

System Architecture

|  |  |
| --- | --- |
| Document ID | uid-sys-arch |
| Version | 1.0 |
| Status | Release |

# Table of contents

[Table of contents 1](#_Toc17824847)

[Introduction 4](#_Toc17824848)

[Glossary 4](#_Toc17824849)

[Table 1.  Glossary of UniquID-specific Terminology 4](#_Toc17824850)

[What is UniquID? 4](#_Toc17824851)

[When to use UniquID (not exhaustive example use cases) 5](#_Toc17824852)

[Vehicle Access Control System 5](#_Toc17824853)

[V2X 6](#_Toc17824854)

[Smart Meters 6](#_Toc17824855)

[Wifi access for guest users 6](#_Toc17824856)

[Zero-touch configuration appliance 6](#_Toc17824857)

[IoT device identification and Cloud access without Certification Authorities 7](#_Toc17824858)

[Secure Medical Document Workflow 7](#_Toc17824859)

[When not to use UniquID 7](#_Toc17824860)

[How UniquID Works 8](#_Toc17824861)

[Overview 8](#_Toc17824862)

[Major System Components 9](#_Toc17824863)

[UniquID Console 9](#_Toc17824864)

[UniquID Backend Services 10](#_Toc17824865)

[Blockchain 10](#_Toc17824866)

[UniquID Command Line Interface (CLI) Tool 10](#_Toc17824867)

[UniquID Agent 11](#_Toc17824868)

[Identity Management 12](#_Toc17824869)

[Permissions Management 13](#_Toc17824870)

[Access Contract 14](#_Toc17824871)

[Capability Contract 15](#_Toc17824872)

[Security 17](#_Toc17824873)

[Advanced Topics 17](#_Toc17824874)

[Backend Services 18](#_Toc17824875)

[Overview 18](#_Toc17824876)

[Identity Self-Provisioning Service (ISPS) 20](#_Toc17824877)

[Permission Management Service 20](#_Toc17824878)

[Asset Registry Service 21](#_Toc17824879)

[Communication Channel (MQTT) 21](#_Toc17824880)

[Contracts 21](#_Toc17824881)

[Contract Creation 22](#_Toc17824882)

[Contract Revocation 23](#_Toc17824883)

[Access Control List 24](#_Toc17824884)

[Message structure and RPC 25](#_Toc17824885)

[Message Signature and Authentication 27](#_Toc17824886)

[UniquID System Administration 28](#_Toc17824887)

[Console 28](#_Toc17824888)

[Command-Line Interface tool (CLI) 29](#_Toc17824889)

[CLI Real-Time Logging 30](#_Toc17824890)

[CLI Custom Scripts 31](#_Toc17824891)

[Appendix 33](#_Toc17824892)

[API Documentation 33](#_Toc17824893)

[Identity Self-Provisioning Service HTTP APIs 33](#_Toc17824894)

[API Overview 33](#_Toc17824895)

[Detailed APIs description 34](#_Toc17824896)

[Get managed devices 34](#_Toc17824897)

[Get managed devices filtered by status 35](#_Toc17824898)

[Get enrollment (orchestration) contracts 36](#_Toc17824899)

[Get enrollment (orchestration) contract for a specific provider 37](#_Toc17824900)

[Create an enrollment (orchestration) contract 38](#_Toc17824901)

[View orchestrators list 39](#_Toc17824902)

[View orchestrator information 39](#_Toc17824903)

[Create a new orchestrator 40](#_Toc17824904)

[Delete an orchestrator 41](#_Toc17824905)

[Get imprinter info 42](#_Toc17824906)

[View imprinter properties 44](#_Toc17824907)

[Upload properties file 44](#_Toc17824908)

[Permission Management Service HTTP APIs 45](#_Toc17824909)

[API Overview 45](#_Toc17824910)

[Detailed API description 46](#_Toc17824911)

[Get all managed devices 46](#_Toc17824912)

[Get info about specific device 48](#_Toc17824913)

[Get list of access contracts 49](#_Toc17824914)

[Create new Access Contract 51](#_Toc17824915)

[Delete an Access Contract 53](#_Toc17824916)

[Get Orchestrator info 54](#_Toc17824917)

[Get Orchestrator properties 56](#_Toc17824918)

[Update Orchestrator properties 57](#_Toc17824919)

[Init Orchestrator 59](#_Toc17824920)

[Create admin 60](#_Toc17824921)

[Create user 61](#_Toc17824922)

[Update user 63](#_Toc17824923)

[View users list 64](#_Toc17824924)

[Delete user 66](#_Toc17824925)

[Login 67](#_Toc17824926)

[Logout 68](#_Toc17824927)

[Create cloud connection 69](#_Toc17824928)

[Update cloud connection 71](#_Toc17824929)

[Delete cloud connector 73](#_Toc17824930)

[View connector 74](#_Toc17824931)

[View all connectors 76](#_Toc17824932)

[Connector status 77](#_Toc17824933)

[Software Components 79](#_Toc17824934)

# 

# Introduction

This document aims to provide you with an understanding of how the UniquID system is designed and the elements of the system interact with each other.  The first section begins with a high-level description of the major system components, and then the following sections describe the operation of these components  in more detail.

# Glossary

Table 1.  Glossary of UniquID-specific Terminology

|  |  |
| --- | --- |
| Term | Description |
| Agent | UniquID’s core software component hosting its own unique cryptographic identity. An Agent can either offer or consume services with other Agents, as defined by a Contract. |
| Access Contract | A class of contract between two UniquID Agents, to grant or revoke access permissions. The Agent providing this contract verifies its validity by using transactions on the underlying blockchain. |
| Access Control List | The field within a Contract defining which resources are accessible and which actions are allowed between the Agents identified in the Contract itself. |
| Capability Contract | A class of contract to temporarily grant access to the Agent’s resources or actions.  The Agent providing access through a Capability Contract doesn’t need connection to the blockchain to verify its validity. |
| Identity Self-Provisioning Service (ISPS) | A backend service to control how the unique identity of each Agent is provisioned in the system and it is distributed leveraging the underlying Blockchain. |
| Organization | A tool to identify  the domain within which a subset of Agents is managed, e.g., devices and applications pertaining to a single facility. |
| Permission Management Service (PMS) | The backend service to create and issue  access contracts between Agents. |

# What is UniquID?

UniquID is the Identity and Access Management system for Internet of Things and other decentralized applications:

1. It enables your enterprise to easily provision IoT devices and applications, providing an automated system to create unique cryptographic identities that work across your enterprise infrastructure and applications, without the need of a central authentication service.
2. System architects, application designers, and administrators can then leverage these unique IoT devices identities to grant and revoke the permission to use specific Resources or perform certain Actions, by issuing UniquID Contracts. These Contracts control how IoT devices are offering and consuming resources and actions by leveraging decentralized blockchain transactions, instead of central authorization servers .

For example,  UniquID can manage the provisioning and control of your enterprise's IoT assets located at the edge of the network, such as industrial  gateways, sensors, actuators, appliances and software applications, leveraging its decentralized and interoperable blockchain to reduce the need for constant connectivity to the cloud, and complex Identity and Access Management system integrations.

UniquID is provided as a Software as a Service offer by default. Self hosting, dedicated instances and other options are available.

Software and system developers simply need to integrate UniquID’s Agent SDK within your devices and applications, and connect them to UniquID’s ISPS and PMS services.  Once connected, these services automatically:

1. Provision new devices and applications on UniquID’s blockchain
2. Enroll their digital identities into your unique Organization
3. Control them by issuing and revoking Contracts.

# When to use UniquID (not exhaustive example use cases)

Developers can use UniquID to mitigate the multiple issues related to device Identity and Access Management (IAM) inside distributed IoT networks. UniquID is designed to support the availability, integrity and confidentiality of IoT devices whenever the traditional, cloud-centric systems are not suitable in terms of maintenance overhead, identity services availability, and computational trust dependencies.

Some examples use cases implementing of UniquID are: .

## **Vehicle Access Control System**

This use case enables the owner of a connected vehicle to use their smartphone/PC to issue and revoke access Contracts to drivers, mechanics, insurance platforms, valet services, parking and any other type of access to the vehicle or the data generated from it, controlling  when and for how long this access will be provided.

The owner can be also a  fleet manager,who can control a large number of vehicles without any change to the system. The Access Control rules are automatically embedded in the Contracts and distributed via the underlying Blockchain, so they are enforced locally (no persistent connection to remote authentication and authorization server are needed).

## **V2X**

In this scenario, a connected vehicle has to communicate in a secure way with other vehicles and the roadside connected equipment, overcoming the limits imposed by different jurisdictions, digital standards and manufacturers. UniquID’s solution is designed to enforce the security of the communication protocols with a distributed solution that is simpler and less expensive to deploy, maintain and scale up than a global centralized PKI of every entity interconnected by the system.

## **Smart Meters**

In this scenario, energy utilities have to bill the supplied electricity to consumers. To measure electricity consumption, the energy company installs smart meters in consumers’ buildings. These meters send energy consumption information through a central authentication and authorization hub at the energy company.

In order to ensure the communication security between the smart meters and the company’s authentication and authorization hub, UniquID’s solution grants a decentralized method to verify that  data is correct before it’s dispatched in the cloud. Also, this solution allows you to encrypt the data in motion without using a remote authentication service, mitigating the risk of MITM-attacks. The distributed nature of UniquID’s underlying blockchain  provides the ability to control a large amount of sensors (smart meters) by simply broadcasting batches of signed transactions.

## **Wifi access for guest users**

Within this scenarion, a wifi provider (e.g. a Telco operator or a Company) can use UniquID’s solution to grant anonymous/guest access to an existing WiFi network, without transmitting any credentials (such as passwords or certificates), nor authenticating the user through the usual *Captive Portal*. With UniquID, the access is tracked and logged by a traditional access control system (e.g. a Radius server), integrated with UniquID Contracts in the back-end. This solution collects the public key of a device from the underlying blockchain, therefore removing the provisioning of the guest in every centralized Radius server that provides WiFi access, reducing guests’ credentials management and maintenance overhead.

## **Zero-touch configuration appliance**

An IoT-enabled appliances manufacturer uses UniquID’s Identity Self-Provisioning Service (ISPS) to identify and pre-provision its products  before they leave the manufacturing facility, loading its configuration only when the appliance is sold to the final client. Once the appliance is paid, the configuration is embedded within a Contract and transmitted via the underlying Blockchain. This operation drastically reduces the installation costs and overhead, while enabling third-party service providers to securely install these appliances without the need for any “administrator” privilege. With the current release, millions of appliances can be remotely managed in this way.

## **IoT device identification and Cloud access without Certification Authorities**

Within this scenario, an IoT devices manufacturer can leverage UniquID’s solution to grant an access to its private/hybrid/public cloud services without managing signed certificates, or hard-coded API passwords.  Above a critical amount of endpoints (~100,000) UniquID’s solution becomes more efficient that traditional CA solutions, by enabling the provisioning and configuration of thousands of devices in a few minutes, just by leveraging a single script. Moreover, the IoT devices manufacturer doesn’t have to deal with wild-card certificates, Certification Authorities  and to set-up/run an internal Public Key Infrastructure (PKI). The blockchain-based identity provided by UniquID is compatible with secure-element based solutions, and is less expensive on large volumes.

## **Secure Medical Document Workflow**

Within  this scenario, a patient can automatically share their medical files with  doctors across hospitals. The medical files are encrypted using keys owned only to the patient, and then stored in the cloud. The decryption keys are exchanged usingUniquID-enabled applications, which leverage blockchain transactions to enforce the authorizations without relying on trusted third parties. This architecture ensures that medical files can be distributed openly, but only the patient's device is capable of enabling access to them.

An example of the process is:

* The patient's medical data is stored in the cloud, in an encrypted form.
* The patient enters into an agreement for the provision of medical services at a clinic or hospital. A key point of the agreement is to permit the binding of the patient's device (a tablet, personal computer or smartphone) to the backend services of the clinic. The doctor’s devices are already bound to the backend services of the clinic.
* The patient visits a new doctor, bringing with them the personal smartphone that holds the Contract to authorize new devices.
* The doctor requests to see the patient's medical data on their work computer.
* The patient scans a QR code from the doctor’s work computer and permits the sharing of their medical data with this doctor’s computer.
* The patient's smartphone provides a one-time key to decrypt the medical data.
* The doctor's computer downloads the patient's encrypted medical data from the cloud and then decrypt it. This allows the doctor to examine the data they require.
* The doctor prescribes a treatment for the patient.
* The patient leaves the clinic and revokes the authorization to their medical data. From this point, the doctor cannot access the patient’s private medical data.

# When not to use UniquID

Although UniquID is an ideal fit for many use cases, there exist a few use cases which is not suitable for.

UniquID, as a stand-alone Identity and Access Management platform, is not efficient to identify and authenticate people: these solutions must provide a trusted process to capture and verify the personal data of the user. This process must follow  controlled procedures, where the binding process between a person and their digital identity should be certified as valid and broadly accepted. As an example, the provisioning of a new user on the blockchain, since irreversible, should be performed within an audited process, which is responsible for the verification of a government-issued document before enabling the access to a banking account. This process can be simplified by adopting a smartphone to issue and revoke credentials: in this case, the user digital identity can be stored inside the device itself, and controlled via biometry (e.g., fingerprint or face recognition). UniquID’s Agent software might run on the smartphone, thus enabling machine-to-machine secure authentication and authorization with the application server on the other end of the connection, but the solution is evidently more complicated if compared to more traditional approaches.

