# Integration Report [OSiP]

## Introduction

Jolocom, alongside Spherity and Accenture, took part in an interoperability focused proof of concept, realized over the period June - August 2019. The scope of the project (defined in more detail in the next sections) was to achieve interoperability across a multi step (i.e. involving credential issuance and exchange) use case.

This informal report attempts to provide a general outline of the project goals, and mostly focuses the implementation efforts identified and undertaken by the participants.
The next section will provide a brief description of the implemented user flow, with later sections focusing on some of the main interoperability related challenges (both foreseen and unforeseen) we've identified and attempted to solve. We hope to capture some of the learnings we've derived from this project.

### Use case scope and description

The use case we agreed to develop around is as follows:
The user (in this case a journalist) can use a wallet application on their phone to receive a verifiable credential (from a trusted issuer service), which can then be presented to a verifier in exchange for a service (in this case admission to an event).

**Assumptions:**

We assume that the Journalist already has an identity wallet on their device (it should not matter if they are using the Spherity or Jolocom wallet). We also assume that before approaching any of the OSIP issuance or verification services (developed and used as part of this project), the Journalist already holds an identity card verifiable credential in their wallet. To satisfy the latter assumption both parties set up simple demo services to provision the wallet with an ID Card verifiable credential.

**Step 1. Credential issuance:**

In the first step of the user story, the journalist visits the OSIP verification platform to apply for a verifiable credential (referred to as the OSIP credential), which they can use to gain free entrance to various event venues with restricted access.
The application process starts with the journalist visiting the OSIP issuer web page and filling out a simple web form, providing their first and last names, alongside their ID card number. For the next step, the journalist can scan a QR code to share their ID card credential using their wallet application. It should be mentioned that sharing the ID card credential can take place either as part of the application process, or at a later point in time, after the application has been submitted. Once the journalist shares their ID card credential, the id card number is matched against the previously submitted forms. If a match is found, the application process is marked as completed.

It should be noted that the web form is not strictly necessary, but it enables the use case where someone can start the application process on behalf of the journalist, at which point the journalist would only need to scan the QR code to provide the lacking ID Card credential.

All submitted applications can be manually reviewed by an employee of the OSIP verification platform, and in case no issues were identified, the application is marked as processed, and the OSIP Credential is issued. At this point the journalist can visit the OSIP verification platform for the last time, to scan a QR code and receive the verifiable credential to their wallet.

**Step 2. Credential consumption:**

As a result of the previous step, the journalist's wallet should hold an OSIP credential, issued by the OSIP verification service, which can be shared with the verifier service to gain access to an event with restricted access. In the context of the demo user flow, the final service was also realized as a web application, with a simple UI rendering a QR code triggering a credential request flow on the user's wallet. Once the user presents a valid OSIP credential, the UI of the verification service updates to reflect it. At this point we consider the flow completed, with the journalist "gaining access to the venue".

*Remark* - The user can trigger the aforementioned interactions either by scanning a QR code using their wallet, or, alternatively, by visiting the issuer / verifier page from their mobile phone browser and tapping on a button (which

embeds a deep link containing the same payload encoded in the QR code). For both the OSIP issuer and verifier services, a button containing a deep link is always available as an alternative to QR codes, to allow for mobile only user flows. This was not explicitly described in the previous two subsections to avoid repetition.

## Areas of interoperability

We tried to identify the broad areas of the stack across which interoperability would need to be achieved. The following general areas of interest have been identified:

- DID methods - each participating party relied on a specific DID method, integrated with the corresponding wallet application and libraries
- Verifiable Credentials - each participating party had support for issuing and presenting signed claims, potentially using different data models, or following different specifications
- Interaction protocols - each participating party used a specific protocol(s) for requesting and issuing verifiable credentials

In order to ensure interoperability given the previously outlined use case, we decided to focus on the 3 aforementioned general areas of the stack. The following subsections provide more context on the individual areas, and the associated interop related challenges / work.

