Technical description of the Waves Enterprise platform

Release 1.15.0

https://wavesenterprise.com

May 08, 2024

PLATFORM INSTALLATION AND USAGE

The Waves Enterprise blockchain platform is a comprehensive distributed ledger system that allows the formation of both public and private blockchain networks to solve various tasks, including those in the corporate and public sectors.

What is blockchain?

Blockchain is a continuous consequent chain of linked blocks that contain some information. This chain is replenished with new blocks. The process of new blocks creation is called *mining*. Each block contains a hash sum of the previous block data. This makes it impossible to change the content of any block after its broadcasting in the network, because it requires modification of all chain blocks at all the nodes of the blockchain.

At the corporate level, the blockchain technology is used for development of **distributed ledger systems**. A distributed ledger system does not have a unified control center, its data are stored simultaneously at all nodes of a network. In order to update data, consensus algorithms are used that automatically confirm that all network nodes have the same data copy.

Such a system provides security of transferred data and resolves the problem of trust between the network participants.

What is the Waves Enterprise blockchain platform designed for?

The Waves Enterprise blockchain platform allows to perform a wide range of business and public tasks:

- Workflow speed-up due to authomatization of business processes and lower number of mediators.

- Protection of data from external modification with the use of encryption and multi-level check of every operation within the network.

- Business applications of any complexity due to wide opportunities of smart contract development and comfortable blockchain integration tools.

- Achievement of mutual trust between participant of business workflow due to guaranteed acceptance of majority opinion in the de-centralized network.

Learn more about private projects based on the Waves Enterprise blockchain platform at our official website.

CONTENTS

1.1 System requirements

Currently Waves Enterprise blockchain platform supports Unix-like systems (for example, popular Linux and MacOS distributives). Waves Enterprise platform can be run effectively on the following operating systems:

- server operating system:

    - CentOS 6/7 (x64);

    - Debian 8/9/10 (x64);

    - Red Hat Enterprise Linux 6/7 (x86);

    - Ubuntu 20.04 (x64).

- workstation operating systems:

    - Ubuntu 20.04 (x64) и выше;

    - macOS Sierra and above.

Hardware and system requirements for the computer where a new Waves Enterprise node is deployed are stated below.

Variant	vCPU	RAM	SSD	JVM operation mode
Минимальные требования	2+	4Gb	50Gb	java -Xmx2048M -jar
Recommended requirements	2+	4+ Gb	50+ Gb	java -Xmx4096M -jar

Hint: `Xmx` flag defines the maximum size of JVM memory available.

1.1.1 Environment requirements for the Waves Enterprise blockchain platform

Important: Waves Enterprise platform is distributed as a Docker image, so there is no need to install any software other than Docker and Docker-compose and to configure the environment. With Docker you can deploy a Docker container from a Docker image which already contains Java, CryptoPro and other *necessary software*.

However, you must purchase the licenses for the *proprietary software* from its manufacturer and then transfer the licenses to the node via the environment variables as described below in the *Installing CryptoPro CSP license* section.

The *open-source components* do not require license keys to be obtained and passed to the node.

Following are the environment requirements for the Waves Enterprise blockchain platform:

- Oracle Java SE 11 (64-bit) or OpenJDK 11 and higher
- Docker CE
- Docker-compose

Installing CryptoPro CSP license

After you have obtained a CryptoPro CSP license, locate the `/configs/node/node.env` env file on the node, and set the environment variables to the values specified in the license:

```
CSP_LICENSE={{ CSP_LICENSE }}
JCSP_LICENSE={{ JCSP_LICENSE }}
COMPANY_NAME={{ COMPANY_NAME }}
```

where

- CSP_LICENSE – CSP license key,
- JCSP_LICENSE – JCSP license key,
- COMPANY_NAME – company name as specified in the JCSP license.

See also

> *External components of the platform*

1.2 Licenses of the Waves Enterprise blockchain platform

The commercial version of the Waves Enterprise blockchain platform is intended for use in the corporate and government sectors and is distributed through user licenses.

Note: The *opensource version* of the Waves Enterprise blockchain platform does not require a license.

The scheme for obtaining a license to use the commercial platform version is as follows:

To access and manage the obtained licenses, the License management service is provided. The specifics of working with it are described in the platform installation manuals:

Deploying a platform with connection to Mainnet

Deployment of the platform in a private network
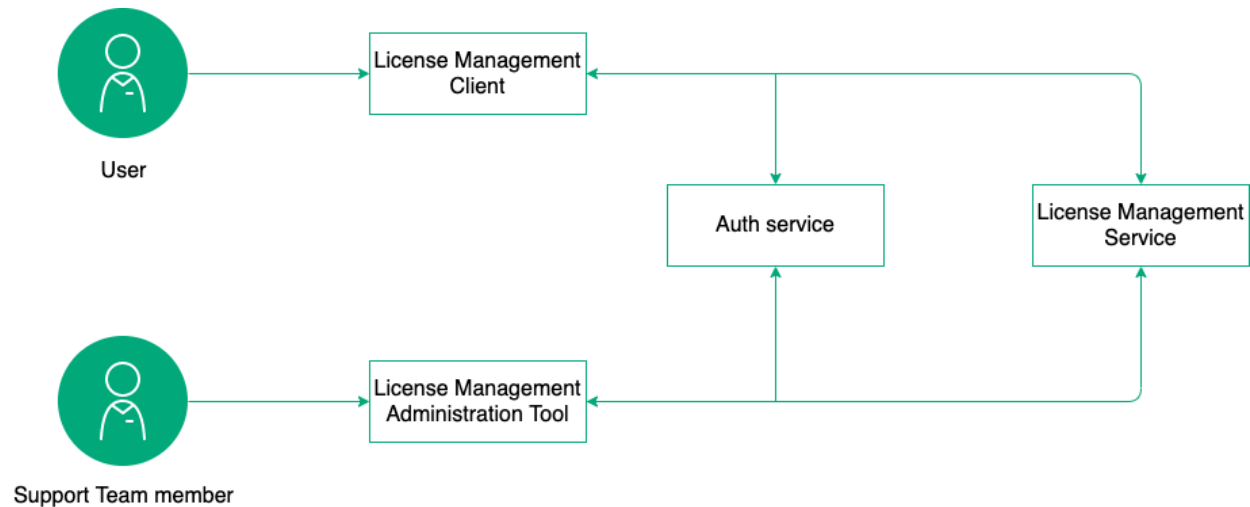
1.2.1 License types

