

---

# **Jolocom-Lib Documentation**

**Jolocom**

**Jun 28, 2019**



---

## Where to go next?

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The Jolocom library at a glance . . . . .	4
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	How to install the Jolocom library . . . . .	5
2.2	How to create a self-sovereign identity . . . . .	5
2.3	Using the identity . . . . .	7
2.4	What can I do now? . . . . .	7
<b>3</b>	<b>Credentials &amp; Signed Credentials</b>	<b>9</b>
3.1	Create a signed credential . . . . .	9
3.2	Validate a signature on a signed credential . . . . .	11
3.3	Working with custom credentials . . . . .	11
<b>4</b>	<b>Public Profile</b>	<b>15</b>
4.1	Adding a public profile . . . . .	15
4.2	Removing your public profile . . . . .	16
4.3	View the public profile . . . . .	16
<b>5</b>	<b>Credential-based Communication Flows</b>	<b>17</b>
5.1	Credential requests . . . . .	17
5.2	Credential issuance . . . . .	20
5.3	What next? . . . . .	22
<b>6</b>	<b>Single Sign On (SSO) with Jolocom</b>	<b>23</b>
6.1	Clone the Github repository . . . . .	23
6.2	Editing the service configuration file . . . . .	24
6.3	Authenticating against the local service . . . . .	24
<b>7</b>	<b>Using custom connectors</b>	<b>25</b>



## **Our Approach**

The Jolocom identity solution aims to be a universal, lightweight, open source protocol for decentralized digital identity and access right management. The protocol is built on to leading open source standards standards and relies on distributed / decentralized systems such as Ethereum and IPFS for identity registration and resolution.

The protocol architecture revolves around three main concepts:

- **Hierarchical Deterministic Key Derivation** , which enables pseudonymous, context-specific interactions through the creation of and control over multiple identities.
- **Decentralized Identifiers (DIDs)** , which are associated with each identity and used during most interaction flows, such as authentication or data exchange.
- **Verifiable Credentials** , which are digitally-signed attestations issued by an identity. The specification can be used to develop a simple way of associating attribute information with identifiers.

Cryptographic keys and DIDs enable the existence of a self-sovereign identity. Keys and verifiable credentials provide the tools required to create complex data while simultaneously preserving simplicity at the core.

This approach allows us to keep the protocol generic while facilitating an unlimited number of specific use cases with varying levels of complexity.

A further component of the protocol architecture calls for the integration of a public, censorship-resistant, decentralized network for anchoring and resolving user identifiers. For this we currently use IPFS for storage and Ethereum for anchoring and indexing identifiers.

