How to interact with the Digital Euro N€XT settlement API

The Digital Euro N€XT settlement API is based on a UTXO model. This implies authorization of transactions is facilitated by confirmation of the predicates of the input UTXO, which is done using digital signatures. To be able to build a valid Digital Euro transaction a keypair is required. Which algorithm to use and the additional steps required for a successful settlement are described in the following sections, which will detail the steps:

- Creating a keypair
- Generating an address
- Funding
- Building a transaction
- Signing a transaction

General `byte[]` encoding

All binary datatypes are encoded in Base64, as defined in the OpenAPI specification. The following java example illustrates how an array of three bytes is encoded to Base64. In the example the array `[0,255,128]` (using signed bytes: `[0,-1,-128]`) will be encoded as the string `"AP+A"`.

```
ByteBuffer b = ByteBuffer.allocate(3);
b.put((byte) 0).put((byte) 255).put((byte) 128);
String encoded = Base64.getEncoder().encodeToString(b.array());
```

Everywhere a `byte[]` is transferred over the API this conversion has to be made.

Creating a keypair

The initial version of the Digital Euro prototype uses ECDSA-256 signatures. Many languages provide tools to generate ECDSA signatures out of the box. See the following example in Java:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("EC");
kpg.initialize(256);
KeyPair kp = kpg.generateKeyPair();
Key pub = kp.getPublic();
Key priv = kp.getPrivate();
```

Generating an address

Addresses used in the prototype are double-hashed public keys. The formula for calculation is `ripemed160(sha256(pubKey))` where pubKey is the public key in X.509 format. Continuing the example above using the cryptographic library *Bouncy Castle* for the Ripemed160 Hash:

```
byte[] pubKey = kp.getPublic().getEncoded();
MessageDigest md = MessageDigest.getInstance("sha-256");
var sha256 = md.digest(pubKey);
byte[] address = new byte[20];
RIPEMD160Digest digest = new RIPEMD160Digest();
digest.update(sha256, 0, 32);
digest.doFinal(address, 0);
```

Funding

The current implementation is a temporary solution of the funding process (a future release is likely to include DCAs for intermediaries). The process is initiated by invoking POST request to the endpoint `/fundingRequest`, containing the following information:

- IBAN for source of funding
- Value to be funded
- Address to be credited

For example the request body of a funding request could look like:

```
{
      "callbackUrl": "http://localhost:8080",
      "iban": "EU000001"
      "amount": 100,
      "receiverAddress": "KVNGhLGXy4rSxGn1c+eUN5l4Z84="
}
```

The UTXO received upon a funding or settlement request has the following structure:

- **Serial number**: a unique identifier of the UTXO.
- **Amount**: the value of the UTXO. In the range of [0,9.223.372.036.854.775.807]

- **Witness program commitment**: the hash of the witness program. This is dependent on witness type. For **P2PKH** it is the public key.

These UTXO can be used as input for consecutive transactions without any decoding or modification.

Building a transaction

Building a transaction is the association of inputs and outputs. Inputs are the UTXO that were received during a previous settlement or funding process and can be used as is - without any decoding or encoding. Outputs are a list of hashed public keys (see *Generating an address*) together with the desired value to spend to this output. The sum of all output-amounts must always match the sum of the input-amounts. An example of such an unsigned transaction is shown below:

```
"inputs": [
    {
        "amount": 100,
        "witnessProgramCommitment": "KVNGhLGXy4rSxGn1c+eUN5l4Z84=",
        "serialNumber": "R0VORVNJU19UWF9JRAAAAAE="
    }
],
"outputs": [
    {
        "amount": 100,
        "witnessProgramCommitment": "GyxGucbbdZeKxDoKP86k+ByR24k="
    }
]
```

Signing a transaction

The final step in building a valid transaction is authorizing the inputs. To do so, each input requires a witness, which contains the following information:

- **WitnessType**: A string representing the type (and structure) of the witness. “P2PKH” for a standard transaction.
- **WitnessProgram**: A `byte[]` representing the X.509 encoded public key corresponding to the hash in the input.
- **WitnessData**: A signature of the transaction hash (sha256(inputs, outputs)). Calculating the transaction hash requires all data in the inputs and outputs to be concatenated. Byte arrays can simply be appended. Numbers are serialized using big endian encoding. A 64bit amount of 100 will thus need to be converted to the array `[0,`

`0, 0, 0, 0, 0, 0, 100]`. This array can then be used to calculate the sha256 transaction id.

```
Signature signature = Signature.getInstance("NONEwithECDSA");
  signature.initSign(privKey);
  signature.update(transactionHash);
  byte[] witnessData = signature.sign();
```