# **Appendix 2 - Version 1.3 \_12/12/2023**

The following sections of code are taken from the source code that was amended so as to run the proof of concept. Each of the sections are delineated by a **~~~~~~** and each section of source code have interlaced screen shots from the code that was ran and committed to my GitHub repository.

# **Abstraction**

## **credential.ts**

This file typically contains constant values that are used throughout the project. Constants can include configuration settings, API endpoints, error codes, or any other values that remain unchanged during runtime.

import {

Credential,

RevocationBitmap,

JwsSignatureOptions,

Jwt,

} from "@iota/identity-wasm/node";

import { Identity } from "./identity";

import { CredentialState } from "../types";

import { Build } from "./utils";

export class BuildCredential extends Build<CredentialState, Jwt> {

withSubject(subject: Identity) {

return this.update((state) => {

state.subject = subject;

return state;

});

}

**//INSIDE OF THE SCRIPT**

The script below imports several types and classes from different modules. These imports include Credential, RevocationBitmap, JwsSignatureOptions, Jwt from the @iota/identity-wasm/node module, Identity from the local ./identity module, and CredentialState from the local ./types module. Additionally, it imports a utility function called Build from the local ./utils module.

The BuildCredential class has a method called withSubject which takes an argument of type Identity. This method updates the state of the BuildCredential instance by setting the subject property to the provided Identity object. It uses the update method inherited from the Build class to update the state in an immutable way.

A screen shot of a computer program

Description automatically generated

The withProperty method takes in two parameters: property and value. This method is part of a class or object that has a state with a properties object.

The purpose of the withProperty method is to update the properties object in the state by adding or modifying a specific property with the given value. It follows an immutable pattern by returning a new state object instead of modifying the existing one directly.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withProperty(property: string, value: string) {

return this.update((state) => {

state.properties[property] = value;

return state;

});

}

The withProperty method is defined with the property and value parameters. It uses the update methodto update the state. Inside the update method, a callback function is passed that takes the current state as an argument (state). Within the callback function, the properties object in the state is accessed using the property parameter as the key, and the value parameter is assigned to it. The modified state object is then returned from the callback function. The withProperty method returns the updated state object.

A computer screen with colorful text

Description automatically generated

The script provided is a method called withVerificationMethod in a TypeScript class or module. This method takes a fragment parameter of type string.

The purpose of this method is to update the verificationMethod property of the current state. It does this by utilising the update method, which is likely part of a state management library or framework.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withVerificationMethod(fragment: string) {

return this.update((state) => {

state.verificationMethod = fragment;

return state;

});

}

A breakdown of the script is shown below:

The withVerificationMethod method is called with a fragment parameter, which represents the new value for the verificationMethod property. Inside the method, the update method is invoked, passing a callback function as an argument. The callback function receives the current state as a parameter. Within the callback function, the verificationMethod property of the state is updated with the value of the fragment parameter. The updated state is then returned from the callback function. Finally, the withVerificationMethod method returns the updated state.

This script follows a functional programming approach by utilising a callback function and immutability. It ensures that the state is not directly mutated but instead returns a new state object with the updated verificationMethod property.

A computer screen with text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

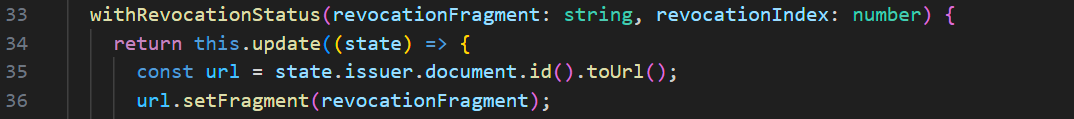
withRevocationStatus(revocationFragment: string, revocationIndex: number) {

return this.update((state) => {

const url = state.issuer.document.id().toUrl();

url.setFragment(revocationFragment);

The withRevocationStatus script is a function that takes two parameters: revocationFragment (a string) and revocationIndex (a number). This function is likely part of a larger codebase or application that deals with revocation status in the context of IOTA.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const revocationStatus = {

id: url.toString(),

type: RevocationBitmap.type(),

revocationBitmapIndex: revocationIndex.toString(),

};

state.revocationStatus = revocationStatus;

return state;

});

}

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

async create() {

return this.finalize(async (state) => {

const credential = new Credential({

issuer: state.issuer.document.id(),

credentialSubject: {

id: state.subject.document.id(),

...state.properties,

},

credentialStatus: state.revocationStatus,

});

console.log("Credential:", JSON.stringify(credential.toJSON(), null, 2));

const jwt = await state.issuer.document.createCredentialJwt(

state.issuer.storage,

state.verificationMethod,

credential,

new JwsSignatureOptions()

);

console.log("Credential token:", JSON.stringify(jwt.toJSON(), null, 2));

return jwt;

});

}

}

The create() function is defined with the async keyword, indicating that it is an asynchronous function. It returns a promise that resolves to a JSON Web Token (JWT).

Within the function, there is a call to this.finalize(), which is assumed to be a method defined elsewhere. The finalize() method takes an asynchronous callback function as an argument, which is defined using the async keyword.

Inside the callback function, a new Credential object is created using the Credential class. The Credential object is initialized with various properties, such as the issuer, credential subject, and credential status. The ...state.properties syntax is used to spread the properties from the state object.

A screen shot of a computer code

Description automatically generated

After creating the Credential object, it is logged to the console using console.log() and JSON.stringify() to convert it to a readable format.

Next, there is a call to state.issuer.document.createCredentialJwt(), which is assumed to be a method defined elsewhere. This method generates a JWT using the provided parameters: the issuer's storage, verification method, credential object, and JwsSignatureOptions. The resulting JWT is logged to the console using console.log() and JSON.stringify(). Finally, the JWT is returned from the create() function.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

## **identity.ts**

**//INSIDE OF THE SCRIPT**

**Organising identity**

The primary purpose of an identity.ts file is to define and export classes, interfaces, or types that represent various aspects of identity within the application. This can include user authentication, authorisation, Access Control and other related functionalities.

import {

IotaIdentityClient,

JwkMemStore,

KeyIdMemStore,

IotaDocument,

JwsAlgorithm,

MethodScope,

RevocationBitmap,

Storage,

} from "@iota/identity-wasm/node";

import { Client, Utils } from "@iota/sdk-wasm/node";

import { BuildCredential } from "./credential";

import { CreatePresentationResponse } from "./presentation";

import {

IdentityWalletOptions,

IdentityState,

CoverStorageDeposit,

RevocationState,

} from "../types";

import { Build, deriveAddressFromMnemonic } from "./utils";

**//INSIDE OF THE SCRIPT**

The provided TypeScript code is a class called IdentityWallet that is used to interact with the IOTA Identity framework. It imports various modules and classes from the @iota/identity-wasm/node, @iota/sdk-wasm/node, and other libraries.

The script defines a class called IdentityWallet that serves as the main component of the Identity Wallet. It has a constructor that takes an IdentityWalletOptions object as a parameter. Inside the constructor, it initializes an instance of the IotaIdentityClient class using the options.client object.

The IotaIdentityClient class is responsible for interacting with the IOTA Identity network. It provides methods for creating and managing identities, generating and verifying credentials, and handling other identity-related operations.

The script also imports other utility functions and classes such as BuildCredential, CreatePresentationResponse, Build, and deriveAddressFromMnemonic. These utilities are likely used for constructing and manipulating credentials, presentations, and other data structures related to the Identity Wallet.

Overall, the script sets up the foundation for an Identity Wallet by initializing the necessary components and dependencies. It provides a starting point for implementing the functionality of the wallet, such as creating and managing identities, generating credentials, and interacting with the IOTA Identity network.

A screen shot of a computer program

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class IdentityWallet {

private client: IotaIdentityClient;

constructor(options: IdentityWalletOptions) {

this.client = new IotaIdentityClient(options.client);

}

The IdentityWallet class is defined using the export keyword, which allows it to be imported and used in other parts of the codebase. It has a private property client of type IotaIdentityClient, which will be used to interact with the IOTA Identity network.

The constructor of the IdentityWallet class takes an options parameter of type IdentityWalletOptions. This parameter is used to configure the IotaIdentityClient instance that will be created and assigned to the client property. The options.client argument passed to the IotaIdentityClient constructor is responsible for providing the necessary configuration for the client, such as the network endpoint and authentication credentials.