You do not need a license to familiarize yourself with the features of the platform. A detailed description of the functionality of the platform and its installation procedure in the trial mode is given in the article *Deploying the platform in the trial mode (Sandbox)*.

The following types of licenses are available for full use of the platform:

- **Trial License** allows you to get acquainted with the platform and technology during the implementation of the partner's pilot project. It is issued under a contract for the duration of the pilot project, or for the time of development and debugging of the product.

- **Commercial license** allows you to use the platform for commercial projects. It is issued for a period determined by the contractual relationship with the partner.

- **Non-commercial license** allows you to use the platform in the implementation of projects not aimed at generating profit. The license is issued for a period determined by the contractual relationship with the partner.

- **Mainnet license** is a special license that allows you to use the *Waves Enterprise Mainnet blockchain network* to exchange data and perform partner transactions. When working in the Mainnet, there are *fees* for the transactions performed. The license is issued free of charge to anyone who has fulfilled the conditions for the connection. The license is valid for one year. After one year, the node owner should request a new license.

Each type of license applies to one node.

To discuss the number of licenses and nodes on your network and other terms of partnership with Waves Enterprise, contact the Waves Enterprise sales team at sales@wavesenterprise.com.

1.2.2 License usage

After you receive the license file, follow these steps:

- If the node is not running, place the license file in the folder whose path is specified in the `license-file` parameter of the node configuration file.

- If the node is running, copy the content of the license file and pass it to the node using the *POST /licenses/upload* API method.

1.2.3 Duration of licenses

The term of the license is negotiated at the conclusion of the contract.

A trial license standard validity period is 3 months.

A Mainnet license is granted for 1 year. When the one year period expires, the node holder must request a new license.

For other projects, the license is issued for any term as agreed upon.

When the license expires, the covered node loses the ability to form new blocks and send new transactions to the network.

See also

Mainnet fees

1.3 Deploying the platform in the trial mode (Sandbox)

To familiarize yourself with the Waves Enterprise blockchain platform, a free trial version running in a Docker container is available to you. No license is required to install and use it, and the blockchain height is limited to 30,000 blocks. With a block round time of 30 seconds, the full operation time of the platform in trial mode is 10 days.

When you deploy the platform in the trial mode, you get a local version of the blockchain where you can test the basic features:

- signing and sending transactions;

- obtaining data from the blockchain;

- installation and call of smart contracts;

- transfer of confidential data between nodes.

You can interact with the platform through gRPC API and REST API interfaces.

1.3.1 Platform installation

Before you start the installation, make sure you have Docker Engine and Docker Compose installed on your machine. Also, familiarize yourself with the blockchain platform *system requirements*.

Note that you may need administrator rights to run commands on Linux (the `sudo` prefix followed by the administrator password).

1. Create a working directory and place the **docker-compose.yml** file into it. You can download this file from the official Waves Enterprise repository on GitHub with the latest platform release or use the wget utility in the terminal:

```
wget https://raw.githubusercontent.com/waves-enterprise/we-node/release-1.15/node/src/
→docker/docker-compose.yml
```

2. Open the terminal and navigate to the directory containing the downloaded **docker-compose.yml** file. Start the Docker container to deploy the platform:

```
docker run --rm -ti -v $(pwd):/config-manager/output wavesenterprise/config-
→manager:latest
```

Wait for the message informing about the end of the deployment:

```
INFO [launcher] WE network environment is ready!
```

This will create 3 nodes with automatically generated credentials. Information about the nodes is available in the **./credentials.txt** file:

```
node-0
blockchain address: 3Nzi7jJYn1ek6mMvtKbPhehxMQarAz9YQvF
public key:         7cLSA5AnvZgiL8CnoffwxXPkpQhvviJC9eywBKSUsi58
keystore password:  OEtrVSL9gzjO87jYx-gIoQ
keypair password:   JInWk1kauuZDHGXFJ-rNXQ
API key:            we

node-1
blockchain address: 3Nxz6BYyk6CYrqH4Zudu5UYoHU6w7NXbZMs
public key:         VBkFFQmaHzv3YMiWLhh4qsCn4prUvteWsjgiiHEpWEp
keystore password:  FsUp3xiX_NF-bQ9gw6tOsA
keypair password:   Qf2rBgBT9pnozLPOkO1yYw
API key:            we

node-2
blockchain address: 3NtT9onn8VH1DsbioPVBuhU4pnuCtBtbsTr
public key:         8YkDPLsek5VF5bNY9g2dxAthd9AMmmRyvMPTv1H9iEpZ
keystore password:  T77fAroHavbWCS6Uir2oFg
keypair password:   bELB4EU1GDd5rS-RId_6pA
API key:            we
```