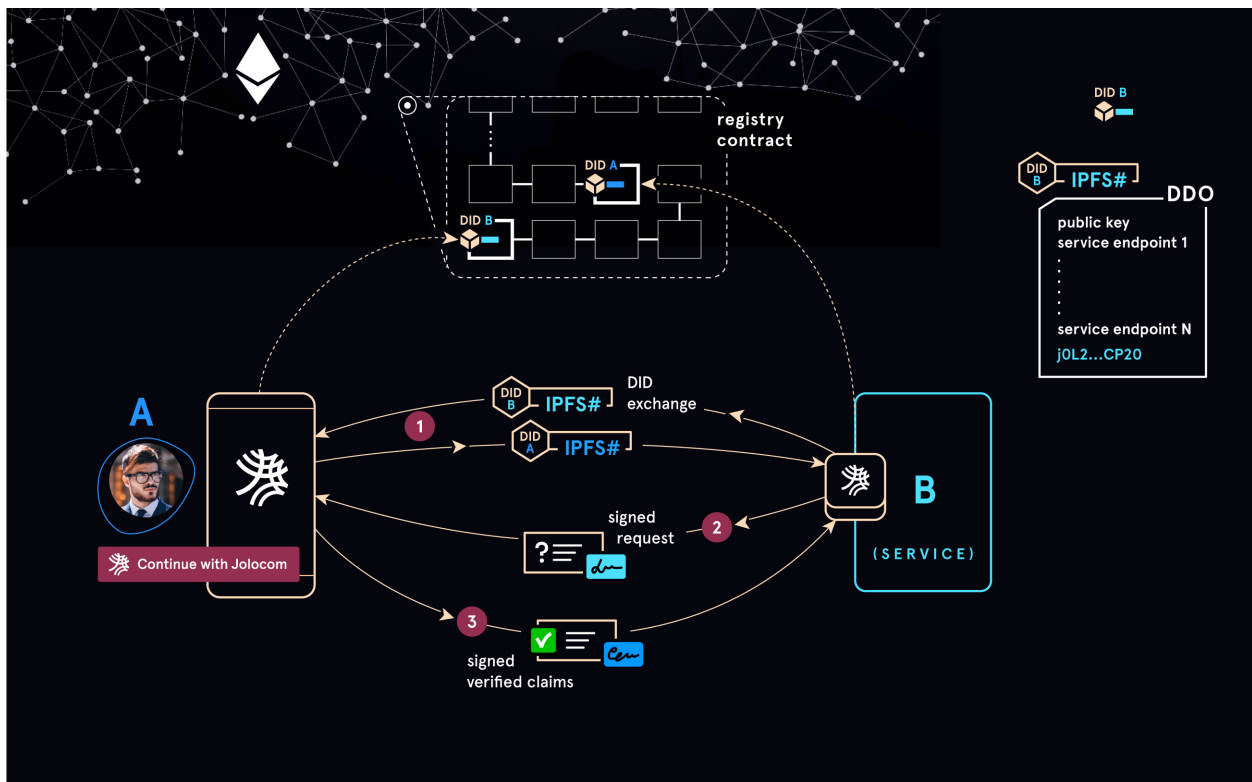
In its most simplistic form, the Jolocom protocol can be used to:

- Create a self-sovereign identity for use by humans, organisations, and smart agents;
- Attach meaningful information to identities in the form of verifiable credentials;
- Easily request and consume verified information about identities in an automated fashion.

We hope it will serve efforts to decentralize the web and digital identity, and enable people, organisations, and smart agents to own and control the data that defines them.



This section provides a very brief overview of the protocol architecture, introduces the concept of self-sovereign identity, and explains how to navigate the Jolocom library.



### System Architecture

The Jolocom protocol is built using the following core specifications:

- [Hierarchical Deterministic Key Derivation](#) , which enables pseudonymous, context-specific interactions through

the creation of and control over multiple identities.

- **Decentralized Identifiers (DIDs)** , which are associated with each identity and used during most interaction flows, such as authentication or data exchange.
- **Verifiable Credentials** , which are digitally-signed attestations issued by an identity. The specification can be used to develop a simple way of associating attribute information with identifiers.

A further component of the protocol architecture calls for the integration of a public, censorship-resistant, decentralized network for anchoring and resolving user identifiers. For this we currently use IPFS for storage and Ethereum for anchoring and indexing identifiers.

**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 The Jolocom library at a glance

The Jolocom library exposes an interface with four main entry points:

- `identityManager` - used for generating, deriving, and managing multiple keys;
- `parse` - used for instantiating various classes offered by the Jolocom library from encoded / serialized forms;
- `registry` - used for adding, retrieving, and modifying data persisted on IPFS and indexed on Ethereum;
- `unsigned` - used for generating unsigned versions of various data structures offered by the Jolocom library.

The next section examines the API in more detail.