By encapsulating the IotaIdentityClient instance within the IdentityWallet class, developers can interact with the IOTA Identity functionality using a higher-level and more intuitive API. This abstraction simplifies the usage of the IOTA Identity features and allows for better code organization and maintainability.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

static new() {

return new BuildIdentityWallet({});

}

generateIdentity() {

const action = async () => {

const network = await this.client.getNetworkHrp();

const mnemonic = Utils.generateMnemonic();

const storage = new Storage(new JwkMemStore(), new KeyIdMemStore());

const address = await deriveAddressFromMnemonic(network, mnemonic);

const document = new IotaDocument(network);

console.log("Address:", address);

return {

client: this.client,

mnemonic,

storage,

address,

document,

};

};

const state = action();

return new BuildIdentity(state);

}

}

The provided TypeScript code appears to be a part of a class or module.

**Method 1:** new()

The new() method is a static method that returns a new instance of the BuildIdentityWallet class. It takes an empty object as an argument ({}) and initializes a new BuildIdentityWallet instance. The purpose of this method is to create a new instance of the BuildIdentityWallet class.

**Method 2:** generateIdentity()

The generateIdentity() method is responsible for generating an identity using the BuildIdentity class. It performs the following steps:

Retrieves the network HRP (Human Readable Part) using the getNetworkHrp() method of the client object. Generates a mnemonic using the generateMnemonic() method of the Utils class. Creates a new instance of the Storage class, passing in a JwkMemStore and a KeyIdMemStore as arguments. Derives an address from the mnemonic and the network using the deriveAddressFromMnemonic() function. Creates a new instance of the IotaDocument class, passing in the network as an argument. Logs the generated address to the console.

Returns an object containing the following properties:

* client: The client object.
* mnemonic: The generated mnemonic.
* storage: The Storage instance.
* address: The derived address.
* document: The IotaDocument instance.

In summary, the generateIdentity() method generates an identity by performing various operations such as retrieving the network HRP, generating a mnemonic, deriving an address, and creating instances of different classes. It then logs the generated address and returns an object containing the necessary information for the identity. Finally, it creates a new instance of the BuildIdentity class using the returned object.

This script provides a convenient way to create a new instance of the BuildIdentityWallet class and generate an identity using the BuildIdentity class.

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program code

Description automatically generated

The provided TypeScript code is a class called BuildIdentityWallet that extends the Build class. This class is responsible for building an IdentityWallet object with the given options.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

class BuildIdentityWallet extends Build<IdentityWalletOptions, IdentityWallet> {



withClient(client: Client) {

return this.update((state) => {

state.client = client;

return state;

});

}

The withClient method is defined within the BuildIdentityWallet class. It takes a client parameter of type Client. This method is used to set the client property of the state object. It uses the update method inherited from the Build class to update the state object with the new client value.

A black screen with yellow and white spots

Description automatically generated with medium confidence

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

create() {

return this.finalize((state) => {

return new IdentityWallet(state);

});

}

}

The create method is defined within the BuildIdentityWallet class. It does not take any parameters. This method is used to finalize the building process and return a new IdentityWallet object based on the current state object.

In summary, the BuildIdentityWallet class provides a way to build an IdentityWallet object with specific options. It has a withClient method to set the client property and a create method to finalize the building process and return the built IdentityWallet object.

A computer screen with text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

class BuildIdentity extends Build<IdentityState, Identity> {

The provided TypeScript code is a class called BuildIdentity that extends the Build class. It is used to build an identity by defining various methods and properties.



The BuildIdentity class is defined, which extends the Build class. It takes two type parameters: IdentityState and Identity. This means that the BuildIdentity class is specific to building an identity with a particular state and identity type.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withVerificationMethod(fragment: string) {

return this.update(async (state) => {

await state.document.generateMethod(

state.storage,

JwkMemStore.ed25519KeyType(),

JwsAlgorithm.EdDSA,

fragment,

MethodScope.VerificationMethod()

);

return state;

});

}

1. The withVerificationMethod method is defined within the BuildIdentity class. It takes a fragment parameter of type string. This method is used to add a verification method to the identity being built.
2. Inside the method, an asynchronous update operation is performed on the state object. The state object represents the current state of the identity being built.
3. Within the update operation, the generateMethod method is called on the state.document object. This method generates a method for the identity using the provided parameters: state.storage, JwkMemStore.ed25519KeyType(), JwsAlgorithm.EdDSA, fragment, and MethodScope.VerificationMethod().
4. Finally, the updated state object is returned.

In summary, the BuildIdentity script provides a class that extends the Build class and includes a method withVerificationMethod to add a verification method to the identity being built. This script can be used to build identities with specific states and identity types.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withRevocationService(fragment: string) {

return this.update(async (state) => {

const revocationBitmap = new RevocationBitmap();

const serviceUrl = state.document.id().join(`#${fragment}`);

const service = revocationBitmap.toService(serviceUrl);

state.document.insertService(service);

return state;

});

}

The withRevocationService script is a TypeScript function that takes a fragment string as a parameter. It is used to add a revocation service to a document state.

A screen shot of a computer code

Description automatically generated

1. The withRevocationService method takes a fragment parameter, which is a string representing a fragment of the document.
2. Inside the method, it calls the update method, which is likely part of a larger class or framework. The update method takes a callback function as an argument.
3. The callback function is an asynchronous function that receives the current state as a parameter. It performs the following steps:
   1. Creating a new instance of the RevocationBitmap class, which is likely responsible for managing revocation information.
   2. Constructs a serviceUrl by joining the document's ID with the provided fragment using the join method.
   3. Converts the serviceUrl into a revocation service using the toService method of the revocationBitmap instance.
   4. Inserts the revocation service into the state.document object.
   5. Finally, it returns the updated state.
4. The withRevocationService method itself returns the result of the update method, which is likely a promise or an observable.

In summary, the withRevocationService script adds a revocation service to a document by updating the document's state. It utilizes the RevocationBitmap class to create the revocation service and inserts it into the document's services. The script follows a functional programming style by using an asynchronous callback function within the update method.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withStorageDepositCovered(coverStorageDeposit: CoverStorageDeposit) {

The withStorageDepositCovered script is a TypeScript function that takes a coverStorageDeposit parameter of type CoverStorageDeposit. This function is used to ensure that the storage deposit required for a specific operation is covered before executing that operation.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

return this.update(async (state) => {

The function uses the update method, which is likely part of a larger framework or library. This method allows for updating the state of an object or system. In this case, it takes an asynchronous callback function (state) => { ... } as an argument.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

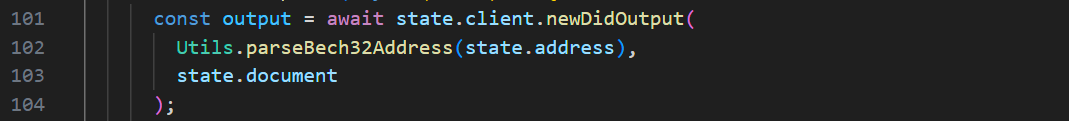
const output = await state.client.newDidOutput(

Utils.parseBech32Address(state.address),

state.document

);

Within the callback function, the script performs the next step. It calls the newDidOutput method on the state.client object. This method likely creates a new output for a decentralized identifier (DID) based on the provided address and document.



const rentStructure = await state.client.getRentStructure();

const tokensRequired = Utils.computeStorageDeposit(output, rentStructure);

await coverStorageDeposit(state.address, tokensRequired);

return state;

});

}

It then retrieves the rent structure by calling the getRentStructure method on the state.client object. The rent structure likely defines the cost of storage for the given system. It calculates the number of tokens required to cover the storage deposit by calling the computeStorageDeposit method on the Utils object. This method takes the output and rent structure as arguments. It covers the storage deposit by calling the coverStorageDeposit function with the address and tokens required as arguments. Finally, returning the updated state.

A screen shot of a computer program

Description automatically generated

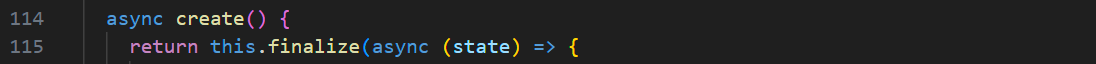
Overall, the withStorageDepositCovered script ensures that the storage deposit is covered before executing the desired operation. It follows a pattern of updating the state within an asynchronous callback function, performing necessary calculations and operations along the way.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

async create() {

return this.finalize(async (state) => {

The next section of TypeScript code appears to be an asynchronous function named create().



