3

i3-MARKET Wallets

3.1 Objectives

i3M-Wallets is a set of technologies that facilitate the interaction of the different i3-MARKET stakeholders with the Backplane API. It manages i3-MARKET identities, data agreements (with signature verification/generation), non-repudiation proofs (generation/verification), and secrets for data encryption/decryption.

All the code has been made publicly available at the i3M-Wallet monorepo [2]. Several packages are provided in this repo, but a standard i3-MARKET user/developer is likely only needing:

- **i3M Server Wallet:** It is an interactionless wallet implementation not requiring any user interaction. It has been designed to be operated by a "machine" or service. Current implementation is in TypeScript/JavaScript and can be easily imported to any JS project with NPM/Yarn.
- **i3M-Wallet Desktop App:** It is a desktop application (Windows, MacOS, and Linux) thought to be operated by end-users. The app can be securely paired to any application, allowing the application to interact with the wallet through an HTTP API. Wallet actions requested by any application will require explicit confirmation of the end-user through the app interface (window).
- **i3M-Wallet Protocol API:** A TypeScript/JavaScript library that can be used to easily connect to an i3M-Wallet Desktop App. It wraps all the functionalities provided by the wallet's HTTP API into convenient class methods. It works in Node.js (both ESM and CJS) and browsers. Follow the pairing example to properly pair your JS application to the wallet and start using the Wallet API.
- **i3M-Wallet OpenAPI Specification:** In order to get a better understanding of what functionalities of the wallet are provided to paired

applications, a developer should analyse the i3M-Wallet OpenAPI Specification [39] or just visualize it online at editor.swagger.io[1].

The complete documentation of every package is provided in every package's README of the open-source public repositories.

3.2 Technical Requirements

i3M-Wallets implement the following requirements:

Identity management:
The wallet implements key functionalities for enabling the self-sovereign identity (SSI) solution of i3-MARKET. These functionalities are described below.

DID management:
The i3-MARKET identity subsystem heavily relies on the use of distributed identifiers (DID) [40]. The wallet should then be able to manage DIDs, specifically:

- **Create DID:** The wallet should be able to create a DID and the complementary cryptographic key for managing it. The keys must be securely stored/managed.
- **Present DID:** The wallet should be able to present a DID upon request and prove ownership of it.
- **Resolve DID:** The wallet should be able to retrieve the public data associated with a DID, including (but not limited to) the public keys associated with the DID.
- **Verify asset signature:** The wallet should be able to verify signatures using the DID of the signer.
- **Sign assets:** The wallet should be able to sign assets using the private key associated with a DID.
- **Deactivate DID:** The wallet should be able to deactivate a DID.

Verifiable Credentials management:
i3M-Wallets handle i3-MARKET identities, which, in the end, are DIDs and a set of Verifiable Credentials issued for those DIDs. A Verifiable Credential [41] is a tamper-evident credential that can be cryptographically verified and

[1]https://editor.swagger.io/?url=https://raw.githubusercontent.com/i3-MARKET-V3-Public-Repository/SP3-SCGBSSW-I3mWalletMonorepo/public/packages/wallet-desktop-openapi/openapi.yaml

stores claims about an identity issued by different entities. An example of a Verifiable Credential could be a university diploma issued to a student. The specific requirements implemented by the i3M-Wallets with regard to the management of Verifiable Credentials are:

- **Verify Verifiable Credential:** Verify the validity of a Verifiable Credential by verifying the signatures of the issuers.
- **Share Verifiable Credential:** Share an owned Verifiable Credential upon request and prove ownership.
- **Store Verifiable Credential:** Store Verifiable Credentials associated with owned identities.

Secure data exchanges:
For a secure data exchange to happen, the i3M-Wallet should support the management of the cryptographic material needed for the secure data exchange, including the storage and verification of the data sharing agreements, the verifiable proofs for the non-repudiation protocol, and the cryptographic material associated with every data exchange. Please refer to Chapter 14 for better understanding the flow of a secure data exchange.

Specifically, an i3M-Wallet implements the following requirements regarding secure data exchanges.

