity, except that this function will not attempt to "anchor" the identity but rather it will try to resolve (as defined by the corresponding DID method specification) an existing identity based on the DID / keys held by the passed `SoftwareKeyProvider` instance.

Note: For further examples of identity creation scenarios, check out the Jolocom-SDK documentation

CHAPTER 2

Signed Credentials

The Jolocom library contains a number of functions and classes that enable the creation and consumption of signed JSON-LD Verifiable Credentials. Any agent can ensure a given credential is valid by verifying that the associated cryptographic signature is correct and was generated using the expected private key.

2.1 Create a signed credential

The easiest way to create a signed credential is by using an instance of the `IdentityWallet` class. For examples of the various ways to instantiate an `IdentityWallet`, refer to the Getting Started section.

```
import { claimsMetadata } from '@jolocom/protocol-ts'

const emailAddressSignedCredential = await identityWallet.create.signedCredential({
  metadata: claimsMetadata.emailAddress,
  claim: { email: 'example@example.com' }
  subject: identityWallet.did // Our own DID, results in a self-issued credential
}, password)

...
```

Notice the `JSON` form of the newly created `emailAddressSignedCredential` is simply a JSON-LD Verifiable credential. The `SignedCredential` class provides a number of methods to easily consume the data from the credential.

```
// The credential in JSON form

{
  // Omitted for brevity
  @context: [...],
  id: 'claimId:d9f45722872b7',
  name: 'Email address',
  issuer: 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777',
  issued: '2018-11-16T22:21:28.862Z',
```

(continues on next page)

(continued from previous page)

```
  type: ['Credential', 'ProofOfEmailCredential'],
  expires: '2019-11-16T22:21:28.862Z',
  claim: {
    email: 'example@example.com',
    id: 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777'
  },
  proof: {
    created: '2018-11-16T22:21:28.861Z',
    type: 'EcdsaKoblitzSignature2016',
    nonce: 'fac9b5937e6f0cbb',
    signatureValue:
↪'922c73134cb81558b337a0b222fac3c7f8418ca46febcd57d903def7134843640644f0086d36a6cf29f975b82eabfa459
↪',
    creator:
↪'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777#keys-1'
  }
}
```

Note: Typings / definitions for credential types the library supports by default are made available through the `@jolocom/protocol-ts` npm package. Alternatively, you can check out the GitHub repositories for the core types, as well as demo types.

It's worth noting that in the aforementioned credential, the `issuer`, the `subject`, and the signature `creator` are the same DID. We refer to this type of credential as *self-signed* or *self-issued*.

To issue a credential to another entity, we simply need to specify the DID of the corresponding `subject`:

```
// You can also pass a custom expiry date for the credential
const customExpiryDate = new Date(2030, 1, 1)
const emailAddressSignedCredential = identityWallet.create.signedCredential(
{
  metadata: claimsMetadata.emailAddress,
  claim: { email: 'example@example.com' },
  subject: 'did:jolo:6d6f636b207375626a656374206469646d6f636b207375626a65637420646964'
},
password,
customExpiryDate
)
```

Note: The custom expiry date is an optional argument (if not present, will default to 1 year from Date.now())

Taking a look at the newly created credential, we can indeed see that the `subject`, denoted by the `claim.id` key, is different:

```
// The credential in JSON form
// All irrelevant / repeating fields have been omitted.

{
  '@context': [ ... ],
  ...
  issuer: 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777',
  claim: {
    email: 'example@example.com',
```

(continues on next page)

(continued from previous page)

```
    id: 'did:jolo:6d6f636b207375626a65637420646964d6f636b207375626a65637420646964'
  },
  proof: EcdsaLinkedDataSignature {
    ...
    creator:
↪'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777#keys-1'
    ...
}
```

2.2 Verifying a signature on a signed credential

Perhaps you would like to present the newly created signed credential to another SSI agent part of an interaction. The (intended) recipient needs to be able to verify that the received credential is valid. This can be done as follows:

```
import { JolocomLib } from 'jolocom-lib'

// The credential will often be received serialized in its JSON form.
const receivedCredential = JolocomLib.parse.signedCredential(json)
const valid = await JolocomLib.util.validateDigestable(receivedCredential)
```

The previous step amounts to resolving the DID document associated with the credential `issuer`, and using the listed public keys to validate the credential signature.