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

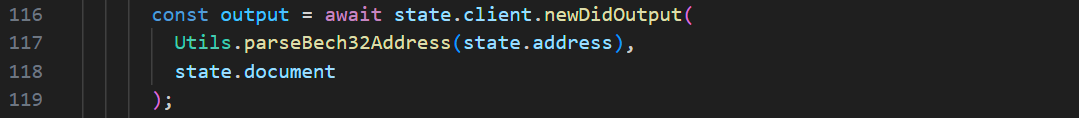
const output = await state.client.newDidOutput(

Utils.parseBech32Address(state.address),

state.document

);

In this snippet of code, the newDidOutput() method is called on the state.client object. It takes two arguments: the result of calling Utils.parseBech32Address() with state.address as the argument, and state.document. The await keyword indicates that the function will wait for the newDidOutput() method to complete before proceeding.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

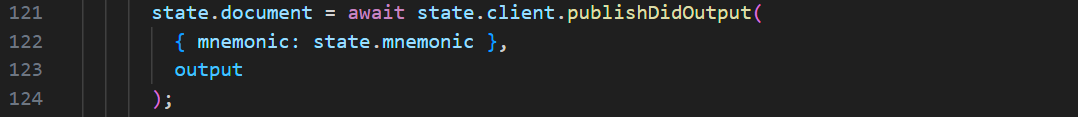
state.document = await state.client.publishDidOutput(

{ mnemonic: state.mnemonic },

output

);

Here, the publishDidOutput() method is called on the state.client object. It takes two arguments: an object with a mnemonic property set to state.mnemonic, and the output variable from the previous snippet. Similar to the previous snippet, the await keyword ensures that the function waits for the publishDidOutput() method to finish before moving on.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

console.log("DID:", state.document.id().toString());

Line 125 logs a message to the console, displaying the string "DID:" followed by the result of calling the id() method on state.document and converting it to a string using toString().



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

console.log(

"DID document:",

JSON.stringify(state.document.toJSON(), null, 2)

);

The console.log function is used to print the DID document to the console. It takes two arguments: a string "DID document:" and the serialized JSON representation of the state.document object.

The JSON.stringify function is used to convert the state.document object to a JSON string. The second argument null is used to specify that no replacer function should be applied during the serialization process. The third argument 2 is used to add indentation to the JSON string, making it more readable.

By logging the DID document to the console, developers can easily inspect and verify the contents of the document during the execution of the identity.ts command. This can be helpful for debugging purposes or for understanding the structure and content of the DID document.

Overall, the console.log statement in the identity.ts command provides a convenient way to display the DID document and its properties, aiding in the development and testing process.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

return new Identity(state);

});

}

}

Then the function returns a new Identity object instantiated with the state variable as the argument. This suggests that the create() function is part of a larger class or module and is responsible for creating a new identity.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

class BuildRevocation extends Build<RevocationState, void> {

withRevocationStatus(revocationFragment: string, revocationIndex: number) {

return this.update(async (state) => {

state.issuer.document.revokeCredentials(

revocationFragment,

revocationIndex

);

The BuildRevocation class is a part of the IOTA Identity library, which is a framework for managing decentralized identities on the IOTA Tangle. This class is responsible for building and managing the revocation status of credentials issued by an identity.

The primary purpose of the BuildRevocation class is to provide a convenient way to update the revocation status of a credential. It extends the Build class, which is a generic class provided by the IOTA Identity library for managing the state of an identity.

The withRevocationStatus method within the BuildRevocation class takes two parameters: revocationFragment and revocationIndex. These parameters represent the revocation fragment and index of the credential that needs to be revoked.

Inside the withRevocationStatus method, the update function is called with an asynchronous callback function. This function updates the state of the identity by invoking the revokeCredentials method on the issuer.document object. The revokeCredentials method is responsible for revoking the specified credential by passing the revocationFragment and revocationIndex as arguments.

A screen shot of a computer program

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const output = await state.issuer.client.updateDidOutput(

state.issuer.document

);

state.output = output;

return state;

});

}

The purpose of the code snippet above is to update the output of the identity.ts command. The following is the functionality of the code.

const output = await state.issuer.client.updateDidOutput(state.issuer.document);

This line of code calls the updateDidOutput function on the client object of the issuer state. The updateDidOutput function takes the issuer.document as an argument and returns the updated output. The await keyword ensures that the function call is asynchronous and waits for the result before proceeding.

The updateDidOutput function is responsible for updating the output of the identity.ts command. It might perform various operations, such as generating a new output based on the provided document or updating an existing output with new information.

state.output = output;

After obtaining the updated output from the updateDidOutput function, the code assigns it to the state.output variable. This allows the updated output to be accessed and used later in the code.

return state;

Finally, the code returns the updated state object. This ensures that any changes made to the state object, including the updated output, are propagated and available for further processing or usage.

A screen shot of a computer

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withStorageDepositCovered(coverStorageDeposit: CoverStorageDeposit) {

return this.update(async (state) => {

if (!state.output) throw "Output not updated.";

const rentStructure = await state.issuer.client.getRentStructure();

const tokensRequired = Utils.computeStorageDeposit(

state.output,

rentStructure

);

The function takes a parameter coverStorageDeposit which represents the storage deposit to be covered.

The function then calls the update method, which is likely part of the larger framework or library being used. This method allows for updating the state of the identity or performing some operation on it.

Inside the update method, the function checks if the state.output exists. If it doesn't, an exception is thrown with the message "Output not updated." This check ensures that the necessary output for the operation is present.

The function then calls getRentStructure on the state.issuer.client object. This method likely retrieves the rent structure information, which defines the cost of storing data on the IOTA network.

Next, the function calls Utils.computeStorageDeposit with the state.output and rentStructure as parameters. This function calculates the amount of tokens required as a storage deposit based on the output and rent structure. The tokensRequired variable will hold this calculated value.

The primary purpose of the withStorageDepositCovered function is to calculate and ensure that the required storage deposit for a specific operation is covered. This is important because the IOTA network requires users to pay a storage deposit to store data on the network. By checking and covering the storage deposit, the function ensures that the operation can proceed without any issues related to insufficient funds for storage.

A screen shot of a computer code

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

await coverStorageDeposit(state.issuer.address, tokensRequired);

return state;

});

}

In the given code snippet, the await coverStorageDeposit(state.issuer.address, tokensRequired); line serves a specific purpose within the identity.ts file. The await keyword is used to pause the execution of the code until the promise returned by the coverStorageDeposit() function is resolved. This ensures that the subsequent code is not executed until the promise is fulfilled.

The coverStorageDeposit() function is likely a custom function defined elsewhere in the codebase. Without further context, we can assume that this function is responsible for covering the storage deposit required for the identity issuer's address. The state.issuer.address parameter passed to the coverStorageDeposit() function represents the address of the identity issuer. This address is used to identify the specific identity being processed.

The tokensRequired parameter represents the number of tokens required to cover the storage deposit. This value is likely calculated based on certain criteria or business rules. By awaiting the coverStorageDeposit() function, the code ensures that the storage deposit is covered before proceeding with any further operations. This is important because the subsequent code might rely on the successful completion of the storage deposit coverage.

Once the coverStorageDeposit() function resolves, the code continues execution, and the return state; statement is reached. This line returns the updated state object, which might contain additional information or changes made during the execution of the coverStorageDeposit() function.

A screen shot of a computer

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

async create() {

return this.finalize(async (state) => {

const rentStructure = await state.issuer.client.getRentStructure();

const output = await state.issuer.client.client.buildAliasOutput({

...state.output,

amount: Utils.computeStorageDeposit(state.output, rentStructure),

aliasId: state.output.getAliasId(),

unlockConditions: state.output.getUnlockConditions(),

});

The create() function is defined as an asynchronous function, indicating that it will perform operations that may take some time to complete. The finalize() function is called with an asynchronous callback function (state) => { ... }. This callback function receives the state object as a parameter, which contains the necessary information for creating the identity.

Inside the callback function, the getRentStructure() function is called on the state.issuer.client object. This function retrieves the rent structure from the IOTA network, which is used to calculate the storage deposit required for the identity. The buildAliasOutput() function is called on the state.issuer.client.client object. This function constructs the output for the identity transaction, including the necessary parameters such as the amount of storage deposit, alias ID, and unlock conditions.

The ...state.output syntax is used to spread the properties of the state.output object into the buildAliasOutput() function call. This ensures that all the properties of the state.output object are included in the constructed output. The amount property of the output is set using the Utils.computeStorageDeposit() function, which calculates the required storage deposit based on the state.output and the rent structure obtained earlier.

The aliasId property of the output is set to the alias ID obtained from the state.output. The unlockConditions property of the output is set to the unlock conditions obtained from the state.output.