- **Store data sharing agreements:** Store data sharing agreements associated with one of the identities managed by the wallet. The data sharing agreements are verified (both schema and signatures) before they are stored. The cryptographic material associated with the agreement, including the freshly created agreement-specific keys are also verified and stored.
- **Sign data sharing agreements:** Sign a data sharing agreement using one of the owned identities.
- **Store non-repudiation proofs:** For the conflict-resolution system to work, the wallet should store non-repudiation proofs for every data exchange, which can later be used to unequivocally prove that the data exchange happened and what was exchanged. Non-repudiation proofs are verified before being stored.

3.3 Solution Design/Blocks

The development of i3M-Wallets is organized in different packages/modules providing different functionalities. All the packages have been made publicly available at the i3M-Wallet monorepo, namely:

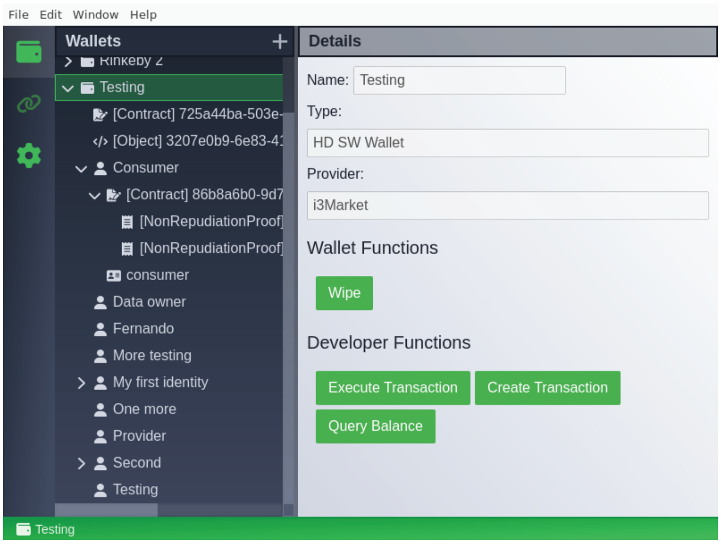
- Base Wallet: base-wallet [42]
- SW Wallet: sw-wallet [43]
- BOK Wallet: bok-wallet [44]
- Wallet Desktop: wallet-desktop [45]
- Server Wallet: server-wallet [46]
- Wallet OpenAPI: wallet-desktop-openapi [47]
- Wallet Protocol: wallet-protocol [48]
- Wallet Protocol API: wallet-protocol-api [49]
- Wallet Protocol Utils: wallet-protocol-utils [50]

The *Base Wallet* [42] package is a high-level implementation of the i3M-Wallet functionalities. It internally uses a crypto wallet. For such a purpose, it uses the so-called KeyWallet interface, which currently has several implementations:

- **SW Wallet** [43]: A software implementation of a hierarchically deterministic wallet, which can be recomputed with a seed.
- **BOK Wallet** [44]: A software implementation of a wallet implemented as a bag of (independent) keys.
- **HW Wallet**: A package allowing the use of IDEMIA's i3-MARKET HW Wallet as the internal KeyWallet. The implementation of this package involves IDEMIA's proprietary code implementing the open-source KeyWallet interface. A video demonstrating how it is used with the i3M-Wallet Desktop application can be watched at [51].
- **Wallet Desktop** [45] is the i3M-Wallet Desktop application providing a secure and convenient user interface to the i3-MARKET Base Wallet. It can be defined as a cross-platform facility tool that eases the communication between a wallet (software or hardware) and the i3-MARKET SDK via an HTTP API. Furthermore, it provides some features like wallet synchronization using a secure cloud vault. It also has a user interface (UI) to display the information of the selected wallet and ask for user consent if any wallet operation needs it. The Figure 3.1 shows an example of the Wallet UI.

If the wallet is to be run by a service instead of an actual person, the *Server Wallet* [46] should be used instead, which is distributed as a JavaScript/TypeScript library that can be easily instantiated from server code.

Both the Server Wallet and the Wallet Desktop expose some functionalities that can be used programmatically from paired applications. The *Wallet OpenAPI* [47] defines the internal HTTP API that is exposed (more details in section of Interface Description). However, for security reasons, the API is



Name: Testing
Type: HD SW Wallet
Provider: i3Market

Figure 3.1 Wallet Desktop UI.

not directly available via HTTP. Indeed, it is encapsulated inside a secure session that is established after successful pairing of the wallet with an application.