### 2.1 How to install the Jolocom library

To begin using the Jolocom protocol, first install the Jolocom library 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
```

**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.

### 2.2 How to create a self-sovereign identity

In broad strokes, the creation of a self-sovereign identity comprises the following steps:

- Instantiate the `SoftwareKeyProvider` class by providing a 32 byte random seed `Buffer`, and a password for encryption
- Use the `keyProvider` to derive two keys, one to control your Jolocom identity, the other to sign Ethereum transactions
- Transfer a small amount of ether to your second key to later pay for updating the registry contract
- Instantiate and use a `JolocomRegistry` to anchor the newly created DID document on the Ethereum network

The following sections present these steps in greater detail.

### Instantiate the Key Provider class

The `SoftwareKeyProvider` class abstracts all functionality related to [deriving key pairs](#) and creating / validating cryptographic signatures. The 32 byte seed used to instantiate the key provider is persisted in the instance, encrypted using the provided password. Therefore, all operations which involve key derivation require the password as well. This is what instantiating a key provider looks like:

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

// Feel free to use a better rng module
const seed = crypto.randomBytes(32)
const password = 'correct horse battery staple'

const vaultedKeyProvider = new JolocomLib.KeyProvider(seed, password)
```

---

**Note:** In the next release, the constructor will be modified to require only the encrypted seed value, thereby reducing the amount of time during which the seed is exposed.

---

### Derive a key to sign the Ethereum transaction

The `vaultedKeyProvider` just instantiated can be used to derive further key pairs necessary to complete the registration. We need to derive a key for signing the Ethereum transaction, which anchors the newly created identity.

```
const publicEthKey = vaultedKeyProvider.getPublicKey({
  encryptionPass: secret
  derivationPath: JolocomLib.KeyTypes.ethereumKey // "m/44'/60'/0'/0/0"
})
```

#### See also:

In the event that one of your keys becomes compromised, you only lose that one key. All other derived keys (including the most important master key) remain secure. Go to [BIP-32](#) if you want to find out more about this derivation scheme. We are currently looking at key recovery solutions in case the master key itself is compromised.

The only arguments that need to be passed to `getPublicKey` are the `derivationPath`, in the format defined in [BIP-32](#), and the `encryptionPass` that was used to create the encryption cipher. The Jolocom library comes equipped with a few predefined paths for generating specific key pairs. The list will expand as new use cases are explored. You can view the available paths as follows:

```
console.log(JolocomLib.KeyTypes)
```

The next step involves transferring a small amount of ether to the Rinkeby address corresponding to the created key pair.

### Transferring ether to the key

In order to anchor the identity on the Ethereum network, a transaction must be assembled and broadcasted. In order to pay for the assembly and broadcasting, a small amount of ether needs to be present on the signing key. There are a few ways to receive ether on the Rinkeby test network, and the library also expose a helper function to assist:

```
await JolocomLib.util.fuelKeyWithEther(publicEthKey)
```

This will send a request to a [fueling service](#) Jolocom is currently hosting.

### Anchoring the identity

The final step to creating a self-sovereign identity is anchoring the identity on Ethereum and storing the newly created DID document on IPFS. For these purposes, the `JolocomRegistry` can be used; it is essentially an implementation of a [DID resolver](#). The creation would look as follows:

```
const registry = JolocomLib.registries.jolocom.create()
await registry.create(vaultedKeyProvider, secret)
```

Behind the scenes, two key pairs are derived from the seed. The first key is used to derive the DID and create a corresponding DID document. The second key is used to sign the Ethereum transaction, adding the new DID to the registry smart contract.

---

**Note:** We intend to add support for [executable signed messages](#) in the next major release, thereby eliminating the need to derive two key pairs.

---

## 2.3 Using the identity

The `create` function presented in the previous section eventually returns an instance of the `IdentityWallet` class, which can be used to authenticate against services, issue credentials, and request data from other identities. Later sections will explore the exposed interface in more detail.

In case you have already created your identity, and would like to instantiate an `IdentityWallet`, you can simply run:

```
/**
 * You will need to instantiate a Key Provider using the seed used for identity_
 * → creation
 * We are currently working on simplifying, and optimising this part of the api
 */

const registry = JolocomLib.registries.jolocom.create()
const IdentityWallet = await registry.authenticate(vaultedKeyProvider, {
  derivationPath: JolocomLib.KeyTypes.jolocomIdentityKey,
  encryptionPass: secret
})
```

## 2.4 What can I do now?

So far, you have successfully created and anchored a digital self-sovereign identity. The subsequent sections cover how to:

- create a public profile and make it available through your DID document;
- issue statements about yourself and others in form of signed [verifiable credentials](#);
- authenticate against other identities, share and receive signed verifiable credentials, and create various interaction tokens;
- use custom connectors for IPFS and Ethereum communication.



