ArcBlock / **abt-did-spec**

ABT DID Protocol

10 stars 5 forks

Star Watch

Code Issues 1 Pull requests 1 Actions Projects Wiki Secur

master

dingpl716 Merge pull request #5 from ArcBlock/v0.3.1 … on 12 Oct 2019 18

View code

README.md

ABT DID

This repository defines the specification of ArcBlock DID Auth Protocol.

Table of Contents

Abstract

Motivation

DID

The decentralized identification provided by ArcBlock.

```
did:abt:z1muQ3xqHQK2uiACHyChikobsiY5kLqtShA
 DID           DID string
```

```
schema
```

DID Type

DID type is the first two bytes of the DID string's binary format. It contains three sections:

1. The first six bits are the *RoleType* of DID, e.g., `account` = 0
2. The following 5 bits denote the *KeyType*, algorithm to convert secret key to public key, e.g., `ED25519` = 0
3. The latter 5 bits represent the *Hash* function to calculate the hash of the public key, e.g., `sha3` = 1

So DID type bytes `0x0C01` can be interpreted as follows:

```
+-------------+-----------+------------+
| 000011      | 00000     | 00001      |
+-------------+-----------+------------+
| application | ed25519   | sha3       |
+-------------+-----------+------------+
```

See appendix for the full list of values.

Create DID

This process is inspired by Bitcoin. The difference is that we use a single SHA3 to replace SHA256 and RIPEMD160 which are used to do double hash in Bitcoin.

- Step 1: Choose the *RoleType*, *KeyType* and *Hash* from above, let's use `application`, `ed25519` and `sha3` in this example.
- Step 2: Choose a secret key randomly, e.g.

  ```
  D67C071B6F51D2B61180B9B1AA9BE0DD0704619F0E30453AB4A592B036EDE644E4852B7091317E36
  ```

- Step 3: Generate the public key of this secret key by using the *KeyType*. So we can get public key

  ```
  E4852B7091317E3622068E62A5127D1FB0D4AE2FC50213295E10652D2F0ABFC7
  ```

- Step 4: Get the *Hash* of the public key

  ```
  EC8E681514753FE5955D3E8B57DAEC9D123E3DB146BDDFC3787163F77F057C27
  ```

- Step 5: Take the first 20 bytes of the public key hash

```
EC8E681514753FE5955D3E8B57DAEC9D123E3DB1
```

- Step 6: Add the DID type bytes `0x0C01` in front of the hash of Step 4

```
0C01EC8E681514753FE5955D3E8B57DAEC9D123E3DB1
```

- Step 7: Get the hash of the extended hash in Step 6

```
42CD815145538F8003586C880AF94418341F9C4B8FA0394876553F8A952C7D03
```

- Step 8: Take the first 4 bytes in step 7

```
42CD8151
```

- Step 9: Append the 4 bytes in step 8 to the extended hash in step 6. This is the binary DID string

```
0C01EC8E681514753FE5955D3E8B57DAEC9D123E3DB142CD8151
```

- Step 10: Encode the binary value by using the Bitcoin Base58 method. Note, we are using IPFS's multibase to encode as `base58_btc` type, it will add a `z` in the beginning of the base58 encoded string

```
zNKtCNqYWLYWYW3gWRA1vnRykfCBZYHZvzKr
```

- Step 11: Assemble the parts and get the full DID

```
did:abt:zNKtCNqYWLYWYW3gWRA1vnRykfCBZYHZvzKr
```

Create Extended DID

One of our main goal is to protect users' privacy. So people do not use the DID generated from their master key to talk to DAPPs, instead, the WALLET automatically generates an extended DID according to the user's master DID and the DAPP's DID and use this extended DID to communicate with the DAPP. The following process is how to create an extended DID:

- Step 1: Convert the `appDid` from the string format to binary format.
- Step 2: Apply sha3 to the binary format of `appDid` .

- Step 3: Take the first 64 bits of the hash.
- Step 4: Split these 64 bits into two 32-bits-long sections denoted as `S1` and `S2`.
- Step 5: Derive the HD secret key by using path `m/44'/ABT'/S1'/S2'/address_index`. Here, `ABT'` is the coin type registered on SLIP44 (ABT's coin type is 260). `address_index` is numbered from index 0 in a sequentially increasing manner.
- Step 6: Convert the HD secret key to `userDid` by using the rules described in DID section.