3. Run the finished configuration:

```
docker-compose up -d
```

If the nodes run successfully, the following message will be displayed:

```
Creating network "platf_we-network" with driver "bridge"
Creating node-2      ... done
Creating node-0      ... done
Creating node-1      ... done
```

The REST API and gRPC API node interfaces are available at the following addresses:

Node	REST API	gRPC API
node-0	localhost:6862	localhost:6865
node-1	localhost:6872	localhost:6875
node-2	localhost:6882	localhost:6885

4. To stop the running nodes, run the following command:

```
docker-compose down
```

1.3.2 Further actions

Sandbox mode of the platform: fixing issues

1. Error when starting the container for platform deployment:

```
2021-02-07 16:26:59,289 INFO [launcher] ./output/configs/nodes/node-0/accounts.conf
2021-02-07 16:27:07,432 INFO [launcher] ./output/configs/nodes/node-1/accounts.conf
2021-02-07 16:27:19,948 INFO [launcher] ./output/configs/nodes/node-2/accounts.conf
2021-02-07 16:27:28,023 INFO [launcher] Creating blockchain section for the node config␣
→files
Traceback (most recent call last):
  File "launcher.py", line 304, in <module>
    create_new_network()
  File "launcher.py", line 228, in create_new_network
    create_blockchain(addresses, nodes)
  File "launcher.py", line 106, in create_blockchain
    network_participants.append(ConfigFactory.from_dict({"public-key": addresses.get_
→keys()[i],
IndexError: list index out of range
```

Cause: Second start of the container.

Solution: Delete the working directory with the platform files and start over by downloading the *docker-compose.yml* file.

2. Platform startup error after successful deployment:

```
ERROR: for node-1 Cannot create container for service node-1: Conflict. The container␣
→name "/node-1" is already in use by container
→"47cfd7a517e160d201ae969b24392ca0bc2b9720c73e7324dac45daaa24814cb". You have to remove␣
→(or rename) that conCreating node-2 ... error

ERROR: for node-2 Cannot create container for service node-2: Conflict. The container␣
→name "/node-2" is already in use by container "ccd28832f1fb5457186e50d5e5Creating node-
→0 ... error
```

```
tainer to be able to reuse that name.

ERROR: for node-0 Cannot create container for service node-0: Conflict. The conCreating␣
→postgres ... error
eb8ac184f88195f1a560ee8ef7ade5c46f899d". You have to remove (or rename) that container␣
→to be able to reuse that name.

ERROR: for postgres Cannot create container for service postgres: Conflict. The container␣
→name "/postgres" is already in use by container
→"d4bc6d758faafcc9b2bc352b9cbcc5dc909f2959059b7abf17db0088916506d1". You have to remove␣
→(or rename) that container to be able to reuse that name.

ERROR: for node-1 Cannot create container for service node-1: Conflict. The container␣
→name "/node-1" is already in use by container
→"47cfd7a517e160d201ae969b24392ca0bc2b9720c73e7324dac45daaa24814cb". You have to remove␣
→(or rename) that container to be able to reuse that name.

ERROR: for node-2 Cannot create container for service node-2: Conflict. The container␣
→name "/node-2" is already in use by container
→"ccd28832f1fb5457186e50d5e58f98ed3b35c944931589a42a0262a205a17393". You have to remove␣
→(or rename) that container to be able to reuse that name.

ERROR: for node-0 Cannot create container for service node-0: Conflict. The container␣
→name "/node-0" is already in use by container
→"7ed421ac8c8c5ca91a916970c1eb8ac184f88195f1a560ee8ef7ade5c46f899d". You have to remove␣
→(or rename) that container to be able to reuse that name.

ERROR: for postgres Cannot create container for service postgres: Conflict. The container␣
→name "/postgres" is already in use by container
→"d4bc6d758faafcc9b2bc352b9cbcc5dc909f2959059b7abf17db0088916506d1". You have to remove␣
→(or rename) that container to be able to reuse that name.
ERROR: Encountered errors while bringing up the project.
```

Cause: Containers of individual nodes or services are already in use by running containers.

Solution: If you need to rebuild the platform again, stop it with the `docker-compose down` command. Use the command `docker stop [container ID]` to stop running containers of nodes and services. You can enter several running container IDs in a row, separated by a space, or stop all containers with the command `docker stop $(docker ps -a -q)`. Then use the command `docker rm [container ID]` to remove them. The IDs of the containers used are available in error reports like the one above. You can remove multiple containers or all used containers with a single command using a similar syntax.

3. Container startup error:

```
ERROR: for nginx-proxy Cannot start service nginx-proxy: driver failed programming␣
→external connectivity on endpoint nginx-proxy␣
→(86add881e45535e666443cb00e6a6cb66f79a906e412d4f78d2db9d81c6d63d7): Error starting␣
→userland proxy: listen tcp 0.0.0.0:80: bind: address already in use

ERROR: for nginx-proxy Cannot start service nginx-proxy: driver failed programming␣
→external connectivity on endpoint nginx-proxy␣
→(86add881e45535e666443cb00e6a6cb66f79a906e412d4f78d2db9d81c6d63d7): Error starting␣
→userland proxy: listen tcp 0.0.0.0:80: bind: address already in use
```

```
ERROR: Encountered errors while bringing up the project.
```

Cause: The 80:80 port on your machine is occupied by another application.