Creating a secure session requires the execution of the *Wallet Protocol* [48], which implements the i3M-Wallet pairing protocol and the agreement of a secure (both encrypted and authenticated) session between the wallet and the paired application.

The Wallet Protocol enables any application to securely connect to the wallet. It solves two problems: the discovery of the wallet and the secure channel creation. The i3M-Wallet pairing protocol is designed to pair applications running in the same machine and to not require any external entity for the process, and it is heavily inspired in the Bluetooth Secure Simple Pairing protocol in use since Bluetooth 2.1 [52]. The protocol has been carefully designed and validated with a formal security analysis using Tamarin's prover [53], a tool that has also been used to validate TLS1.3, among other security protocols. The complete description and design are available in open access repositories.

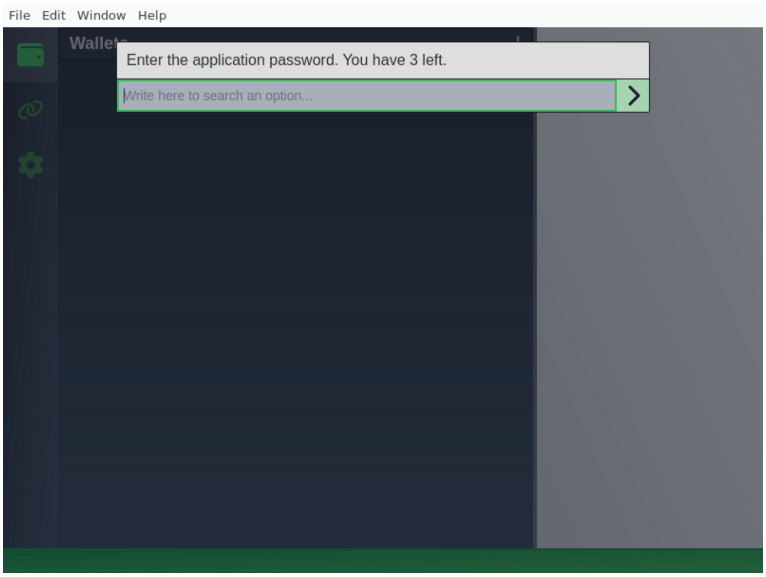
In order to ease the use of the Wallet Protocol a set of libraries have been developed for both Node.js and browser JavaScript. *Wallet Protocol Utils* [50] defines a set of utilities for the pairing, including dialogs for setting the PIN in browser JS apps and Node.js, and session managers for properly managing wallet-protocol's sessions obtained after a successful pairing. Moreover, *Wallet Protocol API* [49] is a TypeScript/JavaScript library that wraps all the in-session encapsulated calls to the wallet's HTTP API into convenient class and methods.

3.4 Diagrams

Wallet Desktop:

Figure 3.3 shows the start-up flow of the Wallet Desktop. The first thing it does is loading the application configuration. It is a JSON file whose path depends on the operative system running the Wallet Desktop:
%APPDATA%\wallet-desktop\config.json on Windows
$XDG_CONFIG_HOME/wallet-desktop/config.json or ∼/.config/wallet-desktop/config.json on Linux
∼/Library/Application Support/wallet-dektop/config.json on macOS