Declare DID

ABT DID method defines the DID document as the account state, but one DID does not have a corresponding state in the blockchain state by default. One must declare a DID in order to let it be visible to others on the chain. Declaring a DID is done by sending a `DeclareTx` transaction to the blockchain. The following is a sample transaction.

```
{
  "hash": "36BBCA0115A52C0F43C42E84CAE368481A0F32B218380721E3DD2B0456D1D294",
  "tx": {
    "from": "z1RMrcjJVwuohBoqAsPaVvuDajQi1fDo8Qx",
    "itx": {
      "__typename": "DeclareTx",
      "data": null,
      "pk": "IWNMqz5IdsqxO0x9iqdlSfMvPkchVc3un8mmLXT_GcU",
      "type": {
        "address": "BASE58",
        "hash": "SHA3",
        "pk": "ED25519",
        "role": "ROLE_ACCOUNT"
      }
    },
    "nonce": 1,
    "signature": "E_BkPhw-WUpkTk5nn_WF4z-8huOBqjl-3vQ122TYCDQiahFlklVJT3I7YUwr8d-pi_
    "signatures": []
  }
}
```

After the declaration, the corresponding state will be created for this DID.

Read DID

To read a DID, one just needs to send a GRPC request to ABT network. The structure of the request is described as follows. The `address` filed is the DID to query. The `keys` field is used to fetch specific parts of the state. If it is omitted, entire account states will be returned. The `height` field can be used to retrieve the older version of the DID documents. If it is omitted, the latest one will be returned.

```
message RequestGetAccountState {
  string address = 1;
  repeated string keys = 2;
  uint64 height = 3;
}
```

The response contains the DID document associated with this DID.

Update DID

Account state could be potentially updated through any kind of transaction and different types of transaction would affect the account state in different aspects. The following is an example of `TransferTx` transaction.

```
{
  "hash": "36BBCA0115A52C0F43C42E84CAE368481A0F32B218380721E3DD2B0456D1D294",
  "tx": {
    "from": "z1RMrcjJVwuohBoqAsPaVvuDajQi1fDo8Qx",
    "itx": {
      "__typename": "TransferTx",
      "to": "z1YgP3zaVdQzB9gC3kHAyTiiMMPZhLzCLDP",
      "data": "The new data to replace the existing one.",
      "pk": "IWNMqz5IdsqxO0x9iqdlSfMvPkchVc3un8mmLXT_GcU",
    },
    "nonce": 1,
    "signature": "E_BkPhw-WUpkTk5nn_WF4z-8huOBqjl-3vQ122TYCDQiahFlklVJT3I7YUwr8d-pi_
    "signatures": []
  }
}
```

It is worth mentioning that old versions of DID document are still stored on the chain due to the natures of the data structure used by blockchain. So this operation is not updating the DID document in place but putting a new version over the existing one.

Revoke DID

An account cannot be erased from a blockchain, but the account owner can send a `SuicideTx` to mark the account as `dead`. This process is irreversible and any transaction that involves a `dead` account is considered as invalid.


```
{
  "hash": "36BBCA0115A52C0F43C42E84CAE368481A0F32B218380721E3DD2B0456D1D294",
  "tx": {
    "from": "z1RMrcjJVwuohBoqAsPaVvuDajQi1fDo8Qx",
    "itx": {
      "__typename": "RevokeTx",
      "pk": "IWNMqz5IdsqxO0x9iqdlSfMvPkchVc3un8mmLXT_GcU",
    },
    "nonce": 1,
    "signature": "E_BkPhw-WUpkTk5nn_WF4z-8huOBqjl-3vQ122TYCDQiahFlklVJT3I7YUwr8d-pi_
    "signatures": []
  }
}
```


Privacy considerations

The ways of how to create, register and manage DIDs in ABT DID method are designed to provide enhanced privacy, improved anonymity and reduced correlation risk.

- Keeping personally-identifiable information (PII) off-ledger.