The *validateDigestable* function will attempt to resolve the issuer of the signed object as part of the signature verification process. By default, a resolver for the did:jolo is used. In case the issuer is anchored on a different network, an additional argument can be passed to the validation function:

```
import { JolocomLib } from 'jolocom-lib'

// The credential will often be received serialized in its JSON form.
const receivedCredential = JolocomLib.parse.signedCredential(json)
const valid = await JolocomLib.util.validateDigestable(
  receivedCredential,
  // A different resolver can be passed to the function
  JolocomLib.didMethods.jun.resolver
)

// Alternatively, an instance of an identity can be passed as well
const valid = await JolocomLib.util.validateDigestable(
  receivedCredential,
  identityWallet.identity
)
```

2.3 Working with custom credentials

Users are free to define custom credential types. The set of possible interactions / use cases would be quite restricted if only types defined in the aforementioned modules could be used. The following sections delve into why you might want to define custom credentials, and how to do so.

Why would I want to define a custom credential type?

Let's assume you want to use verifiable credentials for managing permissions inside your system. You might have one or more trusted identities that issue access credentials to requesters deemed worthy. For these purposes, none of the credential types we currently provide suffice.

Alternatively, consider this scenario: a bar that only allows adults of legal age on the premises. At a certain point, patrons must prove they are over 18 years of age in order to order enter the establishment. Patrons could of course disclose their individual dates of birth, but this is not optimal in light of the fact that more information is disclosed than required for the purposes of the interaction.

An alternative is to adopt an approach based on verifiable credentials. A trusted entity, such as a government authority, could issue signed credentials to all citizens that request such verifications, i.e. an attestation stating that a citizen is of or over a certain age. A citizen could later present such a credential when entering a bar.

This allows citizens to prove that they are allowed to gain entry to the bar, in a verifiable way, without disclosing any additional information.

Defining custom metadata

So far, when creating credentials, `metadata` provided by the `@jolocom/protocol-ts` package has been used. When creating custom credentials, we have to write our own `metadata` definitions.

Let's take another look at the second example use case from the previous section. One of the many possible `metadata` definitions would be:

```
const customMetadata = {
  context: [{
    ageOver: 'https://ontology.example.com/v1#ageOver'
    ProofOfAgeOverCredential: 'https://ontology.example.com/v1
↪#ProofOfAgeOverCredential'
  }],
  name: 'Age Over',
  type: ['Credential', 'ProofOfAgeOverCredential']
  claimInterface: {
    ageOver: 0
  } as { ageOver: number }
}
```

Note: For more documentation on defining custom credential `metadata`, check out this document. Please note that all examples of **creating credentials** and **creating metadata** are currently outdated (updates already in progress).

The extra typing information - `as {ageOver: number}` is only relevant if you use TypeScript. It enables for auto-completion on the `claim` section when creating a `SignedCredential` of this type. If you develop in JavaScript, you can simply omit this line.

Creating and verifying custom credentials

The newly created `metadata` definition can now be used to create a credential:

```
const ageOverCredential = identityWallet.create.signedCredential({
  metadata: customMetadata,
  claim: {
    ageOver: 18
  },
  subject: requesterDid
}, password)
```

(It's that simple!)

It is worth noting that the custom `metadata` definition is only needed for creating credentials. Validating custom credentials is still as simple as:

```
const valid = await JolocomLib.util.validateDigestable(ageOverCredential)
```

CHAPTER 3

Public Profile

Note: This section only applies to the did:jolo did method, or other DID mehtod implementations with a defined publishPublicProfile method.