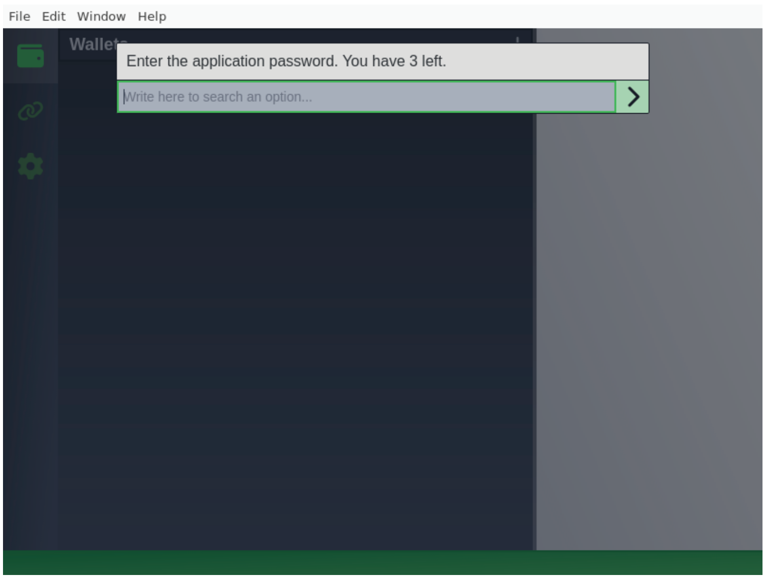
Then the Wallet Desktop initializes the user interface and the so-called extra features, which nowadays is just an encrypted storage for supplementary data (cryptographic material is securely stored/managed by the KeyWallet). Figure 3.2 shows the UI asking for the encrypted storage password.



Enter the application password. You have 3 left.

Write here to search an option...

Figure 3.2 UI password request for the encrypted storage.



Enter the application password. You have 3 left.

Figure 3.3 Wallet start-up flow.

The wallet factory oversees instantiating and the wallet modules. Wallet modules have an entry function that returns a wallet object. On the start-up, it will use the configuration file to select the current wallet.

Finally, it initializes the HTTP API. From now on, the application will listen to the user interface and the HTTP port in parallel; being securely encapsulated inside the HTTP connection, the wallet will receive calls to the internal HTTP API defined by the Wallet OpenAPI. The complete flow diagram is as defined in Figure 3.3.

As an example of invoking one of the wallet's API functionalities, let us follow the flow for generating a signature depicted in Figure 3.4.

As the Wallet Desktop receives an HTTP request in a Wallet Protocol's session, the request encapsulates (both encrypted and authenticated) an HTTP call to perform a signature; the OpenAPI Validation express module uses the OpenAPI definitions present in the package wallet-desktop-openapi to validate the request body. If the request is valid, the sign API handler gets the current wallet selected by the user and it calls its sign function. The current wallet can be any kind of hardware or software wallet supported by Wallet Desktop.