Solution: Stop the containers with the `docker-compose down` command. Then change the **ports** parameter of the **nginx-proxy** section in the **docker-compose.yml** file, selecting a free port:

```
nginx-proxy:
  image: nginx:latest
  hostname: nginx-proxy
  container_name: nginx-proxy
  ports:
    - "81:80"
```

After that the client and REST API will be available at **127.0.0.1:81** or **localhost:81**. The rest of the services will be available at the addresses with their former ports.

4. Error when navigating to 127.0.0.1 or localhost in Mozilla Firefox:

```
SSL_ERROR_RX_RECORD_TOO_LONG
```

Reason: By default, the localhost is accessed via HTTPS, but SSL is not provided when deploying the platform in the Sandbox mode.

Solution: Enter the full address using HTTP: `http://127.0.0.1` or `http://localhost`.

Hint: For the list of error codes for the Waves Enterprise blockchain platform, see *System errors*.

See also

Deploying the platform in the trial mode (Sandbox)

sandbox-monitoring

System errors

See also

Transactions of the blockchain platform

Smart contracts

Confidential data exchange

gRPC tools

REST API methods

1.4 Deploying a platform with connection to Mainnet

In this platform deployment option, all of your transactions will be sent to the Mainnet, Waves Enterprise's core network. When working with the Mainnet, there are *fees* in WEST for each transaction.

To connect to Mainnet, you only need to install one node.

In case you need to deploy a network of multiple nodes with connection to the Mainnet, contact the technical support service.

A Mainnet license is granted free of charge for 1 year to anyone who meets the conditions for connection. When the one year period expires, the node holder must request a new license.

1.4.1 Account creation, token transfer and confirming transaction

Before deploying the node software, create a WE account using the client. Then perform the following steps:

1. In the client, create a blockchain address using the **Address not selected** button in the upper right corner of the application, or using the **Create address** button in the **Tokens** tab. Don't forget to write down or remember the seed phrase! With its help, you will always be able to restore access to your address in case of losing your credentials. After creating the address, click the **Use address** button.

2. Transfer to the created address an amount in WEST that exceeds the generating balance. To do this, go to the **Tokens** tab of the client and click the **Add tokens via Waves Exchange** button. Copy your blockchain address, and then follow the prompts of the exchange service to purchase WEST.

3. Lease any number of WEST tokens to `3NrKDuHjUG7vSCiMMD259msBKcPRm4MvaJu` and save the identifier for this transaction: it will be used to confirm your balance and ownership of your blockchain address. Since tokens are leased to this address, you will be able to revoke them at any time in the future.

1.4.2 Node deployment

Check out the *system requirements for* the blockchain platform.

After successful transfer of tokens, deploy the node:

1. Create a working directory and place the **docker-compose.yml** file in it. You can download this file from the official Waves Enterprise repository on GitHub with the latest platform release or in the terminal using the wget utility:

```
wget https://raw.githubusercontent.com/waves-enterprise/we-node/release-1.15/node/src/
↪docker/docker-compose.yml
```

2. Download the file `mainnet.conf` file from the official Waves Enterprise GitHub repository, selecting the current version of the platform. Then rename it to `private_network.conf` and place it in the root of the working directory.

3. Deploy your node:

```
docker run --rm -ti -v $(pwd):/config-manager/output/ wavesenterprise/config-
↪manager:latest
```

After deploying the node, all generated addresses and passwords will be stored in the **credentials.txt** file in the working directory.

1.4.3 Node connection to the Mainnet

1. Go to the Waves Enterprise Technical Support site and register.

2. Create a **Participant Connection** application for an entity or individual.

3. Fill in all the required fields of the form, in particular, the public key of the node to be connected. If you plan to mine on Mainnet, check the box **I ask for mining rights**.

4. In the **Confirmation of WEST token ownership** field, enter the ID of the transaction by which you leased the tokens to `3NrKDuHjUG7vSCiMMD259msBKcPRm4MvaJu`.

5. Wait for the application review and confirmation of successful registration, and then start the node whose public key you specified in the connection request:

```
docker-compose up -d node-0
```

After starting the container, *the node REST API* will be available at `http://localhost:6862`. To stop your node, run the `docker-compose down` command.

6. To perform mining and send transactions, transfer **50,000 WEST** or more to the address of the connected node.

Hint: To view the status of your Mainnet license, use the `GET /licenses/status` request to the node.

Mainnet fees

The table below shows the fees that are charged to users for transactions on the Waves Enterprise Mainnet.

Trans-action number	Trans-action name	Fee	Description
1	*Genesis transaction*	no fee	Initial binding of the balance to the addresses of the nodes created at blockchain startup
3	*Issue Transaction*	1 WEST	Token issue. Fee can be paid in WEST only
4	*Transfer Transaction*	0.01 WEST	Token transfer
5	*Reissue Transaction*	1 WEST	Token reissue
6	*Burn Transaction*	0.05 WEST	Token burning
8	*Lease Transaction*	0.01 WEST	Token leasing
9	*Lease Cancel Transaction*	0.01 WEST	Cancelling of token leasing
10	*Create Alias Transaction*	1 WEST	Creation of an address alias
11	*MassTransfer Transaction*	0.05 WEST	Mass transfer of tokens. The minimum fee is specified. The fee amount depends on the number of addresses in a transaction. To find out the exact fee amount, use the *POST /transactions/calculateFee* REST method
12	*Data Transaction*	0.05 WEST per kilo-byte	Transaction with data in the form of fields with a key-value pair. The commission is always charged to the author of the transaction. The minimum fee is specified. The fee amount depends on the data size. To find out the exact fee amount, use the *POST /transactions/calculateFee* REST method
13	*SetScript Transaction*	0.5 WEST	Transaction binding a script with a RIDE contract to an account
14	*Sponsorship Transaction*	1 WEST	Sponsorship setting or cancelling
15	*SetAssetScript*	1 WEST	Transaction binding a script with a RIDE contract to an asset
101	*Genesis Permission Transaction*	no fee	Appointment of a first network administrator for further permission granting
102	*Permission Transaction*	0.01 WEST	Granting/removing permissions on an account
103	*Create-*	1	Creation of a Docker smart contract

See also

GET /licenses

Deploying a platform with connection to Mainnet

Node update in the Mainnet

With each new release of the platform, we recommend that you update the nodes connected to Waves Enterprise Mainnet. All the users whose nodes are running on the Mainnet receive an email notifying of the node version update. If you haven't received such an email, contact the technical support team.