 UniquID suffers the same type of complication within *Self-Sovereign Identity* platforms, which leverage a self-defined personal digital identity maintained by a decentralized system, with no central authorities such as governments or credit institutions. UniquID is not a Self-Sovereign Identity system for people, however it includes by design the decentralized sub-system needed to plug Self-Sovereign identities  with IoT devices.

This scenario, once broadly adopted, might unlock the phase where UniquID-enabled devices become efficient also to authenticate human users which are registered under Self-Sovereign identity conditions. The most common device is the smartphone, therefore a UniquID-equipped app, running the Agent, might carry its own unique identity be easily linked to the owner identity, leveraging the same principles that today allow mobile users to operate with their bank accounts. So, as long as a platform can trust a smartphone app running UniquID Agent, and as long as the provisioning of the owner within the mobile app can be trusted, your solution can trust the user indirectly through the UniquID IAM principles (furthermore if the smartphone is protected by the user’s biometry).

Since we can predict a large adoption of biometric authentication by smartphone manufacturers, and we can predict a very large diffusion of personal and wearable IoT devices, UniquID can support the futuristic vision where every time we use a connected device we strengthen our digital authentication by providing a physical context. For example, a person sitting inside a self-driving car cannot be also sitting in front of the enterprise’s computer, or trying to open the office door by using the company badge.

More in general, UniquID solution is designed to simplify the Identity and Access Management of large sets of devices, while keeping the overhead as small as possible. Therefore, even if the system efficiently works with very minimal configurations (2 connected devices or 2 connected applications), you can get all the benefits of the solution when managing hundreds of thousands of IoT devices and interconnected cloud services.

# How UniquID Works

## **Overview**

UniquID enables enterprises to scale the deployment of large numbers of IoT devices, by providing tools that securely automate:

* The provisioning of new devices (by enrolling their digital identity)
* The control of the permissions that grant access to the data generated by these devices.

By leveraging these two principles, UniquID allows you also to manage access permissions between  IoT devices and applications running inside your IT infrastructure, since UniquID Agent is designed to provision and control also standard applications running on cloud instances and server on the premises.

Devices and applications which incorporate the UniquID functionalities are referred to as UniquID *Agents*. *Contracts* between *Agents,* stored and trusted by the blockchain*,* are used to grant and revoke permissions (see the dotted lines).

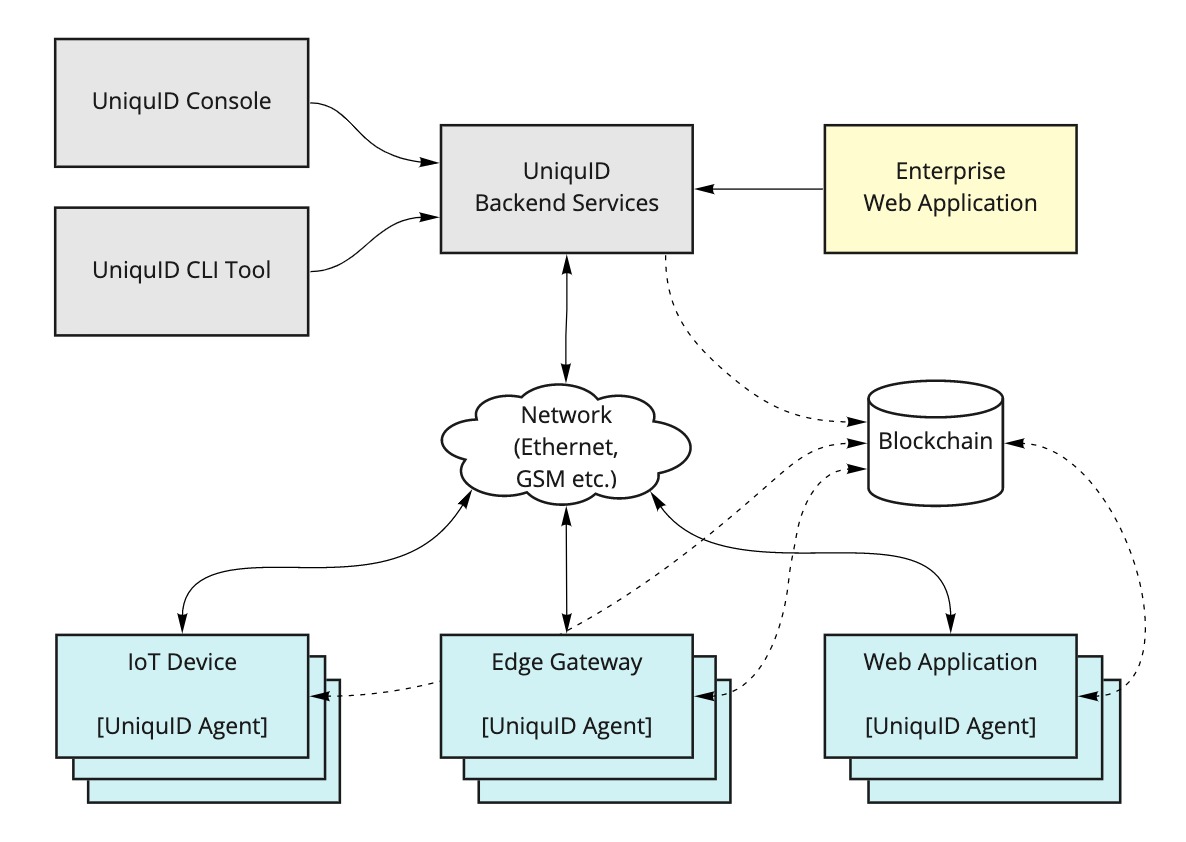


Fig. 1.  UniquID System Components

## **Major System Components**

### **UniquID Console**

The Console is a web application to administer your Organization account and the Agents provisioned into it. t UniquID Console provides the following functionalities:

* Create an account for your Organization.
* Create an administrator user for your Organization.
* Download the credentials file storing the access authorization for the administrator.
* Delete the account for your Organization.

You can access the Console at the address <https://uniquid.com/console> .

### **UniquID Backend Services**

UniquID provides a managed infrastructure which provides the access to the backbone of the system. This system is provided via a Software as a Service (SaaS) solution that removes the need to invest time and effort in the design, development and maintenance of the backend infrastructure. Upon request and for large installations, backend services can run inside clients’ private or hybrid infrastructure.

UniquID’s backend services run  behind HTTPs RESTful APIs which can be integrated within your applications. You can use these APIs to:

* Generate a list of all provisioned Agents\* inside your Organization.
* Generate a list of all current and revoked contracts between Agents.
* Issue a contract to grant permissions between two Agents.
* Delete a contract to revoke permissions between Agents.

\*As mentioned, an Agent can be an IoT device or an application.

### **Blockchain**

UniquID’s system currently leverages the Litecoin public blockchain to store the contracts that grant and revoke permissions between UniquID Agents. The use of this distributed blockchain ledger means that UniquID agents can verify the validity of its entries (which contain the Contracts), thus validating the permissions granted to Agents, without connecting to UniquiD’s Backend Services.  This is the major benefit of using a distributed blockchain technology: since individual Agents do not need to send queries to centralised backend services, they are capable to validate an access request from another Agent by trusting the ledger entries and their integrity (enforced by the underlying Proof of Work algorithm). All the information required by the Agents to authorize the access is available on the public blockchain.

UniquID’s blockchain protocol also provides an immutable record of the contracts created to grant and revoke permissions. This feature provides a permanent record of how and when permissions were changed, which is designed to support audit and traceability purposes.

### **UniquID Command Line Interface (CLI) Tool**

To simplify the usability and the flexibility of the platform, UniquID provides a command-line tool which allows you to easily access all of the methods of UniquID RESTful APIs.  This tool enables the system administrator to control the status of all entities inside the Organization, and issue or revoke access permissions if required.

The tool provides the following functionalities:

* View a list of all provisioned Agents.
* View a list of all contracts between Agents.
* Create a contract to grant permissions between Agents.
* Delete a contract to revoke permissions between Agents.
* Deploy the Agent into an AWS EC2 instance, which automatically integrates AWS IoT Core with your UniquID Organization.
* Delete the EC2 resources which were previously deployed by an AWS IoT Core integration command and restore AWS to its previous state.

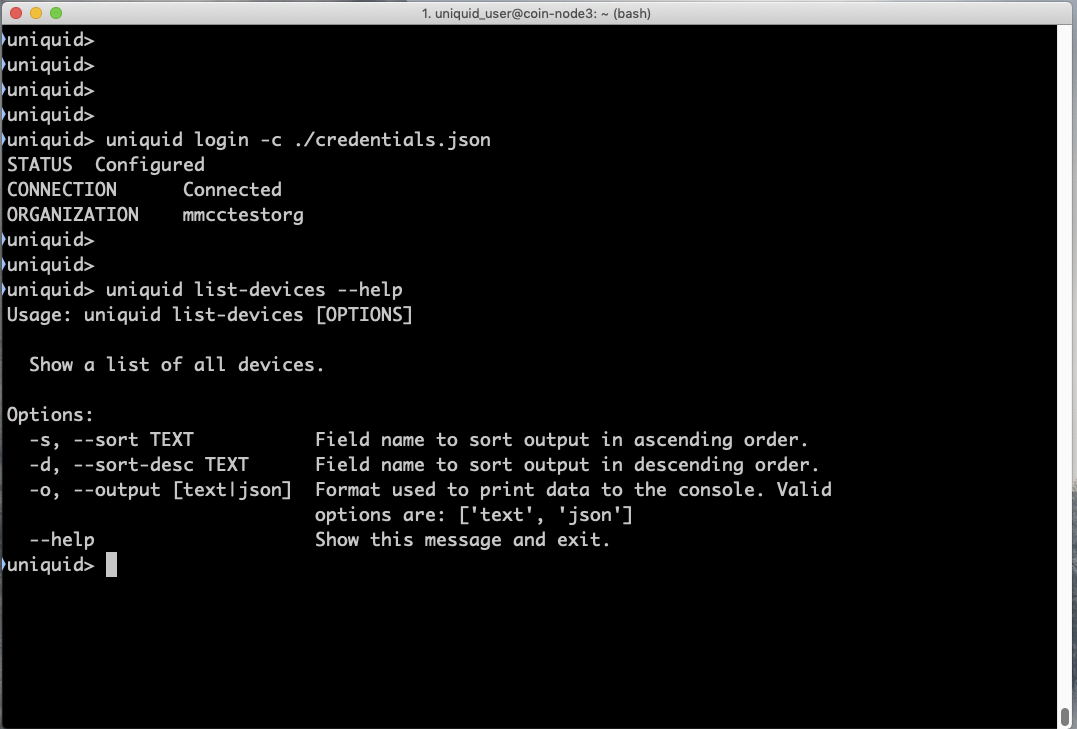


Fig. 2  UniquID Command-line Tool

UniquID CLI tool can be automatically installed using the Python package management tool 'pip' and the command ‘pip install uniquid’.  The CLI provides a conventional shell interface, therefore it is ready to be integrated into automation scripts and your existing deployment processes.

### **UniquID Agent**

The Agent is the core element of the platform, and is responsible to pair any IoT device or application with its own unique identity inside UniquID distributed platform.  
In order to provision an IoT device inside your UniquID Organization, you have to run a valid UniquID *Agent* on it. To leverage UniquID IAM platform capabilities, IoT devices and applications have to leverage their Agent.

Agents can be integrated with the software or firmware of the IoT device or the application, by using UniquID’s Software Development Kit (SDK).  Currently, UniquID SDK supports the following programming languages:

* Java
* Javascript, Typescript and Node.js
* C, C++

There are cases where the Agent cannot be installed directly on IoT devices: the approach remains the same, however UniquID is run at the gateway level, providing a proxy functionality.

## **Identity Management**

UniquID provides a service to create a unique cryptographic identity for each Agent enrolled in the system. The service runs on top of a public blockchain, therefore an Asset’s digital identity is unique among the whole set of customers, organizations and applications/devices (as of today, a total space of 2^256 unique private keys).

UniquID identities can be generated and deployed during one of these exemplification moments:

* After an IoT device is assembled on the production line (e.g. burn-in test)
* After an IoT device is delivered to a distributor and is ready to be provisioned
* After an IoT device is installed at the customer’s premises and is tested
* As soon as an Application instance is run the first time
* Every time a stateless Application is executed (e.g., a Docker container)

Newly instantiated UniquID Agents are designed to randomly generate their own unique identity (by leveraging any locally available entropy generators) and then propagate the resulting public key to the Agents in the system that finalize the provisioning by writing it on the blockchain (the ISPS service). This process of identity self-generated is close to *Self-Sovereign Identity* principles.

