



ZENODYS PROTOCOL WHITE PAPER

---

# Zenodys protocol specification and data format standardization

Q2 - 2018  
VERSION: 0.1

<b>Contents</b>	<b>2</b>
<b>Preface</b>	<b>3</b>
<b>1. Intro</b>	<b>3</b>
<b>2. Digital assets trading overview</b>	<b>4</b>
<b>3. Protocol implementation</b>	<b>5</b>
3.1 Publishing and accepting asset terms	5
3.2 Off-chain asset exchange	5
<b>4. Zenodys Protocol in action</b>	<b>10</b>

## Preface

This document provides an overview of *Zenodys Protocol* and standard assets data format for secure exchange inside *Zenodys Network*.

It's assumed that reader have basics about *Visual Elements* and *Computing Engine* described in the [technical whitepaper](#).

Current protocol implementation is for *.NET Computing Engine*. Protocol improvements, upgrades and implementation for *C Computing Engine* will be implemented in future iterations.

## 1. Intro

*Zenodys Platform* is a development tool based on *Visual Elements*.

Implementation of *Zenodys Protocol* is also in form of *Visual Elements* - users will trade their digital assets in a secure and flexible way by just drag'n'dropping protocol elements, setting their properties and combine them with other existing *Elements* inside the *Platform*.

In general, three types of *Elements* are needed for transferring digital asset from owner to customer:

- **ZenLicenceChecker**: this *Element* is running in *Template* on asset owner's *Computing Engine*. It listens for asset requests and validates them.
- **ZenAssetTransmitter**: *Element* that transmits assets from owner to customer after successful licence check
- **ZenAssetReceiver**: *Element* that runs on customer *Computing Engine*. It initiates request and receives assets.

Their implementation can be found on [GitHub](#)

Those are standard *Visual Element* implementations - like existing ones inside *Zenodys Platform*. Those microservice implementations *Computing Engine* understands and executes.

In general, there are two types of *Elements* - ones that execute some actions (*ZenAssetTransmitter*, *ZenAssetReceiver*) and ones that wait some event to happen (*ZenLicenceChecher*).

It advisable to check [Action](#) and [Eventable](#) *Elements* for better code understanding.

Existing *Elements* and both *Computing Engines* will be shortly open sourced.

## 2. Digital assets trading overview

Digital assets trading is two phase process that contains publishing and accepting asset terms on marketplace and off-chain asset exchange.

Digital assets can be different types. For example it can be data from device, file, algorithm, dApp, *Element*... That's why trading process can be slightly different from type to type, but in general steps are:

- Asset owner publishes new asset to marketplace and uploads *Template* to his *Computing Engine* (or to *Zenodys Node* if asset is hosted on *Zenodys Network*)
- Customer that wants to access digital asset must accept terms written in smart contract and deploy *Template* that will deliver asset to his *Computing Engine*.
- Licence terms are executed each time customer's *Computing Engine* delivers asset.

*Zenodys Marketplace* contains different types of licences:

- Volume based licence that contains quantity how many times asset can be accessed
- Subscription based licence allowing customer access asset for a given period of time.

Other licence types will be added in next versions.

## 3. Protocol implementation

All **security implementation** (digital asset encryption/decryption, authentication, keystore generation...), **smart contract auto generation/deployment, communication and business logic** is implemented inside *Zenodys Platform/Protocol* and executed by just setting the properties and connecting *Visual Elements*. It does not require any cryptographic or programming knowledge from the users.

### 3.1 Publishing and accepting asset terms

Publishing and accepting asset terms is a process that runs on the marketplace dApp. When a user adds new digital assets to the marketplace, a smart contract is automatically generated and published to the Ethereum network. Example of auto generated smart contract for volume based licence can be found on [github](#).

When the customer accepts the terms, [addLicence](#) function that inserts the licence details is called:

- **licenceld** - unique customer's digital asset purchase identification
- **customer** - customer's Ethereum address
- **price** - price in tokens. Token quantity might vary from customer to customer (bonuses and discounts might be included) so it's defined on licence level
- **quantity** - number of permitted accesses to digital asset. Licence is removed when quantity reaches zero. After that customer can extend licence under same or different conditions

### 3.2 Off-chain asset exchange

To start with actual asset exchange, owner and customer must have deployed *Templates* on their *Computing Engines*.

