Positive City Exchange Energy Marketplace Technical Documentation

Version 1.0 - August 2020 - IOTA Foundation

A Brief Overview Of IOTA

IOTA is a third-generation Distributed Ledger Technology (DLT), built using a revolutionary blockless distributed ledger, known as the Tangle. It solves many of the traditional problems with Blockchain. The Tangle is scalable, lightweight and makes it possible to transfer value and store immutable data without fees.

Why IOTA

IOTA develops a secure data communication protocol and zero-fee micropayment solutions for the Internet of Things and machine to machine applications.
Together with a growing ecosystem of corporations, start ups, public enterprises and universities, the IOTA Foundation addresses Digital Trust challenges and co-creates new data driven business models by setting collaboration across business sectors.

Core Benefits

- Open source technology that is developed and maintained by the non-profit IOTA Foundation
- Permissionless and secure network
- IOTA protocol ensures data privacy, immutability and authenticity
- Fee-less enabling a frictionless digital infrastructure for payments and data
- Lightweight core protocol to run on small embedded devices and sensors
- Real-time streaming of data and payments to avoid latency

Solution Summary

The solution described in the document provides the implementation of an energy flexibility trading marketplace. An *energy flexibility* market is a digital marketplace, where three actors interact to trade energy products. Energy flexibility is the energy (electric or thermal) that an asset produces in excess of its immediate needs. This energy flexibility can be transferred by an asset producing it to another one consuming it, for a given time and at a given agreed price.

The actors considered in the energy flexibility marketplace are: energy providers, energy consumers and marketplace operators.

An energy provider is the owner of the asset able to produce energy flexibility. An energy consumer is the owner of the asset that needs to consume energy. A producer's asset can be an EV charging station. A consumer's asset can be an electric vehicle (EV).

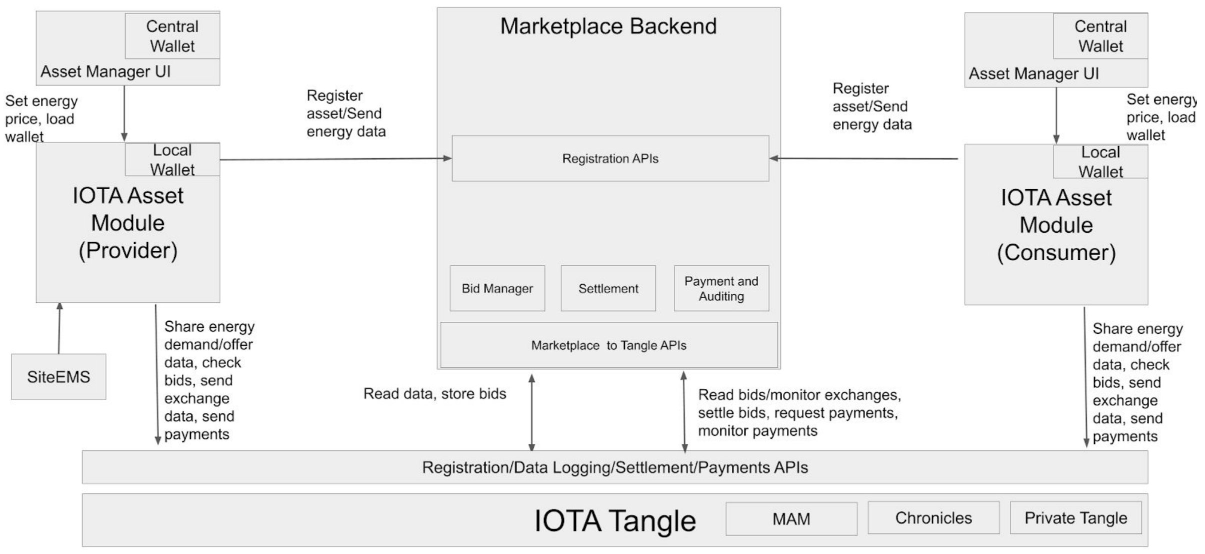
To measure in real-time the demand and offer of an asset, both devices are equipped with an IOTA asset module. The IOTA asset module allows the connected device to measure the requested and produced energy. The IOTA asset module also features an IOTA wallet to process asset to asset payments. An asset can act both as producer and consumer (i.e. prosumer). However at a given time, each asset can only take one role.

At a high level, an energy provider can create and share using the IOTA protocol an immutable contract offering energy (quantity and offered price), while an energy consumer can create one requesting energy (quantity and desired price). The energy marketplace backend matches optimal demand and offer, creates a trading bid and triggers the subsequent phases: energy transfer and direct consumer to provider payment.

The implemented marketplace allows providers and consumers to trade energy flexibility demand and offer in a decentralized way and to directly settle payments. The marketplace backend exposes the ability to connect custom algorithms for the optimal matching of demand and offer and creation of an energy trading contract.

Private/Public keys to sign information and encrypt communication are established during the initial Asset Module registration phase.

Figure below shows the high-level SW architecture.


Central Wallet
Asset Manager UI
Set energy price, load wallet
Local Wallet
IOTA Asset Module (Provider)
SiteEMS
Share energy demand/offer data, check bids, send exchange data, send payments
Register asset/Send energy data
Marketplace Backend
Registration APIs
Bid Manager
Settlement
Payment and Auditing
Marketplace to Tangle APIs
Read data, store bids
Read bids/monitor exchanges, settle bids, request payments, monitor payments
Register asset/Send energy data
Central Wallet
Asset Manager UI
Set energy price, load wallet
Local Wallet
IOTA Asset Module (Consumer)
Share energy demand/offer data, check bids, send exchange data, send payments
Registration/Data Logging/Settlement/Payments APIs
IOTA Tangle
MAM
Chronicles
Private Tangle

The system consists of the following elements:

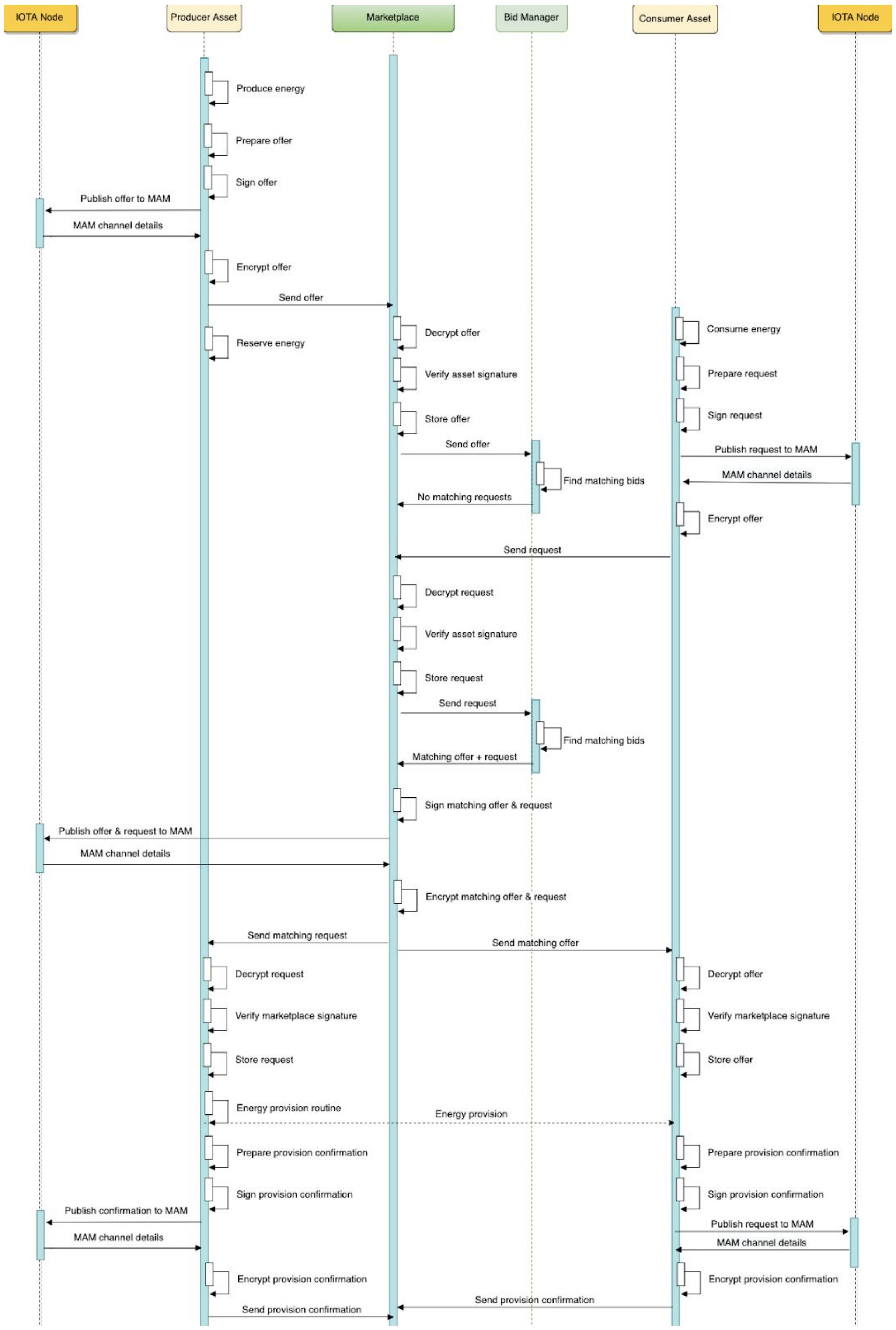
- An *IOTA Asset module*, a prototype HW and SW solution that allows *prosumers*, i.e., energy assets such as solar panels, charging stations, etc that provide and consume energy, to integrate with an energy Marketplace backend. The module allows to exchange information and payments between assets and Marketplace using the IOTA Tangle. 3rd parties energy meters (e.g., ABB SiteEMS) can connect directly to the Asset module to share information with the Marketplace. The module provides a UI for its management.
- A *Marketplace backend* and its related interfaces/APIs. A suite of SW modules that allow the following:
  1. Receives data directly from asset modules;
  2. Use the IOTA Tangle to read/store immutable data, i.e., energy demands and offers bids and agreed bids generated respectively by: IOTA asset modules, and the marketplace bid manager;
  3. Manage energy trading settlements and asset to asset payments