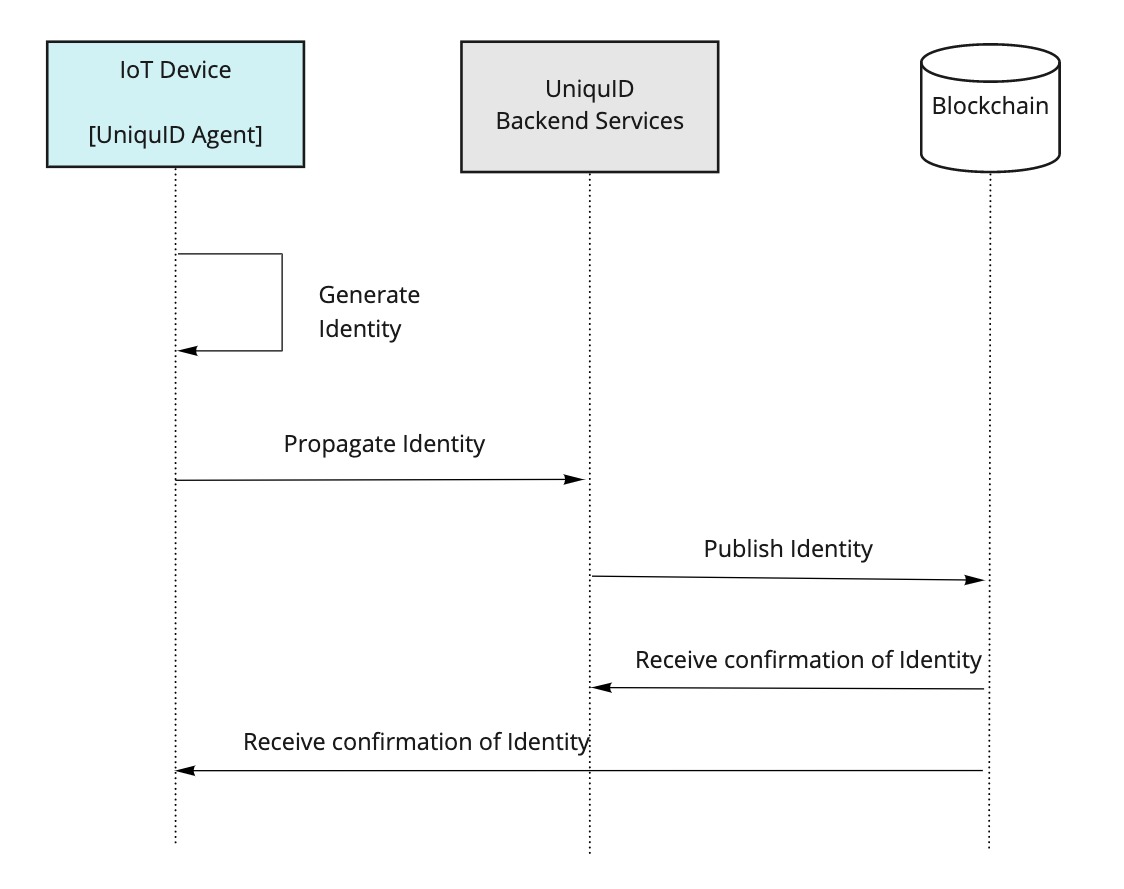


Fig. 3.  Identity Provisioning Sequence

In practical scenarios, when an IoT device is powered for the first time it generates its own UniquID Agent private key and then it broadcasts the resulting public key across a local, trusted channel to the UniquiD backend services.

Once the backend services receive a new Agent identity, an administrator Contract is created. This contract binds the Agent embedded inside the new IoT device to your Organization, granting administration rights.  As soon as the new Contract is written on a new block inside the blockchain, the Organization can grant or revoke permissions between the other existing Agents and the new provisioned IoT device.

## **Permissions Management**

UniquID Permissions Management is designed to grant or revoke permissions between Agents, which can be IoT devices or Applications.

Every Agent can be granted a specific set of permissions toward another Agent by its administrator: as an example, it may provide the rights to execute certain actions (open/close, raise/lower, push/pull) or the rights to read or write data.

Equally, revoking these permissions will take away an Agent’s ability to perform such activities.

Each UniquID Agent evaluates any incoming request as soon as it is received by another Agent, to determine if the permissions exist.

### **Access Contract**

UniquID governs these permissions using Contracts written on the public blockchain. A standard type of UniquID Contract is the *Access Contract*, which contains an array of unsigned integers that maps each of them to the actions that a given IoT device or application can do, such as sending the requested data, or performing the requested action.  If no valid Access Contract is present on the blockchain, the UniquID Agent will deny the request, blocking its execution on the underlying application.

UniquID Agents must have periodic access to the blockchain to download any new Access Contracts. Once retrieved, new Access Contracts are cached locally inside the Agent, so it may not need to query the blockchain every time a new access request is performed. Therefore, an UniquID Agent can validate the request with a lower latency, or without using a WAN connection, improving the system performance.

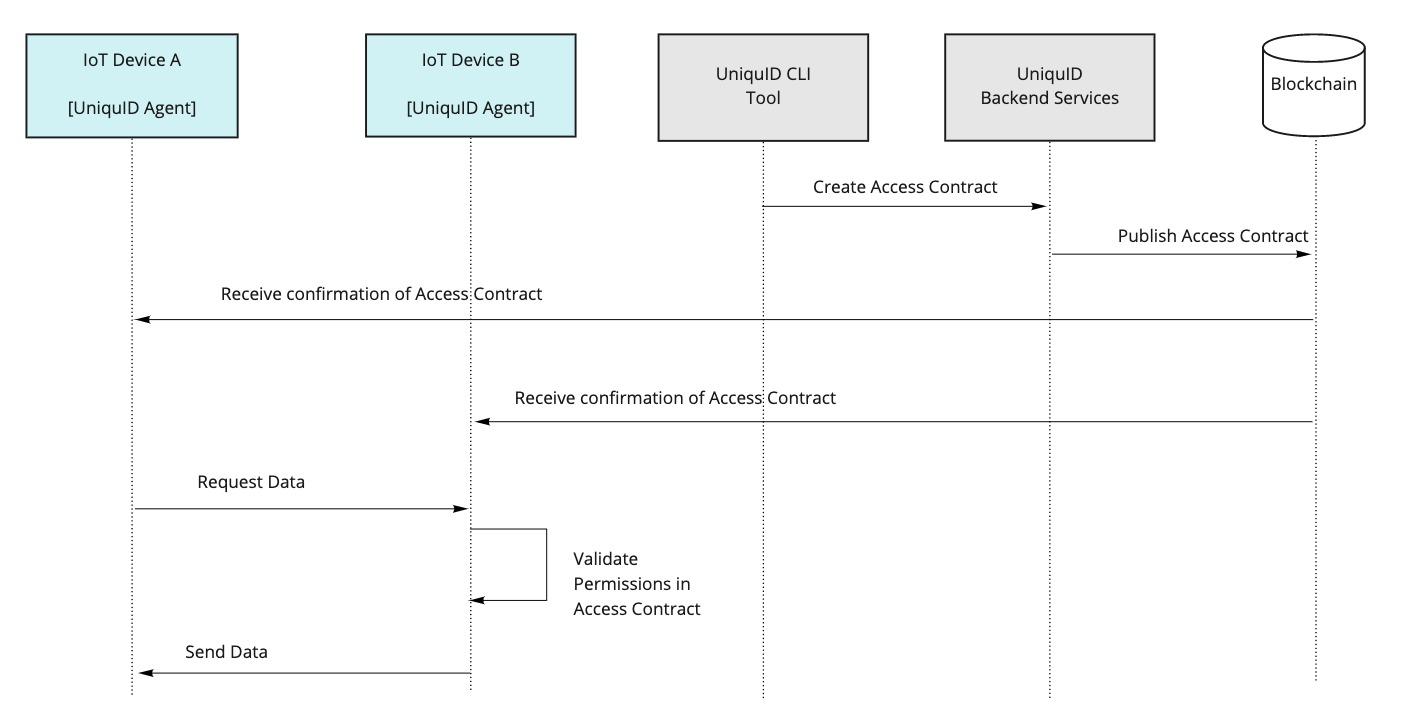


Fig. 4.  Granting Permissions Sequence

Fig. 4 show an example of Access Contract being used between two IoT devices “IoT Device A” requests data from “IoT Device B”.  
“IoT Device B” will always reject these requests unless an Access Contract with “IoT Device A” is in place.

In this example, a valid Access Contract between the two devices has been previously created using the CLI tool and the Backend Services.  As soon as “IoT Device B” downloads the Access Contract from the blockchain, it verifies its validity, stores it in the local cache, and enables “IoT Device A” to request the data.

Please note that Device A, Device B and the Backend Services don’t share a trusted, direct connection to exchange Access Contracts, but leverage instead the underlying public blockchain.

Access Contracts can be issued on the blockchain using the UniquID CLI tool, or by directly sending queries to the UniquID Backend Services RESTful APIs.

The Access Contract is defined inside JSON formatted inputs.

This JSON file must contain three fields:

* The public key of the Provider Agent (defined by the xPub format). The Provider Agent hosts the service or the data we want to authorize toward the User Agent.
* The public key of the User Agent (defined by the same xPub format used above). The User Agent  is the client that needs access to the service or the data hosted by the Provider Agent. As mentioned, the Provider Agent needs the Access Contract written on the blockchain before it can authorize the User Agent to access any available resources.
* The Access Control List (ACL). This field specifies which services or resources are affected by the contract. In the standard Access Contract, they are identified by an array of unsigned integers, which may be used to map a set of instructions in the underlying application.

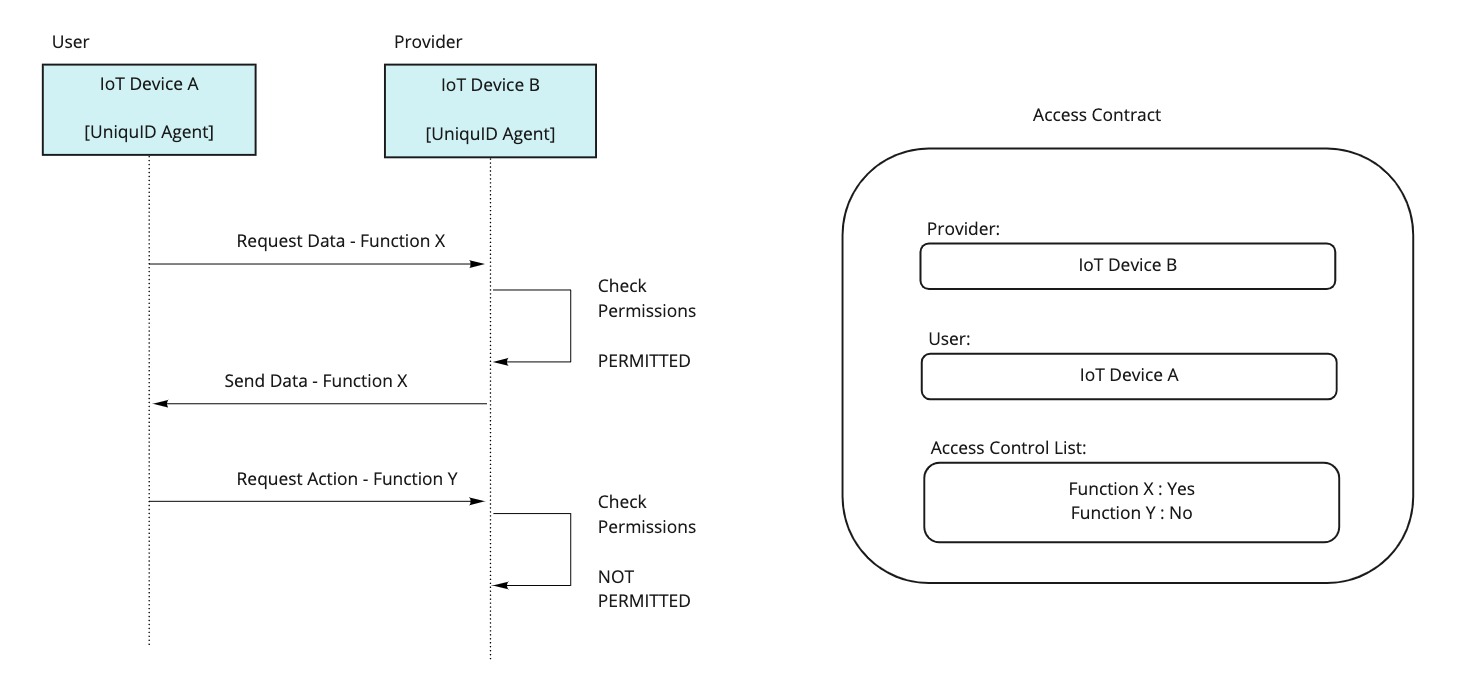


Fig. 5.  Access Contract Sequence

### **Capability Contract**

For large-scale, high-frequency and offline Contract needs, UniquID provides the *Capability Contract* class. This type of contract is designed to grant permissions among a pool of Agents, only for a limited duration of time, without issuing Access Contracts between between each and all of them. The Capability Contract works similarly to a traditional authentication token: once the allotted time expires, the permissions are no longer valid and a new Capability Contract must be issued.

Capability Contracts are designed to be validated by the Provider Agent that has no Internet connectivity, thus has no ways to fetch UniquID’s underlying blockchain. Therefore, a Provider Agent using a Capability Contract is able to authorize a User Agent  in situations where they have limited network and blockchain connectivity.