#### Cryptography Overview

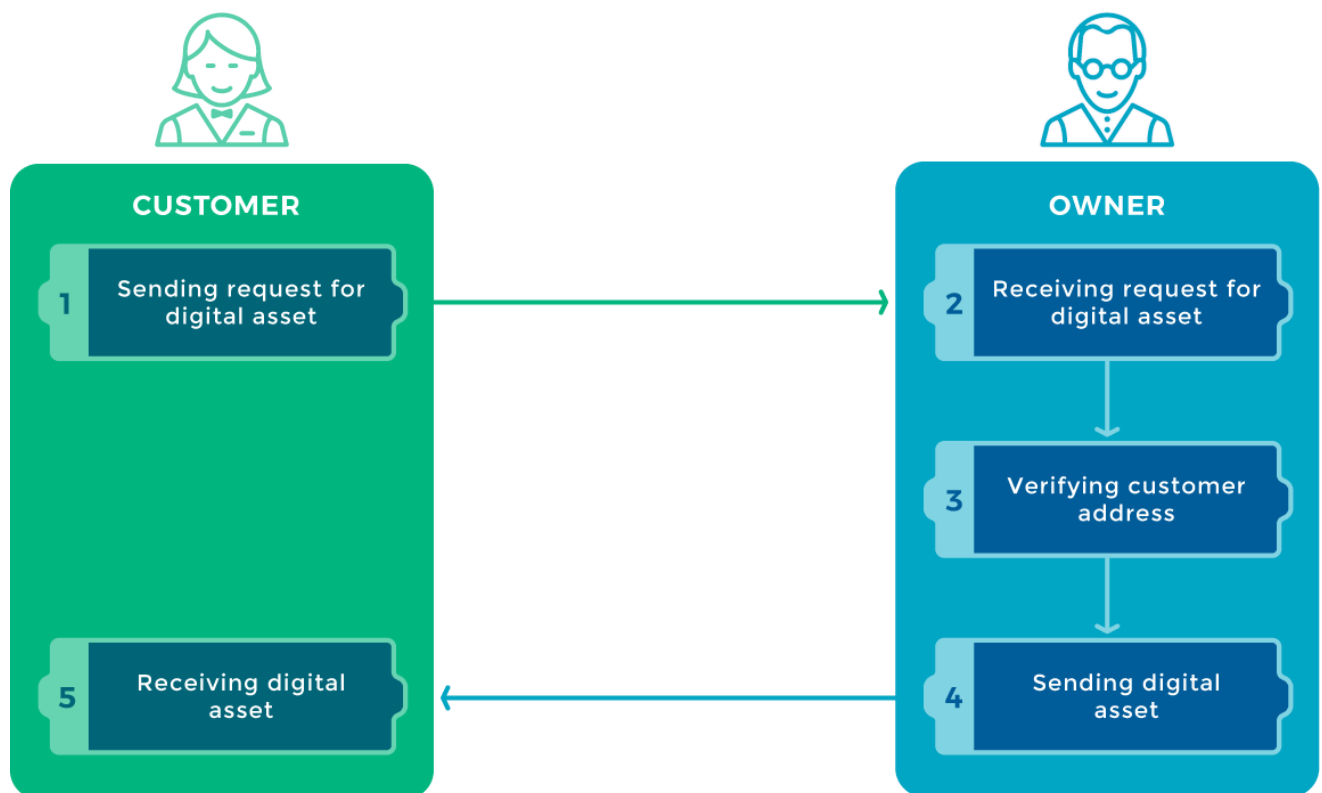
There are two kinds of encryption algorithms:

- For **symmetric-key algorithm**, the same cryptographic key is used for both encryption and decryption. Symmetric-key algorithms are faster and less processor intensive than asymmetric-key algorithms. But there is a problem - same key is used for both encryption and decryption, and this leads to big problem of key distribution from encryption (sender) to decryption side (receiver).

- **Asymmetric key algorithms** require two separate keys - public key is used to encrypt plain text or to verify a digital signature and private key is used to decrypt ciphertext or to create digital signature. Asymmetric key algorithms doesn't have the problem of key transport, but they are computationally costly compared with symmetric key algorithms.