A screen shot of a computer code

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

state.issuer.document = await state.issuer.client.publishDidOutput(

{ mnemonic: state.issuer.mnemonic },

output

);

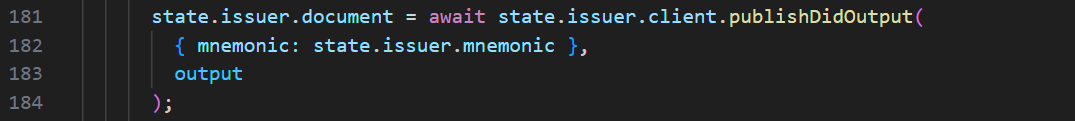
The purpose of the mentioned portion of the identity.ts command is to publish the output of the DID (Decentralized Identifier) document to the IOTA Tangle using the IOTA Identity library.

The state.issuer.client refers to the IOTA Identity client instance that is responsible for interacting with the IOTA Tangle. The publishDidOutput() is a method provided by the IOTA Identity library that allows the user to publish the output of the DID document to the IOTA Tangle.

The publishDidOutput() method takes two parameters:

{ mnemonic: state.issuer.mnemonic }: This parameter is an object that contains the mnemonic of the issuer's DID. The mnemonic is used to derive the private key for signing the DID document.

output: This parameter represents the output of the DID document that needs to be published to the IOTA Tangle. By calling publishDidOutput() with the appropriate parameters, the code publishes the output of the DID document to the IOTA Tangle. This ensures that the DID document is stored in a decentralized and immutable manner, providing trust and verifiability.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

console.log(

"DID document:",

JSON.stringify(state.issuer.document.toJSON(), null, 2)

);

});

}

}

The console.log function is used to print the DID document to the console. It takes two arguments: a string "DID document:" and the serialized JSON representation of the state.issuer.document object.

The JSON.stringify function is used to convert the state.issuer.document object to a JSON string. The second argument null is used to specify that no replacer function should be applied during the serialization process. The third argument 2 is used to add indentation to the output JSON string, making it more readable.

By logging the DID document to the console, developers can easily inspect and verify the contents of the document during the execution of the identity.ts command. This can be helpful for debugging purposes or for understanding the structure and content of the generated DID document.

A computer screen with text

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class Identity {

client: IotaIdentityClient;

address: string;

storage: Storage;

document: IotaDocument;

mnemonic: string;

constructor(state: IdentityState) {

Object.assign(this, state);

}

The given TypeScript code defines a class called Identity. This class has several properties and methods that are used to generate credentials and presentation responses. The Identity class has the following properties:

1. client: An instance of the IotaIdentityClient class.
2. address: A string representing the address of the identity.
3. storage: An instance of the Storage class.
4. document: An instance of the IotaDocument class.
5. mnemonic: A string representing the mnemonic associated with the identity.

A computer screen with text

Description automatically generated

The constructor of the Identity class takes a state parameter of type IdentityState. It uses Object.assign to assign the properties of the state object to the this object.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

generateCredential() {

const action = async () => {

const state = {

issuer: this,

properties: {},

};

return state;

};

const state = action();

return new BuildCredential(state);

}

The generateCredential method is defined within the Identity class. It creates a new action function that returns a state object. The state object has two properties: issuer, which is set to the current instance of the Identity class, and properties, which is an empty object.

The action function is an asynchronous function, indicated by the async keyword. It returns the state object.

The generateCredential method then calls the action function and assigns the returned state object to the state variable. Finally, it creates a new instance of the BuildCredential class, passing the state object as an argument, and returns it.

A computer screen with colorful text

Description automatically generated with medium confidence

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

generatePresentationResponse() {

return new CreatePresentationResponse({

holder: this,

credentials: [],

});

}

The generatePresentationResponse method is another method defined within the Identity class. It creates a new instance of the CreatePresentationResponse class, passing an object as an argument. The object has two properties: holder, which is set to the current instance of the Identity class, and credentials, which is an empty array.

In summary, the given TypeScript code defines a class called Identity with properties and methods for generating credentials and presentation responses. It utilizes other classes such as IotaIdentityClient, Storage, IotaDocument, BuildCredential, and CreatePresentationResponse to achieve its functionality.

A computer screen with text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

generateRevocation() {

const action = async () => {

const state = {

issuer: this,

};

return state;

};

const state = action();

return new BuildRevocation(state);

}

}

The function starts by defining an asynchronous arrow function called action. This function is assigned to the action constant. The purpose of this function is to encapsulate the logic for generating the revocation state. Inside the action function, a state object is created. This object contains the issuer property, which is set to this. In the context of this code snippet, this refers to the current instance of the identity.

The action function then returns the state object. Outside the action function, the action function is invoked and its result is assigned to the state constant. This step executes the logic defined in the action function and retrieves the generated revocation state.

Finally, the function returns a new instance of the BuildRevocation class, passing the state object as an argument. This indicates that the primary purpose of the generateRevocation() function is to create a new BuildRevocation object with the generated revocation state.

A computer screen with text and symbols

Description automatically generated with medium confidence**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

## **presentation.ts**

**Organising presentations**

The primary purpose of a presentations.ts file is to define and implement the user interface (UI) components and their behaviour. These components are responsible for rendering the visual elements of the application and handling user interactions.

**//INSIDE OF THE SCRIPT**

The script begins with import statements that bring in the necessary modules and dependencies. These imports allow the script to use the functionality provided by these modules.

import {

Jwt,

Presentation,

JwsSignatureOptions,

JwtPresentationOptions,

FailFast,

JwsVerificationOptions,

JwtCredentialValidationOptions,

JwtCredentialValidator,

JwtPresentationValidationOptions,

JwtPresentationValidator,

SubjectHolderRelationship,

**JWT and Presentation Types**

The script defines various types related to JWTs and their presentation. These types provide a structured way to represent and manipulate JWTs and their associated data. Examples of these types include Jwt, Presentation, and SubjectHolderRelationship.

**JWS Signature Options**

The script includes the definition of JwsSignatureOptions, which represents the options for signing a JWT using JSON Web Signature (JWS). These options allow customization of the signing process, such as choosing the signing algorithm and providing the signing key.

**JWT Presentation Options**

The script defines JwtPresentationOptions, which represents the options for presenting a JWT. These options control how the JWT is presented and validated, such as specifying the required claims or audience.

**FailFast**

The script includes the FailFast type, which is used to indicate whether the JWT presentation or validation process should stop immediately upon encountering an error or continue processing.

**JWS Verification Options**

The script defines JwsVerificationOptions, which represents the options for verifying the signature of a JWT using JWS. These options allow customization of the signature verification process, such as specifying the expected signing algorithm and providing the verification key.

**JWT Credential Validation Options**

The script includes JwtCredentialValidationOptions, which represents the options for validating the credentials of a JWT. These options allow customization of the credential validation process, such as specifying the required issuer or audience.

**JWT Credential Validator**

The script defines JwtCredentialValidator, which is responsible for validating the credentials of a JWT. This validator uses the provided options to perform the necessary checks, such as verifying the issuer and audience.

**JWT Presentation Validation Options**

The script includes JwtPresentationValidationOptions, which represents the options for validating the presentation of a JWT. These options allow customization of the presentation validation process, such as specifying the required claims or audience.

**JWT Presentation Validator**

The script defines JwtPresentationValidator, which is responsible for validating the presentation of a JWT. This validator uses the provided options to perform the necessary checks, such as verifying the required claims and audience.

**Subject Holder Relationship**

The script includes the SubjectHolderRelationship type, which represents the relationship between the subject and the holder of a JWT. This type provides a way to define and enforce the relationship between the subject and the entity presenting the JWT.

In summary, the provided TypeScript code imports modules, defines types, and sets options related to JWTs and their presentation and validation. These components work together to provide a flexible and customisable way to work with JWTs in TypeScript applications.

A screen shot of a computer

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

} from "@iota/identity-wasm/node";

import {

CreatePresentationValidationState,

Predicate,

PresentationRequestOptions,

PresentationResponseState,

PresentationValidationOptions,

} from "../types";

import { Build } from "./utils";

**Imports**

The provided TypeScript code is a class definition for a PresentationRequest object. This class is used to create a presentation request, which is a request for a presentation of identity-related information. In this script, the imports include various types and options related to presentation validation and response states. Additionally, the Build import is used from a local utils file.

The CreatePresentationRequest class has three methods: withPredicate, withNonce, and create.

**Class Definition**

The script defines a class called PresentationRequest. This class represents a presentation request and contains properties and methods related to creating and managing such requests.