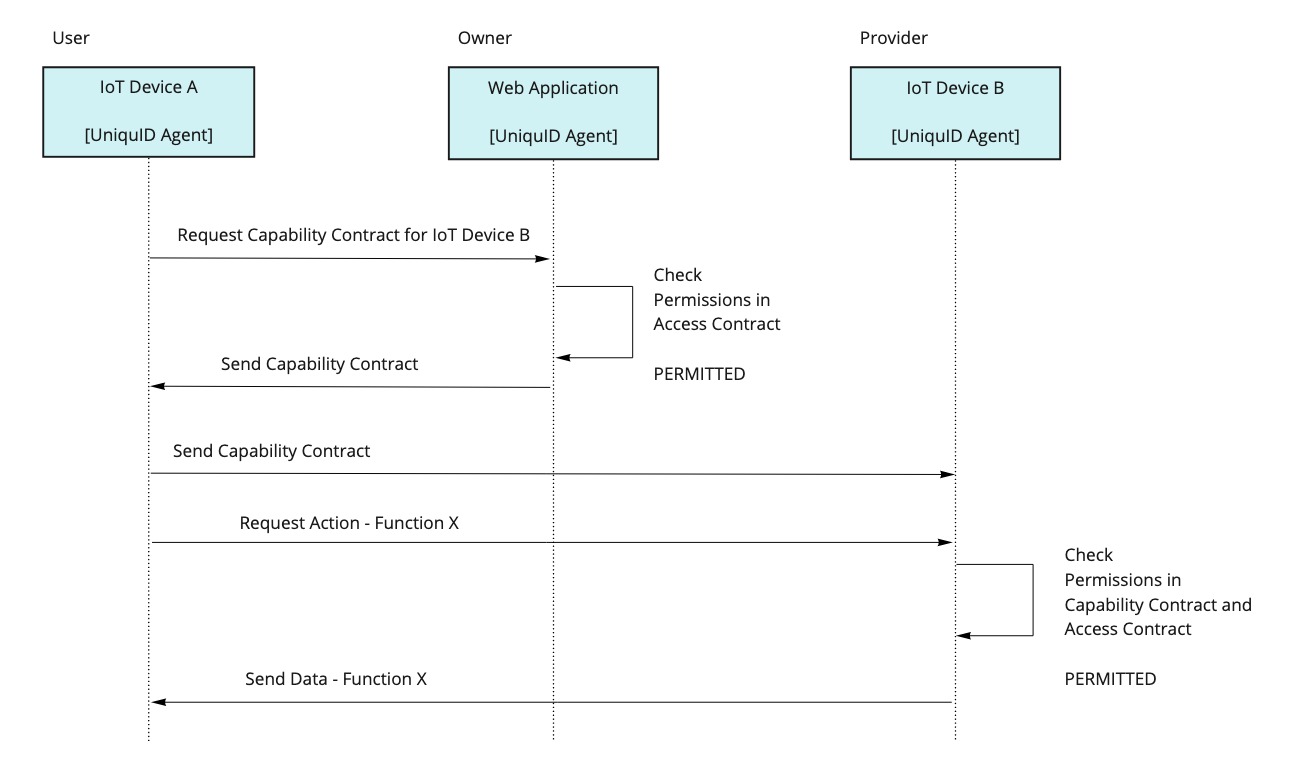


Fig. 6.  Capability Contract Sequence

Fig.6 shows the sequence diagram of a Capability Contract used to grant access between UniquID Agents. In this scenario, there is also the Owner Agent together with the User and Provider Agents there is also the “Owner” Agent.

 The Owner Agent is trusted by the Provider Agent by leveraging a previously issued Access Contract.To create a new Capability Contract, UniquID requires two Access Contracts in place:

* one between the User Agent and the Owner Agent
* one between the Owner Agent and the Provider Agent.

By issuing these two Access Contracts, the administrator puts in place the permissions to leverage, later on, a Capability Contract. Once these two contracts are in place, the User Agent can request the Owner Agent to issue a Capability Contract, such that it gives permission to access the resources provided by the Provider Agent. As usual, upon receiving the Capability Contract, the User Agent can cache it for later usage, until it expires.

During the handshake, the User Agent sends a request to the Provider Agent, providing two key data:

* The Access Contract they have with the Owner Agent
* The Capability Contract, which embeds the Access Control List (ACL).

If both contracts are valid, and the requested resources is allowed by the Capability Contract signed by the Owner Agent, the Provider Agent accepts the request, applying the same logic of the standard Access Contract. By design, theCapability Contract is not stored on the blockchain.

## **Security**

When a new Agent is started, it generates a new Identity and then sends its Name and its public cryptographic information to the Identity Self-Provisioning Service. The ISPS reacts sending a transaction on the Block-Chain giving coins to the Agent. This first transaction, that contains a public key of the ISPS and the hash of a public key of the Agent (Litecoin address), is considered the first contract that establishes the root of trust between the UniquID Permission Management Service and the Agent.

All UniquID contracts are written on the Block-Chain and are then retrieved from each Agent directly from the Block-Chain itself using the standard Block-Chain protocols and thus trusted by that.

Each communication between two UniquID Components (ISPS, PMS and Agents) are secured by UniquID contracts.

It is important to note that the communication doesn't go through the Block-Chain itself but on a different communication channel (today MQTT with signed messages) secured by the information of the UniquID contract.

An UniquID contract basically stores this information:

* a public key of the “provider” Agent (the agent that accept requests)
* The hash of a public key of the “user” Agent (the agent that sends requests)
* The access permissions (the Bitmask)

All the Block-Chain cryptography, and thus the UniquID one, is based on Elliptic Curves.

The public key information stored on the contract, along with the fact that only the Agent knows its private key, permit to securely identify and trust the counterpart and link the interaction to an Access Permission.

Today the identification is enforced using message signature. Each message exchanged between the components are signed using the ECDSA algorithm using the private key related to the public one of the contract. In this way who receives a messages is always able to verify who sent the message, link it to a contract and act accordingly to the permission defined on it. For a detailed description of the message structure and the signature please refer to the section [Message structure and RPC](https://docs.google.com/document/d/1StIkdiUzhJYLTSbyrp6LhhVvU4Tgg-oaZaRmne6cy9Y/edit#heading=h.hgtb09awjqi1).

Other ways to identify and trust the counterpart can be built using the ECDH algorithm. The ECDH algorithm permit to two parties to agree on a common shared secret starting from the knowledge of their own private key and the public key of the counterpart.

# Advanced Topics

This section explains with technical detail how the components of the UniquID system  interact, and the low-level protocol works. This section is not necessary to understand how to use UniquID, but it could be useful to better understand the implications of using an underlying blockchain instead of traditional, centralized services.

UniquID solution is a *layer two application*, built on top of the Litecoin blockchain. UniquID uses the transactions stored inside the blocks  as the infrastructure to implement its IAM system.

## **Backend Services**

### **Overview**

Although UniquID is based on a public blockchain, the system leverages a set of centralized services to configure the Identity and Access Management of the Agents, and correctly issue the transactions on the ledger. These services are:

* Identity Self-Provisioning Service (ISPS, aka “The Imprinter”)
* Permission Management Service (PMS, aka “The Orchestrator”)
* Asset Registry (aka “the Registry”)

These three services provide the management tools to the administrator  of given UniquID Organization. These tools manage all of the blockchain implementation so developers and system designers don’t have to learn it or deal with it.

Each of these services have to interact with the other management services, while also connecting with the devices and the applications running the UniquID Agent .

As of today, UniquID leverages two classes of communication channels:

* Any Identity or Access Contract transaction leverage the peer-to-peer network of of Litecoin nodes, by using standard transactions broadcast
* RPC calls and data exchange between Agents leverages locally available resources (e.g., MQTT / TCP / Bluetooth / WiFi...).

Fig 7 shows UniquID services and how they interact with IoT devices:

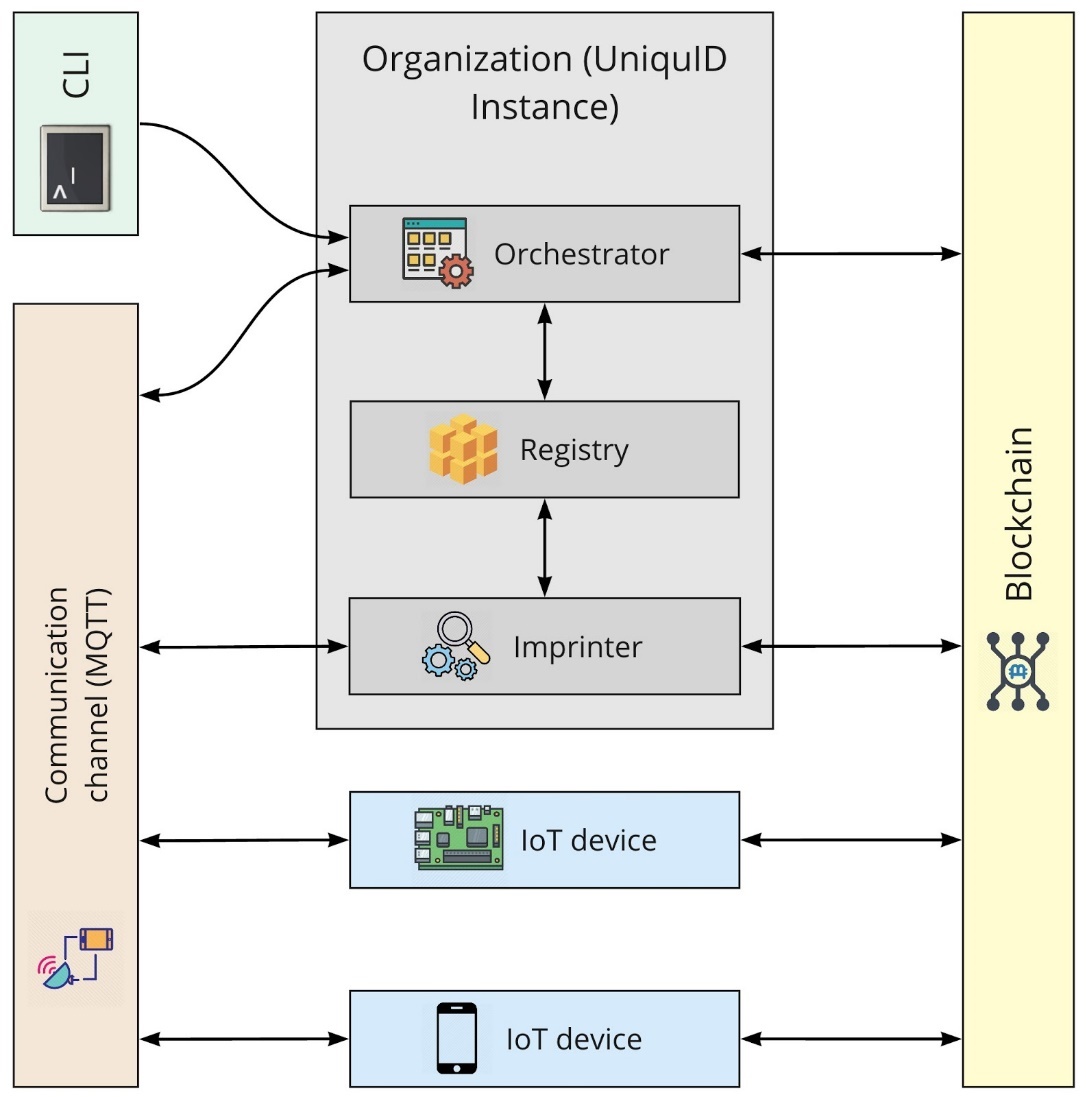


Fig 7. UniquID Backend Services

Within an ideal scenario, UniquID Backend Services are used for the initial provisioning of new Agents and the creation and revocation of Access Contracts between them (e.g., Agents running inside IoT devices and services). In general UniquID Backend Services are used for creating and modifying the configuration settings of the permissions for the whole set of the  Organization’s Agents.

UniquID services can be deployed in two ways:

* Cloud infrastructure deployment - each Organization is located within isolated instances, behind UniquID’s SaaS running on AWS
* On-premises infrastructure deployment - his option supports companies with specific security and compliance needs, for example to run the *Imprinter* and the *Orchestrator* services inside isolated local networks, for air-gap IoT devices provisioning.

### **Identity Self-Provisioning Service (ISPS)**

The ISPS, called Imprinter Service, is a critical component  that manages the provisioning of new Agents (as mentioned, IoT devices or applications) inside the Organization.

The Imprinter works in automatic mode: once a new IoT device connects to the communication channel (e.g., MQTT) the UniquID Agent running on it announces itself. The Imprinter automatically captures this message, and begins the provisioning process by writing a unique record to the blockchain.

Ensuring security of this process  is a key aspect of a correct and secure use of UniquID’s service. Correclty using the *Identity Self-Provisioning* allows the Organization to block attackers from  enrolling unauthorized Agents to the Organization's network, therefore excluding it from the Orchestrator Asset Registry of devices.  and If an Agent is not inside an Orchestrator’s Asset Registry, no Contracts can be issued to it.

On the practical side, UniquID Imprinter Service is necessary to provision new devices or services to the Organization, but it can be performed by third parties, much before the IoT device is deployed. We should assume theImprinter service will be used only once in the lifecycle of an IoT  device, for example, at the manufacturing implant, when the device is built, assembled or tested.-

In the future, a factory might provide IoT devices already registered with UniquID Imprinter service, putting them in a pre-provisioned state,  for future use inside a specific Organization. UniquID designed the Imprinter service to apply a unique digital serial number to a device during the manufacturing process, identifying it by a tamper-resistant transaction on the Litecoin blockchain.

After the new device is  visible on the blockchain, it can be safely dispatched to the store for selling, or supplied to the customer. In this scenario, the customer gets the device ready  to launch the Orchestration process and get provisioned into its own specific Organization.

