Aries Framework JavaScript

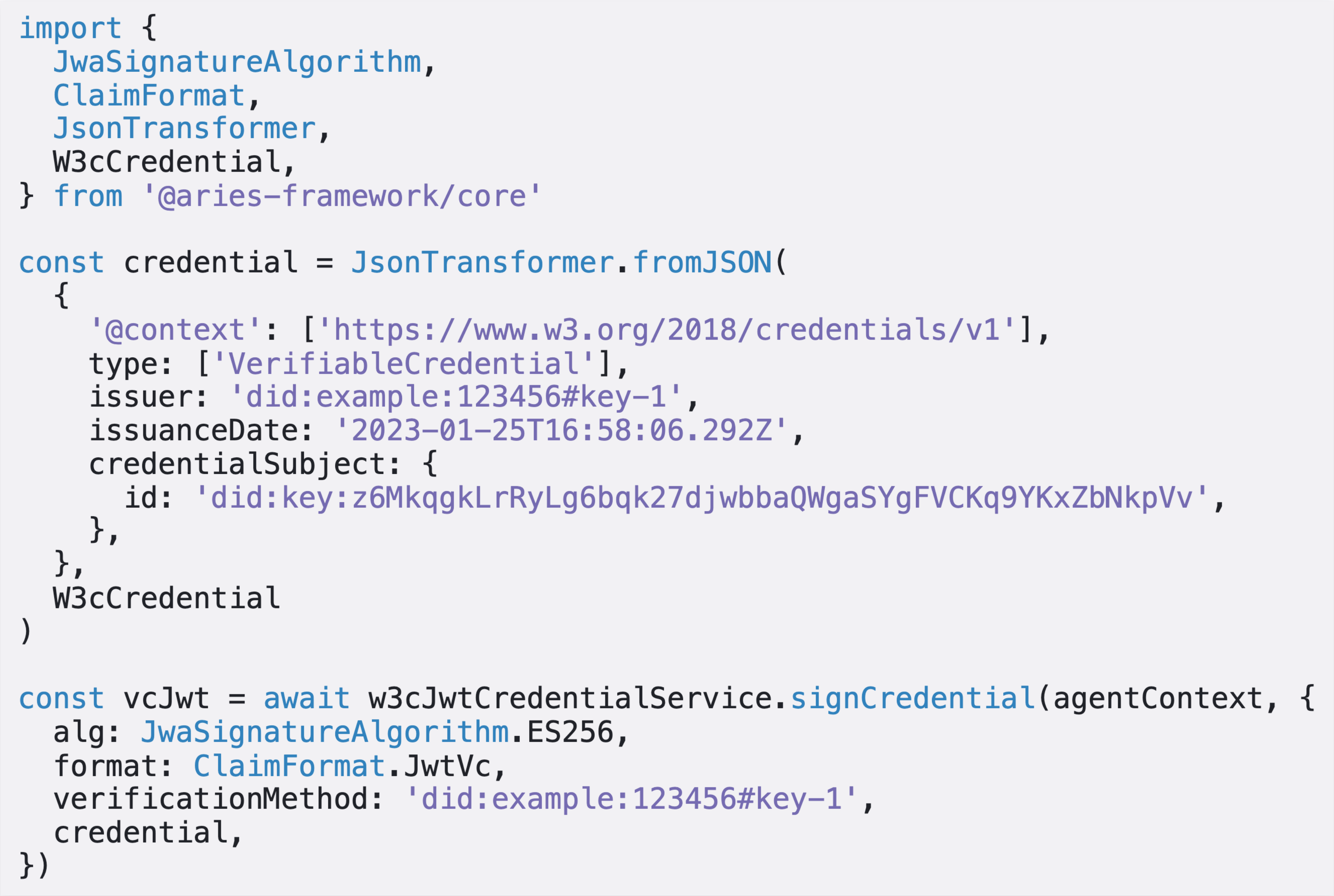
WG Call - June 1st 2023

Agenda

- Update on JWT VCs, extended crypto support and OpenID4VC
- AFJ 0.4.0 Release
- Making it easier for people to get started with AFJ
- Rethinking the Storage Layer