In order to update your node, carry out the following:

1. Download the latest version of the `docker-compose.yml` file from the official Waves Enterprise repository on GitHub selecting the latest release.

2. Place the docker-compose.yml file in the working directory of your node, replacing the old file.

3. If your node is working, stop it:

```
docker-compose down
```

4. After stopping the node, enter the following command:

```
docker-compose up -d node-0
```

The first time you start a node, starting from version 1.4.0, the state migrator will automatically start. The migration is performed automatically and takes a few minutes. If the migration is successful, you will see the `Migration finished successfully` message, and the node will continue to run.

> **Attention:** If you do not use Docker Compose, contact the technical support team for the instructions on how to update the node.

See also

> *Deploying a platform with connection to Mainnet*
>
> *Mainnet: fixing issues*
>
> *Mainnet fees*

Mainnet: fixing issues

When deploying a platform with a connection to Mainnet, it is possible that such errors may occur during the node deployment phase:

```
ERROR: for node-1 Cannot create container for service node-1: Conflict. The container␣
↪name "/node-1" is already in use by container
↪"47cfd7a517e160d201ae969b24392ca0bc2b9720c73e7324dac45daaa24814cb". You have to remove␣
↪(or rename) that conCreating node-2 ... error

ERROR: for node-2 Cannot create container for service node-2: Conflict. The container␣
↪name "/node-2" is already in use by container "ccd28832f1fb5457186e50d5e5Creating node-
```

<div align="right">(continues on next page)</div>

```
↪0 ... error
tainer to be able to reuse that name.

ERROR: for node-0 Cannot create container for service node-0: Conflict. The conCreating␣
↪postgres ... error
eb8ac184f88195f1a560ee8ef7ade5c46f899d". You have to remove (or rename) that container␣
↪to be able to reuse that name.

ERROR: for postgres Cannot create container for service postgres: Conflict. The container␣
↪name "/postgres" is already in use by container
↪"d4bc6d758faafcc9b2bc352b9cbcc5dc909f2959059b7abf17db0088916506d1". You have to remove␣
↪(or rename) that container to be able to reuse that name.

ERROR: for node-1 Cannot create container for service node-1: Conflict. The container␣
↪name "/node-1" is already in use by container
↪"47cfd7a517e160d201ae969b24392ca0bc2b9720c73e7324dac45daaa24814cb". You have to remove␣
↪(or rename) that container to be able to reuse that name.

ERROR: for node-2 Cannot create container for service node-2: Conflict. The container␣
↪name "/node-2" is already in use by container
↪"ccd28832f1fb5457186e50d5e58f98ed3b35c944931589a42a0262a205a17393". You have to remove␣
↪(or rename) that container to be able to reuse that name.

ERROR: for node-0 Cannot create container for service node-0: Conflict. The container␣
↪name "/node-0" is already in use by container
↪"7ed421ac8c8c5ca91a916970c1eb8ac184f88195f1a560ee8ef7ade5c46f899d". You have to remove␣
↪(or rename) that container to be able to reuse that name.

ERROR: for postgres Cannot create container for service postgres: Conflict. The container␣
↪name "/postgres" is already in use by container
↪"d4bc6d758faafcc9b2bc352b9cbcc5dc909f2959059b7abf17db0088916506d1". You have to remove␣
↪(or rename) that container to be able to reuse that name.
ERROR: Encountered errors while bringing up the project.
```

Cause: Containers of individual nodes or services are already in use by running containers.

Solution: Stop the node with the `docker-compose down` command. Use the command `docker stop [container ID]` to stop running containers of nodes and services. You can enter several running container IDs in a row, separated by a space, or stop all containers with the command `docker stop $(docker ps -a -q)`. Then use the command `docker rm [container ID]` to remove them. The IDs of the containers used are available in error reports like the one above. You can remove multiple containers or all used containers with a single command using a similar syntax.

After removing the extraneous containers, turn the platform around again.

See also

Deploying a platform with connection to Mainnet

Node update in the Mainnet

See also

Generators

Licenses of the Waves Enterprise blockchain platform

Contents

- *Deployment of the platform in a private network*
    - *Creation of a node account*
    - *Platform configuration for operation in a private network*
    - *Obtaining a private network license and associated files*
    - *Genesis block signing*
    - *Launching the network*
    - *Attaching the Client application to the private network*

1.5 Deployment of the platform in a private network

If your project or solution requires an independent blockchain, you can deploy your own blockchain network based on the Waves Enterprise platform. Contact the Technical Support Service, and our experts will help you configure the platform delivery to meet the needs of your project.

However, if you need to change any settings or configure the platform by yourself, this section provides a step-by-step guide for deploying and manual configuring the platform for a private network.