The Imprinter doesn’t need any manual command to operate, and runs in automatic mode listening for any new Agent. The service implements a set of RESTful HTTPs APIs functions to control the service status, and the Agents that are being registered in the organization (as mentioned, IoT devices or applications).

You can find links to Imprinter HTTP API specification in Appendix section.

### **Permission Management Service**

The Orchestrator service commands and controls the Contracts that manage the interaction between the IoT devices and the applications inside its Organization. The Orchestrator is designed to create, issue and revoke the Contracts between the Agents, as soon as they were previously provisioned using the Imprinting service. To operate, the Orchestrator service, requires commands issued by RESTful HTTPs API functions.

The list of available API functions is provided in the Appendix section.

The access to an Orchestrator API calls is protected by the accounts management system, which runs a basic API authentication service. In order to issue commands, the administrator has to complete the authentication process and get the secret key to call the API.

### **Asset Registry Service**

The Registry Service provides the repository of Agents under a certain Organization. It is necessary to match the provider Agent name and provider Agent unique blockchain address. For example, when a user Agent (e.g., an IoT device) receives an Access Contract, it needs to identify which is the associated provider Agent to contact.

Essentially, the Asset Registry Service provides the equivalent of a digital label for every Contract, with the name of the provider Agent on it. The provider Agent name is also needed in these two additional scenarios:

 - to *get* the contracts for a specific Agent

* To identify and reach the Agent  to open a session (e.g., using its MQTT topic)

### **Communication Channel (MQTT)**

In the default implementation, we use the MQTT channel for the communication between the Agents.

This Communication channel has two main uses:

* Data exchange between IoT devices, as usually see within the IoT space
* Data exchange between IoT devices and the UniquID Services within their Organization

UniquID also supports communications via TCP/IP, and fully supports low-footprint transport protocols like Bluetooth and unencrypted WiFi. UniquID’s services payload is protocol-agnostic, so it is possible to adopt custom communication transport and protocols (e.g. LoRa).

## **Contracts**

A UniquID *Contract* represents the agreement between UniquID Agents (which can be IoT devices or applications) that enforces the access rights within a UniquID organization.C Contracts  command and control three key principles:

* The identification system to match every Contract with the related Agents
* The communication channel being used for the data exchange (e.g., session authentication)
* The  access to RPC calls sent through the communication channel

UniquID provides contract management functionalities (such as orchestration and revocation) through the Permission Management Service (PMS, aka the *Orchestrator*). Access Contracts are stored in the blockchain by issuing micro-transactions thus, IoT devices trust Contracts changes by keeping the synchronization with the underlying Blockchain - as usually happens with normal cryptocurrency wallets.

A Contract is mandatory to open secure sessions between two IoT devices - of which one is the *User Agent*  and the other a *Provider Agent*. This logical distinction exists only at the Contract level: every Agent within the Organization can be User and Provider, depending on the underlying Contract. As an example, an IoT gateway can be the *User* of cloud IoT applications, and *Provider* of the MQTT broker for local, low-range wireless sensors.

Every Contract can be disabled: during its creation, the administrator elects a third entity, the *Revoker Agent*, which is entitled to disable - revoke - the Contract by issuing a transaction on the blockchain. By default the Revoker Agent is the Provider Agent itself, which controls the process via RPC, however the administrator can elect any other Agent belonging to the same Organization, and disable the contract remotely.

UniquID Contracts are embedded inside standard blockchain transaction, so the logic is governed by standard inputs and outputs composition, as defined by UTXO principles.

The table below describes how transaction inputs and outputs are defined inside standard UniquID Contracts. The OP\_RETURN contains the protocol metadata, storing the Access Control List (ACL). The order of inputs and outputs is fixed: the Provider Agent must use the Output[0] (user address public key hash) to authenticate the User Agent.

Table 2.  New Contract Parameters

|  |  |
| --- | --- |
| Input | Output |
| 0 - provider address | 0 - user address |
|  | 1 - OP\_RETURN |
|  | 2 - revoker address |
|  | 3 - change address\* |

\* the change address is needed to comply with the blockchain protocol

Every Contract is managed through the Orchestrator, which is capable of two main actions: *contract creation* and *contract revocation*.

### **Contract Creation**

*The contract creation* process follows this sequence of actions:

1. The *Orchestrator* creates a contract between two selected Agents, of which one is the *User Agent*, and the other is the *Provider Agent*. The Contract is embedded inside a blockchain transaction object, using standard inputs and outputs.
2. The Orchestrator sends the Contract transaction to the Provider Agent asking the signature
3. The Provider Agent sends back the signed transaction to the Orchestrator
4. The Orchestrator stores locally the signed transaction as a new Contract, and broadcasts it to the blockchain
5. As soon as the Agents are synchronized with the blockchain, they can download and store newly issued transactions as Contracts
6. Agents save the Contracts in the local database, acting as a cryptocurrency wallet
7. *User* and *provider* Agents can use the transaction data to open a  communication and call RPCs

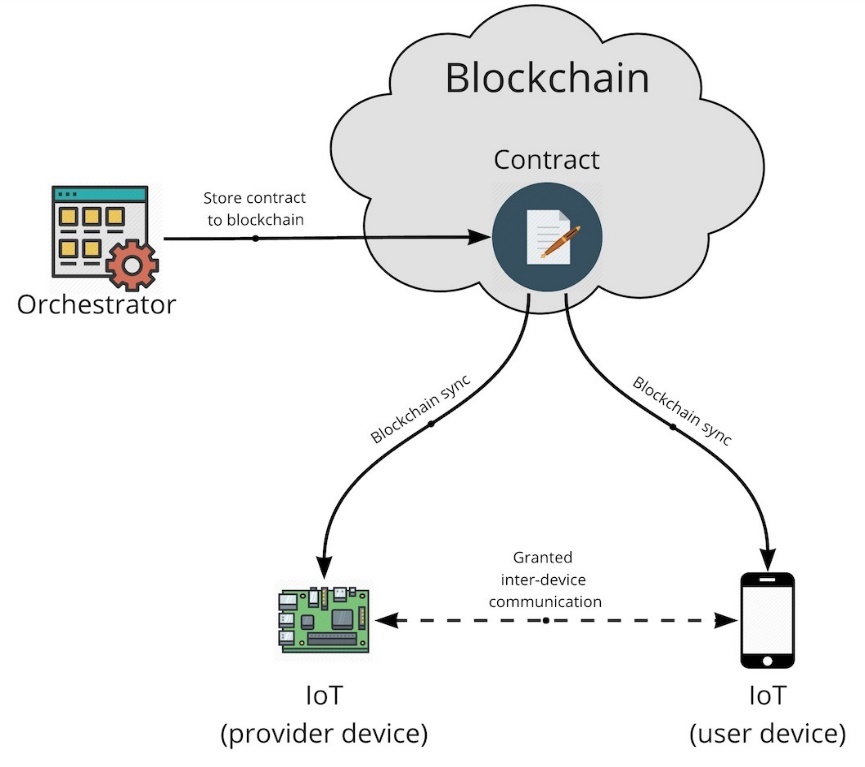


Fig. 8. Contract creation between Provider and User  Agents (e.g. IoT devices).

### **Contract Revocation**

*The Contract* *revocation*, at the high level, works the same of contract creation. At the protocol level, a new transaction is spent on the blockchain, but with a different organization between inputs and outputs:

Table 3. Revocation Contract transaction

|  |  |
| --- | --- |
| Input | Output |
| 0 - revoker address | 0 - provider address |
|  | 1 - user address |
|  | 2 - change address\* |

\* the change address is needed to comply with the blockchain protocol

The Revoker Agent pubkey is stored at the input[0], while Provider and User Agent pubkey hashes are stored at the output[0] and output[1].This transaction is generated by the Orchestrator, signed by the Revoker and broadcasted to the blockchain.

Since the transactions for Contract creation and Contract revocation are stored in the blockchain,security audits may be simplified.

## **Access Control List**

The Access Control List (ACL) is defined in the OP\_RETURN field of each UniquID Contract, and contains the information about the permissions that the Provider Agent can grant to the User.

Each permission is identified by a specific RPC function executed on the device.

Table 4.  Access Control List Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| byte index | bit length | Functionality | Description | Value |
| 0 | 8 | - | version | 0 - for contract v.0 |
| 1 - 4 | 32 | 0-31 | bitmask - execution grants for system reserved RPC | 0 - access denied  1 - access granted |
| 5 - 18 | 112 | 32-143 | bitmask - execution grants for application defined RPC | 0 - access denied  1 - access granted |
| 19 - 79 | 488 | - | unused | Must be 0 |

Based on this schema, every RPC functionality is uniquely identified by an index between 0 and 143 of the OP\_RETURN payload. The first 32 are reserved for UniquID Agent internal use, the remaining 112 are available for custom functions.

Developers that need to embed the Agent inside their IoT device or application have to map the ACL identifiers (from 32 to 143) to the functions/objects hey already have in place. Normally, each ACL index is mapped with a function RPC, so the administrator can enable/disable each of them using the Orchestrator.

Table 5. Connected Car ACL Example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| byte index | bit index | Functionality | Description | Value |
| 0 |  | - | version | 0 - for contract v.0 |
| 1 - 4 |  | 0-31 | bitmask - execution grants for system reserved RPC | 0 - access denied  1 - access granted |
| 5 | 0 | 32 | bitmask - can open and close doors | 0 - access denied  1 - access granted |
| 5 | 1 | 33 | bitmask - can open the trunk | 0 - access denied  1 - access granted |
| 5 | 2 | 34 | bitmask - can turn on the ignition | 0 - access denied  1 - access granted |
| 5 | 3 - 7 | 35-39 | bitmask - not used in the use case | X - don’t care |
| 6-18 |  | 40-143 | bitmask - not used in the use case | X - don’t care |
| 19-79 |  | *-* | unused | 0 |

## **Message structure and RPC**

UniquID Framework defines a structure for the message structure, that is agnostic from the communication protocol. This approach leaves the  developer free to choose the ideal communication protocol (e.g., MQTT, CoAP, HTTPs), based on the existing requirements. The framework adopts message and RPC structure  inspired by JSON-RPC, without strictly following its implementation.

While in the JSON-RPC [...], within the UniquID messaging framework, the User Agent sends a request to the Provider Agent and wait for a response.

When the Provider Agent receives an RPC request, it begins the handshake process, following these steps:

* check if there exists a valid Contract with the User (among the Contracts stored in the local cache)
* if a Contract exists, the Provider compares the RPC request index with the Contract bitmask, to verify if the User has permission to  access that specific functionality.
* If the Contract contains the bit of the requested RPC set to 1, the Provider executes the correspondent method and send back the response to the User, otherwise the Provider discards the request, and doesn't answer.

An example of RPC exchange between User and  Provider might be:

|  |
| --- |
| Request |
| {  "body": {  "Method":32,  "Params":"008000000000c3d9",  "id":1564140714421  },  "signature":"INaJMkHy8rh8SN1+CBjUdGsrnFAaXHVScpbltasEsWE/PLIVhsbmwgYCu3B2VWFbp40FQNULNq9pG6qSiw2gr/E="  } |

Table 6.  Request Message Structure

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| method | number | Method requested by the User Agent (requestor) |
| params | string | Optional parameters used by the method. The application running on top of the Provider Agent implements the function to parse it |
| id | integer/big num | Normally the request’s timestamp. Any response message must contain the same id |
| signature | string | Bitcoin signature of the serialized “body: { }” JSON message component |

|  |
| --- |
| Response |
| {  "Body":{  "Result":"",  "Error":0,  "Id":1564140714421  },  "signature":"IEz3+gpCy1bewFrC1rN+wGqbSIzf1tv5O12xPBFzAGDQCifvwZTqIFa+R9ZzAjMebX/uG5uPVUeFbv29yfUwsa4="  } |

Table 7.  Response Message Structure.

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| result | string | The reply to the requested method. The application running on top of the User Agent implements the function to parse it. Any execution error or exception  can be reported here |
| error | number | Error at messaging level (e.g. malformed JSON). If error == 0 the request is successful and the result is stored in the "result" field |
| id | number | The same id of the Request Message timestamp. |
| signature | string | Bitcoin signature of the serialized “body: { }” JSON message component fields |

### **Message Signature and Authentication**

Every signature in the UniquID Message protocol leverages the “Bitcoin message signature” algorithm. Therefore, the signature is applied to the *digest* of the serialized JSON message.

The serialization algorithm for the request json is:

**sprintf(serializeData,"%d%s" PRId64 "", method, params, id);**

The above example request

{

"body": {

"Method":32,

"Params":"008000000000c3d9",

"id":1564140714421

},

"signature":"INaJMkHy8rh8SN1+CBjUdGsrnFAaXHVScpbltasEsWE/PLIVhsbmwgYCu3B2VWFbp40FQNULNq9pG6qSiw2gr/E="

}

will return:

**char serializeData[] = "32008000000000c3d91564140714421";**

The “Bitcoin message signature” algorithm is executed  as

**base64(recoveriByte || ecdsa( sha256( sha256( "\x18" "Bitcoin Signed Message:" "\n" + varint(serializeDataLen) + serializeData) ) ) )**

Where **||** represent the concatenation of the recoveryByte with the ECDSA algorithm.

The serialization algorithm for the response JSON is:

**sprintf(serializeData,"%d%s" PRId64 "", error, result, id)**

The example above will give:

**char serializeData[] = "01564140714421";**

The signature algorithm is the same as described above for the request.