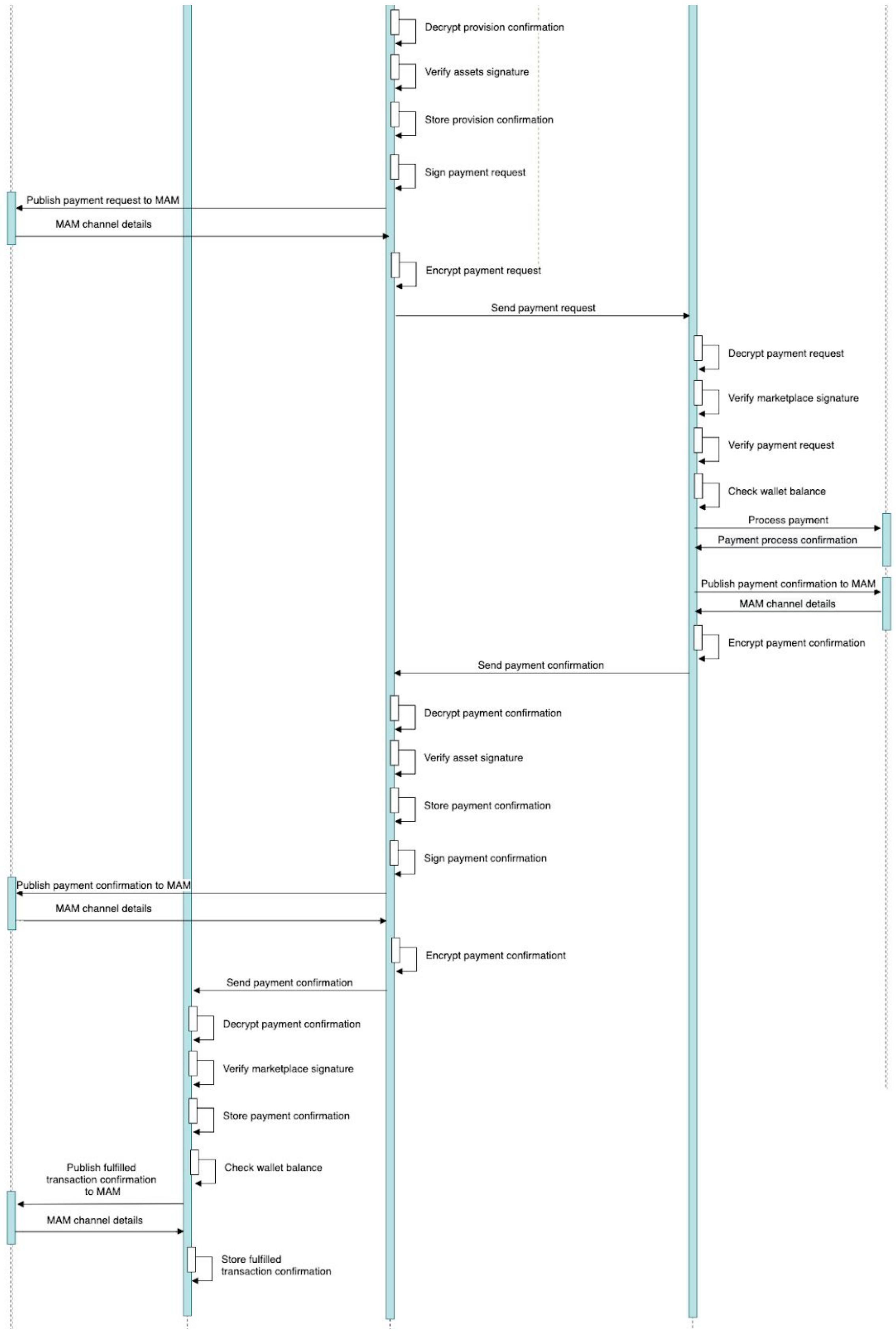
In particular, the IOTA asset module allows energy assets to locally generate energy offers and demands, and to share them with the Marketplace, directly and using the IOTA Tangle.

Use cases & Workflows

Sequence Diagram

The corresponding sequence of steps is provided in the sequence diagram below and further described.


IOTA Node
Producer Asset
Marketplace
Bid Manager
Consumer Asset
IOTA Node
Produce energy
Prepare offer
Sign offer
Publish offer to MAM
MAM channel details
Encrypt offer
Send offer
Reserve energy
Decrypt offer
Verify asset signature
Store offer
Send offer
Find matching bids
No matching requests
Consume energy
Prepare request
Sign request
Publish request to MAM
MAM channel details
Encrypt offer
Send request
Decrypt request
Verify asset signature
Store request
Send request
Find matching bids
Matching offer + request
Sign matching offer & request
Publish offer & request to MAM
MAM channel details
Encrypt matching offer & request
Send matching request
Send matching offer
Decrypt request
Verify marketplace signature
Store request
Decrypt offer
Verify marketplace signature
Store offer
Energy provision routine
Energy provision
Prepare provision confirmation
Sign provision confirmation
Publish confirmation to MAM
MAM channel details
Prepare provision confirmation
Sign provision confirmation
Publish request to MAM
MAM channel details
Encrypt provision confirmation
Send provision confirmation
Encrypt provision confirmation
Send provision confirmation


Decrypt provision confirmation
Verify assets signature
Store provision confirmation
Sign payment request
Publish payment request to MAM
MAM channel details
Encrypt payment request
Send payment request
Decrypt payment request
Verify marketplace signature
Verify payment request
Check wallet balance
Process payment
Payment process confirmation
Publish payment confirmation to MAM
MAM channel details
Encrypt payment confirmation
Send payment confirmation
Decrypt payment confirmation
Verify asset signature
Store payment confirmation
Sign payment confirmation
Publish payment confirmation to MAM
MAM channel details
Encrypt payment confirmationt
Send payment confirmation
Decrypt payment confirmation
Verify marketplace signature
Store payment confirmation
Check wallet balance
Publish fulfilled transaction confirmation to MAM
MAM channel details
Store fulfilled transaction confirmation

Asset module

Energy provider

- Energy produced
    - a new energy offer is created, price and amount are specified
    - energy offer details are signed with asset's private key, encrypted with public key of the marketplace backend and sent to the marketplace
    - energy offer details are signed with asset's private key, and stored in the new MAM channel[1], which is created for the new offer
    - energy offer details (complete transaction) are logged for audit in the local DB asset and communicated to the user module of the asset owner

- Contract received
    - energy provision started to identified energy consuming asset
    - energy production starts and produced energy is logged on the MAM channel
    - contract details are logged in the local DB and communicated to the user module of the asset owner

- Energy provision finished
    - energy pool updated to reflect the reduced energy amount
    - energy provision details are signed with asset's private key, encrypted with public key of the marketplace backend and sent to the marketplace
    - energy provision details are signed with asset's private key, and stored in the new MAM channel, which is created for the new offer
    - energy provision details are logged for audit in the local DB and communicated to the user module of the asset owner

- Payment processing confirmation received
    - payment confirmation details are logged for audit in the local asset's DB and communicated to the user module of the asset owner
    - wallet balance is checked in time intervals
        - incoming payments are evaluated and matched to unpaid contracts
            - Contract is marked as fulfilled, marketplace and energy consuming asset are notified
            - fulfilled energy offer details are encrypted with public key and stored in the MAM channel of the asset module
            - fulfilled energy offer details are encrypted with marketplace public key and sent to the marketplace
        - if no funds received, payment reminder is issued
            - payment reminder is signed with asset private key, encrypted with marketplace public key and sent to the marketplace
            - payment reminder includes a reference to the corresponding transaction

[1] Masked Authenticated Messaging: https://docs.iota.org/docs/mam/1.0/overview

            - if no funds received after 3 payment reminders, claim is issued
                - claim is signed with asset private key, encrypted with marketplace public key and sent to the marketplace
                - claim includes a reference to the corresponding transaction

Energy consumer

- Energy consumed
    - a new energy request is created, price and amount are specified
    - energy request details are signed with asset's private key, encrypted with public key of the marketplace backend and sent to the marketplace
    - energy request details are signed with asset's private key, and stored in the new MAM channel, which is created for the new offer
    - energy request details are logged for audit in the local DB and communicated to the user module of the asset owner

- Contract received
    - energy provision started from identified energy producing asset
    - energy consumption start and consumed energy is logged on the MAM channel
    - contract details are logged in the local DB and communicated to the user module of the asset owner

- Energy provision finished
    - energy pool updated to reflect the increased energy amount
    - energy provision details are signed with asset's private key, encrypted with public key of the marketplace backend and sent to the marketplace
    - energy provision details are signed with asset's private key, and stored in the new MAM channel, which is created for the new offer
    - energy provision details are logged for audit in the local DB and communicated to the user module of the asset owner

- Payment request received
    - payment request is verified
    - payment request details are logged for audit in the local DB and communicated to the user module of the asset owner
    - wallet balance is checked
        - if not enough funds, automated funding from the asset's user wallet is initiated
        - payment request is added to payment queue
        - payment is processed
        - payment confirmation details are signed with asset's private key, encrypted with public key of the marketplace backend and sent to the marketplace
        - payment confirmation details are signed with asset's private key, and stored in the new MAM channel, which is created for the new offer

        - payment confirmation details are logged for audit in the local DB and communicated to the user module of the asset owner

Marketplace module

- receives asset registration
    - stores asset ID, type, revocation address, MAM channel details, public key in a local database

- receives energy offer details
    - decrypts offer details using own private key and verify authenticity (sender signature)
    - stores offer details in local database
    - sends offer details to Bid Manager (/offer API), checks integrity (data stored in the Tangle)
    - logs offer details for audit

- receives energy request details
    - decrypts request details using own private key and verify authenticity (sender signature)
    - stores request details in local database
    - sends request details to Bid Manager (/request API), checks integrity (data stored in the Tangle)
    - logs request details for audit

- receives confirmed bid details from Bid Manager (/match API)
    - creates contract including matched offer and request bids[2]
    - signs contract details using own private key and encrypts the payload using producer's public key
        - sends contract details to energy producer
        - stores contract details to energy producer's MAM channel
    - signs contract details using own private key and encrypts the payload using consumer's public key
        - sends contract details to energy consumer
        - stores contract details to energy consumer's MAM channel
    - stores contract details in local database
    - logs contract details for audit