Zenodys Platform uses both algorithms. Asymmetric key algorithm is used for asset encryption and symmetric key algorithm is used only for encrypting the symmetric key and signatures, which is computationally cost negligible.

### Zenodys Protocol steps



#### Sending request for digital asset

Customer signs digital asset's *licenceId* with his private key. Resulting signature is then encrypted with digital asset owner RSA public key. From signature, customer's Ethereum address is going to be extracted and verified by digital asset owner.

Parameters sent in digital asset request:

- *licenceId*: unique customer's digital asset purchase identification
- *publicKey*: customer's RSA public key used for encrypting AES key which is going to be used for encrypting digital asset
- *encryptedSignature*: signed licenceId used for verifying customer address by asset owner
- *callbackUrl*: url for delivering digital asset.

Functions for sending asset request and encrypting signature are [SendAssetRequest](#) and [SignAndEncryptSignature](#).

### Receiving request for digital asset

Digital Asset Owner receive request and decrypts the signature with his RSA private key. Verification part extracts Ethereum address based on *licenceId* and decrypted signature input. This is implemented in function [WaitRequest](#).

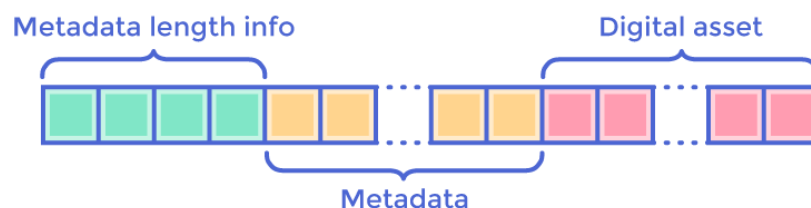
### Verifying customer address

Next step is to make blockchain verification. Once that asset owner has *licenceId* and corresponding customer's Ethereum address, smart contract's [checkLicence](#) function is called. This function verifies if current address belongs to the licence. Smart contract's check licence is called from [ZenLicenceChecker Element](#).

### Sending digital asset

Sending digital asset to customer is performed if verification from previous step succeeds. Here, a metadata stream that contains information about asset encryption and asset's standard values is created. The metadata stream is then combined with digital assets and sent to customer's *Computing Engine*.

Asset and metadata stream are combined into byte array in following order:



Metadata contains following fields:

*Cryptography related*

- Encrypted: True / False
- KeyEncryption: Algorithm used for encrypting AES key (eg "RSA2048")
- DataEncryption: Algorithm used for encrypting digital asset (eg "AES128")
- AESEncryptedKeyValue: Contains Key and IV encrypted values
- DataSignature: Signature that prevents digital asset to be tempered. It contains info about algorithm used for encrypting digital asset signature (eg "HMACSHA256"), value and encrypted key

*Digital asset related*

- AssetType: digital asset's type based on it's physical representation (data, file, image, algorithm, application, dApp...).
- AssetSource: Asset origin. This can be hardware protocol (ZWave, Modbus, RS232, I2C...), database (SqlServer, BigchainDB...) or some other source from where asset originates. If asset is algorithm, application or dApp, source is language used to create asset (R, C, C++, Rust, C#, Python....)
- ResultUnit: Asset unit (kg, km, W....)
- SystemType: Asset system type representation (int, double, bool, string, DataTable, json....)

*AssetSource*, *ResultUnit* and *SystemType* are not mandatory, but it's highly advisable for *Element* creators to provide them when applicable, for tighter integration with *Zenodys Protocol*.

In this step also licence terms are executed.