A feature of this ECDSA signature implementation is that a public key can be calculated by knowing:

* the data itself
* the signature applied to it
* the ECDSA recoveryByte

Therefore,  there’s no need for the Provider Agent to know the public key (relative to the private key) of the User Agent that signed the message. Once the public key is calculated, its validity is calculated by comparing it with the Blockchain Address present on the Contract (which is the hash of the public key itself).

UniquID Agents, by leveraging this feature,  can securely identify and authenticate the source of a given request, leveraging the Contract information instead of a centralized PKI.

# UniquID System Administration

## **Console**

Once you have registered your new Organization on UniquID, you have to download the credentials for your account from the Console web application, by clicking the “Create Admin” button in the top-right section of the main page. The credentials file is needed to authenticate the UniquID command-line tool (CLI) to the UniquID backend services.

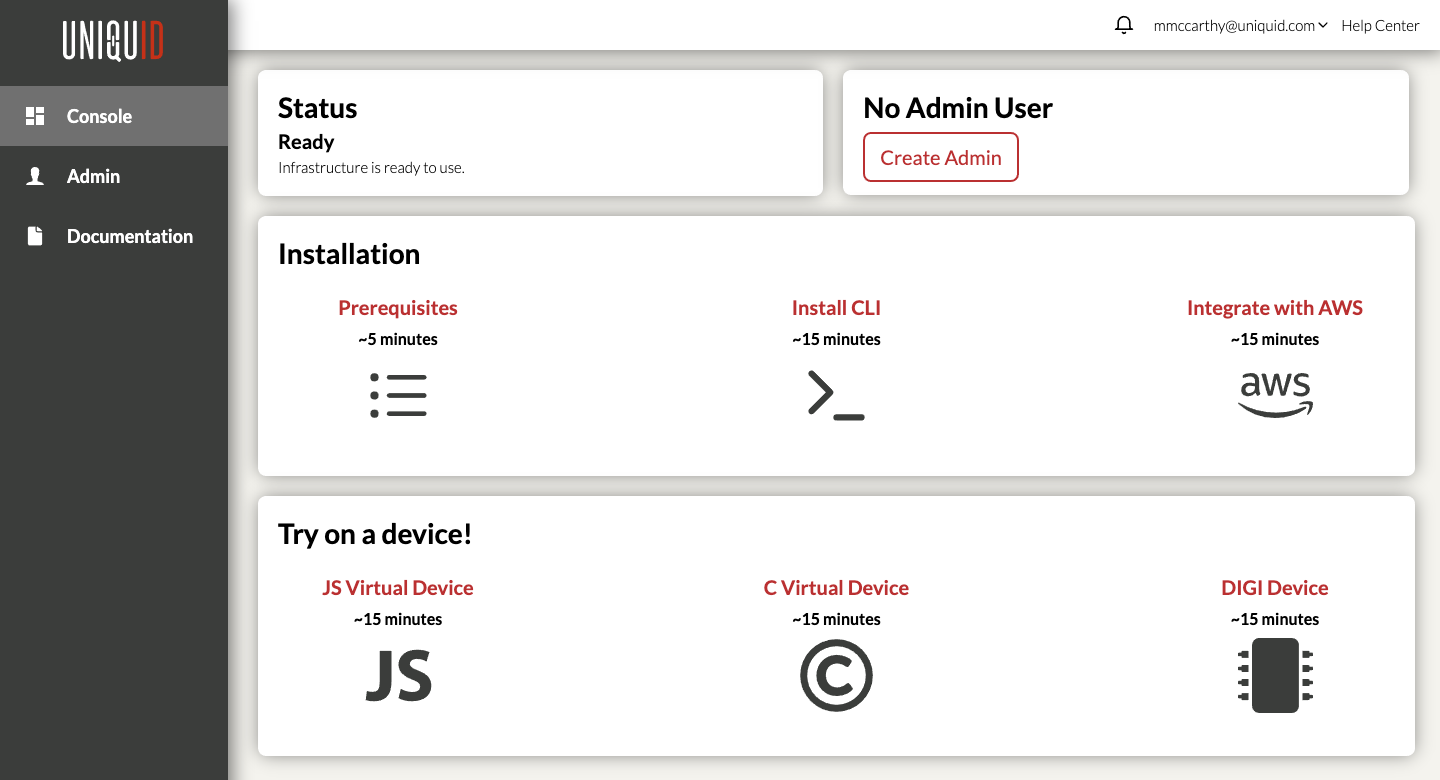


Fig. 9.  UniquID Administration Console

This Console web application also allows you to delete your Organization’s account. WARNING: this action is irreversible, and any metadata provided by UniquID Backend Services will be deleted. Therefore, the Agents you provisioned into the system may no longer function correctly.

## **Command-Line Interface tool (CLI)**

UniquID’s Command-Line Interface (CLI) provides the administrator a tool to interact with the UniquID backend services using standard *\*nix* command-line interface. The commands accept input arguments as files in the JSON format, and data can be printed to the standard output in either text or JSON format.

The tool is developed using Python3 and it can be downloaded from PyPi.org using the ‘pip3’ package installation tool.  Full instructions on how to install the tool are available at:

<https://pypi.org/project/uniquid/>

Once the tool is installed correctly, you will need to download the credentials file for your Organization from the UniquID Console. WARNING: this credentials file must be kept in a secure location as it allows whoever has it in their possession to login to the backend services, and modify the state of the Contracts inside your UniquID Organization.

The CLI tool provides several commands and many of these accept options to modify the command’s behaviour. Table 8 briefly describes the available commands.  Full documentation for each command is available using the ‘uniquid --help’ option, or appending ‘--help’ option to a specific command (e.g., ‘uniquid login --help’).

Table 8.  UniquID CLI Tool Commands

|  |  |
| --- | --- |
| Command | Description |
| login | Use the secret credentials to authenticate the CLI tool with the UniquID backend services |
| logout | Remove the previous authentication between the CLI tool and the UniquUD backend services |
| status | View the status of the authentication between the CLI tool and the UniquID backend services |
| log | View the log generated by the UniquID backend services in real-time |
| list-aliases | View a list of short aliases which are available for the CLI tool’s commands.  All commands have a corresponding alias |
| list-devices | View a list of all of the UniquID Agents, including IoT devices and other applications, whose identities have been imprinted and orchestrated |
| show-device | Show detailed information about a specific UniquID agent |
| list-contracts | View a list of all of the Access Contracts, both active and revoked, which exist between UID Agents |
| show-contract | Show detailed information about a specific Access Contract |
| create-contracts | Create an Access Contract between two UniquID Agents |
| delete-contracts | Revoke an existing Access Contract |
| deploy | Creates the infrastructure and resources necessary to integrate the UniquID system with a third-party system. There are currently two options for this command:   * Basic.  Generate the configuration file which is used to configure UniquID Agents to connect to your organization. * AWS.  Generates the configuration file and certificate which are necessary to connect a UniquID Agent to your organization in the UniquID system and to your organization’s AWS IoT Core infrastructure. This option also creates the necessary infrastructure in AWS so that your UniquID organization is connected to your AWS IoT Core infrastructure |
| undeploy | There is currently one option for this command:   * AWS.  Deletes the AWS infrastructure which was created by previously executing the ‘deploy aws’ command |

The CLI tool creates a log file in your current working directory when you execute a command. One log file is created for each day and the log from all commands are appended to the log file.  The name of the log file is:

uniquid\_cli\_<date in YYMMDD format>.log  
  
The command-line tool can be installed on Linux, OSX and Windows machines. WARNING: the commands ‘deploy’ and ‘undeploy’ are not currently supported on Windows machines.

### **CLI Real-Time Logging**

It is possible to monitor the logs generated by the UniquID backend services in realtime, launching the CLI tool with the ‘uniquid log’ command. This command will display the log containing information about major events in the system. Some of the events which may appear in the log are:

* Time since the last blockchain block was mined
* Identity Self-Provisioning Service (ISPS aka Imprinter) events from the UniquID backend services
* Permission Management Service (PMS aka Orchiestrator) events from the UniquID backend services

### **CLI Custom Scripts**

The CLI tool can be used to automate tasks using custom scripts. These scripts can be written in a programming language such as Python or Ruby. The programming language must provide two key features:

* ability to execute command-line tools and analyze their output
* ability to parse and change data in JSON format

Below, a short example script which is written in Python. This script shows the convenience to automatically capture and analyse the output from the UniquID CLI tool. These scripts can be conveniently streamlined within any existing Identity and Access Management platform, automating the creation and cancellation of Access Contracts or Capability Contracts.  
  
Example Script:

#! /usr/bin/env python3

import json

import shlex

import subprocess

import sys

if \_\_name\_\_ == '\_\_main\_\_':

    # Retrieve a list of UniquID devices in JSON format and

    # convert to a Python dictionary.

    devices\_cmd = 'uniquid ld -o json'

    arg\_list = shlex.split(devices\_cmd)

    result = subprocess.run(arg\_list,

                            stdout=subprocess.PIPE,

                            stderr=subprocess.PIPE)

    if result.returncode != 0:

        sys.exit('List devices command failed')

    devices\_str = result.stdout.decode('utf-8')

    devices\_list = json.loads(devices\_str)

    # Retrieve a list of UniquID contracts in JSON format and

    # convert to a Python dictionary.

    contracts\_cmd = 'uniquid lc -o json'

    arg\_list = shlex.split(contracts\_cmd)

    result = subprocess.run(arg\_list,

                            stdout=subprocess.PIPE,

                            stderr=subprocess.PIPE)

    if result.returncode != 0:

        sys.exit('List contracts command failed')

    contracts\_str = result.stdout.decode('utf-8')

    contracts\_list = json.loads(contracts\_str)

    # create a contract between the first and second devices.

    contract\_dict = {'functions': [38, 39]}

    if len(devices\_list) < 2:

        sys.exit('Must have at least 2 devices to create a contract')

    contract\_dict['provider'] = devices\_list[0]['xpub']

    contract\_dict['user'] = devices\_list[1]['xpub']

    new\_list = [contract\_dict]

    print(new\_list)

    # Create the new contract

    contract\_cmd = 'uniquid cc --input-json \'' + json.dumps(new\_list) + '\''

    print(contract\_cmd)

    arg\_list = shlex.split(contract\_cmd)

    result = subprocess.run(arg\_list,

                            stdout=subprocess.PIPE,

                            stderr=subprocess.PIPE)

    if result.returncode != 0:

        sys.exit('Contract creation command failed')

    contracts\_str = result.stdout.decode('utf-8')

    print(contracts\_str)

    sys.exit(0)

# Appendix

## API Documentation

The appendix contains reference information about UniquID’s API.

## Identity Self-Provisioning Service HTTP APIs

### API Overview

The described API operated with structures that always in JSON format.

All responses have Content-Type: application/json

|  |  |  |
| --- | --- | --- |
| **METHOD TYPE** | **URI** | **DESCRIPTION** |
| GET | /api/v1/devices | Get list of managed devices |
| GET | /api/v1/devices/:status | Get list of managed devices filtered by status |
| GET | /api/v1/enrollment | Get list of the orchestration contracts |
| GET | /api/v1/enrollment/:providerAddress | Get the orchestration contract for the specified Provider Address |
| POST | /api/v1/enrollment | Create a new orchestration contract |
| GET | /api/v1/orchestrators | Get list of Orchestrators |
| GET | /api/v1/orchestrators/:xpub | Get info about specific Orchestrator |
| POST | /api/v1/orchestrators | Create a new Orchestrator |
| DELETE | /api/v1/orchestrators/:xpub | Delete Orchestrator |
| GET | /api/v1/info | Get information about imprinter |
| GET | /api/v1/properties | Get settings information |
| POST | /api/v1/properties | Upload data and saves the file on disk |

### Detailed APIs description

### Get managed devices

The method returns all the devices managed by the current Imprinter.

* If success - method returns a json array of all the devices in the response body.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/devices |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [      {          "name": "device1",          "xpub": "xpub\_device\_1",          "creationTime": 1234567890,          "recipe": "test",          "retries": 0,          "status": "ORCHESTRATED"      },      {          "name": "device2",          "xpub": "xpub\_device\_2",          "creationTime": 1234567890,          "recipe": "test",          "retries": 0,          "status": "IMPRINTED"      }  ] | Success!  name – name of the device  xpub – public key of the device  creationTime – timestamp of device creation  recipe – is always "test"  retries – attempts to change status from \*ING to \*ED  status – can be CREATED, IMPRINTING, IMPRINTED, ORCHESTRATING, ORCHESTRATED |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Get managed devices filtered by status

The method returns all the devices managed by the current Imprinter filtered by status.

* If success - method returns in the response body a json array of all the devices with the specified status.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/devices/:status |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [     {       "name":"device\_name\_1",       "xpub":"public\_key\_1",       "creationTime":1234567890,       "recipe":"test"     },  {       "name":"device\_name\_2",       "xpub":"public\_key\_2",       "creationTime":1234567890,       "recipe":"test"     }  ] | Success!  *name* – name of the device  *xpub*– public key of the device  *creationTime* – timestamp of device creation  *recipe*– is always "test" |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  *error*– text message, describes the error reason |

### Get enrollment (orchestration) contracts

The method returns all the orchestration contracts created by the current Imprinter.