- receives energy provision/consumption confirmation (both producer and consumer)
    - decrypts provision details using own private key, verify authenticity (using send public key)
    - stores provision details in local database

[2] Simple 1to1 matching policies is implemented: (request energy < offered energy && offered price < requested price)

    - logs request details for audit
    - signs payment request details using own private key and encrypts the payload using consumer's public key
        - sends payment request details to energy consumer
        - stores payment request details to energy consumer's MAM channel

- receives payment confirmation (consumer)
    - decrypts payment confirmation details using own private key
    - signs contract details using own private key and encrypts the payload using producer's public key
        - sends confirmation details to energy producer's MAM channel
        - sends confirmation details to energy producer
    - stores payment confirmation details in local database
    - logs payment confirmation details for audit

- receives fulfillment confirmation (producer)
    - decrypts fulfillment confirmation details using own private key
    - stores fulfillment confirmation details in local database
    - logs fulfillment confirmation details for audit

- receives payment reminder (producer)
    - decrypts payment reminder details using own private key
    - signs payment reminder details using own private key and encrypts the payload using consumer's public key
        - sends payment reminder details to energy consumer
        - stores payment reminder details to energy consumer's MAM channel
    - stores payment reminder details in local database
    - logs payment reminder details for audit

- receives claim (producer)
    - decrypts claim details using own private key
    - stores claim details in local database
    - logs claim details for audit
    - optionally: applies claim resolution procedures, e.g. exclude consumers from the marketplace, starts legal procedure.

Asset Module: Technical Details

Remote device configuration

Marketplace script running on the device can be turned on/off and configured remotely from the Device management dashboard of the User module.

To modify device configuration, users should be logged in, select device from the list, and navigate to the Settings tab from where device settings can be adjusted.

Upon the initial device configuration, private/public keypair is generated and stored on the device. See “Key generation” topic for details. Public key is sent to the User module as part of the response.

To process configuration change on the device, the request should contain the correct device UUID, which is known only to the device owner.

Data storage

Marketplace script running on the device stores and reads data related to the transactions, energy consumption, wallet balance, MAM channel details, encryption keys, and logs in the local SQLite database. Content of the database can be viewed using an application such as “DB Browser for SQLite”.

Database file location: `/asset/db/energy_broker.sqlite3`

Job queues

To not block the marketplace script by a single running operation, which can for several seconds or even minutes, Redis job queue is used to schedule and parallelize tasks, handle timeouts and automatically rerun failing or stuck tasks.
To simplify and abstract out all Redis-related operations and configuration, a Bull queue package for handling distributed jobs and messages in NodeJS is used.
All long-running or recurring tasks such as payment processing, transaction verification, encryption/decryption are added to the queue and processed in parallel using Redis.
Bull queue manager provides a GUI to view job details, statistics, and logs It also provides functionality to manually retry or cancel failing jobs.
Bull queue dashboard can be accessed using the configured Device URL by navigating to the https://device_url/dashboard

Since the GUI does not provide authentication functionality, it can be accessed by anyone. Therefore this feature can be disabled using the Device management dashboard of the User module.
To enable/disable Bull GUI on the device, the request should contain the correct device UUID, which is known only to the device owner.

Bull queue configuration file location: `/asset/src/utils/queueHelper.ts`

Key generation

Marketplace script running on the device automatically generates private/public keypair upon the first successful initialization. The keypair is generated using an RSA library with a strong key length of 2048.

Key generation helper file location: `/asset/src/utils/encryptionHelper.ts`

Signing

Each request sent to the Marketplace or User module is signed by the device's private key. Signature is sent in the same request, and also published to the MAM channel of the transaction.
The receiving part should use the device's public key in order to verify the signature.

Encryption/decryption

Each request sent to the Marketplace or User module is encrypted using the recipient's public key.
The receiving part should use its private key in order to decrypt the payload.

Public keys of the Marketplace and User module are communicated to the device in the initial configuration message.

Digital-twin and MAM channels

For every new transaction a digital twin is created and stored in the new MAM channel. Messages in the channel are encrypted using a strong randomly-generated encryption key and stored in the local database.
Channel details are sent along with the transaction details to the Marketplace and User modules for verification purposes.

Payment request verification

Before processing payment for provided energy, payment request is verified by the energy consumer. Following verification steps are performed:

- Transaction ID is valid and can be found in DB
- Transaction status should be “Payment requested”
- Transaction digital twin could be fetched from MAM channel
- Signature of the initial transaction stored in the MAM channel was signed with device’s private key
- Initial transaction stored in DB is the same transaction storen in MAM channel
- Transaction properties from the initial transaction are the same as in the payment request transaction.
- Transaction properties from the contract transaction are the same as in the payment request transaction.

Payment processing

All valid and verified payment requests are added to the payment queue, which is processed according to the `paymentQueueProcessingSpeed` configuration parameter defined individually for every asset.
Payment queue processing consist of the following steps:

- Read up to 10 requests from the queue.
- Calculate total amount to be paid across all requests.
- Check devices’ own wallet balance and compare with the total amount to be paid.
- In case of an insufficient wallet balance send a request to the User module and request funding. This is an automated process.
- Prepare token transfer transactions and send them to the Tangle.
- Check transfer confirmations and verify that transactions were added to the Tangle
- Remove successfully paid requests from the payment queue
- Send payment confirmation to the Marketplace and update transaction status in the local DB

If the device is an energy producing device, instead of payment processing it performs payment confirmation for unpaid transactions.

Wallet funding

Wallet funding request is sent to the User module if wallet balance is not sufficient to pay for all contracts in the current iteration of the payment processing job.
Wallet funding is an automated process, which transfers tokens from the device’s owner wallet to the device wallet.
Wallet funding can be also triggered manually from the Device management dashboard of the User module.

Request resending and cancellation

All pending transactions are evaluated in recurring intervals. Transactions whose pending time exceeds the max. allowed duration will be processed as follows:

- Transactions with statuses 'Energy provision finished', 'Payment requested', 'Payment processed' or 'Claim issued' are considered unpaid. If the device is an energy consumer device, these transactions are added to the payment queue. In case of an energy provider a reminder for unpaid energy will be issued and sent to the Marketplace. See “Claim resolution” for details.
- Transactions with statuses 'Initial offer', 'Initial request' or 'Contract created' will be sent to the Marketplace as cancelled.

Claim resolution

Energy providers can issue payment reminders and claims if the energy was provided according to the contract, but payment was not received or could not be confirmed.
In this case the payment reminder is sent to the Marketplace, and a note is added to the transaction details to indicate the number of issued payment reminders per contract.
If the contract remains unpaid, another payment reminder is issued after the max. allowed waiting time is elapsed. The number of issued payment reminders is incremented by 1.
Once the number of payment reminders reaches 3 and the contract remains unpaid, the transaction will be sent to the Marketplace as a claim, which will have implications for the energy consuming device.

Logging

Results of all significant operations are logged to the internal DB, including error and success responses. The logs can be viewed by inspecting the `log` table of the database.

User notification

Transaction status updates are communicated to the User module of the device owner. Transaction details along with the MAM channel details and user ID are signed with the device's private key, encrypted with the owner's public key and sent to the `notify_event` API endpoint.

Transaction data model

An energy offer transaction consists of the following fields:

```
{
    "providerId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",               /* string */
    "providerTransactionId": "2X8icx565db9fcb3zYTy1o1",         /* string */
    "requesterId": "",                                          /* string */
    "requesterTransactionId": "",                               /* string */
    "energyAmount": 200,                                        /* number */
    "energyPrice": 0.15,                                        /* number */
    "type": "offer",                                            /* string */
    "walletAddress": "YTU9UXWRSZD9VVVAS...ZAZDXSQTVWYTAU",      /* string */
    "timestamp": "1593089013285",                               /* string */
    "location": "Berlin",                                       /* string */
    "status": "Initial offer",                                  /* string */
    "contractId": "",                                           /* string */
    "additionalDetails": ""                                     /* string */
}
```

An energy request transaction consists of the following fields:

```
{
    "providerId": "",                                           /* string */
    "providerTransactionId": "",                                /* string */
    "requesterId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",              /* string */
    "requesterTransactionId": "2X8icx565db9fcb3zYTy1o1",        /* string */
    "energyAmount": 200,                                        /* number */
    "energyPrice": 0.15,                                        /* number */
    "type": "request",                                          /* string */
    "walletAddress": "",                                        /* string */
    "timestamp": "1593089013285",                               /* string */
    "location": "Potsdam",                                      /* string */
    "status": "Initial request",                                /* string */
    "contractId": "",                                           /* string */
    "additionalDetails": ""                                     /* string */
}
```

Marketplace Module: Technical Details

Data storage

Marketplace script stores and reads data related to the transactions, MAM channel details, encryption keys, and logs in the local SQLite database. Content of the database can be viewed using an application such as “DB Browser for SQLite”.

Database file location: `/marketplace/db/marketplace.sqlite3`

Job queues

To not block the marketplace script by a single running operation, which can for several seconds or even minutes, Redis job queue is used to schedule and parallelize tasks, handle timeouts and automatically rerun failing or stuck tasks.
To simplify and abstract out all Redis-related operations and configuration, a Bull queue package for handling distributed jobs and messages in NodeJS is used.
All long-running or recurring tasks are added to the queue and processed in parallel using Redis.

Signing

Each request sent by the Marketplace to an asset is signed by the marketplace’s private key. Signature is sent in the same request, and also published to the MAM channel of the transaction.
The receiving part should use the marketplace’s public key in order to verify the signature.

Encryption/decryption

Each request sent by the Marketplace to an asset is encrypted using the recipient's public key.
The receiving part should use its private key in order to decrypt the payload.

Public keys of the assets are communicated to the Marketplace in the initial `/register` request.

Digital-twin and MAM channels

For every new transaction a digital twin is created and stored in the new MAM channel. Messages in the channel are encrypted using a strong randomly-generated encryption key and stored in the local database.
Channel details are received along with the transaction details by the Marketplace and sent back to the asset after a new message is added to the channel.