### DID Methods

Prior to starting the project, both Jolocom and Spherity were already using DID method implementations, integrated with the respective wallet applications / libraries. It's worth noting that all OSIP services (i.e. the issuer and the verifier) were developed as part of the project, and therefore had no pre-existing dependencies on any specific DID method. The table below lists the DID methods we ended up using in the context of the project:

#### DID Methods used by the different services / actors

| prefix | used by | did method |
|---|---|---|
| did:jolo | Jolocom SmartWallet | Jolocom |
| did:spherity | Spherity Wallet | ETHR DID |
| did:7hirten | OSIP Verifier and Issuance services | ETHR DID |

*Worth noting that although* `did:spherity` *and* `did:7hirten` *both use* `did:ethr`*, different prefixes were needed to disambiguate between different deployments (i.e. public Ethereum testnet vs Private quorum deployment).*

Given the DID methods listed above, we had to ensure that both wallet applications, and the developed services (OSIP issuer and verifier) could seamlessly de-reference and interact with identities anchored on different networks. Taking in consideration the project scope and duration, we decided to delegate DID resolution of identities to a deployed Universal Resolver instance. To elaborate, the OSIP issuer and verifier services would use the universal resolver for all resolution purposes, regardless of the DID of the counterparty, while the wallet applications would delegate resolution of "foreign" (i.e., in the Jolocom case, `did:7hirten` and `did:ethr`) DIDs to the deployed universal resolver instance.
It's also worth noting that as part of the project a modified version of the universal resolver was used, since some of the DID method prefixes listed before (e.g. `did:7hirten`) were not supported by the instance deployed at *uniresolver.io*.

Furthermore, to simplify the interop efforts, we decided to aim for integrating version `0.13` (at that time the latest published version) of the Decentralized Identifiers specification. At this point we have encountered a few unforeseen complications. One problematic point was the spec mandated values for the `@context` field in DID Documents (specifically the inclusion of the `https://www.w3.org/2019/did/v1` entry). During normalization the aforementioned context entry would cause an error to be thrown (because it does not de-reference to a valid JSON-LD context), which would

then lead to a crash (i.e. the universal resolver would not be able to correctly de-reference DID Documents with the context entry present). We had to deviate from the spec in this case and omit the faulty link.

**This issue seems to have been addressed in the latest version of the Decentralized Identifiers specification, and is probably no longer relevant.*

Furthermore, in some cases, the DID Document returned by the universal resolver would include a prefix in the case of some keys, i.e. `sec:publicKeyHex` instead of `publicKeyHex` or `sec:ethereumAddress` (`sec` being an alias defined in the `@context` section, which maps to, for instance, *https://w3id.org/security#*). We could manage to work around this by providing specifically crafted `@context` fields, or by removing all irregularities client (e.g. wallet / OSIP backend) side.

After resolution, the returned DID Documents still needed to be verified / validated. One thing worth mentioning here is that it wasn't always possible to validate the returned DID Document, specifically because not all DID Documents contained a `proof` section. When using ERC 1056, the correctness of the final DID Document is ensured by the correctness of the previously broadcasted transactions. In some cases validation is possible given that the identifier / address is self certifying, but once key rotation operations took place we can't be certain of the correctness of the result.

A similar problem exists with Jolocom DID Documents, which do include a `proof` section containing a signature generated by one of the listed keys. Although the signature itself can be verified, once a key rotation operation has taken place and the DID is no longer self-certifying (i.e. a hash of the public key), it's difficult to be sure that the received DID Document is valid without checking all broadcasted key rotation operations (which are currently not recorded or available). Solving this issue in a general manner was deemed out of scope for the project.

### Verifiable Credentials

In order to make sure that the wallets can seamlessly interact with the various OSIP services, we had to make sure we agree on a format for representing signed credentials. Jolocom already had support for JSON-LD verifiable credentials implemented before the start of the project (as part of Jolocom-Lib). After some initial discussions and evaluation, the project partners agreed on supporting the latest version of the Verifiable Credentials specification (an extract from the corresponding Gitlab issue is included as Appendix A).