* If success - method returns in the response body a json array of all the devices with the specified status.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/enrollment |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [     {        "name":"device\_name\_1",        "providerAddress":"provider\_address\_1",        "userAddress":"user\_address\_1",        "txid":"transaction\_id",        "orchestratorName":"orchestrator\_name\_1",        "xpub":"imprinter\_public\_key"     },  {        "name":"device\_name\_2",        "providerAddress":"provider\_address\_2",        "userAddress":"user\_address\_2",        "txid":"transaction\_id",        "orchestratorName":"orchestrator\_name\_1",        "xpub":"imprinter\_public\_key"     }  ] | Success!  name – name of the device  providerAddress – bip32 address of the provider  userAddress – bip32 address of the provider  txid – unique identifier of the transaction  orchestratorName – name of the orchestrator that manage the device  xpub – public key of the device |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Get enrollment (orchestration) contract for a specific provider

The method returns orchestration contracts created by the current Imprinter for the specified device address.

* If success - method returns in the response body a json array of all the devices with the specified provider address.
* If given providerAddress doesn't exists returns http code 404 Not Found.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/enrollment/:providerAddress |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [     {       "name":"device\_name\_1",       "providerAddress":"provider\_address",       "userAddress":"user\_address\_1",       "txid":"transaction\_id",       "orchestratorName":"orchestrator\_name\_1",       "xpub":"imprinter\_public\_key"     }  ] | Success!  name – name of the device  providerAddress – bip32 address of the provider  userAddress – bip32 address of the provider  txid – unique identifier of the transaction  orchestratorName – name of the orchestrator that manage the device  xpub – public key of the device |
| 404 Not Found | {      "error": "Can't find contract with address"  } | Error! Given providerAddress doesn't exists  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Create an enrollment (orchestration) contract

The method creates a new orchestration contract between an orchestrator and a device.

* If success - method returns http code 200 OK.
* If request body is empty – API returns http code 400 Bad request, with error description
* If an exception throws during the creation process – then return http code 500 internal server error with the error description in the response body.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| POST | /api/v1/enrollment | {     "orchestrator":"xpub\_orchestrator",     "device":"xpub\_device"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK |  | Success! |
| 400 Bad Request | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### View orchestrators list

The method returns the list of all the orchestrators created by the current Imprinter.

* If success - method returns in the response body a json array of all the orchestrators.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/orchestrators |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [     {        "name":"orchestrator\_name\_1",        "xpub":"orchestrator\_1\_public\_key",        "creationTime":1234567890     },     {        "name":"orchestrator\_name\_2",        "xpub":"orchestrator\_2\_public\_key",        "creationTime":1234567890     }  ] | Success!  name – name of the orchestrator  xpub – public key of the orchestrator  creationTime – timestamp of orchestrator creation |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### View orchestrator information

The method returns info about a specific orchestrator.

* If success - method returns in the response body a json array with the orchestrator information.
* If an exception throws during the process – then return http code 500 internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/orchestrators/:xpub |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | [     {        "name":"orchestrator\_name\_1",        "xpub":"orchestrator\_1\_public\_key",        "creationTime":1234567890     }  ] | Success!  name – name of the orchestrator  xpub – public key of the orchestrator  creationTime – timestamp of orchestrator creation |
| 404 Not Found | {      "error": "Can't find orchestrator with xpub"  } | Error! Given xpub doesn't exists  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Create a new orchestrator

The method creates a new orchestrator inside the current imprinter.

* If success - method returns http code 201 created.
* If an exception throws during the process – then return http code 400 bad request with the error description in the response body.
* If the xpub already exist – then return http code 409 Conflict with the error description in the response body.
* If any other exception throws during the creation process – then return http code 500 internal server error with the error description in the response body.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| POST | /api/v1/orchestrators | {     "name":"orchestrator\_name",     "xpub":"orchestrator\_xpub"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 201 OK | {      "name": "orchestrator\_name\_4",      "xpub": "xpub4",      "creationTime": 1534766128988  } | Success!  name – name of the orchestrator  xpub – public key of the orchestrator  creationTime – timestamp of orchestrator creation |
| 400 Bad Request | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |
| 409 Conflict | {      "error": "Resource already exists"  } | Error!  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Delete an orchestrator

The method deletes orchestrator by given xpub.

* If success - method returns http code 410 gone.
* If the xpub doesn’t exist, then return http code 404 Not found error with the error description in the response body.
* If any other exception throws during the creation process – then return http code 500 internal server error with the error description in the response body.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| DELETE | /api/v1/orchestrators/:xpub | {     "name":"orchestrator\_name",     "xpub":"orchestrator\_xpub"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 410 Gone |  | Success! |
| 404 Not Found | {      "error": "Resource not found"  } | Error!  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Get imprinter info

The method returns the information about the current Imprinter.

* If success - method returns the information about the current Orchestrator in the content body and http code 200 OK
* If the request has extra path – method returns http code 400 Bad request with "path not implemented" in the content body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/info |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | {      "name": "ImprinterTest",      "xpub": "tpubDBYesqrK7QKqb5zD2RowX...",      "topupAddress":"mrXWQAJi4Fk8Zt2MLgni...",      "userBalance": 1000000000,      "providerBalance": 0,      "devicesCreated": 0,      "devicesImprinted": 1,      "devicesImprinting": 0,      "devicesOrchestrated": 1,      "devicesOrchestrating": 0  } | Success!  name – name of the Imprinter  xpub – public key of the imprinter  topupAddress – address to use to recharge the Orchestrator  userBalance – coin balance of the "user" bip32 hierarchy  providerBalance – coin balance of the "provider" bip32 hierarchy  devicesCreated – number of devices in Created status  devicesImprinted – number of devices in Imprinted status  devicesImprinting – number of devices in Imprinting status  devicesOrchestrated – number of devices in Orchestrated status  devicesOrchestrating – number of devices in Orchestrating status |
| 400 Bad Request | {      "error": "Path not implemented"  } | Error!  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### View imprinter properties

The method allows to view the properties of the current Imprinter.

* If success - method returns the properties of the current Imprinter in the content body and http code 200 OK
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| GET | /api/v1/properties |  |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK | {      "network": "com.uniquid.regtest",      "mqttBroker":    "tcp://ipaddress:port",      "mqttTopic": "ImprinterTest",      "httpPort": 8090,      "insightApiUrl": "http://ipaddress:port/insight-api",      "registryUrl": "http://ipaddress:port",      "peers": "ipaddress;ipaddress;ipaddress"  } | Success!  network – blockchain network (bitcoin, litecoin, …)  mqttBroker – url of the mqtt broker  httpPort – server listening on this port  insightApiUrl – url of the insight api service  registryUrl – url of the registry service  peers – ip addresses of the peers |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

### Upload properties file

The method allows to upload a settings file to the current Imprinter.

* If success - method returns the information about the current Imprinter in the content body and http code 200 OK
* If request body is empty – API returns http code 400 Bad request, with error description
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **Method** | **Request** | **Body** |
| POST | /api/v1/properties | File to upload |

Response:

|  |  |  |
| --- | --- | --- |
| **Http Code** | **Body** | **Meaning** |
| 200 OK |  | Success! |
| 400 Bad Request | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |
| 500 Internal Server Error | {      "error": "some java exception"  } | Error!  error – text message, describes the error reason |

## Permission Management Service HTTP APIs

### API Overview

The described API operated with structures that always in JSON format.

All responses have Content-Type: application/json

|  |  |  |
| --- | --- | --- |
| **METHOD** | **URI** | **DESCRIPTION** |
| GET | /api/v1/devices | Get all managed devices |
| GET | /api/v1/devices/:xpub | Get info about specific device |
| GET | /api/v1/contracts | Get list of the access contracts |
| POST | /api/v1/contracts | Create new access contracts |
| DELETE | /api/v1/contracts/:txid | Revoke existing access contracts |
| GET | /api/v1/info | Returns information about orchestrator |
| GET | /api/v1/properties | Get orchestrator properties |
| POST | /api/v1/properties | Upload data and saves the file on disk |
| POST | /api/v1/init | Init orchestrator |
| POST | /api/v1/init/admin | Create admin |
| POST | /api/v1/users | Create new user |
| PUT | /api/v1/users/:userId | Update existing user |
| GET | /api/v1/users | Get users list of the tenant |
| DELETE | /api/v1/users/:userId | Delete user |
| POST | /api/v1/login | User login |
| GET | /api/v1/logout | User logout |
| POST | /api/v1/connectors | Create new connector (AWS, Azure, Google) |
| PUT | /api/v1/connectors/:id | Edit existing connector |
| GET | /api/v1/connectors | Get list of all connectors |
| GET | /api/v1/connectors | Get list of all connectors |
| DELETE | /api/v1/connectors/:id | Delete existing connector |
| GET | /api/v1/connectors/:id/status | Test orchestrator ↔ AWS connection |

### Detailed API description

### Get all managed devices

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API returns all managed devices. User must be logged on.

* If success - API returns a JSON array of all the orchestrated devices in the response body.
* If no device found – then returns http code 404 Not found, with error description in the body.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/devices | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | [     {        "name": "device\_name\_1",        "xpub": "public\_key\_1",        "recipe": "test",        "contracts": 1,        "shared": false     }, {        "name": "device\_name\_2",        "xpub": "public\_key\_2",        "recipe": "test",        "contracts": 1,        "shared": false     },     ...  ] | Success!   name – name of the device  xpub – public key of the device  recipe – is always "test"  contracts – number of contracts involving this device, are always 1  shared – flag indicates if this is Shared Device; provided for future evolution. Always false in this version |
| 404 Not Found | {     "error": "No devices found"  } | Error! Orchestrator have no devices   error – text message, describes the error reason |
| 401 Unauthorized | {      "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.   error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.   stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Get info about specific device

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API returns information about the specific device.. To use this API – User must be logged on.

* If success – API returns a JSON array containing just one item in the response body.
* If there is no device with given xpub – then return http code 404 Not found with error description
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/devices/:xpub | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | {     "name":" dev\_name\_1",     "xpub": "pub\_key\_1",     "recipe": "test",     "contracts": 1,     "shared": false  } | Success!   name – name of the device  xpub – public key of the device  recipe - is always "test"  contracts – number of contracts involving this device. Is always 1  shared – flag indicates if this is Shared Device; provided for future evolution. Always false in this version |
| 404 Not found | {     "error": "Device with xpub: 'af3J42if4828c2' not found"  } | Error! Orchestrator have no device with given xpub   error – text message, describes the error reason |
| 401 Unauthorized | {      "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.  error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.   stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Get list of access contracts

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API returns a list of all the Access Contracts of current tenant. To use this API – User must be logged on.

* If success – API returns a JSON array containing all the access contracts in the response body.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/contracts | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | [    {       "provider":            {                 "name":"device\_name\_1",                 "xpub":"device\_xpub",                 "address":"device\_address\_1"             },       "user":            {                 "name":"device\_name\_2",                 "xpub":"device\_xpub\_2",                 "address":"device\_address\_2"             },       "revoker":             {                 "name":"device\_name\_3",                 "xpub":"device\_xpub\_3",                 "address":"device\_address\_3"             },       "txid":"txid",       "recipe":"Access",       "status":"ACTIVE",       "timestamp":123456789,       "functions": [34,35],       "shared": false    },    … ] | Success!  provider – provider device information user – user device information revoker – revoker device information txid – unique identifier of the transaction recipe – is always "Access" status – status of the contract. It can be 'pending', 'active' or 'revoked' timestamp – creation time of the contract functions – functionality enabled by the "provider" to the "user" shared – flag indicates if this is Shared Contract; provided for future evolution. Always false in this version |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials" } | Error! The JSESSIONID is absent, incorrect or expired.  error – text message, describes the error reason |
| 500 Internal Server Error | {     stack: [         {             "error": "some java exception"             "trace": [                 "com.uniquid.servlet.....                 "com.uniquid.dao.....                 ....              ]         },         { …. },         ….     ] } | Error! Unexpected error. Must not appear in release version.  stack – array of cascade of exceptions error – text message, describes the error reason trace – the array of strings contains java trace for error. |

### Create new Access Contract

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API creates a new Access Contract between devices provider and user. This API allows to create contract and if both devices belongs to current tenant. To use this API - User must be logged on.

* If success – API returns http code 200 OK
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If request body is empty – API returns http code 400 Bad request, with error description.
* If provider does not exist in orchestrator database – API returns http code 404 Provider with given Xpub not found.
* If user does not exist in orchestrator database – API returns http code 404 User with given Xpub not found.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/contracts | {     "provider":"device\_xpub",     "user":"device\_xpub",     "functions": [34,35] }  "functions" is optional. If not set, then the orchestrator will allow functions 33 to 37.   Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials" } | Error! The JSESSIONID is absent, incorrect or expired.  error – text message, describes the error reason |
| 400 Bad request | {     "error": "Body can't be empty" } | Error! Client provide incorrect request.  error – text message, describes the error reason |
| 404 Not found | {     "error": "Provider with given Xpub not found" } | Error! There is no provider with given Xpub in database of Orchestrator  error – text message, describes the error reason |
| {     "error": "User with given Xpub not found" } | Error! There is no user with given Xpub in database of Orchestrator  error – text message, describes the error reason |
| 500 Internal Server Error | {     stack: [         {             "error": "some java exception"             "trace": [                 "com.uniquid.servlet.....                 "com.uniquid.dao.....                 ....              ]         },         { …. },         ….     ] } | Error! Unexpected error. Must not appear in release version.   stack – array of cascade of exceptions error – text message, describes the error reason trace – the array of strings contains java trace for error. |

### Delete an Access Contract

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API deletes existing Access Contract. To use this API – User must be logged on.