Update on JWT VCs, extended crypto support and OpenID4VC

- Added support for JWT VCs in the `W3cCredentialService` and `W3cCredentialRecord`
- Integrates with the JWS service and thus supports any JWA Algorithm AFJ and the wallet the wallet supports. Currently `ES256` and `EdDSA` are mainly supported, but adding more is really easy if the crypto is supported



```
import {
  JwaSignatureAlgorithm,
  ClaimFormat,
  JsonTransformer,
  W3cCredential,
} from '@aries-framework/core'

const credential = JsonTransformer.fromJSON(
  {
    '@context': ['https://www.w3.org/2018/credentials/v1'],
    type: ['VerifiableCredential'],
    issuer: 'did:example:123456#key-1',
    issuanceDate: '2023-01-25T16:58:06.292Z',
    credentialSubject: {
      id: 'did:key:z6MkqgkLrRyLg6bqk27djwbbaQWgaSYgFVCKq9YKxZbNkpVv',
    },
  },
  W3cCredential
)

const vcJwt = await w3cJwtCredentialService.signCredential(agentContext, {
  alg: JwaSignatureAlgorithm.ES256,
  format: ClaimFormat.JwtVc,
  verificationMethod: 'did:example:123456#key-1',
  credential,
})
```

`did:jwk`

- Added `did:jwk` resolver and registrar
- Encode a JSON Web Key (JWK) into a did document
- spec
- implementation

```
import { KeyType } from '@aries-framework/core'

const result = await agent.dids.create({
  method: 'jwk',
  options: {
    keyType: KeyType.P256,
  },
})
```

OpenID for Verifiable Credentials

- Extended OID4VCI support to also support JWT credentials and more crypto types (was bound to EdDSA/Ed25519Signature2018 before)
- Integrates with JWS service and Signature suite registry, so automatically supports what AFJ and the wallet supports
- Still only supports receiving credentials, not issuing them.
- No support for AnonCreds credentials.
- Integrates with the did resolver/registrar so can be used with any did method