Another action point included defining the contents of the credentials themselves. This mostly includes agreeing on the credential type, such as `['VerifiableCredential', 'OsipVerifiableCredential']` (which can then be referenced during requests for credentials, as seen in the later examples), defining the contents of the `credentialSubject` (`claim`) section, and including corresponding RDF predicates for the new keys (e.g. *given name* → *https://schema.org/givenName*, etc.) in `@context` entries to ensure no terms are skipped during normalization / signature generation.

One issue encountered along the way was related to validating the signatures included in the `proof` section of presented Verifiable Credentials or DID Documents. The main issue is that different signature verification processes had to be employed depending on the signer (or rather the used DID method).

In the case of Jolocom identities and verifiable credentials, verification implies:

1. De-referencing the DID Document of the signer
2. Extracting the signing key (i.e. `publicKeyHex`)
3. Using the key to verify the signature included in the credential (e.g. using tiny-secp256k1)

In the case of Spherity / OSIP (*did:ethr*) identities and verifiable credentials verification implies:

1. De-referencing the DID Document of the signer
2. Extracting the controlling Ethereum address (i.e. `ethereumAddress`)
3. Recovering the signer's key from the signature (e.g. using elliptic.js)
4. Converting the recovered public key to an Ethereum address and comparing it with the one extracted in step 2

The main complication arose from the fact that regardless of the process employed when generating the signature, the "closest" standardized `proof type` we could use was `EcdsaKoblitzSignature2016`, better options might, of course, be available.

Within the time frame / scope of the project, we were unable to find or define a `proof type` that signaled the usage of the "recover" flow. Given the lack of the aforementioned signal, the verification process was selected based on the DID method of the issuer (which isn't exactly future proof), or based on the length of the signature value (which is far from ideal as well).

### Interaction protocols:

Once we agreed on the specifications to use for modelling identities and credentials, protocols for interaction (i.e. issuing and requesting credentials) had to be agreed upon. We evaluated a few options (namely the Credential Manifest, the W3C CCG work items, the activity and outputs of the DIF Claims / Credentials working group), but generally could not find any comprehensive specifications we could easily agree upon and implement (with the exception of Verifiable Presentations, which we agreed to support as part of the project).

Jolocom already had some simple data structures defined (examples provided later, with general description available here) for modelling basic requests for credentials and credential offers. As part of this project, our intention was to research and possibly adapt some more robust / standardized alternatives, but given the timelines and the required effort / research, this was deemed out of scope.

After researching some alternatives, and discussing Jolocom's current approach, the participants concluded that the simplest and most efficient way to move forward would be to use an improved (where necessary) version of the interaction data structures already used by Jolocom, given that they would be able to satisfy the use case requirements.  We ended up making the following decisions in regards to the data structures to use:

**Exchanging credentials** (e.g. the OSIP verification platform requests an ID Card credential from the user's wallet, and the user shares it):

- **Request** - We used the same format as the one described in this document. No modifications were made, because the data structure could already satisfy the requirements posed by the use case. Examples of the request format, plus some additional context on the decision making process is outlined in Appendix C.
- **Response** - At the start of the project, we were using a minimal data structure (mentioned here, examples included as part of Appendix B) for communicating / sharing credentials with other identities. As part of the project, we decided to adopt an alternative, better documented and more standardized . The most compatible and comprehensively documented alternative we could find were Verifiable Presentations. Some additional context on the decision making process is outlined in Appendix B.

**Issuing credentials** (e.g. the OSIP verification platform advertises that it offers a credential of a specific type, and the user requests / receives it)

- **Offer Request and Response** - Simillarly to the credential request flow, we agreed to reuse the data structures previously developed by Jolocom to model credential offers, and the corresponding offer responses. Additional context, and request / response examples are presented in Appendix D.
- **Communicating the issued credentials** - We decided to use Verifiable Presentations for this step of the process as well. Comments from the previous section (*Exchanging credentials/Response*) apply.

To sum the previous comments up, the final decisions were:

- For requesting credentials - the project partners added support for the data structure already used by Jolocom.
- For sharing credentials - the project partners, including Jolocom, added support for JWT encoded Verifiable Presentations.
- For representing credential offers (i.e. as part of issuance) - the project partners added support for the data structures used by Jolocom.
- For communicating issued credentials (shared by the issuing service) - the project partners, including Jolocom, added support for JWT encoded Verifiable Presentations.