* If success – API returns http code 200 OK
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If contract with given txid not exist – then return http code 404 Not found
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| DELETE | /api/v1/contracts/:txid | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 401 Unauthorized | {      "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 404 Not found | {      "error": "Contract not found"  } | Error! Resource not found.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Get Orchestrator info

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API returns the information about the current Orchestrator. User must be logged on.

* If success – API returns the information about the current Orchestrator in the content body and http code 200 OK
* If the request has extra path – method returns http code 400 Bad request with "path not implemented" in the content body.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/info | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | {        "name":"OrchestratorNJI95H28R4TA",        "publicKey": "tpubDBYT1CEYNPzH9AveZd55....",        "providerBalance":0,        "userBalance":0,        "topupAddress":"mkZusM4szmV7pkWYDvdVTHk28Jm..."  } | Success!    name – name of the Orchestrator  publicKey – Orchestrator’s public key  providerBalance – coin balance of the "provider" bip32 hierarchy  userBalance – coin balance of the "user" bip32 hierarchy  topupAddress – address to use to recharge the Orchestrator |
| 400 bad request | Path not implemented | Error! Extra path information not permitted |
| 401 Unauthorized | {      "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Get Orchestrator properties

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API allows to list the properties of the current Orchestrator. User must be logged on.

* If success – API returns the properties of the current Orchestrator in the content body and http code 200 OK
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/properties | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **REQUEST** | **MEANING** |
| 200 OK | {     "network":"",     "mqttTopic":"",     "mqttBroker":"",     "httpPort":"",     "insightApiUrl":"",     "registryUrl":"",     "imprinterUrl":""  } | Success!    network – blockchain network  mqttTopic – listening topic  mqttBroker – broker for MQTT communication  insightApiUrl – blockchain API service  registryUrl – URL of the registry service  imprinterUrl – URL of the imprinter service |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.  error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Update Orchestrator properties

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API allows to upload a settings file to the current Orchestrator. User must be logged on.

* If success – API returns the information about the current Orchestrator in the content body and http code 200 OK
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If request body is empty – API returns http code 400 Bad request, with error description
* If an exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/properties | File to upload    Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Init Orchestrator

No special authentication requirements for this API call

The API initialize orchestrator (set organization). This organization needs to start working orchestrator and use to communicate between another components of the system. The Init method can be called only once at the beginning. All other APIs couldn't be called before the Init call.

* If success – API returns https code 200 OK.
* If request body is empty – API returns http code 400 Bad request, with error description
* If any other exception throws during the init orchestrator process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/init | {      "orgId": "pepsico"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 403 Forbidden | {     "error": "Settings already initialized"  } | Error! The Orchestrator already initialized. The second call of Init not allowed.  error – text message, describes the error reason |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Create admin

No special authentication requirements for this API call

The API created first (admin) user. This admin user must be used to manage new users. The Init method can be called only once at the beginning. All other APIs couldn't be called before the Init call.

* If success – API returns https code 200 OK.
* If request body is empty – API returns http code 400 Bad request, with error description
* If any other exception throws during the init orchestrator process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/init/admin | {      "userId": "[email@mail.com](mailto:email@mail.com)"      "password": "14f9un34nu34f9u34"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 403 Forbidden | {     "error": "Admin user already initialized"  } | Error! The Orchestrator already initialized. The second call of Init not allowed.  error – text message, describes the error reason |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Create user

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API creates new user inside the current Orchestrator. User must be logged on.

* If success – API returns http code 201 Created.
* If request body is empty – API returns http code 400 Bad request, with error description
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If user with given userId already exists – method fails and return http code 409 Conflict.
* If any other exception throws during the user creation process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/users | {      "userId": "[email@mail.com](mailto:email@mail.com)"      "password": "14f9un34nu34f9u34"  }    Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 201 Created | {     "userId": "[email@mail.com](mailto:email@mail.com)"     // no password in response!  } | Success! |
| 409 Conflict | {      "error": "User already exists"  } | Error! Given userId already exists.  error – text message, describes the error reason |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Update user

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API update user user's information. User must me logged on. In the request body should be only fields that user wants to modify, fields without changes must be skipped.

* If success – API returns http code 200 OK.
* If request body is empty – API returns http code 400 Bad request, with error description
* If given userId is absent returns http code 404 Not found.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the user update process – then return http code 500 Internal server error with the reason description of error.

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | {     "userId": "[new\_email@mail.com](mailto:email@mail.com)"     // no password in response!  } | Success! |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 404 Not found | {     "error": "Can't find user with id"  } | Error! Given userId doesn't exists.  error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| PUT | /api/v1/users/:userId | {      "userId": "[new\_mail@mail.com](mailto:email@mail.com)"      "password":"u34ff9u4n4n39u31f4"  }    Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

### View users list

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API returns the list of all users. User must be logged on.

* If success – API returns http code 200 OK and body with the array of user objects. User objects does not contain password (for security purpose).
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/users | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | [      {          "userId": "first@mail.com"          // no password in response!      },      {          "userId": "second@mail.com"          // no password in response!      },      ...  ] | Success!    userId – unique user's identifier, in our case email. |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Delete user

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API delete user by given userId. User must be logged on.

* If success - API returns http code 410 Gone.
* If user with given userId not exist – then return http code 404 Not found.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| DELETE | /api/v1/users/:userId | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 410 Gone |  | Success! |
| 404 Not found | {     "error": "Can't find user with id"  } | Error! The user with given userId not found  error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Login

*No special authentication requirements for this API call*

The API performs user login with given userId and password.

* If success - method returns JSESSIONID which must be used as a cookie in all session requests.
* If request body is empty – API returns http code 400 Bad request, with error description
* If user with given password is incorrect – method fails and return http code 401 Unauthorized.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/login | {      "userId": "[email@mail.com](mailto:email@mail.com)",      "password": "14f9un34nu34f9u34"  } |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | Set-Cookie: JSESSIONID=298zf09hf012fh2; | Success!    JSESSIONID – session key to perform the API calls from CLI/webApp |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The userId, password combination is incorrect.  error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Logout

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API preforms user logout.

* If success – API returns http code 200 OK.
* If user with given JSESSIONID is incorrect or key is expired – method fails and return http code 401 Unauthorized.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/logout | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is incorrect or expired |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Create cloud connection

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API creates new cloud connection for current tenant (AWS/Azure). User must be logged on. In the request body all fields are mandatory.

* If success – API returns http code 200 OK and object with the information about created cloud connection except secret\_access\_key (for security purpose).
* If request body is empty – API returns http code 400 Bad request, with error description
* If user with given JSESSIONID is incorrect or key is expired – method fails and return http code 401 Unauthorized.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| POST | /api/v1/connectors | {      "name": "My AWS Cloud",      "type": "aws",       // specific fields for given cloud type  }    Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 201 Created | {     "id": 12,      "name": "My AWS Cloud",      "type": "aws",       // specific fields for given cloud type  } | Success!    id – unique cloud connector identifier  name – connector's alias  type – the cloud type (aws, azure, google)  Depending on type – object can contain different additional fields specific to selected cloud. |
| 400 Bad request | {      "error": "Body can't be empty"  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is incorrect or expired  error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Update cloud connection

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

The API updates the existing cloud connector for current tenant by given id. User must me logged on. In the request body should be only fields that user wants to modify.

* If success - API returns http code 200 OK and object with the information about updated cloud connection except password fields like secret\_access\_key (for security purpose).
* If request body is empty – API returns http code 400 Bad request, with error description.
* If request URI doesn't contain id or id is string (must be number only) – return http code 400 Bad request
* If cloud connection with given id doesn't exist – then return http code 404 Not found.
* If user with given JSESSIONID is incorrect or key is expired – method fails and return http code 401 Unauthorized.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| PUT | /api/v1/connectors/:id | {      "name": "New name",      "type": "aws",      // specific fields for given cloud type  }    Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | {     "id": 12,      "name": "New name",      "type": "aws",       // specific fields for given cloud type  } | Success!    id – unique cloud connector identifier  name – connector's alias  type – the cloud type (aws, azure, google)  Depending on type – object can contain different additional fields specific to selected cloud. |
| 400 Bad request | {      "error": "Body can't be empty"  }  {     "error": "Missed or incorrect id...  } | Error! Client provide incorrect request.    error – text message, describes the error reason |
| 404 Not found | {     "error": "Can't find Cloud Connector with id"  } | Error! The cloud connection with given id doesn't exist.  error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is incorrect or expired |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Delete cloud connector

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

This API deletes the existing cloud connector for current tenant by given id. User must me logged on.

* If success - API returns http code 410 Gone.
* If connector with given id doesn’t exist – return http code 404 Not found
* If request URI doesn't contains id or id is string (must be number only) – return http code 400 Bad request
* If user with given JSESSIONID is incorrect or key is expired – method fails and return http code 401 Unauthorized.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| DELETE | /api/v1/connectors/:id | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 410 Gone |  | Success! |
| 404 Not found | {     "error": "Can't find Cloud Connector with id"  } | Error! The cloud connection with given id not found in current tenant.  error – text message, describes the error reason |
| 400 Bad request | {     "error": "Missed or incorrect id...  } | Error! Wrong id given. It must be the number only and it's mandatory.  error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is incorrect or expired  error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### View connector

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

This API returns cloud connector by given id. User must me logged on. API returns cloud connector object with the fields specific to selected cloud. Selected cloud depends on field type which can be aws, azure or google. Connector objects does not contain password information like secretAccessKey.

* If success – API returns http code 200 OK and body with information about cloud connector by given id.
* If cloud connector with given id not exist – then return http code 404 Not found.
* If request URI must have id as number, otherwise, if it's string – then return http code 400 Bad request.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/connectors | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | {     "id": 12,      "name": "My AWS Cloud",      "type": "aws",       // specific fields for given cloud type  } | Success!    id – unique cloud connector identifier  name – connector's alias  type – the cloud type (aws, azure, google)  Depending on type – object can contains different additional fields specific to selected cloud. |
| 404 Not found | {     "error": "Can't find Cloud Connector with id"  } | Error! The cloud connection with given id not found.  error – text message, describes the error reason |
| 400 Bad request | {     "error": "Incorrect id. Must be number only!";  } | Error! Wrong id given. It must be the number only.  error – text message, describes the error reason |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### View all connectors

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

This API returns the list of all existing connectors of current tenant. User must me logged on. API returns list of cloud connector objects with the fields specific to selected cloud. Selected cloud depends on field type which can be aws, azure or google. Connector objects does not contain password information like secretAccessKey.

* If success – API returns http code 200 OK and body with the array of connector objects. Connector objects does not contain password information like secretAccessKey.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/connectors | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK | [     {         "id": 12,         "name": "My AWS Cloud",         "type": "aws",          // specific fields for AWS cloud type     },     {        "id": 18,         "name": "Microsoft Cloud",         "type": "azure",          // specific fields for AZURE cloud type     },     {        "id": 19,         "name": "My Test Google cloud",         "type": "google",          // specific fields for Google cloud type     }     ...  ] | Success!    id – unique cloud connector identifier  name – connector's alias  type – the cloud type (aws, azure, google)  Depending on type – object can contains different additional fields specific to selected cloud. |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

### Connector status

Required the JSESSIONID in the request cookies, user must be logged in (see: Login API)

This API test credentials and connection of given connector by accessKeyId. User must me logged on.

* If success – API returns http code 200 OK and body.
* If the JSESSIONID is absent, incorrect or expired, then return http code 401 Unauthorized with the error description in the response body.
* If any other exception throws during the process – then return http code 500 Internal server error with the reason description of error.

Request:

|  |  |  |
| --- | --- | --- |
| **METHOD** | **REQUEST** | **BODY/HEADERS** |
| GET | /api/v1/connectors/:access\_key\_id/status | Cookie: JSESSIONID=298zf09hf012fh2; |

Response:

|  |  |  |
| --- | --- | --- |
| **HTTP CODE** | **BODY** | **MEANING** |
| 200 OK |  | Success! |
| 503 Service Unavailable |  | Error! Can't connect to AWS. |
| 403 Forbidden |  | Error! Incorrect credentials. Successfully connected to AWS but rejected in credentials verification. |
| 401 Unauthorized | {     "error": "Session expired or incorrect credentials"  } | Error! The JSESSIONID is absent, incorrect or expired.    error – text message, describes the error reason |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |
| 500 Internal Server Error | {      stack: [          {              "error": "some java exception"              "trace": [                  "com.uniquid.servlet.....                  "com.uniquid.dao.....                  ....               ]          },          { …. },          ….      ]  } | Error! Unexpected error. Must not appear in release version.    stack – array of cascade of exceptions  error – text message, describes the error reason  trace – the array of strings contains java trace for error. |

## Software Components

|  |  |
| --- | --- |
| Document | URL |
| Java Core library | <https://github.com/uniquid/uidcore-java> |
| Orchestrator (Java) | <https://github.com/uniquid/massive-orchestrator> |
| Orchestrator library (Java) | <https://github.com/uniquid/uidorchestrator-java> |
| Imprinter (Java) | <https://github.com/uniquid/massive-imprinter-java> |
| Imprinter library (Java) | <https://github.com/uniquid/uidimprinter-java> |
| Registry (Java) | <https://github.com/uniquid/registry-java> |
| Javascript Core Library | <https://github.com/uniquid/uidcore-js> |
| C Core library | <https://github.com/uniquid/uidcore-c> |
| Node Agent (C) | <https://github.com/uniquid/uidagent-c> |