  PII is not stored on chains, only the signatures are stored. When a verifier needs to verify a claim, it asks the peer to be verified for the original data. Due to the nature of DSA algorithms, the it is very hard for the peer to forge the previously signed data.

- DID Correlation Risks and Pseudonymous DIDs

  As illustrated in Request DID Authentication step 1, the way of how to generate application-specific DID enforces pseudonymous DID and privacy across different chains. An user has multiple extended DIDs under one master DID and different extended DIDs are used in different chains. The master DID is never exposed publicly in any way.

- DID Document Correlation Risks

  DID documents of different extended DIDs of the same master DID are also isolated.

Security considerations

The underlayer blockchain also ensured the following security risks:

- Replay attacks
- Man-in-the-middle attacks
- Message insertion attacks
- Deletion attacks
- Modification attacks

Our blockchain based implementation has considered each of following requirements listed in W3C DID specification:

- security assumptions of distributed ledger topology
- policy mechanism used to prove DIDs are uniquely assigned
- integrity protection and update authentication for DID operations
- DID Method-specific endpoint authentication

ABT DID Authentication Protocol

ArcBlock DID (decentralized identification) Authentication Protocol is an open protocol that provides a secure decentralized authentication mechanism by using asymmetric cryptography technology. In this protocol, we define authentication as the process that the WALLET provides answers of the requested claims to the DAPP.

This protocol involves three parties:

- WALLET: The decentralized client side agent of the end user. ArcBlock provides ABTWallet as a solution.
- DAPP: The decentralized application that provides service to users. ArcBlock provides the Forge, a blockchain development framework.
- REGISTRY_CHAIN: The decentralized trust authority. ArcBlock provides the ABT chain as the decentralized trust authority.

The entire authentication protocol contains three processes:

- Pre-knowledge
- Request DID Authentication
- Response DID Authentication (could be multiple rounds)

Pre-knowledge:

Pre-knowledge refers to the process that WALLET gets the information of a DAPP before the real authentication starts. DAPP can provide the information in a QR code or deep link.

The following is an example of the content of a QR code or deep link.

```
https://arcwallet.io/i?action=requestAuth&url=https://example-dapp.io/auth/
```

- `linkPath` : The `linkPath` is located at the beginning of the link and is used to locate the WALLET. In this example, the 'linkPath' is `https://arcwallet.io/i` . This part is configurable, and the SDK allows developers to register their domain for their applications.
  - If the QR code is scanned by a third-party camera, e.g. iPhone, WALLET should be open, and the parameters will be passed to WALLET if it is installed. If WALLET is not installed, an installation page should be open. The same behavior applies to the case when the user clicks such a link. The process illustrated above depends on the deep link technology of different platforms such as iOS and Android.
  - If the QR code is scanned by WALLET, WALLET should ignore this section and parse the parameters.
- `action` : the action WALLET should perform in the next step. Here the action should be `requestAuth` and the WALLET should use `GET` method to access the `url`
- `url` : a x-www-form-urlencoded URL. WALLET uses this url to start the Request DID Authentication process latter.

Request DID Authentication

Request DID Authentication is the step that the WALLET actually starts the authentication process. The WALLET makes a request to the endpoint pointed by the `url` obtained from the previous step. The following is the step-by-step break down of what's going to happen of this process:

1. WALLET makes a request to the endpoint.

   ```
   GET https://example-dapp.io/auth
   ```

2. The response returned by the DAPP shall contain two must-have fields `appPk` and `authInfo` and one optional field `appSession` .

   - `appPk` : The DAPP's public key, encoded by Bitcoin Base58.
   - `authInfo` : A standard JWT token.