---

## Credentials & Signed Credentials

---

The Jolocom library contains a number of functions and classes that enable the creation and consumption of signed [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.

### 3.1 Create a signed credential

The easiest way to create a signed credential is by using an instance of the `IdentityWallet` class. If you have yet not created an identity, check out the [Getting Started](#) section. If you have already created an identity, you can obtain an identity wallet by authenticating, as defined in [section 2.2](#).

```
import { JolocomLib } from 'jolocom-lib'
import { claimsMetadata } from 'cred-types-jolocom-core'

const password = 'correct horse battery staple'

const emailAddressSignedCredential = await identityWallet.create.signedCredential({
  metadata: claimsMetadata.emailAddress,
  claim: { email: 'example@example.com' }
  subject: identityWallet.did // Our own DID, referred to as 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

{
  @context: [
```

(continues on next page)

(continued from previous page)

```

{
  id: '@id',
  type: '@type',
  cred: 'https://w3id.org/credentials#',
  ...
  ProofOfEmailCredential: 'https://identity.jolocom.com/terms/
↪ProofOfEmailCredential',
  schema: 'http://schema.org/',
  email: 'schema:email'
}
],
id: 'claimId:d9f45722872b7',
name: 'Email address',
issuer: 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777',
issued: '2018-11-16T22:21:28.862Z',
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:** All credential types the library supports by default are made available through the `cred-types-jolocom-core` npm package. Alternatively, you can check out the [GitHub repository](#).

It's worth noting that in the aforementioned credential, the issuer, the subject, and the signature creator each share 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:

```

const emailAddressSignedCredential = identityWallet.create.signedCredential({
  metadata: claimsMetadata.emailAddress,
  claim: { email: 'example@example.com' },
  subject: 'did:jolo:6d6f636b207375626a656374206469646d6f636b207375626a65637420646964'
}, password)

```

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 ommited.

{
  '@context': [ ... ],

```

(continues on next page)

(continued from previous page)

```

...
issuer: 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777',
claim: {
  email: 'example@example.com',
  id: 'did:jolo:6d6f636b207375626a656374206469646d6f636b207375626a65637420646964'
},
proof: EcdsaLinkedDataSignature {
  ...
  creator:
↪ 'did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777#keys-1'
  ...
}

```

## 3.2 Validate a signature on a signed credential

Perhaps you would like to present the newly created signed credential to a service or some other entity with a Jolocom identity as part of an interaction. The (intended) recipient needs to be able to verify that the credential received is valid. Validating a received credential proceeds 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 by using the listed public keys to validate the credential signature.

If you already know the public key corresponding to the signing party, it is not necessary to resolve the DID document:

```

import { JolocomLib } from 'jolocom-lib'

const receivedSignedCredential = JolocomLib.parse.signedCredential.fromJSON(received)
const issuerPublicKey = Buffer.from(
↪ '030d4792f4165a0a78f7c7d14c42f6f98decfa23d36e8378c30e4291711b31961f', 'hex')

/**
 * Please note that this will NOT fail if the signer has marked the public key as
↪ compromised or invalid;
 * the signature is simply being verified, without checking against any external
↪ resources.
 */

console.log(await JolocomLib.keyProvider.verifyDigestable(issuerPublicKey,
↪ signedCred)) // true

```

## 3.3 Working with custom credentials

Users are free to define custom credential types. The number of types of interactions would be quite restricted if only types defined by Jolocom 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 authentic. For these purposes, none of the credential types we currently provide suffice.

Or 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 a verification, 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 `cred-types-jolocom-core` 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, remove this line.

