# **Appendix 1 - 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.

# **Hans-example**

## **Index.ts**

This file is commonly the entry point of the application or module. It might import and initialise other modules, set up the server, or define the main logic of the application.

**Organising index**

The index.ts file is typically the main file of a TypeScript project. It acts as the entry point of the application, where the execution begins. When you run a TypeScript project, the code inside the index.ts file is executed first. It sets up the initial environment, imports necessary modules, and starts the application's execution flow. Another important role of the index.ts file is to manage module exports and imports. It serves as a central hub for exporting and importing modules within the project.

**//INSIDE OF THE SCRIPT**

import { IdentityWallet } from "../abstraction/identity";

import { Verifier } from "../abstraction/verifier";

import { delay, Duration } from "../abstraction/utils";

import PROOF\_OF\_CONCEPT from "./proof-of-concept";

**Import Statements**

The code begins using an import statement that imports specific modules from different files. These modules are:

IdentityWallet from "../abstraction/identity"

Verifier from "../abstraction/verifier"

delay and Duration from "../abstraction/utils"

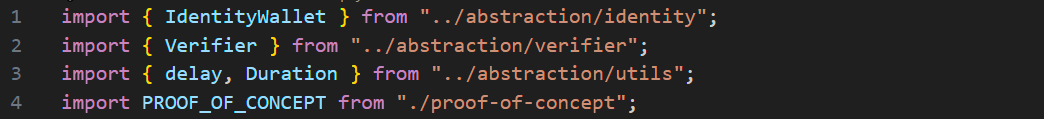
PROOF\_OF\_CONCEPT from "./proof-of-concept"

These import statements bring in the necessary functionality from the respective files found to be used in the script.

**Abstraction Function**

The abstraction function is a concept in programming that allows developers to hide complex implementation details and provide a simplified interface for interacting with a system or module. It abstracts away the complexity and exposes only the necessary functionality to the user.

In the context of the provided code snippet, the abstraction function is l used to abstract away the implementation details of the identity wallet, verifier, and utility functions. It provides a simplified interface for interacting with these components, making it easier for developers to use them without needing to understand the underlying complexity.

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

async function main() {

// Create a wallet to manage identities.

const wallet = await IdentityWallet.new()

.withClient(PROOF\_OF\_CONCEPT.client)

.create();

**// Create a wallet to manage identities.**

The provided TypeScript code is an asynchronous function named main(). It demonstrates the creation of a wallet to manage identities using the IdentityWallet class. The following is a breakdown of the script step by step:

The script starts with the declaration of an asynchronous function named main(). Asynchronous functions allow for non-blocking execution, which is useful when dealing with operations that may take some time to complete, such as network requests or file operations.

Inside the main() function, a wallet is created using the IdentityWallet.new() method. This method returns a promise that resolves to an instance of the IdentityWallet class.

The withClient() method is then called on the IdentityWallet instance, passing PROOF\_OF\_CONCEPT.client as an argument. This method configures the wallet with a client, which could be a specific service or application that the wallet will interact with.

Finally, the create() method is called on the IdentityWallet instance. This method creates the wallet and returns a promise that resolves to the created wallet.

Overall, the script demonstrates the creation of a wallet using the IdentityWallet class, configuring it with a client, and then creating the wallet itself. This script can be used as a starting point for managing identities within a TypeScript application.

A screen shot of a computer code

Description automatically generated

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

// Use the wallet to generate identities for Hans and the college.

console.log("Generating an identity for Hans...");

const hans = await wallet

.generateIdentity()

.withVerificationMethod("verify")

.withStorageDepositCovered(PROOF\_OF\_CONCEPT.coverStorageDeposit)

.create();

console.log("Explorer URL:", await PROOF\_OF\_CONCEPT.linkToExplorer(hans));

The purpose of the script in the index.ts file is to generate an identity for Hans using the IOTA wallet library.

The script starts by logging a message to the console, indicating that it is generating an identity for Hans.

The wallet object is then used to generate the identity. The generateIdentity() function is called on the wallet object, which returns a builder object. This builder object allows us to specify additional parameters for the identity generation.

In this case, the withVerificationMethod("verify") function is called on the builder object. This function sets the verification method for the identity to "verify". The verification method is used to prove the authenticity of the identity.