*Note on encoding*: As shown in the examples included in Appendix B, when encoding Verifiable Presentations, we've opted for the JWT encoded approach, which might seem odd, given that we use JSON-LD Verifiable Credentials. Some of the reasoning is outlined in Appendix B.

Besides the listed arguments, encoding Verifiable Presentations as JWTs resulted in a simpler implementation, because all other interaction data structures the project partners supported were already encoded as JWTs, which allowed us to reuse the validation / interaction handling code, without having to treat verifiable presentations as a special case scenario (i.e. in the context of normalization / signature validation). We identified no strong reason / necessity to couple the encoding format of the Verifiable Credentials to the encoding format of the Verifiable Presentation.

To broadcast interaction requests, the appropriate data structures (e.g. credential request / credential offer) would be Base64Url encoded and communicated to the wallet either via a QR code, or via deep linking. Both cases would use the same encoded payload. Once the wallet has parsed and validated the request, the corresponding response would be assembled and communicated to the requester via HTTPS (a POST request to the `callbackURL` included in the signed interaction request), or via deep linking (also using the value of the `callbackURL` key).
In order to achieve interoperability on the deep linking layer, we had to agree on a common scheme / prefix. For demo purposes we decided to use the `ssi://` prefix, which both wallet implementations would be able to handle. It should be mentioned that none of the parties intend to continue supporting the `ssi://` prefix outside of the completed project, to not complicate or impede any larger standardization efforts.

### Open TODOs:

- [ ] Add diagrams (reconsider, is there anything particular useful we can illustrate here?)
- [ ] Format appendices (make sure they read well)

---

**Appendix A:**

This is an excerpt of the Gitlab issue related to selecting the format for communicating credentials to the requester.

***Issue body:***

> In order to make sure the credential can be exchanged across implementations we need to ensure the data structures are well defined (e.g. as per specific version of the Verifiable Credential spec) and supported across both implementations.
> Some initial ideas on how to proceed (please feel free to edit / contribute)
> - Evaluate current implementations from both parties
> - Find the path of least resistance to enable interoperability (e.g. both parties agree on one spec version to support, both parties support each other's version in addition to their own, etc...)
> Contributions and suggestions from all parties are encouraged. A similar approach for DID documents can be taken as well. For now this is a general overview before we can produce more specific and actionable issues. Members of the @jolocom team will add some initial contributions soon.

***Later comment:***

> We (Jolocom) would currently propose we use Verifiable Credentials to model attestations. We currently support a slightly outdated version of the spec, and would, as part of this project, aim to update the implementation so we can support the latest version.
>
> The following test suite was made available by the W3C, and can be used to ensure that our implementations are spec compliant. (we can aim to pass the `basic` and maybe even `advanced` test suites).
> We can even choose to add the produced reports to the published implementation report.

> *Please note that this issue is only concerned with the format of the credentials themselves, and the format of credential requests / responses (e.g. verifiable presentations) will be tackled separately.*

---

**Appendix B:**

This is an excerpt of the Gitlab issue related to selecting the format for communicating credentials to the requester.

> **Issue body:**
>
> Once we agree on the format of the verifiable credentials (issue #2), we need to agree on the data formats and protocols for exchanging them between identities (i.e. credential requests / offers / verifiable presentations). In the comments we can briefly outline the approaches / data structures that the individual partners currently take, and agree on an optimal way forward.
>
> Some initial ideas / tentative data structure.
> One part of the flow we can already start working on is presenting verifiable credentials to another identity (which could be used both during credential based authentication, and when receiving credentials from an issuance service). We would need to agree on a data structure that includes the credentials, and on a serialization format.
>
> The W3C spec addresses this need using Verifiable Presentations. Multiple presentation types are conceivable (e.g. basic presentation including the credentials directly, zero knowledge proof presentation including data derived from verifiable credentials to satisfy some specific predicates, selective disclosure presentation including a Merkle Tree based structure with some attributes disclosed and some hashed, etc.)
>
> The simplest verifiable presentation type that satisfies our needs would look something like this (example 13 from the verifiable credential spec):