## Metadata structure example:

```
<AssetInfo>
  <Cryptography>
    <Encrypted>True</Encrypted>
    <KeyEncryption algorithm="RSA2048"></KeyEncryption>
    <DataEncryption algorithm="AES128">
      <AEEncryptedKeyValue>
        <Key>
          TIvULvz8WwX0j1TuUikwGYkD4jJHDR8SRQvQnN3jH/1pK2XVjkiJLCK/IqbCYL8kRUKA4KnhDhDBM3...
        </Key>
        <IV>
          cQor7SNS1VngTM76g4ZiGhiH4KuKBMAMzdicfjdCXsSrAjdU4nEmZvqTuVnN3U4hd3WoSI8xYoLw7wHy...
        </IV>
      </AEEncryptedKeyValue>
    </DataEncryption>
    <DataSignature algorithm="HMACSHA256">
      <Value>FAA3fUwsfh80QKyrnXqSYb/XyuhiFm82McGG0F/j6S4=</Value>
      <EncryptedKey>
        sgJ8F12Ra84e+g9o0GBZscBmpV4hHPu1Tg9U/HUiEjKU1VBgm/ytVPGJKKtp3K7igqyY0HgDIY7N45QR...
      </EncryptedKey>
    </DataSignature>
  </Cryptography>
  <AssetType>Data</AssetType>
  <AssetSource>ZWave</AssetSource>
  <ResultUnit>W</ResultUnit>
  <SystemType>Int32</SystemType>
</AssetInfo>
```

Asset transmission is implemented inside [ZenAssetTransmitter](#) Element.

### Receiving digital asset

This is final step where digital asset is received on customer's *Computing Engine*. When received, metadata is extracted from byte array and the digital asset is decrypted. The asset can then be processed in the customer's environment.

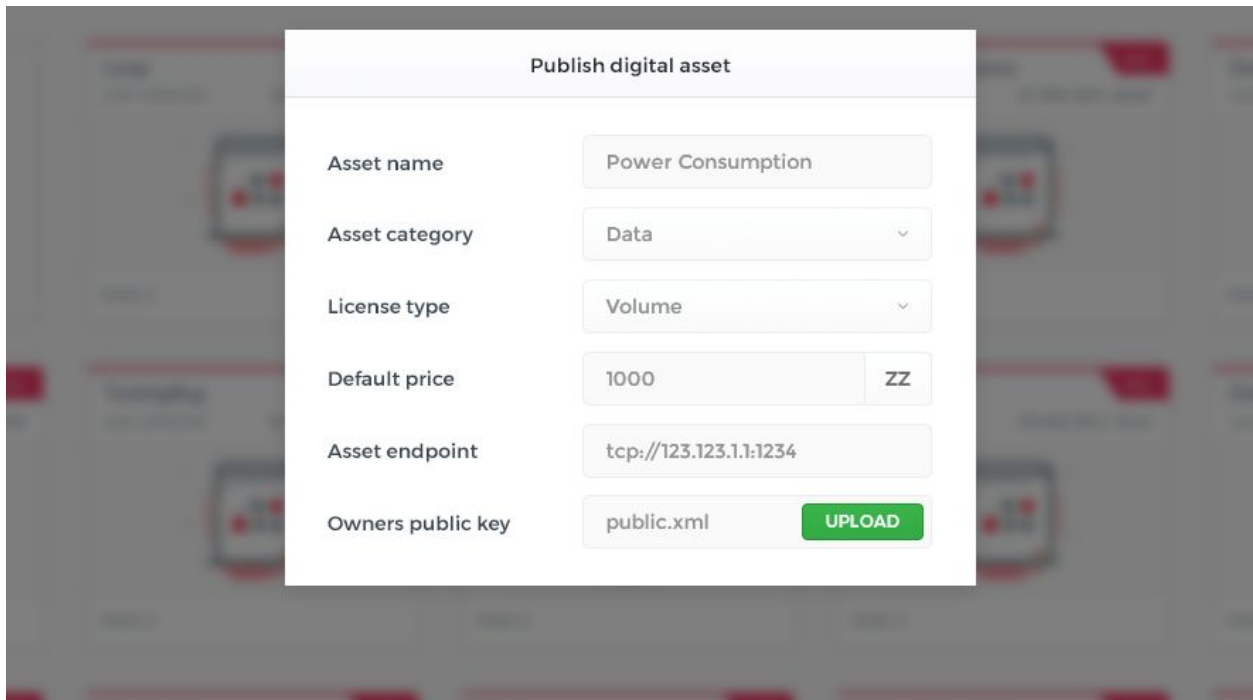
This is implemented inside [WaitAssetResponse](#) function.

## 4. Zenodys Protocol in action

This section demonstrates digital asset trading. It's energy consumption trading based on volume licence.

### Publishing digital asset on marketplace

Asset owner publishes terms on marketplace.