- `appSession` : The encrypted information for the DAPP to process this authentication session. WALLET must return it to DAPP with no change. It is up to the DAPP to how to encrypt this field. The following is an example response payload.


```
{
  "appPk": "zBdZEnbDJTijVVCx4Nx68bzDPPMFwVizSRorvzSS3SGG2",
  "authInfo": "eyJhbGciOiJFZDI1NTE5IiwidHlwIjoiSldUIn0.eyJleHAiOjE1NDg4MDM0MjIsI
  "appSession": ""
}
```


The header and body part of `authInfo` displayed above decodes as


```
{
  "alg": "Ed25519",
  "typ": "JWT"
}
```

```
{
  "iss": "did:abt:zNKtCNqYWLYWYW3gWRA1vnRykfCBZYHZvzKr",
  "iat": 1548703422,
  "nbf": 1548703422,
  "exp": 1548803422,
  "appInfo": {
    "name": "The name of the DAPP",
    "description": "The description of the DAPP.",
    "logo": "https://example-dapp/logo"
  },
  "chainInfo": {
    "host": "https://example-dapp/api"
  },
  "action": "responseAuth",
  "url": "https://example-dapp/auth",
  "requestedClaims": [
    {
      "type": "authPrincipal",
      "description": "Please set the authentication principal.",
    }
  ]
}
```


- `iss` : The issuer of this JWT payload. In this example, it is the DAPP's DID generated from `appPk` .
- `iat` , `nbf` and `exp` : Follows the JWT standard.

- `appInfo` : The basic information of this DAPP, such as name, description, logo and so on.
- `chainInfo` : The basic information of this chain. The protocol right now only supports `chainHost` in side this field. The `chainHost` is the ForgeWeb endpoint of this chain.
- `url` : A must-have field that will be used by the WALLET in Response DID Authentication process.
- `action` : Tells what action should the WALLET perform in next step. Here it should be `responseAuth` and the WALLET shall use `POST` method to access the `url` .
- `requestedClaims` : The verifiable claims required by the DAPP's. In this example, the type of the requested claim is `authPrincipal` which usually is the first claim required by the DAPP in the authentication process. This claim means that the DAPP is asking the WALLET to set the authentication principal before they can continue to the reset of operations. The authentication principal is the user's DID whose private key shall be used to sign all of the requests sent from the WALLET to the DAPP.

3. After gets the response, the WALLET should do following verifications:
   i. Verifies if the `iat` is later than the request is sent.
   ii. Verifies if the response has expired by using `exp` .
   iii. Verifies if the signature matches the `appPk` and if the `appPk` matches the appDid in the `iss` field.
4. **(TBD)** The WALLET could (may under users' request) ask a registry blockchain for the metadata of the DAPP, `trustLevel` for example. ArcBlock provides ABT chain as a registry chain.
5. **(TBD)** The `trustLevel` can be used by the WALLET when displaying requested claims to user. For the DAPP whose `appDid` cannot be found on registry blockchain, the WALLET should make the entire page with �high risk mark. If an DAPP is asking verifiable claims whose required trust_level is higher than the `appDid` s', the WALLET should display those claims with high risk mark.
6. The WALLET needs to determine the DID to use to set as the authentication principal. If the user has accessed the DAPP before, then the WALLET should directly use extended DID as the `authPrincipal` . Otherwise, the WALLET shall create an extended DID before any other operations.

Response DID Authentication

This step can be thought of as that the WALLET is answering the questions asked by the DAPP through the `requestedClaims` field and the DAPP would possibly return more questions depending on the answers.

After determined the value of the `userDid` to respond to the DAPP, the WALLET shall organize the response just in following format:

```
{
  "userPk": "",
  "userInfo": "",
  "userSession": "",
  "appSession": "",
}
```

- `userPk` : The corresponding public key of the `userDid` .
- `userInfo` : The signed authentication object in JWT format.
- `userSession` : The encrypted information for the WALLET to process this authentication session. DAPP must return this filed back to WALLET with no change. It is up to the WALLET to encrypt this part.
- `appSession` : The encrypted information for the DAPP to process this authentication session. WALLET must return it to DAPP with no change.

Just like the `authInfo` , the `userInfo` is also a standard JWT object which can be decoded as follows in this example:

```
{
  "alg": "Ed25519",
  "typ": "JWT"
}
{
  "iss": "userDid",
  "iat": "1548713422",
  "nbf": "1548713422",
  "exp": "1548813422"
}
```

The fields `iss` means the issuer of this JWT payload. It is also the **authentication principal** determined in the previous step.