The withStorageDepositCovered(PROOF\_OF\_CONCEPT.coverStorageDeposit) function is also called on the builder object. This function covers the storage deposit required for the identity creation. The PROOF\_OF\_CONCEPT.coverStorageDeposit variable is used to determine the amount of storage deposit to cover.

Finally, the create() function is called on the builder object to create the identity. The resulting identity object is assigned to the hans variable.

A screen shot of a computer code

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

console.log("Generating an identity for the college...");

const college = await wallet

.generateIdentity()

.withVerificationMethod("verify")

.withRevocationService("revoke")

.withStorageDepositCovered(PROOF\_OF\_CONCEPT.coverStorageDeposit)

.create();

console.log("Explorer URL:", await PROOF\_OF\_CONCEPT.linkToExplorer(college));

The purpose of the script in the index.ts file is to generate an identity for a college using the IOTA protocol.

The first line logs a message indicating that an identity for the college is being generated.

The wallet.generateIdentity() function is called to create a new identity for the college. This function returns a WalletIdentity object. The withVerificationMethod("verify") method is used to specify the verification method for the identity. In this case, the verification method is set to "verify".

The withRevocationService("revoke") method is used to specify the revocation service for the identity. Here, the revocation service is set to "revoke". The withStorageDepositCovered (PROOF\_OF\_CONCEPT.coverStorageDeposit) method is used to cover the storage deposit for the identity. The PROOF\_OF\_CONCEPT.coverStorageDeposit value is passed as an argument to this method.

Finally, the create() method is called to create the identity. The resulting identity is stored in the college variable.

A screen shot of a computer code

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

// Create a verifier for the CRU.

const cru = await Verifier.new().withClient(PROOF\_OF\_CONCEPT.client).create();

**// Create a verifier for the CRU.**

The provided TypeScript code is responsible for creating a verifier for the CRU (Create, Read, Update) operations. The script shall be explained out further below.

Here it can be seen that a verifier is being created for the CRU operations. The Verifier class is being used, and the new() method is called to create a new instance of the verifier. The withClient() method is then called on the newly created verifier, passing PROOF\_OF\_CONCEPT.client as an argument. Finally, the create() method is called on the verifier, which returns a promise that resolves to the created verifier.

It's important to note that the code snippet is using the await keyword before calling the new() and create() methods. This indicates that the code is running in an asynchronous context, and the execution will pause until the promise returned by the create() method is resolved.

Overall, this script creates a verifier for the CRU operations by utilizing the Verifier class and chaining method calls to configure and create the verifier.



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

// Let the college issue a credential for Hans.

console.log("Generating a credential for Hans issued by the college...");

const credential = await college

.generateCredential()

.withSubject(hans)

.withProperty("degree", "Degree")

.withVerificationMethod("verify")

.withRevocationStatus("revoke", 1)

.create();

**// Let the college issue a credential for Hans.**

The given TypeScript code is responsible for generating a credential for Hans the holder of the credential, issued by Freie Universität Berlin (college), who are the issuers. See below for an explanation of the code generated.

In the snippet below, a message is logged to the console, indicating that a credential is being generated for Hans and it is being issued by the college. This line serves as an informational statement to provide feedback during the execution of the script.

A computer screen with text

Description automatically generated

In this snippet, a credential object is created using a chain of method calls. Let's break down each method call:

1. The generateCredential() method is called on the college object to initiate the generation of a credential.
2. The withSubject(hans)method sets the subject of the credential to the person named Hans. The hans variable is expected to hold the necessary information about Hans.
3. The withProperty("degree", "Degree") This method adds a property to the credential, specifying the degree and its value as "Degree". This allows the credential to include additional information about the subject.
4. The withVerificationMethod("verify") method sets the verification method for the credential to "verify". It defines the mechanism through which the authenticity of the credential can be verified.
5. The withRevocationStatus("revoke", 1) method sets the revocation status of the credential to "revoke" with a value of 1. It indicates that the credential can be revoked if necessary.
6. And the create() method finalizes the creation of the credential and returns the credential object.#

A computer screen with text

Description automatically generated

In summary, the script generates a credential for Hans, issued by the college, with specific subject, property, verification method, and revocation status. The generated credential is then stored in the credential variable for further use.