Note: The order of creating node accounts, signing the genesis block and starting the network in the *commercial version* of the platform may be different from that described in this section. This procedure is presented in the documentation for the commercial version of the platform. For more information, contact the Waves Enterprise sales team by email: sales@wavesenterprise.com.

1.5.1 Creation of a node account

Create accounts for each node of your future network.

A node account includes an address and a key pair – a public key and a private key.

To generate the keys use the AccountsGeneratorApp utility, which is included in the *generator* package. You can download this package from the official repository of Waves Enterprise on GitHub selecting the platform version you use.

The address and the public key will be shown on the command line during account creation using the **generator** utility. Node's private key is written to the key storage file `keystore.dat`, which is placed in the directory of the node.

To create an account, the `accounts.conf` configuration file is used, which contains the *account generation* parameters. This file is located in the directory of each node.

To create a node account, go to its directory and place the downloaded **generator-x.x.x.jar** (where x.x.x is the number of the blockchain platform release) file into it. Then run it entering the `accounts.conf` file as an argument:

```
java -jar generator-x.x.x.jar AccountsGeneratorApp accounts.conf
```

When you create a key pair, you can make up your own password to protect the node's key pair. Later on, you can use it manually every time you start your node, or you can set global variables to ask for the password at system startup. See the description of the *account generator* for more information on how to use the password for a node key pair.

If you do not want to use a password to protect the key pair, press the `Enter` key, leaving the field blank.

The following messages will be displayed as a result of the utility operation:

```
2021-02-09 16:03:18,940 INFO  [main] c.w.g.AccountsGeneratorApp$ - 1 Address:␣
→3Nu7MwQ1eSmDVwBzrN1nyzR8wqb2yzdUcyN; public key:␣
→F4ytnnS6H72ypCEpgNKYftGotpdX83ZxtWRX2dyGzDiA
2021-02-09 16:03:18,942 INFO  [main] c.w.g.AccountsGeneratorApp$ - Generator done
```

A `keystore.dat` file will be created in the directory of the node, which contains the account's public key.

1.5.2 Platform configuration for operation in a private network

Following files are used for configuration of the platform:

- The `node.conf` is the main configuration file of a node, which defines its operating principles and a set of options.
- The `api-key-hash.conf` is a configuration file for generating `api-key-hash` and `privacy-api-key-hash` field values; it is used to configure node authorization when authorization by `api-key` hash method is selected. The guidelines for working with this configuration file will be given when configuring the authorization method of the node.

Note: You can setup node configuration parameters in a single file or in several files, including one file into another, for example:

```
include required(file("network.conf"))
include required(file("local.conf"))
```

Put the parameters common for all nodes in one file and set unique node parameters (such as `owner-address`) in a separate file for each node.

Below is a step-by-step guide on how to manually configure a single node to work on a private network. If you have multiple nodes deployed on your network, you will need to perform similar configuration steps for each of them.

Step 1. General configuration of the platform

This step configures cryptography, consensus, Docker smart contract execution and mining. All the parameters required for this are located in the **node.conf** file.

Platform installation and usage

General platform configuration: cryptography

The type and parameters of the cryptographic algorithm used in the blockchain are set in the `crypto` section of the node configuration file. The `crypto` section is used to initialize the cryptography before reading the complete node configuration file.

```
crypto {
  type = WAVES
}
```

- **type** – *cryptography* type; use the `WAVES` value for Waves cryptography algorithms. If the `waves-crypto` parameter is present in the configuration file and is set to `yes`, then the `type` parameter is assigned the `WAVES` value;

Note: The `node.waves-crypto` field with `yes` and `no` values is still supported, but it is not planned to use it in the platform future versions. Instead, the `type` field in the `crypto` section will be used.

See also

> *Deployment of the platform in a private network*
>
> *Cryptography*

Installation and usage of the platform

General platform configuration: consensus algorithm

The Waves Enterprise blockchain platform supports three consensus algorithms – **PoS**, **PoA** and **CFT**. Detailed information about the consensus algorithms used can be found in the *Consensus algorithms* article.

The consensus settings are located in the `consensus` block of the `blockchain` section:

```
consensus {
    type = ""
    ...
}
```

Select the preferred consensus type in the `type` field. Available values: `pos`, `poa`, and `cft`.

type = "pos" or the commented consensus block

If you do not select a consensus type in this field, leaving it blank, the default **PoS** algorithm will be used. This option is equivalent to selecting the `pos` value. In this case, other fields in the `consensus` block are not required, you only need to configure the PoS mining operation in the `genesis` block:

```
consensus {
    type = "pos"
}

...

genesis {
    average-block-delay = "60s"
    initial-base-target = 153722867
    initial-balance = "16250000 WEST"


    ...
}
```

Note: When you use the **PoS** algorithm (`consensus.type = pos`) and some other fields' values are specified in the `consensus` section, they are ignored. For example

```
consensus {
    type = "pos"
    round-duration = 5500ms
}
```

The `round-duration` field value will not be taken into account.

The following parameters of the `genesis` block in the `blockchain` section are responsible for mining with PoS:

- `average-block-delay` – average block creation delay. The default value is **60 seconds**.

- `initial-base-target` – the initial base number to regulate the mining process. The higher the value, the more often the blocks are created. Also, the value of the miner balance affects the use of this parameter in mining – the higher the balance of the miner, the lower the value of `initial-base-target` becomes when calculating the queue of node-miner in the current round.