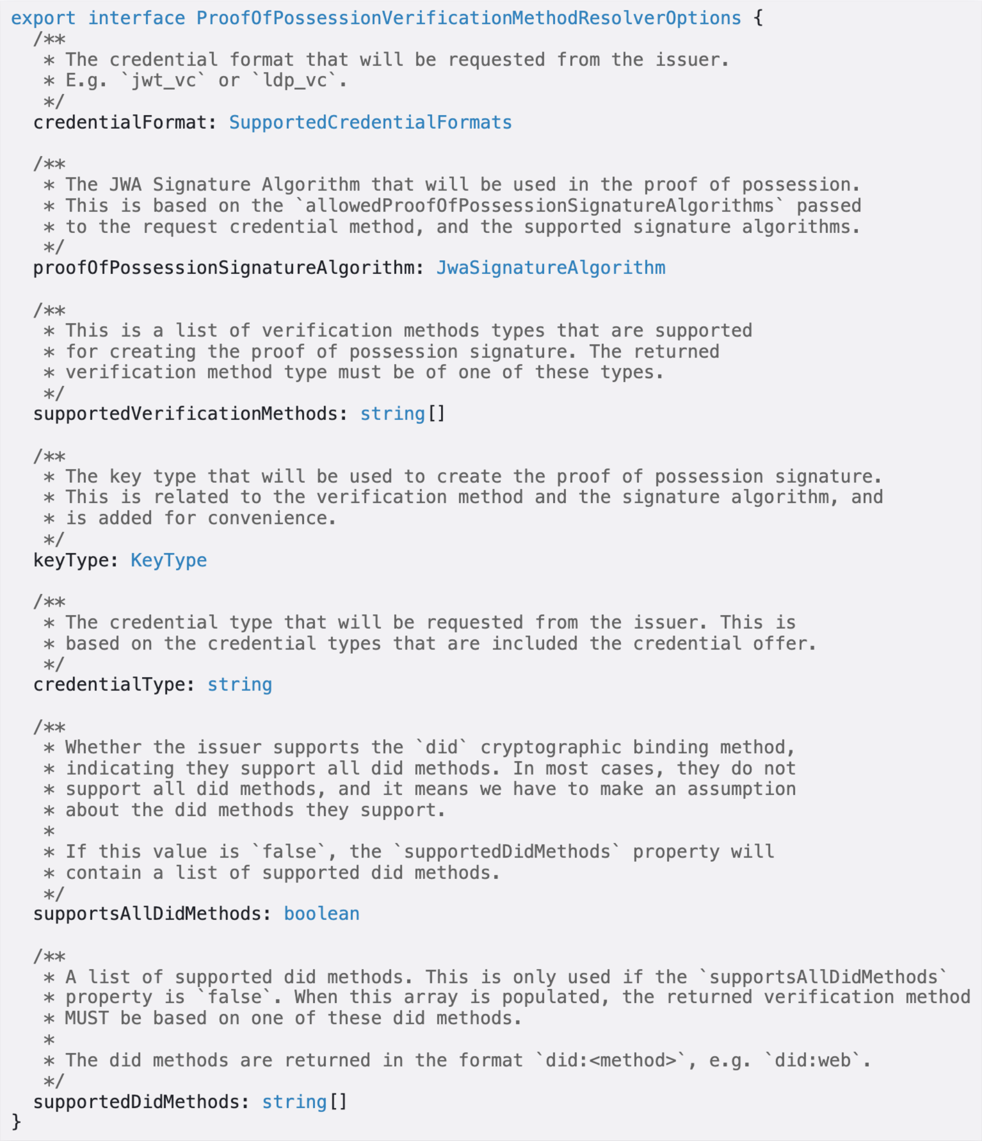
Requesting a credential

```
import { ClaimFormat, JwaSignatureAlgorithm } from '@aries-framework/core'

const result = await agent.dids.create({
  method: 'jwk',
  options: {
    keyType: KeyType.P256,
  },
})

const didJwk = DidJwk.fromDid('did:jwk:xxx')
const verificationMethod = didJwk.didDocument.dereferenceVerificationMethod(
  didJwk.verificationMethodId
)

const w3cCredentialRecords =
  await agent.modules.openId4VcClient.requestCredentialUsingPreAuthorizedCode({
    issuerUri: 'openid-initiate-issuance://?issuer=xxx',
    verifyCredentialStatus: false,
    allowedProofOfPossessionSignatureAlgorithms: [JwaSignatureAlgorithm.EdDSA],
    allowedCredentialFormats: [ClaimFormat.JwtVc],
    proofOfPossessionVerificationMethodResolver: (/* options */) =>
      verificationMethod,
  })
```



```typescript
export interface ProofOfPossessionVerificationMethodResolverOptions {
  /**
   * The credential format that will be requested from the issuer.
   * E.g. `jwt_vc` or `ldp_vc`.
   */
  credentialFormat: SupportedCredentialFormats

  /**
   * The JWA Signature Algorithm that will be used in the proof of possession.
   * This is based on the `allowedProofOfPossessionSignatureAlgorithms` passed
   * to the request credential method, and the supported signature algorithms.
   */
  proofOfPossessionSignatureAlgorithm: JwaSignatureAlgorithm

  /**
   * This is a list of verification methods types that are supported
   * for creating the proof of possession signature. The returned
   * verification method type must be of one of these types.
   */
  supportedVerificationMethods: string[]

  /**
   * The key type that will be used to create the proof of possession signature.
   * This is related to the verification method and the signature algorithm, and
   * is added for convenience.
   */
  keyType: KeyType

  /**
   * The credential type that will be requested from the issuer. This is
   * based on the credential types that are included the credential offer.
   */
  credentialType: string

  /**
   * Whether the issuer supports the `did` cryptographic binding method,
   * indicating they support all did methods. In most cases, they do not
   * support all did methods, and it means we have to make an assumption
   * about the did methods they support.
   *
   * If this value is `false`, the `supportedDidMethods` property will
   * contain a list of supported did methods.
   */
  supportsAllDidMethods: boolean

  /**
   * A list of supported did methods. This is only used if the `supportsAllDidMethods`
   * property is `false`. When this array is populated, the returned verification method
   * MUST be based on one of these did methods.
   *
   * The did methods are returned in the format `did:<method>`, e.g. `did:web`.
   */
  supportedDidMethods: string[]
}
```

SIOPv2 and OpenID4VP

- Currently working on SIOPv2 and OpenID4VP
- Allows to share/prove credentials based on DIF Presentation Exchange (v1)
- OpenID4VP Spec
- SIOPv2 Spec
- RP sends authorization request to Self Issued OpenID Provider (SIOP), wallet responds with a VP based on DIF Presentation Definition
- More updates soon

AFJ 0.4.0 Release

When is it going to happen?