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

**// Let the college issue a credential for Hans.**

**// Let the CRU request credentials from Hans.**

**// In this proof of concept, the request and response are simply passed to**

**// the appropriate functions. In a production grade program, it is advised**

**// to implement a standardized exchange specification, like [OpenID] (**<https://openid.github.io/OpenID4VP/openid-4-verifiable-presentations-wg-draft.html>**).**

console.log("Requesting a presentation from Hans...");

const request = await cru

.generatePresentationRequest()

.withPredicate(PROOF\_OF\_CONCEPT.notOlderThan(Duration.seconds(5)))

.withNonce(PROOF\_OF\_CONCEPT.generateNonce())

.create();

The given TypeScript code demonstrates the process of requesting a presentation from a source named "Hans." Again, the script is broken down below:

The first line console.log("Requesting a presentation from Hans..."); is a simple console log statement that outputs a message indicating the intention to request a presentation from a person named "Hans." This line serves as a visual indicator for the developer and does not affect the functionality of the script.

The second line const request = await cru initiates a variable named request and uses the await keyword to wait for the completion of the subsequent asynchronous operation.

The third line .generatePresentationRequest() is a method call on the cru object. It generates a presentation request, which is a request for specific information or data.

The fourth line .withPredicate(PROOF\_OF\_CONCEPT.notOlderThan(Duration.seconds(5))) is a method call on the presentation request object. It sets a predicate for the request, specifying that the requested data should not be older than 5 seconds. This ensures that the requested information is up to date.

The fifth line .withNonce(PROOF\_OF\_CONCEPT.generateNonce()) is another method call on the presentation request object. It sets a nonce, which is a unique identifier for the request. This helps prevent replay attacks and ensures the integrity of the request.

The sixth line .create(); finalizes the creation of the presentation request and returns the result. The result is then assigned to the request variable.

A screen shot of a computer code

Description automatically generated

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

**// Let Hans gather credentials that meet the requested requirements in a presentation**

**// and generate a response.**

console.log("Generating a presentation response for Hans...");

const response = await hans

.generatePresentationResponse()

.withNonce(request.nonce)

.withCredential(credential)

.withVerificationMethod("verify")

.create();

The given TypeScript code is responsible for generating a presentation response for Hans who is the holder of the credentials. The script performs a series of operations to gather credentials that meet certain requirements and generate the desired response.

breaking down the script step by step:

1. The script starts by logging a message to the console, indicating that a presentation response is being generated for Hans.
2. The hans object, which represents Hans, is used to initiate the generation of the presentation response.
3. The generatePresentationResponse() method is called on the hans object. This method is responsible for creating a presentation response.
4. The withNonce() method is chained to the generatePresentationResponse() method. It takes a nonce parameter from the request object and sets it as a property of the presentation response.
5. The withCredential() method is chained to the previous method. It takes a credential parameter and adds it to the presentation response. This step ensures that the generated response includes the required credential.
6. The withVerificationMethod() method is chained to the previous method. It sets the verification method for the presentation response to "verify". This step specifies the method that will be used to verify the authenticity of the response.
7. Finally, the create() method is called to generate the presentation response. The response is stored in the response variable for further use.

In summary, the script allows Hans to generate a presentation response by gathering credentials that meet their specific requirements. The response is created using the generatePresentationResponse() method, and various methods like withNonce(), withCredential(), and withVerificationMethod() are used to customize the response. The generated response is then stored in the response variable for further processing or usage.

A black screen with text

Description automatically generated

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

**// Let the CRU verify the response signatures and credentials and validate**

**// the credentials against the requirements of the request.**

**// NOTE: This validation should succeed.**

console.log("Validating the presentation response from Hans...");

await cru

.generatePresentationValidation()

.withRequest(request)

.withResponse(response)

.create();

The given TypeScript code is a code snippet that demonstrates the validation of a presentation response from a person named Hans. The script utilizes the console.log function to print a message indicating the validation process. It then uses the await keyword to ensure that the generatePresentationValidation function is executed asynchronously.

The generatePresentationValidation function is likely a method of an object or a class instance. It is chained with the withRequest, withResponse, and create methods. These methods are used to pass the request and response objects to the generatePresentationValidation function, allowing it to perform the validation.

The purpose of this script is to validate the presentation response received from Hans. It is important to note that the code snippet provided is incomplete, as it does not show the definition of the cru object or the request and response objects.