- `initial-balance` – the initial balance of the network. The greater the share of the miner's balance from the initial balance of the network, the smaller the value of `initial-base-target` becomes for determining the miner node of the current round.

type = "poa"

To configure the PoA consensus algorithm, add the following parameters to the `consensus` block:

```
consensus {
    type = "poa"
    round-duration = "17s"
    sync-duration = "3s"
    ban-duration-blocks = 100
    warnings-for-ban = 3
    max-bans-percentage = 40
}
```

- `round-duration` – length of the block mining round in seconds.
- `sync-duration` – the block mining synchronization period in seconds. The total round time is the sum of `round-duration` and `sync-duration`.
- `ban-duration-blocks` – the number of blocks for which the miner node is banned.
- `warnings-for-ban` – the number of rounds during which the miner node receives warnings. At the end of this number of rounds, the node is banned.
- `max-bans-percentage` – percentage of miner node from the total number of nodes in the network that can be banned.

type = "cft"

The basic settings of the CFT consensus algorithm are identical to those of the PoA consensus algorithm:

```
consensus {
    type: cft
    warnings-for-ban: 3
    ban-duration-blocks: 15
    max-bans-percentage: 33
    round-duration: 7s
    sync-duration: 2s
    max-validators: 7
    finalization-timeout: 4s
    full-vote-set-timeout: 4s
}
```

In comparison with the PoA, the CFT has the following additional configuration parameters needed to validate blocks in a voting round:

- **max-validators** – limit of validators participating in a current round.
- **finalization-timeout** – time period, during which a miner waits for finalization of the last block in a blockchain. After that time, the miner will return the transactions back to the UTX pool and start mining the round again.
- **full-vote-set-timeout** – optional parameter which defines, how much time a miner will wait for the full set of votes from all validators after the end of the round (node configuration file parameter: `round-duration`).

While configuring CFT, please note the following recommendations:

- The `sync-duration` parameter must be different from zero. It is recommended to set the value **from 1 to 5 seconds**, depending on the size and complexity of transactions.

- Approximate calculation of the `finalization-timeout` parameter: (`round-duration` + `sync-duration`) / 2. It is not recommended to underestimate this value to speed up finalization: if the miner gathers the necessary number of votes before the end of this time, it will immediately release the finalizing microblock.

- If there is a large number of miners in the network, limit the number of round validators by the `max-validators` parameter. The validator selection mechanism will ensure that all validators rotate evenly across rounds. Too many validators can adversely affect network performance. The recommended range of values is: **from 5 to 10**.

- If the network is running under constant load, set the `full-vote-set-timeout` parameter. Until this timeout expires, the miner waits for a full set of votes from the validators. If the validator encounters any problem, the network uses the `full-vote-set-timeout` to create an additional time slot that allows the lagging validator to complete synchronization. The recommended value is `sync-duration` * 2, it should not exceed `sync-duration` + `finalization-timeout`.

See also

Consensus algorithms

Deployment of the platform in a private network

General platform configuration: mining

General platform configuration: execution of smart contracts

Installation and usage of the platform

General platform configuration: execution of smart contracts

A number of parameters for working with smart contracts are configured in the following two sections of the node configuration file:

- `node.docker-engine` – Docker smart contracts execution parameters,

- `node.wasm` – WASM smart contracts execution parameters.

Docker smart contracts execution configuration

To work with *Docker smart contracts*, a node uses two connection types, for each of which you can configure TLS:

1. The connection to the docker host, the remote machine on which the smart contracts run. This machine uses a docker library that accesses the socket using its protocols. You can enable the secure connection option for it. Such a connection is referred to as "docker-TLS" in this documentation. The docker-TLS connection is configured in the `node.docker-engine.docker-tls` section of the node configuration file; this setting is described below in this section;

2. The connection the running smart contract opens towards the node using gRPC protocol. This is an API connection as the connection point of the smart contract to the node is the same as for any other user or application. This API is configured in the `node.api.grpc` section. For instance, you can *enable or disable TLS* for it. You can find an example of such a configuration in the *Examples of node configuration files* section.

Note: The TLS protocol is not available in the *opensource* version of the platform.

If you are going to develop and execute Docker smart contracts in your blockchain, set their execution parameters in the `docker-engine` section of the node configuration file:

```
docker-engine {
  enable = yes
  use-node-docker-host = yes
  # docker-host = "unix:///var/run/docker.sock"
  execution-limits {
    startup-timeout = 10s
    timeout = 10s
    memory = 512
    memory-swap = 0
    }
  reuse-containers = yes
  remove-container-after = 10m
  allow-net-access = yes
  remote-registries = [
    {
     domain = "myregistry.com:5000"
     username = "user"
     password = "password"
    }
  ]
  check-registry-auth-on-startup = no
  # default-registry-domain = "registry.wavesenterprise.com"
  contract-execution-messages-cache {
    expire-after = 60m
    max-buffer-size = 10
    max-buffer-time = 100ms
    utx-cleanup-interval = 1m
    contract-error-quorum = 2
  }
  contract-auth-expires-in = 1m
  grpc-server {
    # host = "192.168.97.3"
    port = 6865
    }
  remove-container-on-fail = yes
  docker-tls {
    tls-verify = yes
    cert-path = "/node/certificates"
    }
  contracts-parallelism = 8
}
```

- `enable` – enable transaction processing for Docker contracts.

- `use-node-docker-host` – set the parameter to `yes` to define the IP address of the gRPC API available to the contracts. This will read the IP address from the `/etc/hosts` file inside the node container. Also, in order for the contracts to access the node, their containers will be connected to the same docker network in which the node container was created.

- `docker-host` – Docker daemon address (optional). If this field is commented out, the address of the daemon will be taken from the system environment.

- `startup-timeout` – time taken to create the contract container and register it in the node (in seconds).