A public profile can be attached to an identity to make it easy for any identity with which you interact to easily resolve your identity. This is especially relevant for interactions with online services, as a public profile can be used to advertise interaction conditions, as well as various attestations.

Before you start, be sure to initialize the `IdentityWallet` class as outlined in the Getting Started section.

3.1 Adding a public profile

We currently model public profiles as simple `SignedCredential` instances, each containing the following claims: `about`, `url`, `image`, and `name`.

Before we can publish the credential, we need to first create it:

```
import { claimsMetadata } from 'jolocom-lib'

const myPublicProfile = {
  name: 'Jolocom',
  about: 'We enable a global identity system'
}

const publicProfileCredential = await identityWallet.create.signedCredential({
  metadata: claimsMetadata.publicProfile,
  claim: myPublicProfile,
  subject: identityWallet.did
}, password)
```

Add the newly created public profile to your identity:

```
JolocomLib.didMethods.jolo.registrar.updatePublicProfile(
  softwareKeyProvider,
  password,
  identityWallet.identity,
  publicProfileCredential
).then(successful => console.log(successful)) // true
```

At this point, the public profile will be published / advertised as defined by the employed DID method instance.

3.2 Removing your public profile

```
identityWallet.identity.publicProfile = undefined

JolocomLib.didMethods.jolo.registrar.update({
  identityWallet.didDocument,
  softwareKeyProvider,
  password
}).then(successful => console.log(succesful)) // true
```

Please note that due to the way that IPFS handles the concept of deletion, this delete method simply unpins your public profile from its corresponding pin set, and allows the unpinned data to be removed by the "garbage collection" process. Accordingly, if the data has been pinned by another IPFS gateway, complete removal of stored information on the IPFS network cannot be ensured.

3.3 View the public profile

If the identity has an associated public profile credential, and the correct resolver is used (e.g. in this case `JolocomLib.didMethods.jolo.resolver`), the `Identity` instance returned as a result of resolution will have the associated public profile field populated.

```
JolocomLib.didMethods.jolo.resolver.resolve(identityWallet.did).then(identity => {
  console.log(identity.publicProfile) // Should contain the previously published␣
↪signed credentials
})
```

CHAPTER 4

Credential-based Communication Flows

This section offers an overview of the interaction flows supported by the Jolocom Library.

Identities can interact in incredibly complex ways. We currently support a number of quite simple interaction flows, and intend to greatly expand the list in future releases.

Note: The following sections assume you have already created an identity. If you have not yet created an identity, check out the Getting Started section.

4.1 Credential requests

Many services require their users to provide certain information upon signing up. The Jolocom library provides a simple way for services to present their requirements to users who wish to authenticate through. This is done by creating and broadcasting what we refer to as a "Credential Request". First, the aforementioned request must be generated:

Create a Credential Request

```
// An instance of an identityWallet is required at this point
const credentialRequest = await identityWallet.create.interactionTokens.request.share(
↪{
  callbackURL: 'https://example.com/authentication/',
  credentialRequirements: [{
    type: ['Credential', 'ProofOfEmailCredential'],
    constraints: []
  }],
}, password)
```

Note: Documentation on `constraints` and how they can be used to create even more specific constraints will be added soon.

We also allow for simple constraints to be encoded as part of the credential request. If we want to communicate that only credentials issued by a particular `did` should be provided, we can do the following:

```
import {constraintFunctions} from 'jolocom-lib/js/interactionTokens/credentialRequest'

// An instance of an identityWallet is required at this point
  const credentialRequest = await identityWallet.create.interactionTokens.request.
↪share({
    callbackURL: 'https://example.com/authentication/',
    credentialRequirements: [{
        type: ['Credential', 'ProofOfEmailCredential'],
        constraints: [constraintFunctions.is('issuer', 'did:jolo:abc...')]
      }]
  },
  password)
```