Logging

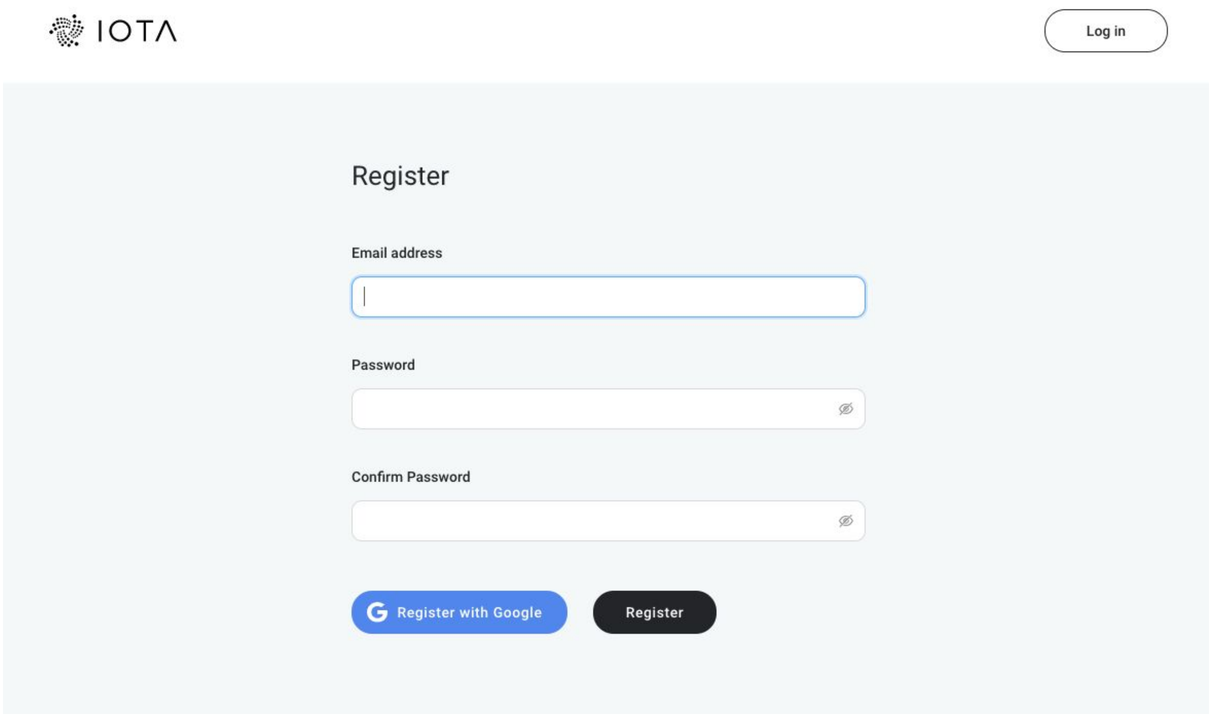
Results of all significant operations are logged to the internal DB, including error and success responses. The logs can be viewed by inspecting the `log` table of the database.

User Module: Functionality Description

The user module can be tested here: https://cityexchange-energymarketplace.web.app/[3]

Registration

New users can register using their Email and password, or by using their Google account.



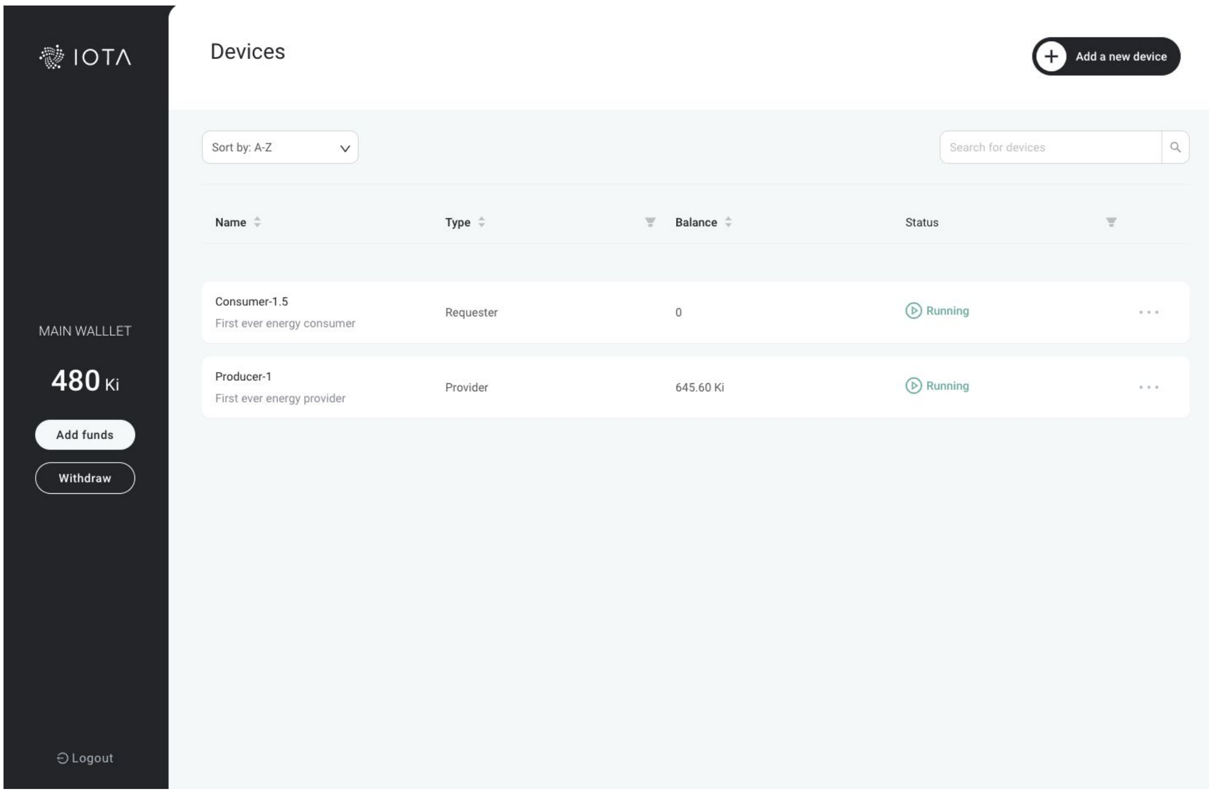
Assets overview

After logging in, users will be presented with the list of assets, including asset name, type, wallet balance and status. Additional info can be viewed by clicking on the asset row in the table.

Users wallet balance and action buttons to add and withdraw funds are displayed on the left sidebar.

Logout option is available at the bottom of the sidebar.

[3] Contact michele@iota.org for a test account

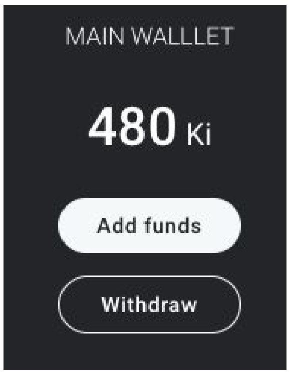


Name	Type	Balance	Status
Consumer-1.5 First ever energy consumer	Requester	0	Running
Producer-1 First ever energy provider	Provider	645.60 Ki	Running

Main wallet funding and tokens withdrawal

Main user's wallet can be funded from the exchange for the production-ready application. Tokens can be withdrawn back to the user's exchange account by pressing the "Withdraw" button.

In the current PoC the devnet tokens will be added to the user's main wallet after pressing the "Add funds" button. Devnet tokens can be withdrawn from the user's main wallet and sent back to IOTA by pressing the "Withdraw" button.

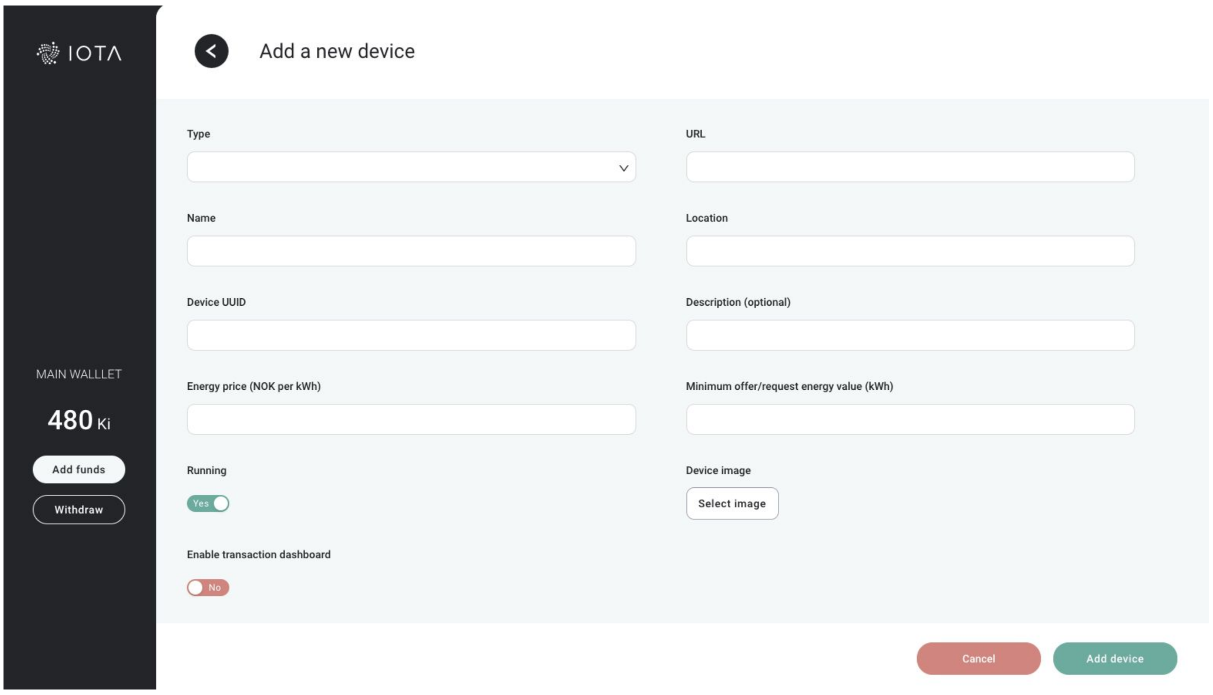


New asset onboarding

Users can onboard new assets by pressing the “Add a new device” button on the Overview page.

