Issuance

Issue a Verifiable Credential Suggest Edits

This tutorial will walk you through issuing verifiable credentials using the verite sdks. For additional context, see Issuance Flow . A complete example of building an issuer is available at our demo-issuer demo.

For the sake of this demo, we will be using Decentralized Identifiers (DIDs) to identify the issuer (you) and subject (the person or entity the credential is about), as well as JSON Web Tokens (JWTs) as the means of signing and verifying the credentials. Strictly speaking, you do not need to use JWTs, but as they are industry-standard and tooling extensively available, they are used throughout this sample implementation.

In practice, you are not limited to using DIDs to identify and prove ownership; anything with a public/private key pair can effectively be used to identity a credential's subject and later prove that identity. Public blockchain addresses offer less privacy and rotation/recovery capabilities than DIDs, but can simplify VC handling in some architectures where DIDs are inhibitive. To make them subjects of verifiable credentials, blockchain addresses can be expressed as DID-like URI's using the did:pkh method , based on the CAIP-10 URI scheme . To prove these, you will need a signature to recover the public key of a given address; the off-chain signatures used to "connect wallet" in most contemporary web3 apps and dApps is adequate for such purposes.

Prerequisites: Issuer Setup

See issuer setup instructions .

Step 1: Create a DID for your issuer

In order to issue a credential, you must have some way of identifying yourself as an Issuer. This allows the credential to be "verifiable" by a 3rd party.

To start, you should create a DID keypair.

JavaScript import { randomDidKey } from "verite" import { randomBytes } from "crypto"

const issuerDidKey = randomDidKey(randomBytes) That keypair should look like the following:

JavaScript { id: 'did:key:z6Mkf2wKCqtkNcKB9kRdHnEjieCLJPSfgwuR19fxBhioAwR7#z6Mkf2wKCqtkNcKB9kRdHnEjieCLJPSfgwuR19fxBhioAwR7', controller: 'did:key:z6Mkf2wKCqtkNcKB9kRdHnEjieCLJPSfgwuR19fxBhioAwR7', publicKey: Uint8Array(32) [...], privateKey: Uint8Array(64) [...] } Keep this keypair safe. You will be using this to sign all Verifiable Credentials. If you lose a keypair, you won't be able to sign future credentials with the same key, and may temporarily lose the ability for verifiers to verify your credentials.

Optional step: Use did:web

In order to leverage trust already established by your domain name, you can expose your did:key (created above) on your domain via the did:web method and use that as reference.

For example, instead of listing the issuer of your VCs as did:key:z6Mkf2wKCqtkNcKB9kRdHnEjieCLJPSfgwuR19fxBhioAwR7 , you could issue your VCs from did:web:example.com .

To do this, you need to create a .well-known/did.json file on your domain, which will allow did resolvers to find your public keys. An added benefit of did:web is that it will allow you to rotate your keys at your desire; previously-issued VCs will no longer verify against a rotated key.

We won't go into the specifics in this article, but you can read more about did:web here , and you can set up a did:web in minutes using this tutorial .

Step 2: Receive a subject's (end-user) DID

In order to issue a Verifiable Credential, you must know where to issue the credential. This can be a subject's DID or any other public key (such as an ethereum address, etc). In this case, we'll continue using DIDs.

Generally, you would follow the Presentation Exchange (PEx) flow as a means of allowing the subject to submit their DID as well as a request for a specific type of VC.

To do so, you need to create a Credential Manifest showcasing what attestations you offer as Verifiable Credentials. In this example, we will offer a "Know Your Customer, Anti-Money Laundering" attestation (KYC/AML Attestation), meaning we are compliant with US regulations and have checked your account to determined you are not a bad actor. The benefit of this type of VC is that another service can be compliant with regulations without any personal information exposed in the VC.

JavaScript import { buildKycAmlManifest } from "verite"

const manifest = buildKycAmlManifest({ id: issuerDidKey.controller }) This manifest contains instructions for the subject to use to request a VC.

The subject uses that manifest to build a “Credential Application”, which serves as a request for a VC. For example, a subject could create the application as such:

JavaScript import { randomDidKey, composeCredentialApplication } from "verite"

// The subject needs a did:key, generate a random one: const subjectDidKey = randomDidKey(randomBytes)

const application = await composeCredentialApplication(subject, manifest)

Step 3: Issue the Verifiable Credential

Once the subject has requested a VC and submitted their DID (as part of their credential application), the Issuer can create a VC.

For our example, we're building a VC containing a KYC/AML Attestation. This attestation is quite simple in Verite. It defines an authority which has performed proper KYC/AML checks on the subject (in this example the authority is Verite, with the DID of did:web:verite.id), so that will used in the issuer field.

From the Credential Application, the issuer also confirms:

- Who to issue the VC to -- the subject identifier is the
- holder
- field from the VP in the Credential Application* That the subject actually controls the identifier by signing a proof (along with the challenge)
- Any other inputs requested by the manifest (if specified)

We transport Verifiable Credentials using a Verifiable Presentation. The presentation allows the issuer to bundle the credentials before sending to the subject, along with signed proof.

Putting this together, we can do the following:

JavaScript import { composeFulfillmentFromAttestation, validateCredentialApplication, buildIssuer, KYCAMLAttestation } from "verite"

const decodedApplication = await validateCredentialApplication(application)

const attestation: KYCAMLAttestation = { type: "KYCAMLAttestation", process: "https://verite.id/definitions/processes/kycaml/0.0.1/usa", approvalDate: new Date().toISOString() } const credentialType = "KYCAMLCredential" const issuer = buildIssuer(issuerDidKey.subject, issuerDidKey.privateKey)

const presentation = await composeFulfillmentFromAttestation(issuer, decodedApplication.holder, manifest, attestation, credentialType) The subject can then decode the presentation and store their Verifiable Credential for future use.

That’s it. You have now issued a Verifiable Credential.

You can view this demo as a full working example in oudemo-issuer demo. Updated5 months ago *Table of Contents * * Prerequisites: Issuer Setup * * Step 1: Create a DID for your issuer* * * Optional step: Use did:web * * Step 2: Receive a subject’s (end-user) DID * * Step 3: Issue the Verifiable Credential