The screenshot shows a 'Publish digital asset' form with the following fields and values:

Field	Value
Asset name	Power Consumption
Asset category	Data
License type	Volume
Default price	1000 ZZ
Asset endpoint	tcp://123.123.1.1:1234
Owners public key	public.xml

An 'UPLOAD' button is visible next to the Owners public key field.

- **Asset name**  
Name of the asset
- **Asset category**  
Can be data, file, algorithm, application...
- **Licence type**  
Type of licence. Can be volume or subscription based
- **Default price**  
Price in tokens
- **Asset endpoint**  
Url on which asset can be reached

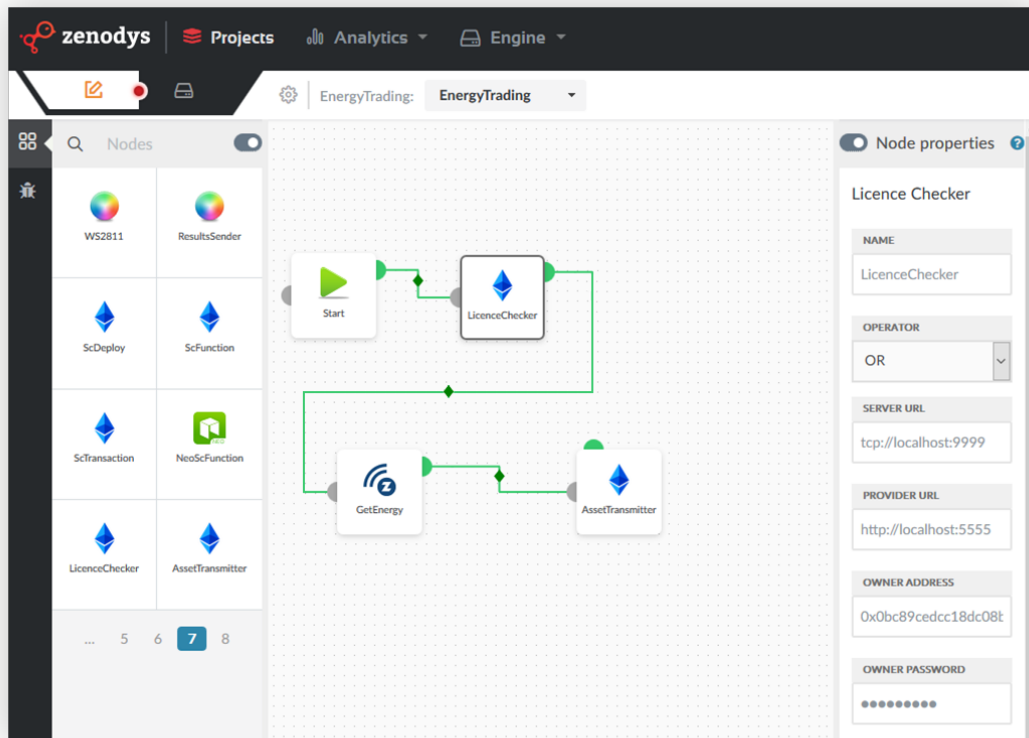
- **Owners public key**

RSA public key used for encrypting signatures and keys on digital access distribution. Keystore is generated automatically by *Computing Engine*.

Based on entered information, a smart contract is automatically generated and deployed to Ethereum blockchain. Example of generated smart contract is on [GitHub](#)

### Preparing Computing Engine to host asset

Asset owner deploys template on *Computing Engine*. Two standard Zenodys Protocol Elements are required on owner side - *ZenLicenceChecker* that waits for customers requests and *ZenAssetTransmitter* that sends asset to customer.



### Workflow description

Loop entry point is Start Element that immediately moves to LicenceChecker that stops workflow and waits for buyers digital asset request.

When request arrives it validates the licence. If licence validation succeeds it measures power consumption (eg Fibaro ZWave smart plug).