- Asset type can be set as Producer or Consumer.
- URL points to the asset’s API entry point.
- Device UUID should match the UUID of the physical device, which is also hardcoded in the asset configuration file under `/asset/src/config.json`
- Energy price is specified in IOTAs per kWh
- Minimum offer/request energy value: for energy providers, once the available energy amount exceeds this value, an energy offer will be created and sent to the Marketplace. For energy consumers, once the available energy amount drops under this value, an energy request will be created and sent to the Marketplace[4].
- Running flag specifies if the marketplace script on the device should immediately start operation or remain disabled.
- Enable transaction dashboard flag specifies if the GUI of the Bull queue should be enabled or disabled.

Internally a new wallet will be generated for the asset, configuration will be sent to the device and the marketplace script will be activated. The device in turn will register itself with the Marketplace, generate a keypair and return its public key back to the owner’s backend. Asset data along with the public key is stored in the internal database, the key will be used to verify the signature of the asset.

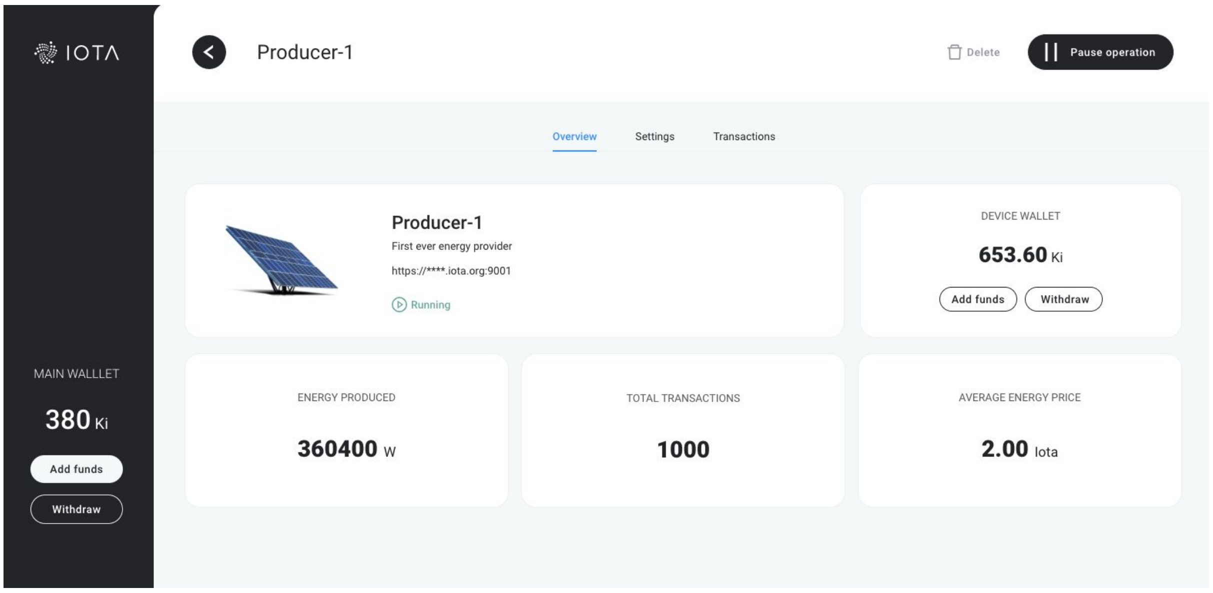


Type	URL
Name	Location
Device UUID	Description (optional)
Energy price (NOK per kWh)	Minimum offer/request energy value (kWh)
Running: Yes	Device image
Enable transaction dashboard: No	

[4] Offers are valid for 24 hours; duration is configurable in the settings and can be set to a minimum of 1 hour

Asset details overview

Asset details are shown on the overview page, including aggregated produced/consumed energy, number of fetched transactions and average energy price.



Producer-1
First ever energy provider
https://****.iota.org:9001
Running

DEVICE WALLET
653.60 Ki

MAIN WALLLET
380 Ki

ENERGY PRODUCED
360400 W

TOTAL TRANSACTIONS
1000

AVERAGE ENERGY PRICE
2.00 Iota

Asset management

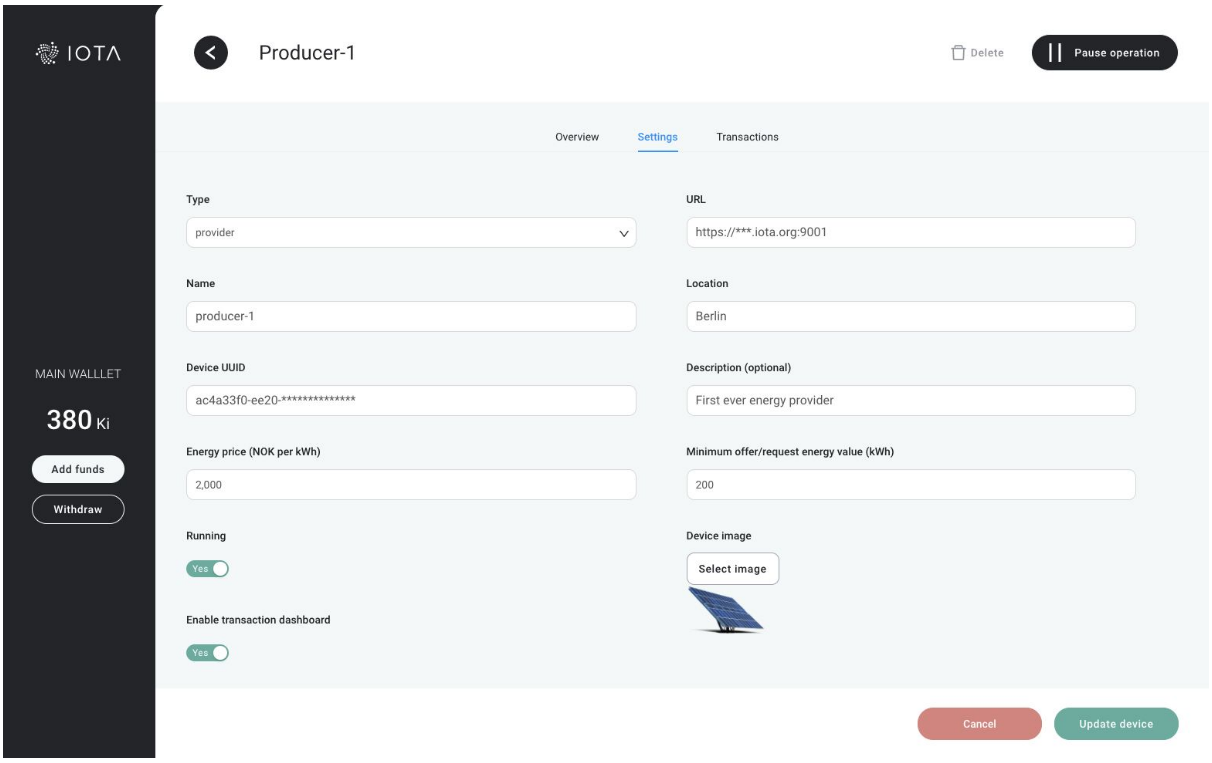
Asset marketplace script running on the device can be turned off by pressing the "Pause operation" button. It is re-enabled again by pressing the "Continue operation" button. An asset can be removed from the list, including its transactions and the associated wallet, by pressing the "Delete" button.

Internally the status flag will be sent to the device and the marketplace script will be activated or deactivated.

Asset configuration updates

Existing asset configuration can be updated by navigating to the Settings tab.

Internally the updated configuration will be sent to the device. The device will update the registration data with the Marketplace.