A black screen with colorful text

Description automatically generated

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

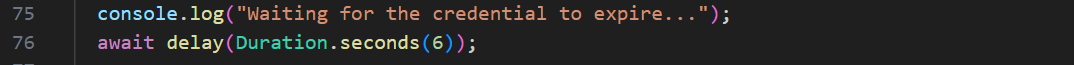
**// Wait for a certain number of seconds.**

console.log("Waiting for the credential to expire...");

await delay(Duration.seconds(6));

The provided TypeScript code is using the await keyword to pause the execution of the code for 6 seconds. It achieves this by calling the delay function and passing it the duration of 6 seconds.

The delay function is not a built-in TypeScript function, but it is commonly used in asynchronous programming to introduce a delay or pause in the execution of code. It is often used in scenarios where you need to wait for a certain period of time before proceeding with the next steps.



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

**// Let the CRU verify the response signatures and credentials and validate**

**// the credentials against the requirements of the request.**

**// NOTE: This validation should fail.**

console.log("Validating the presentation response from Hans again...");

await cru

.generatePresentationValidation()

.withRequest(request)

.withResponse(response)

.create();

The given TypeScript code is a code snippet that demonstrates the validation of a presentation response from a person named Hans. The script utilizes the console.log function to print a message indicating the validation process. It then uses the await keyword to ensure that the generatePresentationValidation function is executed asynchronously.

The generatePresentationValidation function is likely a method of an object or a class instance. It is chained with the withRequest, withResponse, and create methods. These methods are used to pass the request and response objects to the generatePresentationValidation function, allowing it to perform the validation.

The purpose of this script is to validate the presentation response received from Hans. It is important to note that the code snippet provided is incomplete, as it does not show the definition of the cru object or the request and response objects.

A black screen with colorful text

Description automatically generated

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

**// Revoke the credential of Hans.**

console.log("Revoking the credential...");

await college

.generateRevocation() **(We left it here)**

.withRevocationStatus("revoke", 1)

.withStorageDepositCovered(PROOF\_OF\_CONCEPT.coverStorageDeposit)

.create();

console.log("Revoking the credential...");: This line simply logs a message to the console, indicating that the credential revocation process is about to begin.

await college.generateRevocation(): This line calls the generateRevocation method on the college object. This method is responsible for generating a revocation object that will be used to revoke the credential.

.withRevocationStatus("revoke", 1): This line sets the revocation status of the generated revocation object. In this case, the status is set to "revoke" with a revocation index of 1. This means that the credential with index 1 will be revoked.

.withStorageDepositCovered(PROOF\_OF\_CONCEPT.coverStorageDeposit): This line sets the storage deposit coverage for the revocation object. The PROOF\_OF\_CONCEPT.coverStorageDeposit value is used to cover the storage deposit required for the revocation process.

.create(): This line creates the revocation object and initiates the revocation process. Once the revocation object is created, it will be broadcasted to the IOTA network, and the corresponding credential will be revoked.

A computer screen with text

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

**// Let the CRU verify the response signatures and credentials and validate**

**// the credentials against the requirements of the request.**

**// NOTE: This validation should fail.**

console.log("Validating the presentation response from Hans again...");

await cru

.generatePresentationValidation()

.withRequest(request)

.withResponse(response)

.create();

}

main();

The provided code snippet demonstrates the process of validating a presentation response using the IOTA DID framework. This line simply logs a message to the console, indicating that the presentation response from a specific entity named "Hans" is being validated. It serves as a progress indicator or informational message.

A screen shot of a computer program

Description automatically generated

In this part of the code, the cru object is used to generate a presentation validation. The generatePresentationValidation() function is called on the cru object, which returns a presentation validation object. This object allows us to specify the request and response objects for validation.

The withRequest(request) method is used to provide the request object, which likely contains the necessary information for validating the presentation response. Similarly, the withResponse(response) method is used to provide the response object that needs to be validated.

Finally, the create() method is called on the presentation validation object to initiate the validation process. This method likely performs the necessary checks and validations based on the provided request and response objects.

A screen shot of a computer program

Description automatically generated

This line calls the main() function, which is likely the entry point of the program. It is responsible for executing the code and initiating the validation process.

Overall, the code snippet demonstrates the process of validating a presentation response within the IOTA DID framework. It logs a message indicating the validation process and uses the cru object to generate a presentation validation, providing the necessary request and response objects for validation. Finally, the create() method is called to initiate the validation process.