More Response DID Authentication

After the WALLET set the authentication principal, the DAPP could send more claims back to the WALLET depending on its business logic. It is just like a customer manager in a bank could ask a serious of questions before he or she can process the business service for the customer. So let's see another example response that a DAPP could return to the WALLET after the first round authentication.

```
{
  "appPk": "zBdZEnbDJTijVVCx4Nx68bzDPPMFwVizSRorvzSS3SGG2",
  "authInfo": "eyJhbGciOiJFZDI1NTE5IiwidHlwIjoiSldUIn0.eyJleHAiOjE1NDg4MDM0MjIsImlhd
  "appSession": "",
  "userSession": ""
}
```

The header and body part of `authInfo` displayed above decodes as

```
{
  "alg": "Ed25519",
  "typ": "JWT"
}
{
  "iss": "did:abt:zNKtCNqYWLYWYW3gWRA1vnRykfCBZYHZvzKr",
  "iat": 1548703422,
  "nbf": 1548703422,
  "exp": 1548803422,
  "appInfo": {
    "name": "The name of the application",
    "description": "The description of the application.",
    "logo": "https://example-dapp/logo"
  },
  "action": "responseAuth",
  "url": "https://example-app/auth",
  "requestedClaims": [
    {
      "type": "profile",
      "description": "Please fill in basic information.",
      "items": ["fullName", "mobilePhone", "mailingAddress"]
    },
    {
      "type": "agreement",
      "description": "The user data usage agreement.",
      "meta": {
        "name": "user_agreement",
      },
      "uri": "https://document-1.io",
      "hash": {
        "method": "sha2",
        "digest": "The hash result of the document's content"
      }
    },
    {
      "type": "agreement",
      "description": "The service agreement",
      "meta": {
        "name": "service_agreement"
```

```
      },
      "uri": "ipfs://document-2",
      "hash": {
        "method": "sha3",
        "digest": "The hash result of the document's content"
      }
    }
  ]
}
```

- `meta` : An optional field in each requested claim, the WALLET **MUST** return this field to the DAPP without any change.
- `description` : The description to display to end user.

So in this example, the DAPP is asking the WALLET to provide some basic information of the user. The WALLET shall prompt a user to let him or her to finish this claims and send back to the DAPP again. We will talk about how to process each type claim in next section.

Revoke DID Authentication

TBD

Verifiable Claims

Verifiable claims is a list of claim item. Different types of claims have some common attributes even though they are designed to server under different scenarios.

- `type` : This is a must-have field for types of claims items.
- `description` : The message to display to end users, a must-have field for all types of claims items. The WALLET could omit this field when returning the claims for the sake of network bandwidth.
- `meta` : An optional field available for all types of claim item. It is up to the DAPP to put some information there so that it can easily process the claims sent back from the WALLET. The WALLET **MUST** return this field in each claim item without any changes.

The supported types of claims are described as follows:

AuthPrincipal

This is the first claim to ask by the DAPP, it is used to informing the WALLET to set the authentication principal for the entire authentication process. The authentication principal is the `iss` field of the JWT payload sent from the WALLET.

To ask the WALLET to set up authentication principal, the DAPP should respond the WALLET with this claim:

```
{
  "requestedClaims": [
    {
      "type": "authPrincipal",
      "description": "Please set the authentication principal.",
    }
  ]
}
```

Sometimes a DAPP may want the WALLET to prove that it is a specific user. In such cases, the DAPP can add the field `target` in the claim item. The WALLET must set the authentication principal to the specific DID.

```
{
  "requestedClaims": [
    {
      "type": "authPrincipal",
      "description": "Please set the authentication principal to start this atomic s
      "target": "did:abt:z1fw9Ycbb7cJnj1NUm6hyuSYHuTHEwph8yH"
    }
  ]
}
```

Profile

Profile is the verifiable claim used to gather users' basic information such as name, email, age and so on. The requested information is put in the `items` sub-field. For example, when the DAPP requires users to provide their first name, mobile phone number and mailing address, it could respond a claim to the WALLET in following way.

```
{
  "requestedClaims": [
    {
      "type": "profile",
      "description": "Please provide the basic information.",
      "items": ["fullName", "mobilePhone", "mailingAddress"]
    }
  ]
}
```

The WALLET shall answer this claim in such way:

```
{
  "requestedClaims": [
    {
      "type": "profile",
      "fullName": "Alice Bean",
      "mobilePhone": "123456789",
      "mailingAddress": {
        "addressLine1": "456 123th AVE",
        "addressLine2": "Apt 106",
        "city": "Redmond",
        "state": "WA",
        "postalCode": "98052",
        "country": "USA"
      }
    }
  ]
}
```

Please see the appendix for the list of all profile items.