Producer-1

Type: provider

URL: https://***.iota.org:9001

Name: producer-1

Location: Berlin

Device UUID: ac4a33f0-ee20-**************

Description (optional): First ever energy provider

Energy price (NOK per kWh): 2,000

Minimum offer/request energy value (kWh): 200

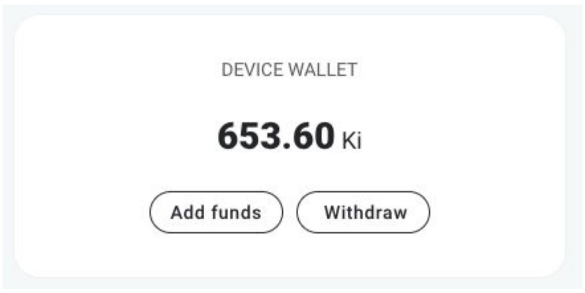
Running: Yes

Device image

Enable transaction dashboard: Yes

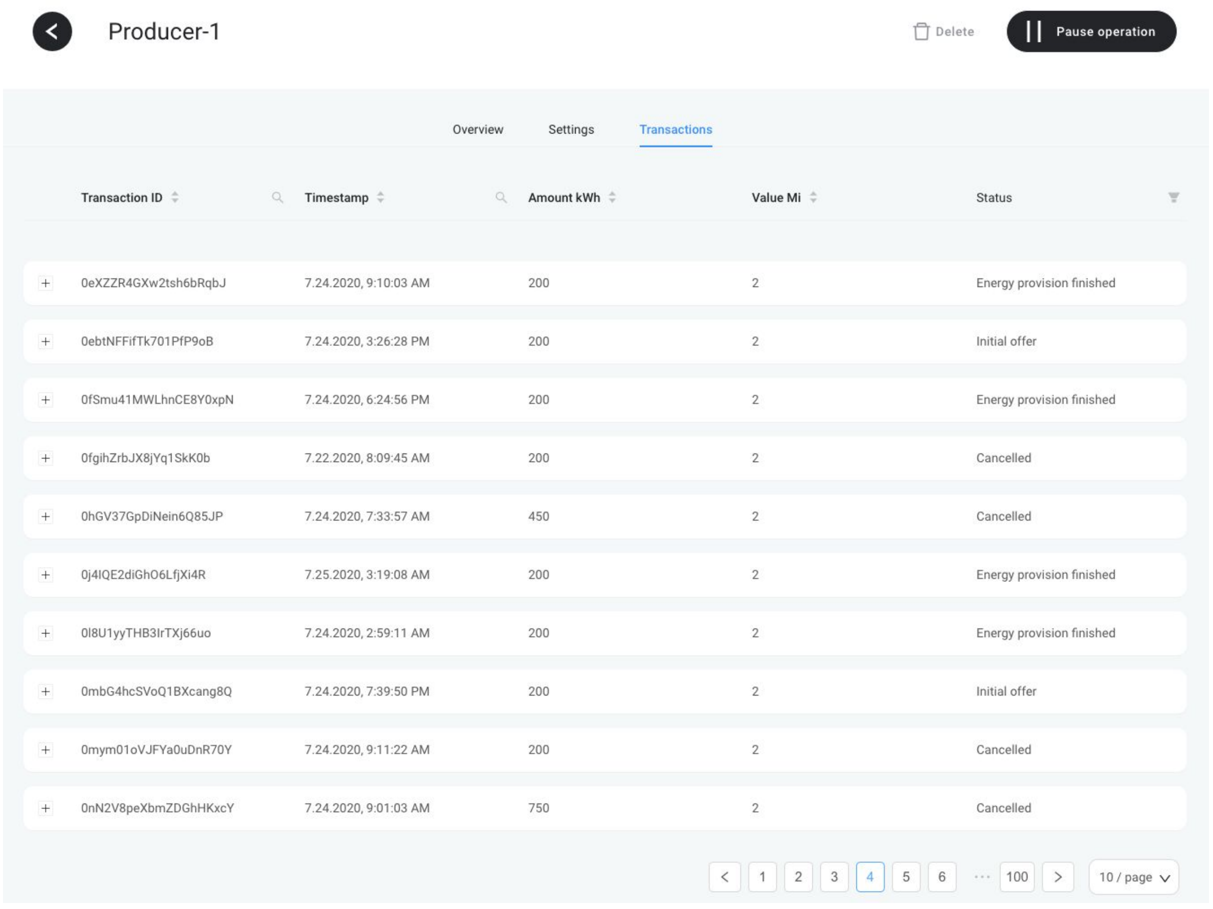
Asset wallet funding and tokens withdrawal

Internally the asset’s wallet will be funded from the owner’s wallet. Wallet balance will be updated.



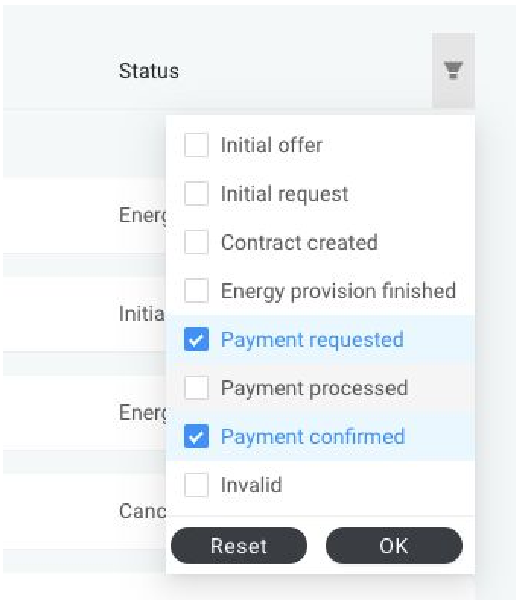
Transactions overview

Asset’s transactions are displayed in the table. Each entry is clickable, it expands all events of the given transaction.



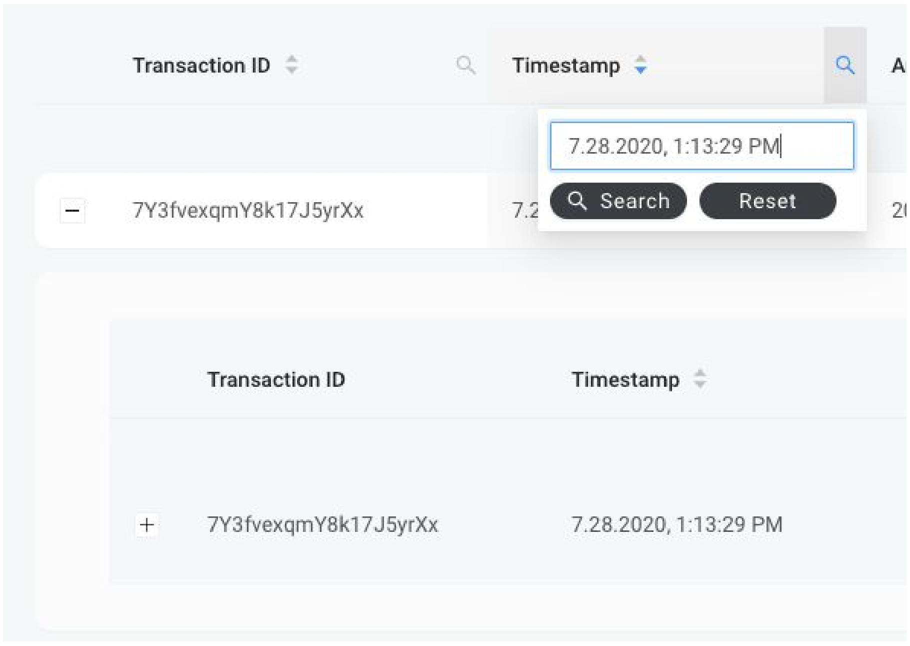
	Transaction ID	Timestamp	Amount kWh	Value Mi	Status
+	0eXZZR4GXw2tsh6bRqbJ	7.24.2020, 9:10:03 AM	200	2	Energy provision finished
+	0ebtNFFifTk701PfP9oB	7.24.2020, 3:26:28 PM	200	2	Initial offer
+	0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:56 PM	200	2	Energy provision finished
+	0fgihZrbJX8jYq1SkK0b	7.22.2020, 8:09:45 AM	200	2	Cancelled
+	0hGV37GpDiNein6Q85JP	7.24.2020, 7:33:57 AM	450	2	Cancelled
+	0j4IQE2diGhO6LfjXi4R	7.25.2020, 3:19:08 AM	200	2	Energy provision finished
+	0l8U1yyTHB3IrTXj66uo	7.24.2020, 2:59:11 AM	200	2	Energy provision finished
+	0mbG4hcSVoQ1BXcang8Q	7.24.2020, 7:39:50 PM	200	2	Initial offer
+	0mym01oVJFYa0uDnR70Y	7.24.2020, 9:11:22 AM	200	2	Cancelled
+	0nN2V8peXbmZDGhHKxcY	7.24.2020, 9:01:03 AM	750	2	Cancelled

The table can be filtered by transaction status, please use the filter icon as shown on the screenshot below.



Initial offer
Initial request
Contract created
Energy provision finished
[✓ Payment requested]
Payment processed
[✓ Payment confirmed]
Invalid

The table can be filtered by transaction ID or timestamp, please use the search icon as shown on the screenshot below.



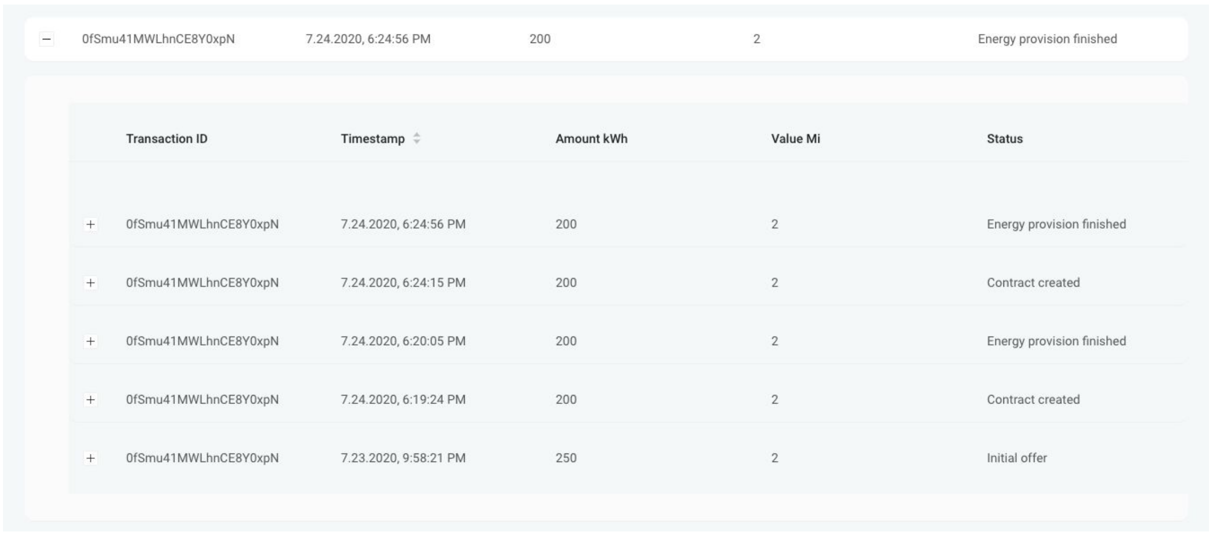
Transaction ID	Timestamp
7Y3fvexqmY8k17J5yrXx	7.2