A screen shot of a computer

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class PresentationRequest {

nonce: string;

predicates: Predicate[];

constructor(options: PresentationRequestOptions) {

Object.assign(this, options);

}

}

The PresentationRequest class has two properties:

1. nonce: This property represents a unique identifier for the presentation request. It is of type string.
2. predicates: This property is an array of Predicate objects. Each Predicate represents a condition or requirement for the requested presentation. The Predicate type is imported from the external library.

The class also has a constructor that takes in an options parameter of type PresentationRequestOptions. The constructor assigns the properties of the options object to the corresponding properties of the PresentationRequest instance using Object.assign().

Overall, this script provides the foundation for creating and managing presentation requests in TypeScript. It utilizes imports from external libraries and defines a class with properties and a constructor to handle the request creation process.

A computer screen shot of a program code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class CreatePresentationRequest extends Build<

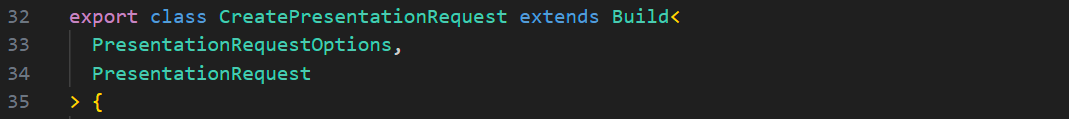
PresentationRequestOptions,

PresentationRequest

> {

**Class Definition**

The CreatePresentationRequest script is a TypeScript class that provides a convenient way to create a presentation request object. It is designed to be used in conjunction with other classes and functions to build and finalise a presentation request.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withPredicate(predicate: Predicate) {

return this.update((state) => {

state.predicates.push(predicate);

return state;

});

}

**Method:** withPredicate

The withPredicate method is used to add a predicate to the presentation request. It takes a Predicate object as a parameter and returns an updated instance of CreatePresentationRequest. Inside the method, it uses the update method to modify the state of the object by adding the predicate to the predicates array. The update method takes a callback function that receives the current state and returns the updated state.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withNonce(nonce: string) {

return this.update((state) => {

state.nonce = nonce;

return state;

});

}

**Method:** withNonce

The withNonce method is used to set the nonce of the presentation request. It takes a nonce string as a parameter and returns an updated instance of CreatePresentationRequest. Similar to the withPredicate method, it uses the update method to modify the state of the object by assigning the nonce parameter to the nonce property.

A black screen with a black background

Description automatically generated with medium confidence

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

create() {

return this.finalize(async (state) => {

return new PresentationRequest(state);

});

}

}

**Method:** create

The create method is used to finalize the creation of the presentation request. It does not take any parameters and returns a promise that resolves to a new instance of PresentationRequest. Inside the method, it uses the finalize method to asynchronously execute a callback function that receives the current state and returns a new object.

A computer screen with text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

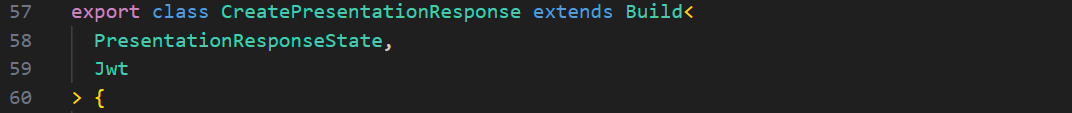
export class CreatePresentationResponse extends Build<

PresentationResponseState,

Jwt

> {

The TypeScript code provided is a class called CreatePresentationResponse that extends the Build class. It represents a response object for creating a presentation.



The CreatePresentationResponse class extends the Build class, which indicates that it inherits properties and methods from the Build class. The Build class is not provided in the code snippet, so we can assume it is defined elsewhere.

The CreatePresentationResponse class has three methods: withNonce, withVerificationMethod, and withCredential.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withNonce(nonce: string) {

return this.update((state) => {

state.nonce = nonce;

return state;

});

}

The withNonce(nonce: string) method takes a nonce parameter of type string. It updates the state object by assigning the nonce value to the state.nonce property. It then returns the updated state object.

A black background with yellow lights

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withVerificationMethod(fragment: string) {

return this.update((state) => {

state.verificationMethod = fragment;

return state;

});

}

The withVerificationMethod(fragment: string): This method takes a fragment parameter of type string. Similar to the withNonce method, it updates the state object by assigning the fragment value to the state.verificationMethod property. It also returns the updated state object.

A computer screen shot of text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withCredential(credential: Jwt) {

return this.update((state) => {

state.credentials.push(credential);

return state;

});

}

The withCredential(credential: Jwt) method takes a credential parameter of type Jwt. It updates the state object by pushing the credential object into the state.credentials array. It then returns the updated state object.

The TypeScript code provided is a class called CreatePresentationValidation that extends the Build class. This class is responsible for validating a presentation request and response using JSON Web Tokens (JWTs).

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

create() {

return this.finalize(async (state) => {

if (!state.nonce) throw "Nonce must be set.";

if (!state.verificationMethod) throw "Verification method must be set.";

if (!state.credentials.length)

throw "At least one credential must be added.";

const presentation = new Presentation({

holder: state.holder.document.id(),

verifiableCredential: state.credentials,

});

The given TypeScript code is a method called create() that returns a Promise. This method is part of a larger codebase and is responsible for creating a presentation token using the Presentation class.

The create() method starts by calling the finalize() method, which is a custom implementation. This method takes a callback function as an argument and executes it asynchronously. Inside the callback function, several checks are performed using conditional statements. If any of these checks fail, an error is thrown with an appropriate error message.

These checks ensure that the required properties (nonce, verificationMethod, and credentials) are set before proceeding. Once the checks pass, a new instance of the Presentation class is created. This class represents a verifiable presentation and requires the holder (document ID) and verifiableCredential (an array of credentials) as parameters.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

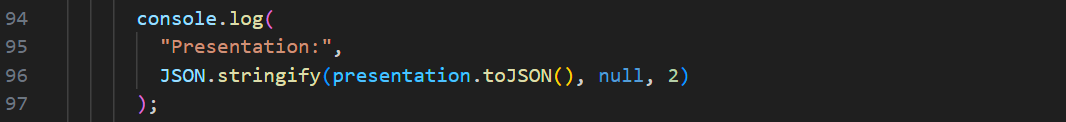
console.log(

"Presentation:",

JSON.stringify(presentation.toJSON(), null, 2)

);

The console.log() function is used to print the JSON representation of the presentation object. This is helpful for debugging and understanding the structure of the presentation. The next step involves creating a presentation token (JWT) using the createPresentationJwt() method. This method is likely part of the holder.document object and takes several arguments: the holder.storage, verificationMethod, presentation, JwsSignatureOptions, and JwtPresentationOptions. The JwsSignatureOptions and JwtPresentationOptions are likely classes that provide additional options for creating the JWT.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const jwt = await state.holder.document.createPresentationJwt(

state.holder.storage,

state.verificationMethod,

presentation,

new JwsSignatureOptions({ nonce: state.nonce }),

new JwtPresentationOptions()

);

The purpose of the mentioned code snippet is to create a JSON Web Token (JWT) for a presentation in the IOTA Identity framework. The state.holder.document.createPresentationJwt: This is a method call that belongs to the document object of the holder state. It is used to create a JWT for a presentation.

The state.holder.storage parameter represents the storage mechanism used by the holder to store the presentation. The state.verificationMethod parameter specifies the verification method to be used for the presentation. It is typically a public key or a DID (Decentralized Identifier) associated with the holder.

Along with the parameter presentation contains the actual presentation data that needs to be included in the JWT. The parameter new JwsSignatureOptions({ nonce: state.nonce })specifies additional options for the JWS (JSON Web Signature) signature. In this case, it includes a nonce value from the state object. And the new JwtPresentationOptions(): This parameter represents additional options for the JWT presentation.

A computer screen with text on it

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

console.log("Presentation token:", JSON.stringify(jwt.toJSON(), null, 2));

return jwt;

});

}

}

In the given code snippet of code above, the console.log statement is used to print the presentation token in a human-readable format. The JSON.stringify(jwt.toJSON(), null, 2) function call converts the jwt object to a JSON string with indentation of 2 spaces.

The primary purpose of this code is to provide a visual representation of the presentation token for debugging or informational purposes. By logging the presentation token, developers can easily inspect its contents and verify its correctness during the execution of the program.

The return jwt; statement is used to return the jwt object from the function. This allows the presentation token to be used or further processed by other parts of the code.