### Creating and verifying custom credentials

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

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

(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)
```



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.

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

Add the newly created public profile to your identity:

```
/**
 * Typescript accessors are used to get
 * and set values on the identityWallet instance
 * @see https://www.typescriptlang.org/docs/handbook/classes.html
```

(continues on next page)

(continued from previous page)

```
*/  
identityWallet.identity.publicProfile = publicProfileCred
```

---

**Note:** `Typescript` accessors are used to get and set values on the `identityWallet` instance

---

So far you have been making changes to your identity only locally. Now, you can commit the changes to IPFS and Ethereum.

```
await registry.commit({  
  identityWallet,  
  vaultedKeyProvider,  
  keyMetadata: {  
    encryptionPass: secret,  
    derivationPath: JolocomLib.KeyTypes.ethereumKey  
  }  
})
```

In order to update your public profile, simply create a new credential, add it to your `identityWallet`, and commit the changes.

## 4.2 Removing your public profile

```
identityWallet.identity.publicProfile = undefined  
  
await registry.commit({  
  identityWallet,  
  vaultedKeyProvider,  
  keyMetadata: {  
    encryptionPass: secret,  
    derivationPath: JolocomLib.KeyTypes.ethereumKey  
  }  
})
```

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.

## 4.3 View the public profile

Viewing the public profile associated with an identity is easy:

```
console.log(identityWallet.identity.publicProfile)
```

An instance of the `SignedCredential` class is returned.

---

## 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.

---

### 5.1 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)
```

---

**Note:** For further documentation and examples explaining how to create and send credential requests, check the [API documentation](#), the [generic backend documentation](#), the [integration tests](#), and finally [this collection of simple snippets](#).

---

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, and prevent some replay attacks.

In order to make the credential request consumable by the [Jolocom SmartWallet](#) the [JSON Web Token](#) must further be encoded as a QR code that can be scanned by the wallet application. The `credentialRequest` can be encoded as follows:

```
// Will be deprecated in future releases in favor of more user-friendly and intuitive_
↪ways to encode data

const jwtEncoded = credentialRequest.encode()
const QREncoded = new SSO().JWTtoQR(jwtEncoded)
```

---

**Note:** The JWT encoded interaction token can also be sent to the [Jolocom SmartWallet](#) via “Deep Links”. An example client application making use of this flow can be [here](#).

---

### 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 credential 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,
 * will be populated automatically based on the request starting from next major_
 ↪release
```

(continues on next page)

(continued from previous page)

```

*/
const credentialResponse = await identityWallet.create.interactionTokens.response.
  ↪share({
    callbackURL: credentialRequest.payload.interactionToken.callbackURL,
    suppliedCredentials: [signedEmailCredential.toJSON()] // Provide signed_
  ↪credentials of requested type
  },
  encryptionPass, // 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 to filter 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 encoded as a JSON Web Token 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, 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.
 */

const validResponse = credentialResponse.satisfiesRequest(credentialRequest)
const registry = JolocomLib.registries.jolocom.create()

```

(continues on next page)

(continued from previous page)

```
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
}
```

## 5.2 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 pinged by the client device with a 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*jbb5WdcAvaYluVdCjX1XVg.png'
    },
    background: {
      url: 'https://i.imgur.com/OMrldei.png',
    },
    text: {
      color: '#05050d'
    }
  }
}
```

(continues on next page)



(continued from previous page)

```

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 standardisation efforts. An example of such standardisation initiatives is the [Credential Manifest proposal](#).

The `renderInfo` is used to describe how a credential should be rendered and is currently supported by the Jolocom Smartwallet. It allows for a variety of graphical descriptors in it's format:

```

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 credential 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'
    },
  ],
}

```

(continues on next page)

(continued from previous page)

```
{
  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.

An example implementation for an issuance service can be found [here](#).

## 5.3 What next?

Some additional bits of high level documentation related to the supported interaction flows are available [here](#)

With the reasoning behind the credential request and response flows unpacked, it's time to put it all to the test! Head to the next section to learn how to set up your own service for interacting with Jolocom identities.