By default the generated credential request will be valid for 1 hour. Attempting to scan or validate the requests after the expiry period will fail. In case you would like to specify a custom expiry date, the following is supported:

```
// You can also pass a custom expiry date for the credential, supported since v3.1.0
const customExpiryDate = new Date(2030, 1, 1)

// An instance of an identityWallet is required at this point
  const credentialRequest = await identityWallet.create.interactionTokens.request.
↪share({
    expires: customExpiryDate,
    callbackURL: 'https://example.com/authentication/',
    credentialRequirements: [{
        type: ['Credential', 'ProofOfEmailCredential'],
        constraints: []
      }]
  },
  password)
```

Note: The expiration date can be passed in a similar manner when creating other interaction token types as well (e.g. Authentication, Credential Offer, etc. . .)

Note: For further documentation and examples explaining how to create and send credential requests, as well as other interaction messages, refer to the API documentation, the the integration tests, and the Jolocom SDK documentation.

The easiest way to make the credential request consumable for the client applications is to encode it as a JSON Web Token. This allows us to easily validate signatures on individual messages, as well as prevent replay attacks.

Note: The interaction messages can be sent to other agents for processing via any preferred channel. In order to allow SmartWallet users to consume the message, it can be encoded as a QR code, or via "Deep Links"..

Consume a Signed Credential Request

Once the encoded credential request has been received on the client side, a corresponding credential response should be prepared and sent:

```
const credentialRequest = JolocomLib.parse.interactionToken.fromJWT(enc)
identityWallet.validateJWT(credentialRequest)
```

Note: The `validateJWT` method will ensure the interaction request is not expired, and that it contains a valid signature.

Create a Credential Response

Once the request has been decoded, we can create the response:

```
/**
 * The callback URL has to match the one in the request,
 */

const credentialResponse = await identityWallet.create.interactionTokens.response.
↪share({
    callbackURL: credentialRequest.payload.interactionToken.callbackURL,
    suppliedCredentials: [signedEmailCredential.toJSON()] // Provide signed␣
↪credentials of requested type
  },
  password, // The password to decrypt the seed for key generation as part of signing␣
↪the JWT
  credRequest // The received request, used to set the 'nonce' and 'audience' field␣
↪on the created response
)
```

The credential supplied above (conveniently) matches what the service requested. To ensure that no credentials other than those corresponding to the service requirements are provided, the following method can be used:

```
// We assume the client application has multiple credentials persisted in a local␣
↪database
const localCredentials = [emailAddressSignedCredential, phoneNumberCredential]
const localCredentialsJSON = localCredentials.map(credential => credential.toJSON())

// The API will change to take instances of the SignedCredential class as opposed to␣
↪JSON encoded credentials
const validCredentials = credentialRequest.applyConstraints(localCredentialsJSON)

console.log(validCredentials) // [emailAddressSignedCredential]
```

Once the credential response has been assembled, it can be encoded and sent to the service's callback URL:

```
const credentialResponseJWT  = credentialResponse.encode()
```

Consume a Signed Credential Response

Back to the service side! The credential response has been received and the provided data is ready to consume. First, decode the response:

```
const credentialResponse = await JolocomLib.parse.interactionToken.
↪fromJWT(receivedJWTEncodedResponse)
await identityWallet.validateJWT(credentialResponse, credentialRequest)
```

Note: The `validate` method will ensure the response contains a valid signature, is not expired, lists our `did` in the `aud` (audience) section, and contains the same `jti` (nonce) as the request.

After decoding the credential response, the service can verify that the user passed the credentials specified in the request:

```
/**
 * We check against the request we created in a previous step
 * this requires the server to be stateful. We are currently
 * exploring alternatives (such as embedding the request in the response token).
 */

const validResponse = credentialResponse.satisfiesRequest(credentialRequest)

if (!validResponse) {
  throw new Error('Incorrect response received')
}

const providedCredentials = credentialResponse.getSuppliedCredentials()

const signatureValidationResults = await JolocomLib.util.
↪validateDigestables(providedCredentials)

if (signatureValidationResults.every(result => result === true)) {
  // The credentials can be used
}
```

4.2 Credential issuance

The Jolocom Library also allows for the issuance of verifiable credentials. Similarly to the flow outlined in the previous subsection, a "Credential Offer" needs to be created and broadcast.