A black screen with white text

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class CreatePresentationValidation extends Build<

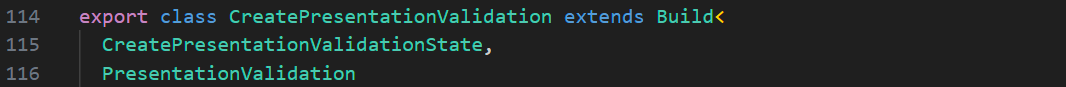
CreatePresentationValidationState,

PresentationValidation

> {

The TypeScript code defines a class called CreatePresentationValidation. This class extends the Build class and has two methods: withRequest and withResponse.

The CreatePresentationValidation class is defined as an exported class, which means it can be imported and used in other parts of the codebase. It extends the Build class, which suggests that it inherits some functionality from the Build class.



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

withRequest(request: PresentationRequest) {

return this.update((state) => {

state.request = request;

return state;

});

}

withResponse(response: Jwt) {

return this.update((state) => {

state.response = response;

return state;

});

}

The CreatePresentationValidation class has two methods: withRequest and withResponse. These methods are used to set the request and response properties of the class's state. The withRequest method takes a PresentationRequest parameter and sets the request property of the state to the provided value. Similarly, the withResponse method takes a Jwt parameter and sets the response property of the state.

A computer screen shot of a program code

Description automatically generated

Both methods use the update method to update the state of the class. The update method takes a callback function that receives the current state as a parameter and returns the updated state. In the callback function, the request or response property is set to the provided value, and then the updated state is returned.

Overall, the CreatePresentationValidation class provides a way to set the request and response properties of its state using the withRequest and withResponse methods, respectively. This can be useful for managing and validating presentation data in a TypeScript application.

In conclusion, the script provided defines a TypeScript class called CreatePresentationValidation that extends the Build class and provides methods for setting the request and response properties of its state.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

create() {

return this.finalize(async (state) => {

let validated = false;

try {

if (!state.request) throw "No request to validate.";

if (!state.response) throw "No response to validate.";

The create() method is responsible for performing validation on a request and response object. Let's break down the script into snippets to better understand its functionality.

A screen shot of a computer

Description automatically generatedIn the section of code above, the finalize() method is called with an asynchronous function as its argument. The function takes a state parameter, which is an object containing the request and response properties. If either the request or response property is false (null, undefined, etc.), an error is thrown indicating that there is no request or response to validate.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

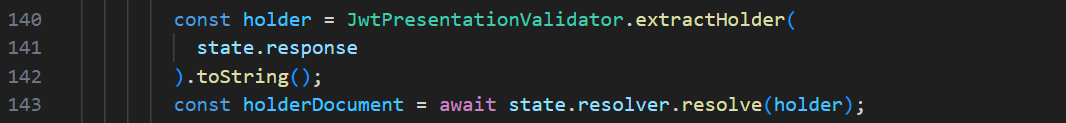
const holder = JwtPresentationValidator.extractHolder(

state.response

).toString();

const holderDocument = await state.resolver.resolve(holder);

The following section then extracts the holder from the state.response object using the JwtPresentationValidator.extractHolder() method. The holder is then converted to a string representation. The holderDocument is obtained by resolving the holder using the state.resolver.resolve() method, which likely retrieves the document associated with the holder.



Then, a validatePresentationOptions object is created using the JwtPresentationValidationOptions class. It is initialised with a presentationVerifierOptions object, which is created using the JwsVerificationOptions class. The nonce property of the presentationVerifierOptions object is set to the nonce property of the state.request object.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const validatePresentationOptions =

new JwtPresentationValidationOptions({

presentationVerifierOptions: new JwsVerificationOptions({

nonce: state.request.nonce,

}),

});

const validatedPresentation = state.presentationValidator.validate(

state.response,

holderDocument,

validatePresentationOptions

);

The state.presentationValidator.validate() method is then called with the state.response, holderDocument, and validatePresentationOptions as arguments. This likely performs some validation on the presentation using the provided options.

A screen shot of a computer program

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const validateCredentialOptions = new JwtCredentialValidationOptions({

subjectHolderRelationship: [

holder,

SubjectHolderRelationship.AlwaysSubject,

],

});

In the next step in the code, a validateCredentialOptions object is created using the JwtCredentialValidationOptions class. It is initialized with a subjectHolderRelationship property, which is an array containing the holder and SubjectHolderRelationship.AlwaysSubject values. This likely specifies the relationship between the subject and holder in the credential validation process.

A computer screen with text

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const credentials = await Promise.all(

validatedPresentation

.presentation()

.verifiableCredential()

.map((unknown) => {

const credential = unknown.tryIntoJwt();

if (!credential) throw "Credentials must be a JWT";

return credential;

})

In this snippet, the script starts by extracting the validatedPresentation and accessing its presentation property. It then retrieves the verifiableCredential array and applies the map function to each element. Within the map function, it attempts to convert each unknown credential into a JWT using the tryIntoJwt method. If the conversion fails, an exception is thrown. Otherwise, the JWT credential is returned.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

.map(async (credential) => {

const issuer =

JwtCredentialValidator.extractIssuerFromJwt(

credential

).toString();

const issuerDocument = await state.resolver.resolve(issuer);

Next, another map function is applied to the array of JWT credentials. Within this map function, the script extracts the issuer from each JWT credential using the extractIssuerFromJwt method. The issuer is then converted to a string representation. The script then awaits the resolution of the issuer document using the state.resolver.resolve method.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

return state.credentialValidator

.validate(

credential,

issuerDocument,

validateCredentialOptions,

FailFast.AllErrors

)

.credential();

})

);

After obtaining the issuer document, the script proceeds to validate the JWT credential using the state.credentialValidator.validate method. It passes the JWT credential, issuer document, validateCredentialOptions, and FailFast.AllErrors as arguments. The validation process returns a validated credential. All the validated credentials are collected using the Promise.all method, which awaits the completion of all the asynchronous operations. The resulting array of validated credentials is stored in the credentials variable.

A computer screen shot of a program

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

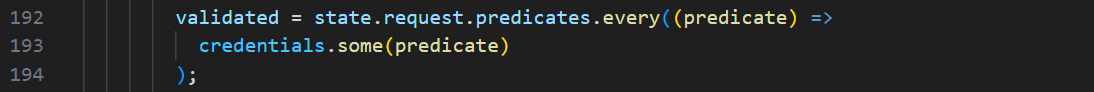
The purpose of the code snippet in question is to validate a presentation based on a set of predicates and credentials.

validated = state.request.predicates.every((predicate) =>

credentials.some(predicate)

);

In this part, the every method is used to iterate over each predicate in the state.request.predicates array. The every method returns true if all predicates return true when applied to the credentials array using the some method. Essentially, this code checks if all the required credentials are present in the credentials array.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

console.log("Presentation", validated ? "valid" : "invalid");

This line logs the result of the validation to the console. If validated is true, it logs "Presentation valid", otherwise it logs "Presentation invalid".

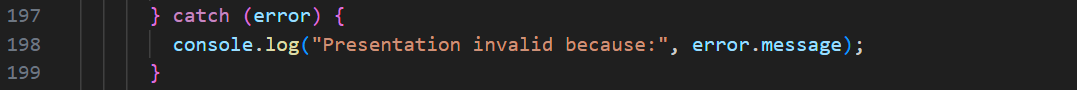
**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

} catch (error) {

console.log("Presentation invalid because:", error.message);

}

The catch block is used to handle any errors that occur during the validation process. If an error occurs, it logs the error message to the console along with the "Presentation invalid because:" prefix.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

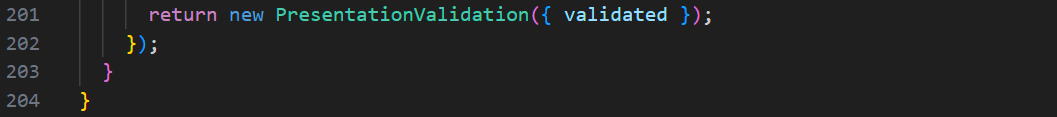
return new PresentationValidation({ validated });

});

}

}

Finally, the code returns a new instance of the PresentationValidation class with the validated value as a parameter. This allows the caller of the presenatation.ts command to access the validation result.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class PresentationValidation {

validated: boolean;

constructor(options: PresentationValidationOptions) {

Object.assign(this, options);

}

}

Lastly, the constructor of the PresentationValidation class takes an argument called options, which is of type PresentationValidationOptions. The options argument is then assigned to the current instance of the class using Object.assign(this, options). This allows for the initialisation of the validated property with the value provided in the options object.