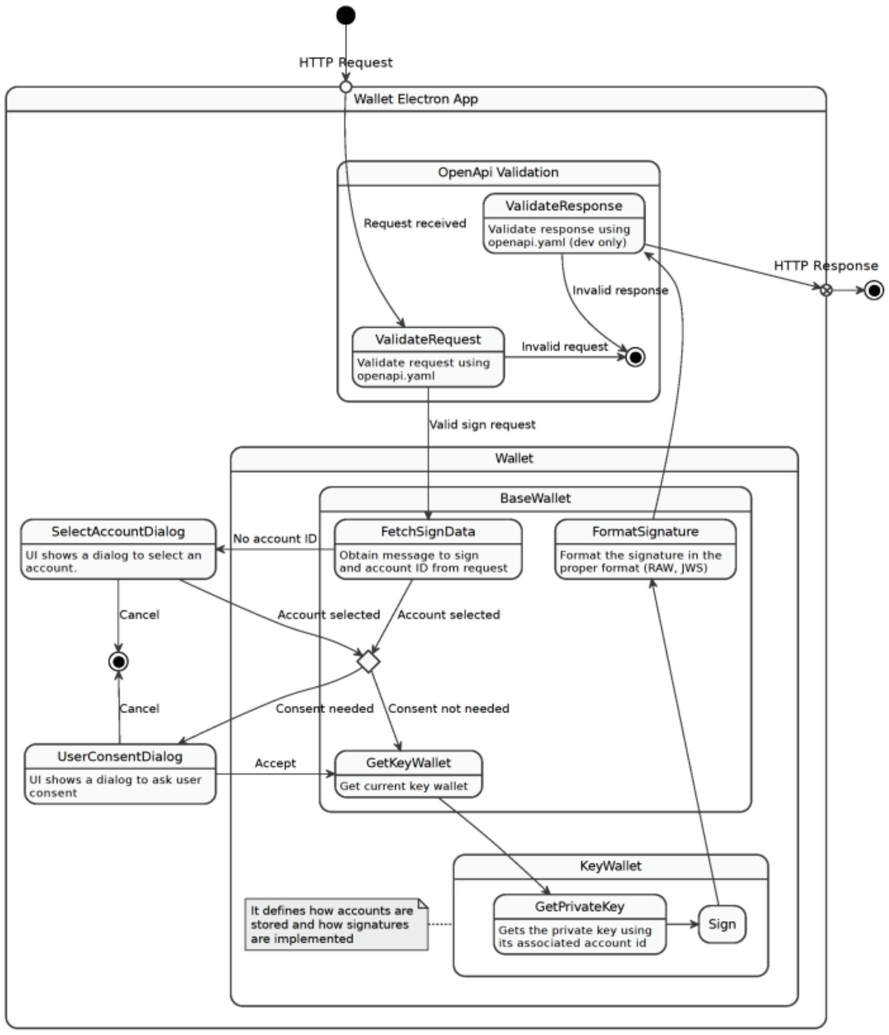


Figure 3.4 Wallet signature flow.

BaseWallet is a class present in the base-wallet package that offers a default implementation of a wallet. Nonetheless, it requires an object implementing the KeyWallet interface to work. KeyWallet defines the low-level implementation of the wallet: it can store keys and use them to sign. Note that by splitting the wallet in BaseWallet and KeyWallet, software wallet (SW) and hardware wallet (HW) can share high-level wallet functionalities, such as signature formatting or key recovering.

To perform a signature, an account must be selected. If the API request does not contain an account ID, the Wallet Desktop will show the list of accounts inside the wallet so that the user can select one. Then, the Base-Wallet will check the application configuration to check if signatures need user consent. If true, the wallet application will show a dialog.

Once the account ID is retrieved and the user consents the signature, the KeyWallet is now able to perform a signature. First, it will get the private key associated with the current account ID, and then it will sign the requested message.

Since the signature format is a common functionality of any wallet, KeyWallets must return signatures in DER encoding so that the BaseWallet can format it in the requested format.

Finally, the Wallet Desktop application builds the response message. On development, the OpenApi Validation module uses the OpenApi definitions to verify the response format (Figure 3.4). If it is correct, the response is sent back to the application encapsulated inside the Wallet Protocol's secure session.

OIDC Authentication:

Wallet Desktop can be used in conjunction with the i3-MARKET OIDC provider to authenticate users. This flow is added here so that views of flows using the Wallet are in this book. However, more detailed information should be available at deliverables in "Trust, Security and Privacy Solutions for Securing Data Marketplaces" at https://www.i3-market.eu/research-and-technology-library/.

Figure 3.5 shows how to perform this authentication flow. As a summary, it specifies how the OIDC provider uses the i3-MARKET SDK to create a selective disclosure request asking a set of verifiable claims to the Wallet Desktop. If the user has them and accepts the disclosure, the Wallet Desktop will answer with a list of verifiable claims along with a proof of ownership. This proof consists of a signature of the disclosure response using the private key of the user DID.

The flow starts when an OpenID Connect relying party (OIDC RP) redirects a user to the OIDC provider using a scope. The scope is a string that specifies which verifiable claims are requested. The scope supported are:

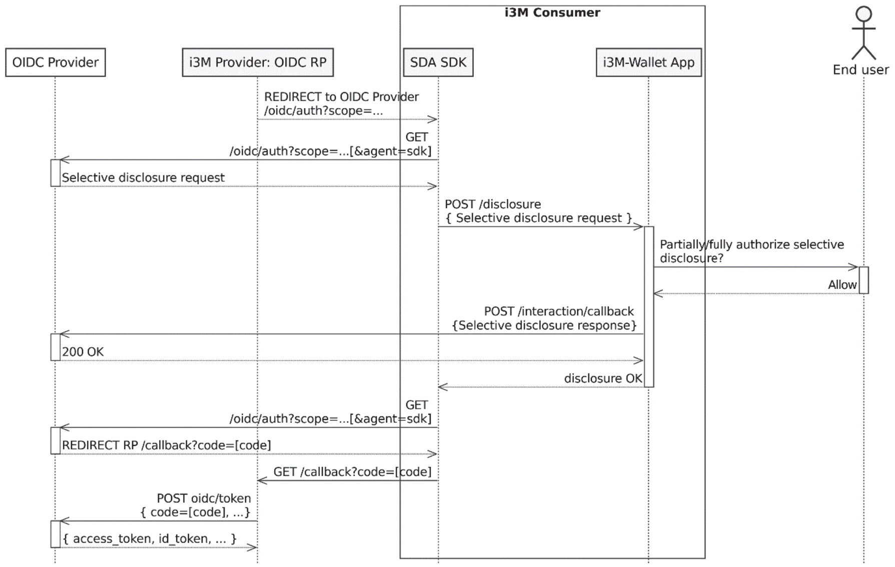

i3M Consumer
OIDC Provider
i3M Provider: OIDC RP
SDA SDK
i3M-Wallet App
End user
REDIRECT to OIDC Provider /oidc/auth?scope=...
GET /oidc/auth?scope=...[&agent=sdk]
Selective disclosure request
POST /disclosure { Selective disclosure request }
Partially/fully authorize selective disclosure?
Allow
POST /interaction/callback {Selective disclosure response}
200 OK
disclosure OK
GET /oidc/auth?scope=...[&agent=sdk]
REDIRECT RP /callback?code=[code]
GET /callback?code=[code]
POST oidc/token { code=[code], ...}
{ access_token, id_token, ... }

Figure 3.5 OIDC authentication using Wallet Desktop and the i3-MARKET SDK (sequence diagram).

- **openid:** The standard OpenID scope. It asks the OIDC provider to return an *idToken*, which is a jwt token containing the user information.
- **vc:** This scope is used to add the verifiable claims inside the *idToken*.
- **vc:<*claim*>:** It notifies to the OIDC provider that users may want to present an optional claim called <*claim*>. It is useful to ask optional claims like some extra user profile information.
- **vce:<*claim*>:** It notifies to the OIDC provider that users must present a valid claim called <*claim*> to proceed with the authentication. It can be used to ask users to present a claim that demonstrates their role (consumer, provider, data owner, etc.).

An example of scope is *"openid vc vce:consumer vc:profile"*. Using this scope, the OIDC provider builds and signs a selective disclosure request. It will be sent to the Wallet Desktop so as to obtain the required claims.

The Wallet Desktop HTTP API runs locally on the user computer. Cloud servers cannot access it directly so that the disclosure request must be sent by the user's computer. An approach is to create a simple frontend that only sends the selective disclosure request to the Wallet Desktop.

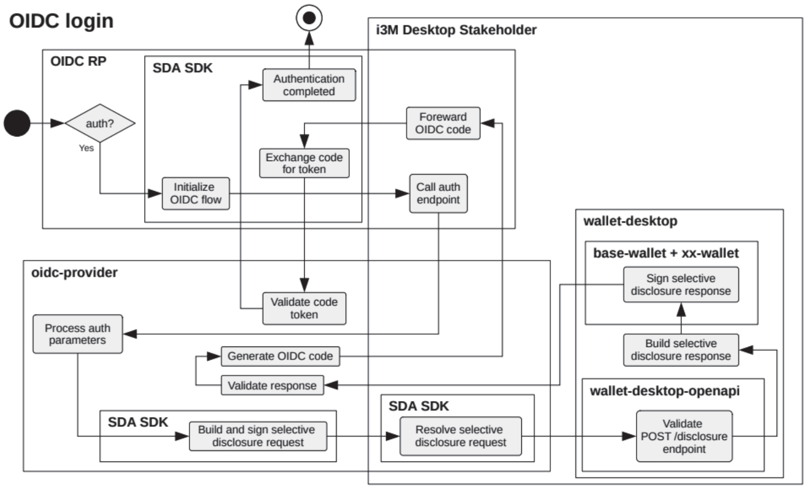


Figure 3.6 OIDC authentication using the wallet (activity diagram).