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "urn:uuid:3978344f-8596-4c3a-a978-8fcaba3903c5",
  "type": ["VerifiablePresentation", "CredentialManagerPresentation"],
  "verifiableCredential": [{ ... }] // W3C Verifiable Credentials,
  "proof": [{ ... }] // Linked data signature, format as the one used on the Verifiable Credentials.
}
```

> Unfortunately, the specification does not mandate one specific way of serializing the verifiable presentation when sending it to another party, as described in section 6, at least 3 formats are possible:
>
> JSON + JWT
> JSON-LD + JWT
> JSON-LD + Linked Data Proofs

This document attempts to contrast the three approaches.

It is worth noting that these different serialization formats can be used for both Verifiable Credentials, and Verifiable Presentations (as shown in the various examples from the aforementioned section of the spec). This means that if we choose to use the JSON-LD + LD Proofs format (as suggested in #2 (closed)), we should try to use a similar / complementary approach for verifiable presentations.

...

**Comment 1:**

The verifiable presentation shown as an example in the issue body is a JSON-LD verifiable presentation + a linked data proof. The outlined verifiable presentation could be sent as is to the other party, as the spec mentions in 6.3.2:
Unlike the use of JSON Web Token, no extra pre- or post-processing is necessary.
But in reality we would most likely have to encode them if we pass them through deep links and such.

Some advantages of this approach are:

- Logic and libraries can be reused across processing verifiable credentials and verifiable presentations (since they are both instances of linked data graphs consisting of a body + proof).
- Linked Data Proofs allow for easier modelling of signature sets, signature chains.
- Linked Data Proofs can be extended to add support for different proof suites.

Some disadvantages of this approach are:
- Not as popular, JWT / JWS / JWE based approaches have seen a lot more adoption for encoding and securing credentials in transit.

- Given that verifiable presentations might be passed as URL query parameters / deep links, they would have to be encoded as base64, or some other url safe format, which is not mandated by the spec, and might have interop consequences.

- **No clear spec for how to encrypt the verifiable presentation** (using either symmetric or asymmetric crypto), Linked Data Proofs only provide support for signatures.

- Linked Data Proofs are more verbose compared to JWS, and a lot of the benefits (e.g. signature chains / sets) are not too useful in the case of verifiable presentations (since we assume the presentations would contain one signature, from by the holder).

*Outline of Linked Data Proofs available here*

**Comment 2:**

**JSON-LD + JWT**
Similar to the approach outlined above, but instead of using a simple JSON-LD document containing a proof section, a JWT is used. An example as given by the spec (example 30):

```
{
  "iss": "did:example:ebfeb1f712ebc6f1c276e12ec21",
  "jti": "urn:uuid:3978344f-8596-4c3a-a978-8fcaba3903c5",
  "aud": "did:example:4a57546973436f6f6c4a4a57573",
  "iat": 1541493724,
  "exp": 1573029723,
  "nonce": "343s$FSFDa-",
  "vp": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "type": ["VerifiablePresentation", "CredentialManagerPresentation"],
    // base64url-encoded JWT as string
    "verifiableCredential": ["..."]
  }
}
```

The Verifiable presentation is represented as a linked data document, associated with the `vp` key of the JWT.

Keys one would normally find in the JSON-LD document are replaced (where possible) with JWT keys, e.g. iss (issuer), jti (id), exp (expirationDate), iss (issuer). The full list of claim names that MUST be used is listed in the specification.

The header section of the JWT would look as follows:

```
{
    "alg": "RS256",
    "typ": "JWT",
    "kid": "did:example:abfe13f712120431c276e12ecab#keys-1"
}
```

Some advantages of this approach are:

- JWT / JWS / JWE are supported much better among the various SSI projects. Libraries and tooling can be reused.
- More lightweight, and although some extensibility is missing compared to LD-Proofs, we would reason given the use case for presentations, it's not a big loss.
- Clear path towards encryption, and a potential future interop touching point with the Hyperledger infrastructure, e.g. aries-RFC 0019 can be supported.
- Validating signatures does not require normalization, or any expensive operations.

Some disadvantages of this approach are:
- Not fully standardized (marked as feature at risk + examples are non normative).
- Leaves open questions regarding how the included credentials should be encoded.

**Appendix C:**

> In this case, the QR code simply encodes a request for credentials. During a brief call we presented our simple approach to generating credential requests. The format of our request looks as follows:

```
{
  "interactionToken": {
    "credentialRequirements": [
      {
        "type": [
          "Credential",
          "ProofOfDemoIdCredential"
        ], // Same as the "type" key in the Verifiable Credential,
        "constraints": [
          {
            "==": [
              {
                "var": "issuer"
              },
              "did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777"
            ] // For constraints we use the JsonLogic package (<http://jsonlogic.com/>), these can be easily client side using
          }
        ]
      }
    ],
    "callbackURL": "<https://botm-university-demo.jolocom.com/authenticate/>"
  },
  "typ": "credentialRequest",
  "iat": "1562238821834",
  "exp": "1562242421834", // In case the interaction token is passed via an email, it might make sense to extend the expiry tim
  "iss": "did:jolo:6b8e4245e9863a976b475b7a7c1dc70e290c9fbf0de9eec751d821944658b564#keys-1", // The DID of the requesting servi
  "jti": "a3396a442ecdc"
}
```

> The `base64` encoded version of the token looks as follows:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NksifQ.eyJpbnRlcmFjdGlvblRva2VuIjp7ImNyZWRlbnRpYWxSZXF1aXJlbWVudHMiOlt7InR5cGUiOlsiQ3JlZGVudGlh
```

> In our case, this token can either be rendered as a QR code, or as an deep link, using the `jolocomwallet://consent/<JWT>`
> Also during the call we agreed we could start by implementing this request format, due to it's simplicity. As feedback is gathered, we can iterate to enable for features not yet supported.
> The format of the response has been outlined in #33

**Appendix D**

Current example of what we use for credential offers:

```
{
  "payload": {
    "interactionToken": {
      "callbackURL": "<https://906f927d.ngrok.io/receive/>",
      "offeredCredentials": [
        {
          "type": "TesterCredential",
          "renderInfo": {
            "logo": {
              "url": "<https://miro.medium.com/fit/c/240/240/1*jbb5WdcAvaY1uVdCjX1XVg.png>"
            },
            "background": {
              "url": "<https://jolocom.io/wp-content/themes/jolocom/images/Solution-hero-mobile.jpg>"
            },
            "text": { "color": "#ffffff" },
            "renderAs": "document"
          },
          "metadata": { "asynchronous": false }
```

```
        }
      ]
    },
    "typ": "credentialOfferRequest",
    "iat": 1564486939433,
    "exp": 1564490539433,
    "iss": "did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777#keys-1",
    "jti": "7147887a03582"
  },
  "signature": "145b8ad4c954155ee8823b727b00559c7592611d0f40ac5e0354588525dcd647775a49e8a81e0b93594c4c9b791c4b2fa0afa8ef4a8d87f
  "header": { "typ": "JWT", "alg": "ES256K" }
}
```


This is what encoded in the QR Code.

The response then is:


```
{
  "payload": {
    "interactionToken": {
      "callbackURL": "<https://906f927d.ngrok.io/receive/>",
      "selectedCredentials": [
        {
          "type": "TesterCredential"
        }
      ]
    },
    "typ": "credentialOfferResponse",
    "iat": 1564487486767,
    "exp": 1564491086767,
    "iss": "did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777#keys-1",
    "aud": "did:jolo:b2d5d8d6cc140033419b54a237a5db51710439f9f462d1fc98f698eca7ce9777",
    "jti": "0c4fa25944a68"
  },
  "signature": "39beccbafb0722c9a48d01bd174a043e91b1e72a15fd47d23381ca0dabbb8a9d3b8cd1258a5e6bd2d45feef73bda30d4f869bd69ee27c7f
  "header": {
    "typ": "JWT",
    "alg": "ES256K"
  }
}
```