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

## **utils.ts**

This file can contain utility functions or helper methods that can be reused across different parts of the project. It might include functions for data manipulation, string formatting, date/time operations, or other common tasks.

**Organising utils**

The primary purpose of a utils.ts file is to gather and centralise utility functions and helper classes in one place. By organising these functions in a separate file, we can easily locate and reuse them throughout the codebase. This promotes code modularity, maintainability, and reduces code duplication.

**//INSIDE OF THE SCRIPT**

import { Client, SecretManager } from "@iota/sdk-wasm/node";

import { COIN\_TYPE } from "../constants";

import { Duration, Timestamp } from "@iota/identity-wasm/node";

export { Duration } from "@iota/identity-wasm/node";

**Import Statements**

The script begins with import statements that bring in necessary dependencies. It imports the Client and SecretManager classes from the @iota/sdk-wasm/node package, the COIN\_TYPE constant from a local constants file, and the Duration and Timestamp classes from the @iota/identity-wasm/node package.

The last line exports the Duration class from the @iota/identity-wasm/node library. This allows other modules or files to import and use the Duration class directly without having to import the entire library.

A screen shot of a computer code

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export abstract class Build<T, U> {

protected state: Promise<T>;

constructor(state: T | Promise<T>) {

this.state = (async () => state)();

}

update(callback: (state: T) => Promise<T> | T): this {

this.state = this.state.then(callback);

return this;

}

finalize(callback: (state: T) => Promise<U> | U): Promise<U> {

return this.state.then(callback);

}

}

**Abstract Class Definition**

The provided TypeScript code is an abstract class called Build that takes two generic types, T and U. The class has the following members:

state: A protected property of type Promise<T>, which will hold the state of the build process.

constructor: The constructor function that initializes the state property. It accepts a parameter state of type T or Promise<T>. The constructor immediately invokes an async function that resolves the state value and assigns it to the state property.

update: A method that takes a callback function callback as a parameter. The callback function accepts the current state (state) as an argument and returns a new state value of type T or Promise<T>. The update method updates the state property by chaining the callback function to the existing state using the then method. It returns the instance of the Build class to allow method chaining.

finalize: A method that takes a callback function callback as a parameter. The callback function accepts the current state (state) as an argument and returns a final value of type U or Promise<U>. The finalize method resolves the state property using the then method and passes the resolved value to the callback function. It returns a Promise<U> representing the final result of the build process.

The Build class provides a foundation for building and updating a stateful process. It allows for method chaining and provides a way to finalize the process by returning a promise that resolves to the final result.

A screen shot of a computer program

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export async function deriveAddressFromMnemonic(

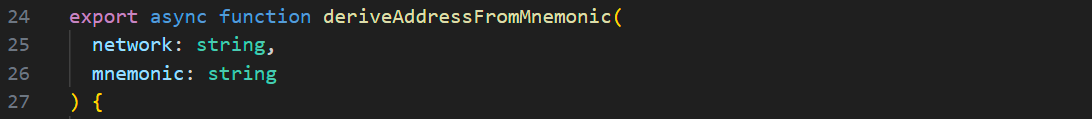
network: string,

mnemonic: string

) {

The script is a function called deriveAddressFromMnemonic that takes two parameters: network (a string) and mnemonic (a string). The function uses BIP32 to derive an address that can be controlled with the given mnemonic.

The deriveAddressFromMnemonic script is a TypeScript function that generates an address using a mnemonic and a specified network.



The script starts with the function declaration export async function deriveAddressFromMnemonic(network: string, mnemonic: string). This function takes two parameters: network and mnemonic. The network parameter represents the network on which the address will be generated, and the mnemonic parameter is the secret phrase used to derive the address.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

// Use BIP32 to derive an address that can be controlled with the mnemonic.

const addresses = await new SecretManager({

mnemonic,

}).generateEd25519Addresses({

bech32Hrp: network,

coinType: COIN\_TYPE,

accountIndex: 0,

range: { start: 0, end: 1 },

});

// Only return the first address.

return addresses[0];

}

The script creates an instance of the SecretManager class and passes the mnemonic as a parameter. The SecretManager class is responsible for generating addresses based on the provided mnemonic.

Then, the script calls the generateEd25519Addresses method on the SecretManager instance. This method generates Ed25519 addresses based on the specified parameters. The parameters include the bech32Hrp (Bech32 human-readable part) which represents the network prefix, the coinType which specifies the type of coin, the accountIndex which determines the account index, and the range which specifies the range of addresses to generate.

The generateEd25519Addresses method returns an array of addresses. However, in this script, only the first address is returned using return addresses[0].

A screen shot of a computer program

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export async function getTokensAvailable(client: Client, address: string) {

const outputIdsResponse = await client.basicOutputIds([

{ address },

{ hasExpiration: false },

{ hasTimelock: false },

{ hasStorageDepositReturn: false },

]);

const outputResponses = await client.getOutputs(outputIdsResponse.items);

In the following script the function begins by making an asynchronous call to the basicOutputIds method of the client object. This method takes an array of options as its argument. The options specify various conditions for the outputs to be retrieved. In this case, the options include filtering outputs based on the provided address, as well as excluding outputs with expiration, timelock, and storage deposit return.

A screen shot of a computer code

Description automatically generated

The basicOutputIds method returns a response object, which contains an array of output IDs. These output IDs represent the outputs that match the specified conditions. The function then proceeds to make another asynchronous call to the getOutputs method of the client object, passing the output IDs as its argument. This method retrieves the actual output data for the given output IDs.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

const tokensAvailable = outputResponses

.map((outputResponse) => BigInt(outputResponse.output.amount))

.reduce((sum, value) => sum + value, BigInt(0));

return tokensAvailable;

}

The response from the getOutputs method is an array of output responses. Each output response contains information about a specific output, including the amount of tokens it holds. The function uses the map method to extract the amount property from each output response and converts it to a BigInt value. The BigInt function is used to handle large integer values that may exceed the maximum safe integer limit in JavaScript.

The resulting array of BigInt values representing the amounts of tokens held by the outputs is then passed to the reduce method. The reduce method iterates over the array and accumulates the values by adding them together. The initial value for the accumulation is BigInt(0).

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export function delay(duration: Duration) {

const currentTime = Date.parse(Timestamp.nowUTC().toRFC3339());

const targetTime = Date.parse(

Timestamp.nowUTC().checkedAdd(duration).toRFC3339()

);

const difference = targetTime - currentTime;

return new Promise((resolve) => setTimeout(resolve, difference));

}

Finally, the function returns the total sum of tokens available as a BigInt value.

**The** delay **Function**

A screen shot of a computer code

Description automatically generated

The delay function takes a parameter duration of type Duration. It calculates the current time and the target time by using the Date.parse method and the Timestamp.nowUTC().toRFC3339() method. The difference variable is then calculated as the difference between the target time and the current time.

Finally, the function returns a Promise that resolves after the calculated difference time using the setTimeout function.

The purpose of the delay function is to introduce a delay in the execution of code. It can be used to simulate asynchronous behaviour or to pause the execution of a function for a specific duration.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

## **verifier.ts**

This file is responsible for verifying or validating certain conditions or data. It could include functions for input validation, data integrity checks, or business rule verification.

**Organising verifier**

The primary purpose of a verifier.ts file is to validate and verify the input data or the state of the program. It acts as a gatekeeper, checking if the provided data meets certain criteria or conditions before proceeding with the execution of the code. This verification process helps in preventing errors, bugs, and potential security vulnerabilities.

**//INSIDE OF THE SCRIPT**

The script provided is a TypeScript implementation of a Verifier class. This class is responsible for generating presentation requests and performing presentation validations in the context of IOTA Identity. It builds from import statements that import various modules and classes from external libraries.

import {

JwtPresentationValidator,

JwtCredentialValidator,

Resolver,

IotaIdentityClient,

EdDSAJwsVerifier,

} from "@iota/identity-wasm/node";

import { VerifierOptions } from "../types";

import {

CreatePresentationRequest,

CreatePresentationValidation,

} from "./presentation";

import { Client } from "@iota/sdk-wasm/node";

import { Build } from "./utils";

These import statements bring in the necessary dependencies for the Verifier class to function properly. The @iota/identity-wasm/node package provides the necessary modules for working with IOTA Identity, while the ./types, ./presentation, and ./utils files contain custom types, presentation creation logic, and utility functions, respectively.