Now!

AFJ 0.4.0 Release

- 0.1.0 release of shared components made today
- After that, AFJ 0.4.0 will be released
- Documentation will be switched to make 0.4.0 the current version
- Still things to do, but we'll make incremental patch releases

Notes about 0.4.0

Experimental features are not covered by semver, and thus may experience breaking changes before 0.5.0

- Implementing your own `AnonCredsRegistry` and AnonCreds service implementation. Using the default implementations (Indy SDK, AnonCreds RS) is fine.
- Shared component libraries (Aries Askar, Indy VDR, AnonCreds RS)
- OpenID4VC Client, and JWT VCs
- Multi-tenancy - Update Assistant NOT integrated yet with tenants module

Notes about 0.4.0

Indy SDK to Askar Migration scripts has limitations

- Mainly focused on holder/verifier role migration in React Native
- Issuer records are not migrated
- Multi-tenancy NOT supported

Making it easier for people to get started with AFJ

- Now that 0.4.0 is released we should make documentation and ease of use main priority
- Simplify usage of demo (docker setup)
- Add documentation for missing parts
  - Sharing Proofs
  - OpenID4VC
  - DID Module
  - ??

Rethinking Storage in Aries Framework JavaScript

- By default AFJ has used Indy SDK as the storage layer
- Recently added support for Aries Askar
- Both libraries have the same model for storage:
    - Store encrypted blob of data by category and id
    - Add tags (also encrypted) to be able to query records
    - Supports unencrypted tags to be able to do advanced queries
        - greater than, less than, etc..

Limitations

- Encrypted blob of data makes it hard to work with concurrent processes modifying the same data
  - Incrementing counters (for AnonCreds revocation)
  - Selectively updating values (instead of overwriting the whole record)
  - Askar supports locking which helps. Not integrated in AFJ yet

Limitations

- Custom encryption layer, while there's already databases out there that have solved this problem
- Are we re-implementing the wheel?
- Migration of records is hard. All records need to be retrieved, decrypted, updated and re-encrypted and stored again.

Limitations

- No good way to do sorting and pagination
- Sorting requires all data to be retrieved and decrypted (can use unencrypted tags, but you need to know beforehand exactly what you're going to sort on)
- Pagination is supported based on offset and limit, but no good way to handle with changes in the dataset that you're querying (cursors, etc..)
- https://github.com/hyperledger/indy-sdk/issues/2431

Thinking of other solutions

- Can we use a 'normal' database, and optionally leverage the native encryption features a database provides?
  - SQLite Encryption Extension
- Cloud providers often have encryption of data by default
  - > Cloud SQL customer data is encrypted when stored in database tables, temporary files, and backups. External connections can be encrypted by using SSL, or by using the Cloud SQL Auth proxy.

Benefits

- Use all the features a normal database has to provide
- Better migration
- Normalization of data
- Sorting, LIKE filtering, better pagination (cursors)
- Probably better performance, as building on top of a database that's optimized to be fast.

Rethinking the Storage Model

- AFJs current record and storage model is not designed for this.
- How would we define the database structure for records?
  - Ideally in a way that supports multiple backends
- ORM
  - Veramo uses TypeORM
  - Define classes for your data model (same as now)
  - Usually provides a migration interface
- Domain specific language
  - E.g. Prisma
    - Mostly agnostic of database, but not really
    - built in migrations
    - No need for classes, types are generated
- Query builder
  - Can abstract language specific features
  - Need to define models for in TS ourselves?
  - E.g. Knex

Thoughts?

Next week

- Ariel will present on the new Wallet API
- Any other topics people would like to discuss?