7.28.2020, 1:13:29 PM

Transaction ID	Timestamp
7Y3fvexqmY8k17J5yrXx	7.28.2020, 1:13:29 PM

Transaction details

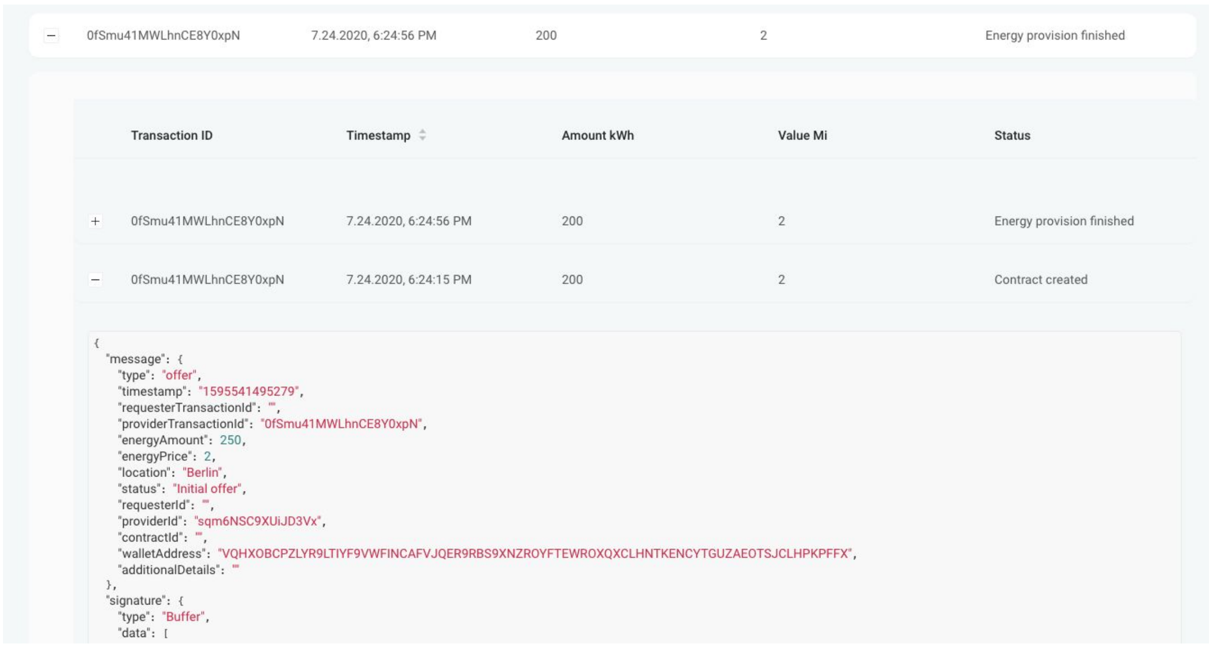
Once the transaction is expanded, all registered events of the transaction are shown.



0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:56 PM	200	2	Energy provision finished

Transaction ID	Timestamp	Amount kWh	Value Mi	Status
0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:56 PM	200	2	Energy provision finished
0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:15 PM	200	2	Contract created
0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:20:05 PM	200	2	Energy provision finished
0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:19:24 PM	200	2	Contract created
0fSmu41MWLhnCE8Y0xpN	7.23.2020, 9:58:21 PM	250	2	Initial offer

The data stored for each event can be verified for integrity and compared with the information stored securely and encrypted in the IOTA Tangle by clicking on the event row. In this case the MAM channel of the transaction will be fetched, decrypted and displayed as JSON content.



−	0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:56 PM	200	2	Energy provision finished

	Transaction ID	Timestamp	Amount kWh	Value Mi	Status
+	0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:56 PM	200	2	Energy provision finished
−	0fSmu41MWLhnCE8Y0xpN	7.24.2020, 6:24:15 PM	200	2	Contract created

```
{
  "message": {
    "type": "offer",
    "timestamp": "1595541495279",
    "requesterTransactionId": "",
    "providerTransactionId": "0fSmu41MWLhnCE8Y0xpN",
    "energyAmount": 250,
    "energyPrice": 2,
    "location": "Berlin",
    "status": "Initial offer",
    "requesterId": "",
    "providerId": "sqm6NSC9XUiJD3Vx",
    "contractId": "",
    "walletAddress": "VQHXOBCPZLYR9LTIYF9VWFINCAFVJQER9RBS9XNZROYFTEWROXQXCLHNTKENCYTGUZAEOTSJCLHPKPFFX",
    "additionalDetails": ""
  },
  "signature": {
    "type": "Buffer",
    "data": [
```

API Specification

Asset module

Init

Remote asset initialisation or configuration update. The payload is sent unencrypted, but the asset accepts the payload only when the provider UUID matches the internal UUID of the device, which is only known to the owner of the device.

Endpoint

POST https://[asset_URL]/init

Request

```
{
    "exchangeRate": 0.15,                                               /* number */
    "maxEnergyPrice": 20,                                               /* number */
    "minOfferAmount": 200,                                              /* number */
    "assetOwnerAPI": "https://energymarketplace.com",                   /* string */
    "marketplaceAPI": "https://marketplace.com:5000",                   /* string */
    "assetId": "sTm6NSC9XUiJD3Vx",                                      /* string */
    "assetName": "energy-producer-XYZ",                                 /* string */
    "assetDescription": "Device model XYZ, type ABC",                   /* string */
    "type": "provider",                                                 /* string */
    "assetOwner": "SpXe6LkNNHSbidheZKzDLzq5ZQ33",                       /* string */
    "network": "devnet",                                                /* string */
    "location": "Berlin",                                               /* string */
    "assetWallet": {                                                    /* object */
       "seed": "CNYAOXQJOYQOMNWEBH...DIZDLHEHSXELMRTLC9999",            /* string */
       "address": "YLSDLWPLLLFUYNFE9GJ...9OWHRQDXCBIXLLSTGSJ",          /* string */
       "keyIndex": 0,                                                   /* number */
       "balance": 0                                                     /* number */
     },
    "marketplacePublicKey": "-----BEGIN PUBLIC KEY-----5UC9ud",         /* string */
    "assetOwnerPublicKey": "-----BEGIN PUBLIC KEY-----7Fsvkg3",         /* string */
    "status": "running",                                                /* string */
    "dashboard": "enabled",                                             /* string */
    "uuid": "ba4a33f0-ee20-41e4-9fcd-9f91ecf77d0f",                     /* string */
    "url": "https://device.com:7001",                                   /* string */
}
```

Response

```
{
   "success": true,
   "publicKey": "-----BEGIN PUBLIC KEY-----uwIv0kCrRlbYoDmyr6HYbqAbuRU15"
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Contract

After a match between energy offer and energy request was found by the Bid Manager, and the contract between energy provider and energy requester is established, this endpoint is used to communicate contract details. The request payload consists of the offer transaction object, request transaction object, and the contract ID. Both contract partners, energy provider and energy requester, will receive the same `/contract` request.
As a result of the `/contract` request the energy provider can immediately start energy provision to the consumer.

Endpoint

POST https://[asset_URL]/contract

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Payment

Once energy provision is finished, and energy provision confirmation sent to the Marketplace, the energy requester will receive the `/payment` request from the Marketplace. Encrypted payload consists of the contract transaction object with updated status and timestamp.
As a result of the `/payment` request the energy requester verifies if the request is valid by comparing a number of transaction parameters with the data stored in the local DB and in the corresponding MAM channel. If the payment request is valid, it will be added to the payment queue and processed together with other payment requests. See “Payment processing” topic.

Endpoint

POST https://[asset_URL]/payment

Request

```
{
     "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
    "success": true
}

In case of an error:
{
    "success": false,
    "error": [error message]
}
```

Payment sent

Once the energy requester reports successful payment processing to the Marketplace, the energy provider will receive the `/payment_sent` request from the Marketplace. Encrypted payload consists of the payment request transaction object with updated status and timestamp.
As a result of the `/payment_sent` request the energy provider logs the transaction into the local DB and verifies payment in defined time intervals.

Endpoint

POST https://[asset_URL]/payment_sent

Request

```
{
     "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Payment confirmed

Once the energy provider reports successful payment confirmation to the Marketplace, the energy requester will receive the `/payment_confirmed` request from the Marketplace. Encrypted payload consists of the payment confirmation transaction object with updated status and timestamp.
As a result of the `/payment_confirmed` request the energy requester logs the transaction into the local DB and communicates to the User module.

Endpoint

POST https://[asset_URL]/payment_confirmed

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Cancel

Pending and unpaid transactions that remain unpaid when the max. allowed waiting time is elapsed, can be cancelled. Both asset types can receive `/cancel` requests from the Marketplace. Encrypted payload consists of the transaction object with updated status and timestamp.
As a result of the `/cancel` request the asset logs the transaction into the local DB and communicates to the User module. Energy providers will additionally unreserve energy that was reserved for the offer.

Endpoint

POST https://[asset_URL]/cancel

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Claim

Energy providers can issue claims for transactions that remain unpaid after 3 payment reminder requests. Energy consumers can receive `/claim` requests from the Marketplace. Encrypted payload consists of the transaction object with updated status and timestamp.
As a result of the claim, energy consuming devices can be excluded from the Marketplace, and a penalty can be issued to the device's owner.

Endpoint

POST https://[asset_URL]/claim

Request

```
{
     "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
    "success": true
}
```

In case of an error:

```
{
    "success": false,
    "error": [error message]
}
```

Produce energy

Energy providers equipped with physical units to produce energy, will notify the script running on the device about the additional generated energy, available to be offered on the Marketplace.

The payload is sent unencrypted, but the asset accepts the payload only when the provider UUID matches the internal UUID of the device, which is only known to the owner of the device or the physical unit.

As a result of the `/produce` request, available energy amount is updated in the internal DB. If the total available energy exceeds the `minOfferAmount` value, a new energy offer is created and sent to the marketplace.

Endpoint

POST https://[asset_URL]/produce

Request

```
{
    "energy": 300,                                                /* number */
    "uuid": "ba4a33f0-ee20-41e4-9fcd-9f91ecf77d0f"               /* string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Consume energy

Energy consumers equipped with physical units to consume energy, will notify the script running on the device about the consumed energy values.

The payload is sent unencrypted, but the asset accepts the payload only when the provider UUID matches the internal UUID of the device, which is only known to the owner of the device or the physical unit.

As a result of the `/consume` request, available energy amount is updated in the internal DB. If the total available energy drops below the `minOfferAmount` value, a new energy request is created and sent to the marketplace.

Endpoint

POST https://[asset_URL]/consume

Request

```
{
    "energy": 300,                                          /* number */
    "uuid": "ba4a33f0-ee20-41e4-9fcd-9f91ecf77d0f"          /* string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Marketplace module

Register

Initial asset registration with Marketplace. The request payload consists of the asset's data such as ID, public key, type, location, URL. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/register` request, the payload is decrypted, signature is verified, then asset info is stored in the local database of the marketplace.

Endpoint

POST https://[marketplace_URL]/register

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Offer

Energy producing assets issue energy offers and communicate them to the marketplace. The payload consists of the offer transaction object.
The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/offer` call, the payload is decrypted, signature is verified, then the offer transaction is stored in the internal database of the Marketplace and communicated to the Bid Manager to find a matching request transaction.