**Import Statements**

A screen shot of a computer code

Description automatically generated

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

export class Verifier {

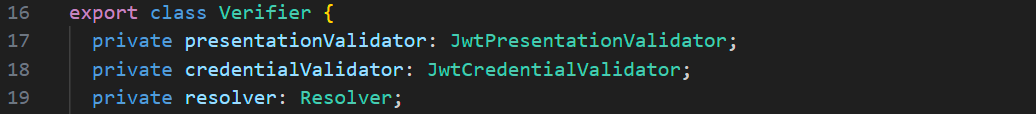
private presentationValidator: JwtPresentationValidator;

private credentialValidator: JwtCredentialValidator;

private resolver: Resolver;

The Verifier class has three private properties: presentationValidator, credentialValidator, and resolver. These properties are used to store instances of the JwtPresentationValidator, JwtCredentialValidator, and Resolver classes, respectively.

**Verifier Class**



**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

constructor(options: VerifierOptions) {

const client = new IotaIdentityClient(options.client);

const verifier = new EdDSAJwsVerifier();

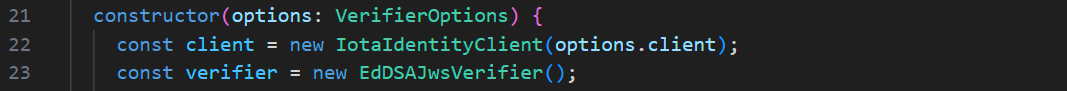
The purpose of the constructor in the verifier.ts command is to initialize the necessary components for verifying digital signatures using the IOTA Identity framework.

options: VerifierOptions: The constructor takes an options parameter of type VerifierOptions. This parameter allows the caller to pass in specific configuration options for the verifier.

const client = new IotaIdentityClient(options.client): This line creates a new instance of the IotaIdentityClient class, which is responsible for interacting with the IOTA Tangle and fetching the necessary data for verification. The options.client parameter is passed to the IotaIdentityClient constructor, allowing the client to be configured with specific settings.

const verifier = new EdDSAJwsVerifier(): This line creates a new instance of the EdDSAJwsVerifier class, which is responsible for verifying digital signatures using the EdDSA algorithm. The EdDSAJwsVerifier class provides methods to verify JSON Web Signatures (JWS) that are used in the IOTA Identity framework.

By initializing the IotaIdentityClient and EdDSAJwsVerifier instances within the constructor, the verifier is set up with the necessary components to perform signature verification. This allows the verifier to interact with the IOTA Tangle and validate digital signatures using the EdDSA algorithm.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

this.presentationValidator = new JwtPresentationValidator(verifier);

this.credentialValidator = new JwtCredentialValidator(verifier);

this.resolver = new Resolver({ client });

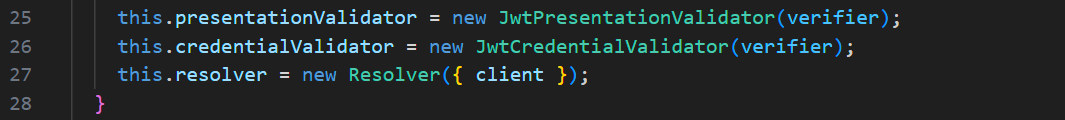
}

In using the presentationValidator object is an instance of the JwtPresentationValidator class. Its primary purpose is to validate the integrity and authenticity of the presentation tokens used in the verification process. Presentation tokens are typically JSON Web Tokens (JWTs) that contain claims about the presentation, such as the subject, issuer, and expiration time. The JwtPresentationValidator class provides methods to verify the signature and validate the claims of the presentation token.

credentialValidator: This object is an instance of the JwtCredentialValidator class. Its main purpose is to validate the integrity and authenticity of the credential tokens used in the verification process. Credential tokens are also JWTs that contain claims about the credential, such as the subject, issuer, and expiration time. The JwtCredentialValidator class provides methods to verify the signature and validate the claims of the credential token.

resolver: This object is an instance of the Resolver class, which is responsible for resolving and retrieving the necessary information from the underlying IOTA network. It acts as a bridge between the verifier and the IOTA network. The Resolver class provides methods to interact with the IOTA network, such as retrieving DID documents, verifying DID signatures, and resolving DID URLs.

By initializing these objects, the code sets up the necessary components for the verification process. The presentationValidator and credentialValidator objects handle the validation of presentation and credential tokens, respectively, while the resolver object facilitates the interaction with the IOTA network.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

static new() {

return new BuildVerifier({});

}

When the new() method is called, it internally creates a new instance of the BuildVerifier class by invoking the constructor with an empty object {} as an argument. The constructor of the BuildVerifier class is responsible for initializing the object with any necessary properties or configurations.

By encapsulating the object creation logic within the new() method, it provides a clean and concise way to create instances of the BuildVerifier class. This can be particularly useful when you want to create multiple instances of the class or when you want to abstract away the details of object creation.

**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

generatePresentationRequest() {

return new CreatePresentationRequest({

predicates: [],

});

}

The generatePresentationRequest() function returns a new instance of the CreatePresentationRequest class. This class is likely defined elsewhere in the codebase and represents a presentation request object.

The CreatePresentationRequest constructor takes an object as an argument, which can include various properties to define the presentation request. In the code snippet, the only property specified is predicates, which is an empty array.

The predicates property is used to define the conditions or requirements that the prover needs to satisfy in order to fulfil the presentation request. Predicates can be used to specify attributes, such as age, qualifications, or any other relevant information that the verifier wants to verify.

By passing an empty array as the value of predicates, the code snippet indicates that the verifier does not require any specific predicates from the prover. This means that the verifier is not requesting any specific attributes or conditions to be fulfilled in the presentation.

A black screen with colorful text

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

generatePresentationValidation() {

return new CreatePresentationValidation({

resolver: this.resolver,

presentationValidator: this.presentationValidator,

credentialValidator: this.credentialValidator,

});

}

}

The generatePresentationValidation() function in the verifier.ts file serves the purpose of creating an instance of the CreatePresentationValidation class. This class is responsible for performing validation checks on a presentation, which is a data structure that contains verifiable credentials.

resolver: This parameter represents the resolver object. In the context of IOTA, a resolver is responsible for resolving decentralized identifiers (DIDs) and retrieving associated data from the underlying distributed ledger. The resolver is used to fetch the necessary information required for validating the presentation.

presentationValidator: This parameter represents the presentation validator object. The presentation validator is responsible for verifying the integrity and authenticity of the presentation. It checks if the presentation has been tampered with or if it has been issued by a trusted entity.

credentialValidator: This parameter represents the credential validator object. The credential validator is responsible for validating the verifiable credentials contained within the presentation. It verifies the authenticity and integrity of each credential, ensuring that they have not been tampered with and have been issued by trusted entities.

By creating an instance of the CreatePresentationValidation class with the provided parameters, the generatePresentationValidation() function sets up the necessary components for validating a presentation. This function encapsulates the logic required to perform the validation process, making it easier to reuse and maintain the code.

A screen shot of a computer code

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

class BuildVerifier extends Build<VerifierOptions, Verifier> {

withClient(client: Client) {

return this.update((state) => {

state.client = client;

return state;

});

}

The BuildVerifier class extends the Build class, which is a generic class that takes two type parameters: VerifierOptions and Verifier. This indicates that the BuildVerifier class is responsible for constructing and configuring a Verifier instance.

The withClient method takes a client parameter of type Client, which represents the IOTA client that will be associated with the verifier. The Client class is typically used to interact with the IOTA network, send transactions, and retrieve information from the Tangle.

Inside the withClient method, the update method is called on the BuildVerifier instance. The update method is a utility method provided by the Build class, which allows you to update the internal state of the instance. It takes a callback function as an argument, which receives the current state as a parameter and returns the updated state.

In the callback function, the state.client property is assigned the value of the client parameter passed to the withClient method. This effectively sets the associated client for the verifier.

Finally, the updated state is returned from the update method, allowing for method chaining or further configuration of the verifier instance.

A screen shot of a computer

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**

create() {

return this.finalize((state) => {

return new Verifier(state);

});

}

}

The create() method is a member function of a class, and it is typically used to instantiate objects or perform any necessary setup before using the object. In this case, it is responsible for creating a new instance of the Verifier class.

The method starts by calling the finalize() method, passing a callback function (state) => { return new Verifier(state); } as an argument. The finalize() method is likely defined elsewhere in the codebase and performs some additional operations before returning the final result.

The callback function (state) => { return new Verifier(state); } is executed within the finalize() method. It takes a parameter state and uses it to create a new instance of the Verifier class by calling its constructor new Verifier(state).

Once the Verifier instance is created, it is returned as the result of the create() method.

A black background with colorful text

Description automatically generated**~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~**