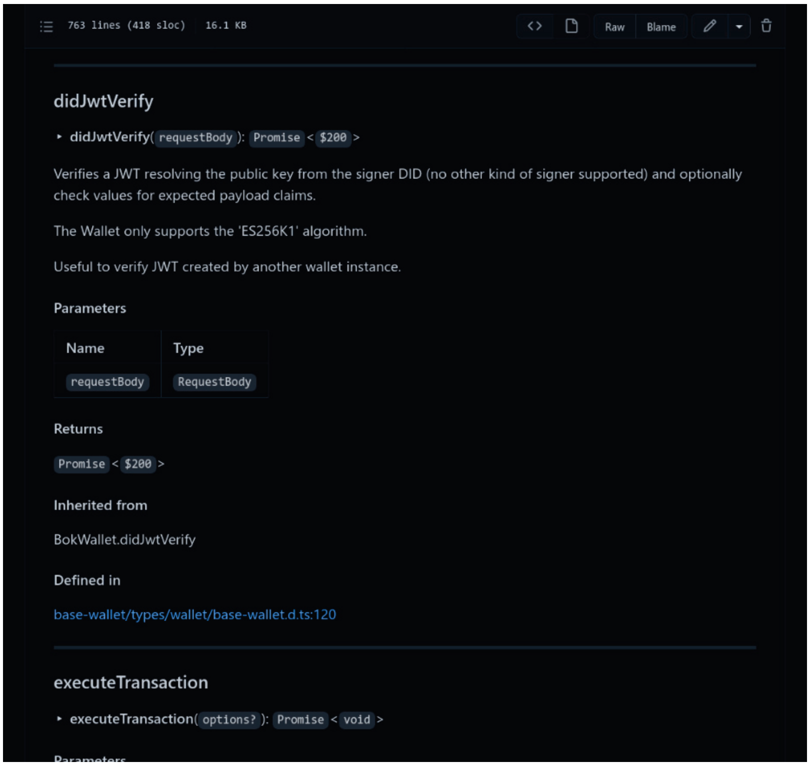
Then, the wallet displays a dialog to individually consent each verifiable claim. If the user allows the disclosure, the accepted verifiable claims will be sent back to the OIDC provider. After verifying all the claims, the OIDC provider will deliver an *idToken* and an *accessToken* to the i3M provider. Figure 3.6 is an equivalent diagram but putting more emphasis on where each block acts.

3.5 Interfaces

All the i3M-Wallet packages, but the i3M-Wallet Desktop App, are provided as TypeScript/JavaScript packages. The code is properly commented and TypeDoc has been used to convert comments in the TypeScript source code into rendered HTML documentation. As a result, the documentation is conveniently available when coding and also as HTML pages that can be accessed from the package's READMEs. As an example, Figure 3.7 shows a fragment of the API for the Server Wallet package.

The Wallet Desktop application follows the Wallet OpenAPI Specification. It is a REST-like API with four entities: identity, selective disclosure, resource, and transaction. It also has a set of helper functions.

- **Identity:** It can be used to create or list the DID of the user.



didJwtVerify

▸ didJwtVerify(`requestBody`): `Promise` < `$200` >

Verifies a JWT resolving the public key from the signer DID (no other kind of signer supported) and optionally check values for expected payload claims.

The Wallet only supports the 'ES256K1' algorithm.

Useful to verify JWT created by another wallet instance.

Parameters

Name	Type
`requestBody`	`RequestBody`

Returns

`Promise` < `$200` >

Inherited from

BokWallet.didJwtVerify

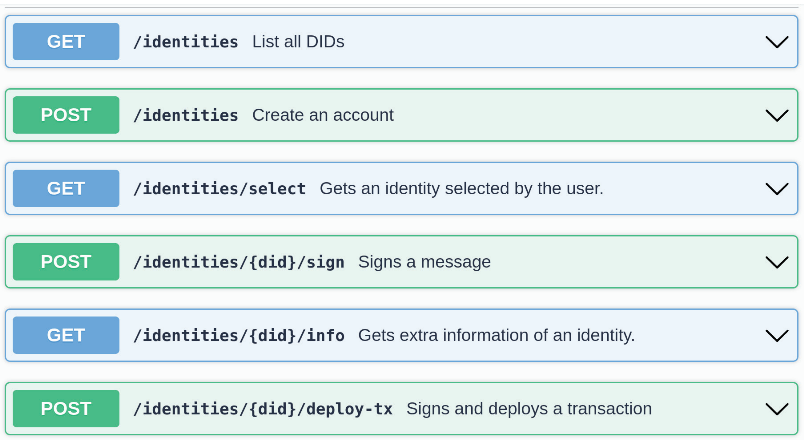
Defined in

base-wallet/types/wallet/base-wallet.d.ts:120

executeTransaction

▸ executeTransaction(`options?`): `Promise` < `void` >

Figure 3.7 A fragment of the Server Wallet API.