A screen shot of a computer program

Description automatically generated

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

## **Proof-of-concept.ts**

The provided TypeScript code is a proof-of-concept implementation that showcases various functionalities related to IOTA Identity and IOTA tokens.

The Proof-of-Concept file plays a crucial role in the development process. It serves as a prototype or a demonstration of a concept or idea before implementing it fully in a project. One of the primary purposes of a POC.ts file is to showcase the core functionality of a feature or a system. It helps stakeholders, such as project managers or clients, to visualise and understand how the final product will work. By providing a tangible representation of the proposed solution, a POC.ts file helps in gathering feedback and making informed decisions about the project's direction.

**//INSIDE OF THE SCRIPT**

The provided TypeScript code is a proof-of-concept implementation that showcases various functionalities related to IOTA Identity and IOTA tokens.

**Usage of the Script**

The script provides a foundation for working with IOTA Identity and IOTA tokens. It demonstrates how to interact with the IOTA network, request tokens from a faucet, check token balances, and perform operations related to IOTA Identity.

The provided TypeScript code is a proof-of-concept implementation that utilizes the @iota/identity-wasm and @iota/sdk-wasm libraries to interact with the IOTA Tangle, a distributed ledger technology. Let's break down the script and understand its components.

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

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

import { EXPLORER, NODE, FAUCET } from "../constants";

import { Identity } from "../abstraction/identity";

import { delay, Duration, getTokensAvailable } from "../abstraction/utils";

The first line imports the Credential and Timestamp classes from the @iota/identity-wasm/node module. These classes are used for managing credentials and timestamps within the IOTA DID framework.

The second line imports the Client class from the @iota/sdk-wasm/node module. This class provides functionality for interacting with the IOTA network, such as sending transactions and retrieving data.

The third line imports the EXPLORER, NODE, and FAUCET constants from the "../constants" module. These constants likely represent URLs or addresses related to the IOTA network, such as the IOTA explorer, a specific IOTA node, and a faucet for obtaining test tokens.

The fourth line imports the Identity class from the "../abstraction/identity" module. This class is a custom abstraction or wrapper around the IOTA DID functionality, providing higher-level methods for managing identities specific to Hans.

The fifth line imports the delay, Duration, and getTokensAvailable functions from the "../abstraction/utils" module. These functions are likely utility functions used within the IOTA DID framework, such as for delaying execution, calculating durations, or retrieving the available token balance.

A screen shot of a computer code

Description automatically generated

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

The provided code snippet is a TypeScript class called ProofOfConcept that is related to the IOTA DID (Decentralized Identifier) framework

class ProofOfConcept {

client: Client;

constructor() {

this.client = new Client({ nodes: [NODE] });

}

**Class Definition**

The code snippet starts by defining a class called ProofOfConcept. This class is used to encapsulate the functionality related to the IOTA DID framework.

**Client Initialization**

Inside the class, there is a property called client of type Client. This property is used to interact with the IOTA network and perform various operations related to the DID framework.

In the constructor of the class, an instance of the Client class is created and assigned to the client property. The Client class is part of the IOTA JavaScript library and provides a convenient way to interact with the IOTA network.

The Client class is initialized with a configuration object that specifies the nodes to connect to. In the provided code snippet, the nodes property is set to [NODE], which suggests that the actual node address is expected to be provided as a parameter when creating an instance of the ProofOfConcept class.

A black screen with colorful text

Description automatically generated

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

The linkToExplorer function is a utility function in the IOTA DID framework that is used to generate a link to the IOTA Identity Explorer for a given identity.

async linkToExplorer(identity: Identity) {

const did = identity.document.id().toString();

return `${EXPLORER}/identity-resolver/${did}`;

}

The linkToExplorer function takes an identity object of type Identity as its parameter. This object represents an identity within the IOTA DID framework.

Inside the function, the identity.document.id() method is called to retrieve the unique identifier (DID) of the identity. The toString() method is then used to convert the DID to a string.

Finally, the function returns a string that represents the link to the IOTA Identity Explorer for the given identity. The link is constructed by concatenating the EXPLORER constant (which represents the base URL of the Identity Explorer) with the /identity-resolver/ path segment and the DID of the identity.