---

## Single Sign On (SSO) with Jolocom

---

The best way to get some hands on experience with the Jolocom library and identity protocol is to try it out for yourself! In this section we examine how to deploy a demo service capable of interacting with Jolocom identities.

### 6.1 Clone the Github repository

To begin, clone the Jolocom demo sso repository and install all dependencies:

```
# clone the repository and navigate to the new folder
git clone https://github.com/jolocom/demo-sso.git; cd ./demo-sso

# install all dependencies
yarn install
# or
npm install
```

---

**Note:** In order to ensure that the application works correctly, you will also need [redis](#) installed on your local machine. The demo application makes use of the `redis-server` and `redis-cli` commands to launch and to connect to a local database upon start.

---

To ensure no errors occurred during installation, we can attempt to start the service:

```
# Ensure you are in the 'demo-sso' folder
yarn start
```

If everything is setup properly, you should see the following printed message after a few seconds: *'Demo service started, listening on port 9000'*.

If you don't see the message or encounter an unexpected error, get in touch with the Jolocom team for troubleshooting support.

Once you see the message, you're ready to move on.

## 6.2 Editing the service configuration file

If we open the `config.ts` - file located in the project root directory, we will notice that there are three options we can configure:

- `seed` - A 32 byte `Buffer` of random bytes, used for deriving key pairs, as explained in the the [Getting Started](#) section.
- `password` - The password used to encrypt the `seed` on the instance. Must be provided for any operations involving key derivation.
- `serviceUrl` - The url that can be used reach the deployed service, if you are testing locally, the default value should suffice.
- `credentialRequirements` - The types of credentials required by the service. By default the service requires a `ProofOfNameCredential`, with no associated constraints.

---

**Note:** Additional documentation on the `credentialRequirements` section will be added soon.

---

After the fields have been configured, the service can be started by running `yarn start`.

## 6.3 Authenticating against the local service

With the local service running, open a browser and navigate to `http://localhost:9000/`. Tapping the button “Continue with Jolocom” on the landing page that loads will generate a credential request (as defined [here](#)), encode it as a QR code, and display the resulting image.

At this point the presented request can be scanned using the Jolocom SmartWallet to generate the corresponding credential response for sharing with the service.

The `demo-ss0` repository also includes a script that can be run to simulate the client completing the authentication for testing and development purposes. Further documentation on the authentication test script can be found [here](#).

---

## Using custom connectors

---

By default, all identities created using the Jolocom library are indexed in a contract deployed on the Rinkeby test network, and the corresponding DID documents are stored on IPFS.

The interaction with the corresponding networks is delegated to two components:

- IPFS connector
- Ethereum connector

Using custom connectors (e.g. to experiment on a private network) is also supported.

You can also supply your custom implementations of both connectors, in case your identities are indexed on a private Ethereum deployment, or you would like to connect to a custom IPFS cluster. A custom implementation might look as follows:

```
class CustomEthereumConnector implements IEthereumConnector {
  async resolveDID(did: string) {
    console.log(`Intercepted request for ${did}`)
    return fetchFromCacheIfAvailable(did)
  }

  async updateDIDRecord(args: IEthereumResolverUpdateDIDArgs) {
    console.log(`Intercepted request for ${args.did}, updating to ${args.newHash}`)
    return queueUpdateRequest(args)
  }
}

class CustomIpfsConnector implements IipfsConnector {
  async storeJSON({ data, pin }: { data: object; pin: boolean }) {
    ...
  }

  async catJSON(hash: string) {
    ...
  }
}
```

(continues on next page)

(continued from previous page)

```
    async removePinnedHash(hash: string) {
      ...
    }
  }

const customRegistry = JolocomLib.registries.jolocom.create({
  ethereumConnector: new CustomEthereumConnector(),
  ipfsConnector: new CustomIpfsConnector()
})
```

---

**Note:** Using only one custom connector is also supported.

In the event a connector is not provided when instantiating the `registry`, the default implementation provided by Jolocom will be used.

---

In some cases, it might make sense to define connectors that rely fully on databases maintained in a centralised manner. The current library API supports this use case as well.