GET `/identities` List all DIDs

POST `/identities` Create an account

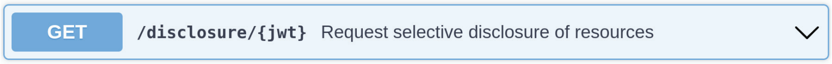
GET `/identities/select` Gets an identity selected by the user.

POST `/identities/{did}/sign` Signs a message

GET `/identities/{did}/info` Gets extra information of an identity.

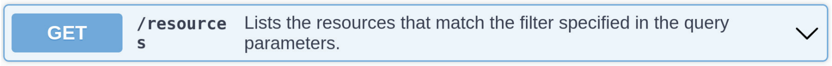
POST `/identities/{did}/deploy-tx` Signs and deploys a transaction

- **Selective disclosure:** Used to get a set of resources proving its ownership. They are signed with the requester private key.

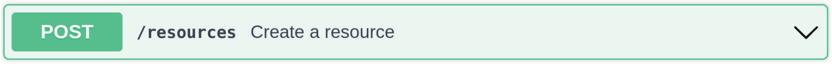


GET `/disclosure/{jwt}` Request selective disclosure of resources

- **Resource:** Besides identities and secrets, the wallet *may* securely store arbitrary objects in a secure vault. The list of requests for resource is shown in the following diagram:

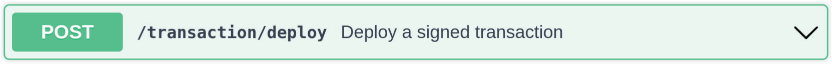


GET `/resources` Lists the resources that match the filter specified in the query parameters.



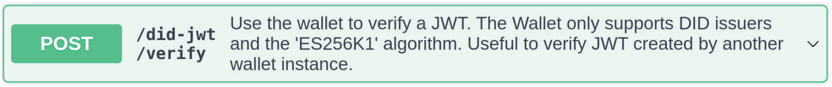
POST `/resources` Create a resource

- **Transaction:** Endpoints for deploying signed transactions to the DLT the wallet is connected to.

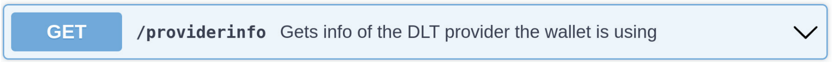


POST `/transaction/deploy` Deploy a signed transaction

- **Utils:** Additional helper functions.



POST `/did-jwt/verify` Use the wallet to verify a JWT. The Wallet only supports DID issuers and the 'ES256K1' algorithm. Useful to verify JWT created by another wallet instance.



GET `/providerinfo` Gets info of the DLT provider the wallet is using

3.6 Background Technologies

The Wallet Desktop application uses Electron [54] to build a cross-platform application. Electron is a framework for creating native applications using web technologies using chromium. It also has implemented the Node.js core libraries so that the application can easily access to the operative system functionalities.

The libraries of the wallet monorepo use Ethers.js and Veramo to implement the integration of the Wallet with the i3-MARKET DLT:

– Ethers.js [55] is a complete and compact library for interacting with the Ethereum blockchain. It was originally designed for use with ethers.io and has since expanded into a more general-purpose library.

– Veramo [56] is a JavaScript Framework for Verifiable Data that was designed from the ground up to be flexible and modular, which makes it highly scalable. It can run on several environments: node, mobile, and browser. Its main utility is to make easy the use of DIDs, Verifiable Credentials, and data-centric protocols to bring next-generation features to users.

Wallet OpenAPI meets the OpenAPI specification [57]. Wallet Desktop validates all inputs against the OpenAPI schema using express-openapi-validator [58].

Documentation for the different packages has been made available, thanks to TypeDoc [59].