Agreement

Agreement is another commonly used type of claim. It stands for the agreements that a DAPP asks the user to sign. An `agreement` claim type contains following fields:

- `uri` : An URI points to the content of the agreement.
- `hash` : An object where `method` sub field specifies the algorithm (sha3, sha2 and so on) used, and `digest` sub field is the hash result.
- `agreed` : A boolean value added by the WALLET to indicate if the user agrees the agreement.
- `sig` : The DSA signature of the `hash` .

When a DAPP wants a user to sign agreements, it should add a list of such claim items in the response.

```
{
  "requestedClaims": [
    {
      "type": "agreement",
      "description": "The user data usage agreement.",
      "meta": {
        "name": "user_agreement."
      },
```

```
        "uri": "https://document-1.io",
        "method": "sha2",
        "digest": "The hash result of the document's content"
      },
      {
        "type": "agreement",
        "description": "The service agreement",
        "meta": {
          "name": "service_agreement"
        },
        "uri": "ipfs://document-2",
        "method": "sha3",
        "digest": "The hash result of the document's content"
      }
    ]
  }
```

When see this response, WALLET should prompt the user to sign the agreements. Later, the WALLET should submit a list of signed claim items back to the DAPP. If the user agrees, WALLET shall add the `agreed` and the `sig` field. If the user declines, then the WALLET just need to add the `agreed` field with value `false`. No signature is required in this situation.

```
  {
    "requestedClaims": [
      {
        "type": "agreement",
        "uri": "https://document-1.io",
        "method": "sha2",
        "digest": "The hash result of the document's content",
        "meta": {
          "name": "user_agreement."
        },
        "agreed": true,
        "sig": "user's signature against the doc digest plus AGREED."
      },
      {
        "type": "agreement",
        "uri": "ipfs://document-2",
        "method": "sha3",
        "digest": "The hash result of the document's content",
        "meta": {
          "name": "service_agreement"
        },
        "agreed": false
      }
    ]
  }
```

Asset

The Asset claim item is used to let the WALLET to provide a asset DID to the DAPP. Usually the DAPP would need to verify the ownership of the asset against the DID set in the authentication principal. To ask the WALLET to provide an asset DID, the DAPP shall send the claims in following way:

```
{
  "requestedClaims": [
    {
      "type": "asset",
      "description": "Please provide the coupon you own.",
      "meta": {
        "id": "coupon",
      }
    }
  ]
}
```

The WALLET shall return the response in following way:

```
{
  "requestedClaims": [
    {
      "type": "asset",
      "meta": {
        "id": "coupon",
      },
      "asset": "did:abt:zje1uzZTCZN551EWGLyyCEW9AM2wAdjymfHb"
    }
  ]
}
```

Signature

Signature is a commonly used claim between the WALLET and the DAPP. When the DAPP need to guide the WALLET user through a certain business operation, they will probably end up with signing and broadcasting a transaction to the blockchain. In this situation, the DAPP usually assembles a transaction and sends to the user and asks for signature. The DAPP is supposed to send the origin full payload and the type URL of the transaction to the WALLET so that the WALLET can validate the hash and display transaction to the user.

Besides the common fields among all types of claim items, the Signature claim item has following extra fields:

- `typeUrl` : The type url of this transaction. This field helps the WALLET to decode and display the raw transaction.
- `origin` : The Base58 encoded payload of the transaction. This payload should be decoded by using the `typeUrl` .
- `method` : The Hash method to use to generate the message digest of the `origin` . This value should be equal to the hash method in user's DID.
- `digest` : The message digest of the `origin` . This is the data that the WALLET signs. WALLET will validate if this digest matches with the origin data.
- `sig` : The signature to return.

```
{
  "requestedClaims": [
    {
      "type": "signature",
      "description": "Please sign this exchange transaction.",
      "typeUrl": "fg:t:transaction",
      "origin": "base58 encoded data",
      "method": "sha3",
      "digest": "base58 encoded digest",
      "meta": {
        "id": 12345
      }
    }
  ]
}
```