By using this linkToExplorer function, developers can easily generate a link to the IOTA Identity Explorer for a specific identity. This allows them to view and explore the details and transactions associated with that identity within the IOTA DID framework.

A screen shot of a computer code

Description automatically generated

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

The provided code snippet is a function called requestTokens that takes an address as a parameter.

async requestTokens(address: string) {

const tokensAvailableBefore = await getTokensAvailable(

this.client,

address

);

await this.client.requestFundsFromFaucet(FAUCET, address.toString());

while (

tokensAvailableBefore === (await getTokensAvailable(this.client, address))

) {

await delay(Duration.seconds(1));

}

}

The const tokensAvailableBefore = await getTokensAvailable(this.client, address);

Is a line of code that calls the getTokensAvailable function, passing in the this.client object and the address parameter. It assigns the returned value to the tokensAvailableBefore variable. This line is used to get the number of tokens available before requesting additional tokens.

The await this.client.requestFundsFromFaucet(FAUCET, address.toString());

Is a line of code that calls the requestFundsFromFaucet function on the this.client object, passing in the FAUCET and address.toString() parameters. It awaits the completion of this function, which requests additional tokens from a faucet. The address.toString() converts the address parameter to a string before passing it to the function.

The while (tokensAvailableBefore === (await getTokensAvailable(this.client, address))) { await delay(Duration.seconds(1)); }

Is a block of code that creates a loop that continues until the number of tokens available after requesting additional tokens is different from the tokensAvailableBefore value. It uses the getTokensAvailable function to get the current number of tokens available, comparing it with the tokensAvailableBefore value. If they are equal, it waits for 1 second using the delay function from the Duration module.

A screen shot of a computer code

Description automatically generated

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

coverStorageDeposit = async (address: string, tokensRequired: bigint) => {

while ((await getTokensAvailable(this.client, address)) < tokensRequired) {

await this.requestTokens(address);

}

};

**coverStorageDeposit**

The following snippet of code defines an asynchronous function called coverStorageDeposit. It takes two parameters: address of type string and tokensRequired of type bigint.

The function uses a while loop to check if the tokens available for the given address are less than the tokensRequired. If the condition is true, it calls the requestTokens function and awaits its completion. This process continues until the required tokens are available.

A screen shot of a computer code

Description automatically generated

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

notOlderThan(duration: Duration) {

return (credential: Credential) => {

const oldestIssuanceDate = Date.parse(

Timestamp.nowUTC().checkedSub(duration).toRFC3339()

);

const issuanceDate = Date.parse(credential. IssuanceDate().toRFC3339());

return issuanceDate > oldestIssuanceDate;

};

}

**notOlderThan**

This snippet defines a function called notOlderThan that takes a parameter duration of type Duration. The function returns another function that takes a parameter credential of type Credential.

Inside the returned function, it calculates the oldestIssuanceDate by subtracting the duration from the current UTC timestamp using the checkedSub method. It then converts the result to the RFC3339 format using the toRFC3339 method. It then retrieves the issuanceDate from the credential object and converts it to the RFC3339 format. And finally, it compares the issuanceDate with the oldestIssuanceDate and returns true if the issuanceDate is greater than the oldestIssuanceDate, indicating that the credential is not older than the specified duration.

A screen shot of a computer code

Description automatically generated

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

generateNonce() {

return Math.floor(Math.random() \* Math.pow(2, 32)).toString();

}

}

**generateNonce**

This snippet defines a function called generateNonce that generates a random nonce. It uses the Math.random() method to generate a random number between 0 and 1, multiplies it by Math.pow(2, 32) to get a random number between 0 and 2^32, and then converts it to a string.

The generated nonce is used for various purposes, such as ensuring uniqueness or adding randomness to cryptographic operations.

A computer screen with text

Description automatically generated

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

The provided code snippet is written in TypeScript and exports a new instance of the ProofOfConcept class.

export default new ProofOfConcept();

The ProofOfConcept class represents a proof of concept implementation within the IOTA DID framework. It is likely a custom class developed to demonstrate or test specific functionalities related to the framework.

By creating a new instance of the ProofOfConcept class, the code snippet initialises and sets up the proof of concept implementation. This could involve configuring various components, initializing data structures, and connecting to the IOTA network.

A computer screen with text

Description automatically generated

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