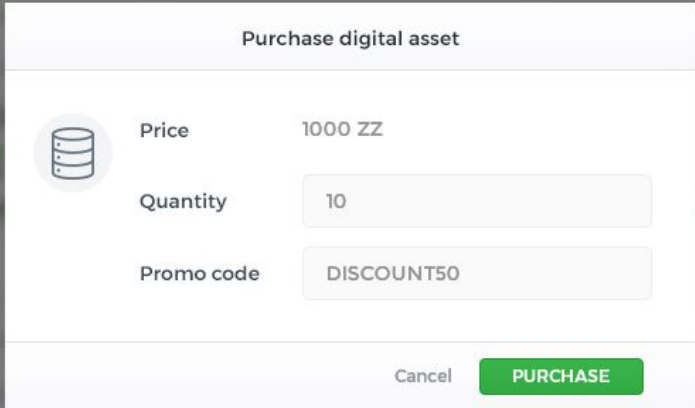
AssetTransmitter then encrypt power consumption, prepares it to standardized format and transfers it to customer.

LicenceChecker Element and proceeding workflow is executed on every customer's request.

An asset in this example is simple, but it could be additionally processed and merged with other digital assets.

### Purchasing digital asset

Customer purchase asset on marketplace.



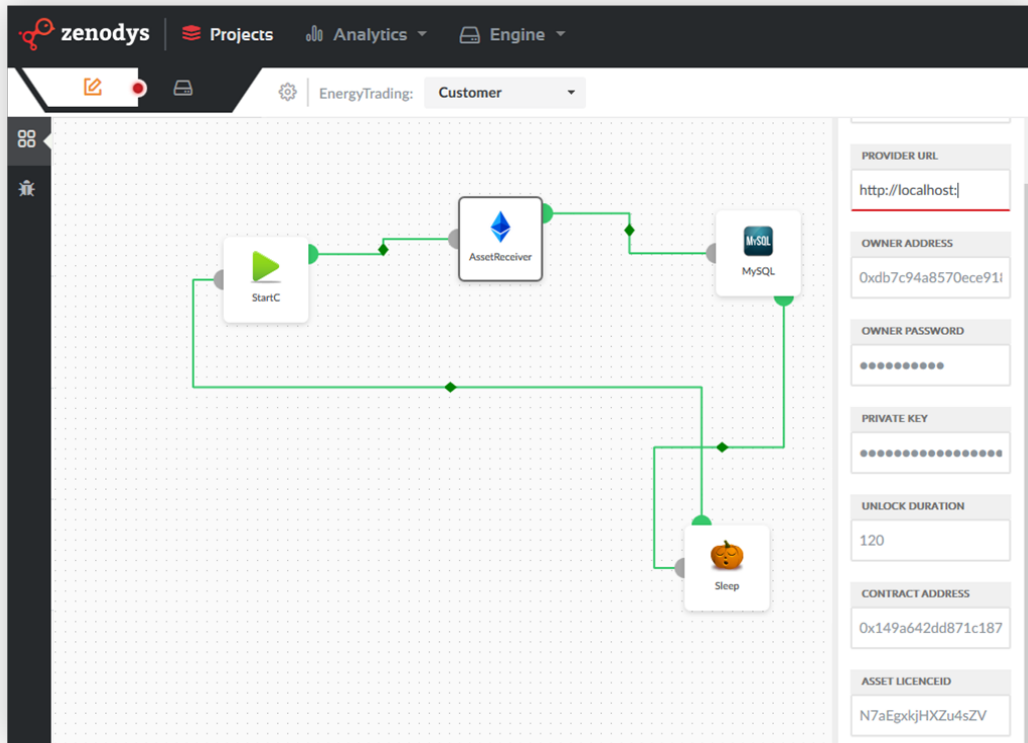
The screenshot shows a modal dialog titled "Purchase digital asset". On the left, there is a database icon. The dialog contains three input fields: "Price" with the value "1000 ZZ", "Quantity" with the value "10", and "Promo code" with the value "DISCOUNT50". At the bottom right, there are two buttons: "Cancel" and "PURCHASE".

- **Price**  
Asset price in tokens
- **Quantity**  
Customer enters quantity.
- **Promo code**  
Promo code can contain bonuses, discounts etc. Price is automatically calculated based on *Promo code* discounts.

Licence is automatically saved to [smart contract](#) by calling "addLicence" function

## Preparing Computing Engine to access asset

Customer deploys Template on Computing Engine. Standard protocol Element AssetReceiver is required that sends asset request and receive it.



### Workflow description

Loop entry point is Start Element. Then AssetReceiver sends request for asset and waits until it arrives. In next step, it's saved to database. Additionally, the asset could be processed before storing it.