The WALLET shall return the signature in following format:

```
{
  "requestedClaims": [
    {
      "type": "signature",
      "typeUrl": "fg:t:transaction",
      "origin": "base58 encoded data",
      "method": "sha3",
      "digest": "base58 encoded digest",
      "meta": {
        "id": 12345
      },
      "sig": "the value to return"
    }
  ]
}
```

Use Cases

The ABT DID Authentication protocol is a generic peer-to-peer protocol that can be used in any case where authentication is required. It can be used in but is not limited to following scenarios:

- User registration.
- User logon.
- Signing documents.
- Requesting/issuing certificate.
- Applying for VISA.
- Peer-to-peer information exchange.

Registry Blockchain (TBD)

Registry blockchain is the place where DAPP DID should be registered. It is the decentralized authority guiding wallets whether or not the DAPP it is asking for is trustworthy. A registry blockchain should provide at least following information of an application: `trustLevel` .

Trust level

Trust level is a number to relatively show how trustworthy an DAPP is. The registry blockchain is responsible for maintaining the trust level of an DAPP. For example, an DAPP can stake ABT token on ABT chain to increase its trust level. If the DAPP did something evil, it will be punished, and its trust level will drop through voting.

APIs

WALLET APIs

- Calculates the userDid for this appDid.

  ```
  cal_userDid(root_user_sk, appDid) returns userDid
  ```

Registry blockchain side APIs

- The function called by the WALLET to get the metadata of `appDid`

  ```
  getAppInfo(appDid) returns {trust_level, app_info}
  ```

ForgeSDK

- Helper function to construct the encoded challenge to be signed.

```
construct_challenge(alg, appDid, nbf, exp, iat, callback, claims \\ [])
returns  challenge
```

- Helper function to verify the signature and DID of a challenge.

```
verify_challenge(challenge, pk)
verify_did(pk, did)
```

References:

- iOS Universal Link
- Android App Link
- URL Encoded Form Data
- JWT

Appendix

DID Role Type

DID Role Type	Value	Comments
`account`	0	
`node`	1	Fixed to `sha2` and `ed25519`
`device`	2	
`application`	3	
`smart_contract`	4	
`bot`	5	
`asset`	6	
`stake`	7	
`validator`	8	Fixed to `sha2` and `ed25519`
`group`	9	

DID Role Type	Value	Comments
`tx`	10	
`tether`	11	Fixed to `sha2` and `ed25519`
`swap`	12	Fixed to `sha2` and `ed25519`
`delegate`	13	
`any`	63	

Hash Algorithm

Hash Algorithm	Value	Comments
`keccak`	0	
`sha3`	1	
`keccak_384`	2	
`sha3_384`	3	
`keccak_512`	4	
`sha3_512`	5	
`sha2`	6	Only for `node` , `validator` , `tether` and `swap`

DSA Algorithm

DSA Algorithm	Value	Comments
`ed25519`	0	
`secp256k1`	1	

Profile items

Name	Type	Comments
`billingAddress`	address	
`birthday`	string	
`companyAddress`	address	

Name	Type	Comments
`companyName`	string	
`driverLicense`	string	
`firstName`	string	
`fullName`	string	
`gender`	string	
`highestEducationDegree`	string	
`homeAddress`	string	
`homePhone`	string	
`languages`	string	
`lastName`	string	
`locale`	string	
`mailingAddress`	address	
`maritalStatus`	string	
`middleName`	string	
`mobilePhone`	string	
`nationalId`	string	
`nationality`	string	
`passport`	string	
`personalEmail`	string	
`photo`	string	
`placeOfBirth`	string	
`primaryOccupation`	string	
`socialSecurityNumber`	string	
`taxpayerIdNumber`	string	
`timezone`	string	

Name	Type	Comments
`workEmail`	string	
`workPhone`	string	

Releases

No releases published

Packages

No packages published

Contributors 3

- **dingpl716** Peiling Ding
- **sunboshan**
- **mave99a** Robert Mao

Environments 1

github-pages Active

Languages

- **Makefile** 100.0%