Create a Credential Offer

Firstly, a credential offer needs to be created:

```
const credentialOffer = await identityWallet.create.interactionTokens.request.offer({
  callbackURL: 'https://example.com/receive/',
  offeredCredentials: [{
   type: 'idCard'
  }, {
   type: 'otherCredential'
  }]
})
```

The endpoint denoted by the `callbackURL` key will be used by the client device to send response to the offer.

The CredentialOffer objects may also contain additional information in the form of `requestedInput`, `renderInfo` and `metadata` (which currently supports only a boolean `asynchronous` key).

A more complex offer can be created as follows:

```
import { CredentialRenderTypes } from 'jolocom-lib/interactionTokens/
↪interactionTokens.types'

const idCardOffer: CredentialOffer = {
  type: 'idCard',
  renderInfo: {
    renderAs: CredentialRenderTypes.document,
    logo: {
      url: 'https://miro.medium.com/fit/c/240/240/1*jbb5WdcAvaY1uVdCjX1XVg.png'
    },
```

(continues on next page)

(continued from previous page)

```
    background: {
      url: 'https://i.imgur.com/0Mrldei.png',
    },
    text: {
      color: '#05050d'
    }
  }
  metadata: {
    asynchronous: false // Is the credential available right away?
  },
  requestedInput: {} // What is required to receive the credential, e.g. residence␣
↪permit credential, etc.
  }
}
```

Note: The `metadata.asynchronous` and `requestedInput` keys are not currently used, and act as placeholders. We are awaiting further standardization efforts. An example of such standardization initiatives is the Presentation Exchange specification proposal.

The `renderInfo` is used to describe how a credential should be rendered and is currently supported by the Jolocom Smartwallet. The currently supported options are:

```
enum CredentialRenderTypes {
  document = 'document',
  permission = 'permission',
  claim = 'claim',
}
export interface CredentialOfferRenderInfo {
  renderAs?: CredentialRenderTypes
  background?: {
    color?: string // Hex value
    url?: string // URL to base64 encoded background image
  }
  logo?: {
    url: string // URL to base64 encoded image
  }
  text?: {
    color: string // Hex value
  }
}
```

Consume a Credential Offer

On the client side, we can decode and validate the received credential request as follows:

```
const credentialOffer = JolocomLib.parse.interactionToken.fromJWT(enc)
identityWallet.validateJWT(credentialRequest)
```

Note: The `validateJWT` method will ensure the interaction request is not expired, and that it contains a valid signature.

Create a Credential Offer Response

To create a response for a credential offer, the callbackURL and the selected credentials must be used:

```
const offerResponse = await identityWallet.create.interactionTokens.response.offer({
  callbackURL: credentialOffer.callbackURL,
  selectedCredentials: [
    {
      type: 'idCard'
    },
    {
      type: 'otherCredential'
    }
  ]
}, secret, credentialOffer)
```

Note: The structure of the response will change as we add support for the aforementioned `requestedInput` field.

Transferring the credential to the user

The credential offer response is sent back to the `callbackURL` provided by the service. At this point, the service can generate the credentials and transfer them to the user. There are a few way to accomplish the last step, currently the service simply issues a `CredentialResponse` JWT containing the credentials. We intend to use Verifiable Presentations for this step once the specification matures.

For an example of a demo service which can undergo credential issuance flows (as well as other interaction flows supported by the Jolocom SmartWallet), refer to the Jolocom Demo interaction service.

Note: These interaction flows are also exposed through the Jolocom SDK, as described in the corresponding SDK documentation section