Endpoint

POST https://[marketplace_URL]/offer

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Request

Energy consuming assets issue energy requests and communicate them to the marketplace. The payload consists of the request transaction object.
The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/request` call, the payload is decrypted, signature is verified, then the request transaction is stored in the internal database of the Marketplace and communicated to the Bid Manager to find a matching offer transaction.

Endpoint

POST https://[marketplace_URL]/request

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Match

Response is sent from Bid manager to Marketplace. The payload is sent unencrypted, as Marketplace and Bid manager are located in the same network. As a result of the `/match` call, a payload containing the offer transaction object, request transaction object, and the contract ID is created, signed by the Marketplace and sent to both contract partners, energy provider and energy requester.

Endpoint

POST https://[marketplace_URL]/match

Request

```
{
    "offer": {
        "providerId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",           /* string */
        "providerTransactionId": "2X8icx565db9fcb3zYTy1o1",     /* string */
        "energyPrice": 0.15,                                    /* number */
        "walletAddress": "YTU9UXWRSZVVAS...ZAZDXSQTWYTAU",      /* string */
        "additionalDetails": ""                                 /* string */
        ...                        /* additional irrelevant offer details */
    },
    "request": {
        "requesterId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",          /* string */
        "requesterTransactionId": "2X8icx565db9fcb3zYTy1o1",    /* string */
        "energyAmount": 200,                                    /* number */
        "additionalDetails": ""                                 /* string */
        ...                      /* additional irrelevant request details */
    }
}
```

Response

```
{
   "success": true
}
```

Provision

Once energy provision is finished, an energy provision confirmation is sent to the Marketplace. Encrypted payload consists of the contract transaction object with updated status and timestamp. Both asset types can send the provision confirmation to the Marketplace.
The payload consists of the request transaction object. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/provision` call, the payload is decrypted, signature is verified, then a payment request is sent to the energy consuming asset. The transaction is stored in the internal database of the Marketplace.

Endpoint

POST https://[marketplace_URL]/provision

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Payment processing

The energy requester reports successful payment processing to the Marketplace. Encrypted payload consists of the payment transaction object with updated status and timestamp. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/payment_processing` call, the payload is decrypted, signature is verified, then a payment confirmation is sent to the energy provider. The transaction is stored in the internal database of the Marketplace.

Endpoint

POST https://[marketplace_URL]/payment_processing

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Payment confirmation

The energy provider reports payment confirmation to the Marketplace. Encrypted payload consists of the payment transaction object with updated status and timestamp. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/payment_confirmation` call, the payload is decrypted, signature is verified, then a payment confirmation is sent to the energy requester. The transaction is stored in the internal database of the Marketplace.

Endpoint

POST https://[marketplace_URL]/payment_confirmation

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Cancel

Transactions that remain in its initial state after the max. allowed waiting time is elapsed, will be cancelled. Both asset types can send `/cancel` requests to the Marketplace. Encrypted payload consists of the transaction object with updated status and timestamp. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/cancel` request the Marketplace logs the transaction into the local DB. If the transaction already contains the contractID, the contract partner other than the sender of the request will be notified.

Endpoint

POST https://[marketplace_URL]/cancel

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Claim

Energy providers can issue claims for transactions that remain unpaid after 3 payment reminder requests. Encrypted payload consists of the transaction object with updated status and timestamp. The payload is sent encrypted with the public key of the Marketplace and signed by the asset.
As a result of the `/claim` request, the Marketplace notifies the energy consumer and can optionally perform additional steps like excluding the asset from the marketplace or issue a penalty to the asset's owner. The Marketplace logs the transaction into the local DB.

Endpoint

POST https://[marketplace_URL]/claim

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
    "error": [error message]
}
```

Bid manager module

Offer

New offer data received and processed by the Marketplace module is communicated to the Bid manager. The payload is sent unencrypted, as Marketplace and Bid manager are located in the same network.
As a result of the `/offer` call, Bid manager performs lookup for a matching request. If a match is found, a matching request is removed from the internal database, and both transactions are bundled and communicated to the Marketplace, otherwise the offer is stored in the internal database.

Endpoint

POST https://[bidmanager_URL]/offer

Request

```
{
    "providerId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",              /* string */
    "providerTransactionId": "2X8icx565db9fcb3zYTy1o1",        /* string */
    "energyAmount": 200,                                       /* number */
    "energyPrice": 0.15,                                       /* number */
    "type": "offer",                                           /* string */
    "walletAddress": "YTU9UXWRSZD9VVVAS...ZAZDXSQTVWYTAU",     /* string */
    "timestamp": "1593089013285",                              /* string */
    "additionalDetails": ""                                    /* string */
}
```

Response

```
{
    "success": true
}
```

In case of an error:

```
{
    "success": false,
    "error": [error message]
}
```

Request

New request data received and processed by the Marketplace module is communicated to the Bid manager. The payload is sent unencrypted, as Marketplace and Bid manager are located in the same network.
As a result of the `/request` call, Bid manager performs lookup for a matching offer. If a match is found, a matching offer is removed from the internal database, and both transactions are bundled and communicated to the Marketplace, otherwise the request is stored in the internal database.

Endpoint

POST https://[bidmanager_URL]/request

Request

```
{
    "requesterId": "1MVaAFB7tvU8qp6rYgmE7pkoATT2",              /* string */
    "requesterTransactionId": "2X8icx565db9fcb3zYTy1o1",        /* string */
    "energyAmount": 200,                                        /* number */
    "energyPrice": 0.15,                                        /* number */
    "type": "request",                                          /* string */
    "timestamp": "1593089013285",                               /* string */
    "additionalDetails": ""                                     /* string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Remove

Called when the offer or request was cancelled. As a result of the `/remove` call the offer or request is removed from the internal database.

Endpoint

POST https://[bidmanager_URL]/remove

Request

```
{
    "requesterTransactionId": "2X8icx565db9fczYTy1o1",           /* string */
    "type": "request"                                            /* string */
    ...                             /* additional irrelevant request details */
}

OR

{
    "providerTransactionId": "1MVaAFB7tvU8qrYgmE7TT2",           /* string */
    "type": "offer"                                              /* string */
    ...                               /* additional irrelevant offer details */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

User module

Fund

Energy consumers can request wallet funding from the asset's owner, if the wallet balance is not sufficient to pay for the consumed energy.

The payload consists of the asset's wallet address, current balance, asset type and asset ID. The payload is sent encrypted with the public key of the asset's owner and signed by the asset.
As a result of the `/fund` call, the payload is decrypted, signature is verified, then a token transfer is initiated from the asset's owner wallet to the asset's wallet.

Endpoint

POST https://[user_module_URL]/fund

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}

In case of an error:
{
   "success": false,
   "error": [error message]
}
```

Notify event

All asset types notify their owners about the new offers/request and any change in the status of the existing requests.
The payload consists of the transaction object. The payload is sent encrypted with the public key of the asset's owner and signed by the asset.
As a result of the `/notify_event` call, the payload is decrypted, signature is verified, then the transaction is stored in the database for audit.

Endpoint

POST https://[user_module_URL]/notify_event

Request

```
{
    "encrypted": "bxC9mdhuJU6RZrb7iy...LoPN8XCNfusy6W==" /* Base64 string */
}
```

Response

```
{
   "success": true
}
```

In case of an error:

```
{
   "success": false,
   "error": [error message]
}
```

Glossary Of Terms

- MAM - Masked Authentication Messaging is a second layer data communication protocol which adds functionality to emit and access an encrypted data stream, like RSS, over the Tangle https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e

Additional Resources

- Dashboard - https://cityexchange-energymarketplace.web.app
- Project Repository - https://github.com/iotaledger/iota-energy-broker
- MAM Repository - https://github.com/iotaledger/mam.js

Project Partners

The work has been developed as part of the EU CxC H2020 project, ref number, focusing on creation of positive energy districts. The following project partners have been involved in the design and testing of the marketplace developed by IOTA Foundation.

Powel

Powel develops and delivers ICT software solutions and technology in an international market to companies in the power industry and municipalities. In the DSO (distribution system operation) industry the solutions from Powel are used for planning, operation, documentation, work process operation, and optimization regarding distribution system management (DMS). For municipalities Powel delivers solutions for documentation and management of technical infrastructure including water and sewage. Powel also delivers smart city services for use by managers and/or citizens. A significant part of Powel solutions and services is power generation planning and optimization in addition to market bidding, trade and settlement. Powel holds excellent industry domain knowhow and 50% of its revenue comes from consultancy services while the other part is from software products and solutions. Powel technology includes data management from smart meters and distribution management systems in smart electricity grids.

Powel is working on integrating with its algotrader

NTNU

NTNU is the largest university in Norway, with 14 faculties and 70 departments and divisions. NTNU has more than 39 000 students and more than 4 600 person-years in academic or scientific positions. The university uses its main scientific profile in technology and the natural sciences and its cross-disciplinary competency to meet global challenges, summarized by its vision: Knowledge for a better world. Three out of four Strategic Research Areas at NTNU – Sustainability, Energy and Health – contribute directly to sustainable urbanization, delivering creative innovations with far reaching social and economic impact in close collaboration with cities, industries, authorities and civil society.

NTNU will offer deployment and testing of IOTA Asset Module and marketplace functionalities in the university campus and using assets deployed by the project.

IOTA Foundation

The IOTA Foundation (IF) is a non-profit organization supporting the development and adoption of the IOTA Tangle, a permissionless Distributed Ledger Technology (DLT), particularly suitable for creating trusted information and value sharing across multi-stakeholder ecosystems. IOTA technology is open source. IF's agile approach to solutions creation leverages on identifying industry problems with real stakeholders, building PoCs and collaboratively testing and validating or refining assumptions.

IOTA is built for the decentralization of machine to machine transactions and communication. With its core technology IOTA supports the free access for everyone in a dedicated environment to participate and benefit from new sources of data being generated by all participants of that decentralized system. In addition, IOTA is a leading innovation facilitator leveraging innovative start-ups and ecosystems. Therefore IOTA envisions a tremendous potential for the future of access management - related to new data sources and physical systems at the same time. In IOTA's ecosystem a couple of relevant innovative solutions have already been developed - ranging from biometric palm vein authentication to Digital IDs for entire cities.

Contact Us

Web: https://www.iota.org
Blog: https://blog.iota.org
Discord: https://discord.iota.org
Twitter: @iotatoken
Email: contact@iota.org