- `timeout` – the time allocated for contract execution; the field specifies the value of the parameter and units of measurement.

- `memory` – memory limit for the contract container (in megabytes).

- `memory-swap` – allocated amount of virtual memory for the contract container (in megabytes).

- `reuse-containers` – using one container for several contracts when using the same Docker image. To enable this option, specify `yes`, to disable - `no`.

- `remove-container-after` – the time interval of container inactivity, after which it will be removed.

- `allow-net-access` – permission to access the network.

- `remote-registries` – Docker registry addresses and authorization settings.

- `check-registry-auth-on-startup` – check authorization for Docker registries at node startup. To enable this option, specify `yes`, to disable - `no`.

- `default-registry-domain` – default Docker registry address (optional). This parameter is used if no repository is specified in the contract image name.

- `contract-execution-messages-cache` – settings section of the cache with the execution statuses of Docker contracts transactions;

- `expire-after` – time to store the status of the smart contract.

- `max-buffer-size` and `max-buffer-time` – settings for size and time of the status cache.

- `utx-cleanup-interval` – when the specified interval elapses, invalid transactions (with Error status) are removed from the UTX pool of a non-miner node. 1m is used by default.

- `contract-error-quorum` – the minimum number of transaction Error (business error) statuses received from different miner-nodes, after which the smart-contract call transaction is removed from the UTX pool of a non-miner node. 2 is used by default.

- `contract-auth-expires-in` – lifetime of the authorization token used by smart contracts for calls to the node.

- `grpc-server` – gRPC server settings section for Docker contracts with the gRPC API.

- `host` – network address of the node (optional).

- `port` – port of the gRPC server. Specify the listening port for gRPC requests used by the platform.

- `remove-container-on-fail` – removes the container if an error occurred during its startup. To enable this option, specify `yes`, to disable – `no`.

- `tls-verify` – enable or disable TLS; if you specify `yes`, certificates are searched for in the directory specified in the `certs-path` parameter; if you specify `no`, certificates are not searched for.

- `certs-path` – the path to the directory where TLS certificates are stored. `{node.directory}/certificates` is used by default.

- `contracts-parallelism` – the parameter determines the number of *parallel transactions of all containerized smart contracts*. The default value is 8.

WASM smart contracts execution configuration

If you are going to develop and execute *WASM smart contracts* in your blockchain, set their execution parameters in the `node.wasm` section of the node configuration file:

```
wasm
  {
    fuel-limit = 20000
  }
```

- `fuel-limit` – limit on the number of contract execution conventional units (similar to gas for Ethereum). One bytecode instruction corresponds to one unit of fuel. Use the parameter to limit the time of a WASM smart contract transaction execution by a virtual machine, thus excluding cases where the contract could be performed indefinitely.

See also

Precise platform configuration: TLS

Deployment of the platform in a private network

Development and usage of smart contracts

General platform configuration: consensus algorithm

General platform configuration: mining

Smart contracts

Installation and usage of the platform

General platform configuration: mining

The blockchain mining parameters are set in the `miner` section of the node configuration file:

```
miner {
  enable = yes
  quorum = 2
  interval-after-last-block-then-generation-is-allowed = 10d
  no-quorum-mining-delay = 5s
  micro-block-interval = 5s
  min-micro-block-age = 3s
  max-transactions-in-micro-block = 500
  max-block-size-in-bytes = 1048576
  min-micro-block-age = 6 s
  minimal-block-generation-offset = 200ms
  pullin-buffer-size = 100
  utx-check-delay = 1s
}
```

- `enable` – activation of the mining option. Enable – `yes`, disable – `no`.

- `quorum` – required number of miner nodes to create a block. A value of 0 will generate blocks offline and is used only for test purposes in networks with one node. When specifying this value, take into account that your own miner node does not sum up with the value of this parameter, i.e. if you specify `quorum = 2`, then you need at least **3** miner nodes for mining.

- `interval-after-last-block-then-generation-is-allowed` – enable block generation only if the last block is not older than the specified time period (in days).

- `micro-block-interval` – an interval between microblocks (in seconds).

- `min-micro-block-age` – the minimum age of the microblock (in seconds).

- `max-transactions-in-micro-block` – the maximum number of transactions in the microblock.

- `minimal-block-generation-offset` – the minimum time interval between blocks (in milliseconds).

- `pulling-buffer-size` – size of transactions buffer. The higher the value of the parameter, the longer the transactions group.

- `utx-check-delay` – UTX pool inspection delay. The miner periodically inspects the pool to make sure if it is empty or not. 1 s is used as the default value. The parameter value must be equal to or more than 100 ms.

The mining settings depend on the planned size of transactions on your network.

Mining settings and consensus algorithm

Also, blockchain mining is closely related to the chosen consensus algorithm. The following parameters of the `miner` section must be taken into account when configuring the consensus parameters:

- `micro-block-interval` – an interval between microblocks (in seconds).

- `min-micro-block-age` – the minimum age of a microblock. The value is specified in seconds and must not exceed the value of `micro-block-interval`.

- `minimal-block-generation-offset` – a minimal time interval between blocks (in milliseconds).

The values of microblock creation parameters must not exceed or otherwise conflict with the values of `average-block-delay` for **PoS** and `round-duration` for **PoA** and **CFT**. The number of microblocks in a block is not limited but depends on the size of the transactions included in the microblock.

UTX settings

The unconfirmed transactions pool (UTX) has a rebroadcasting mechanism, which allows the network to recover faster if any failures occur – for example, if network connectivity between nodes is lost. In such cases, transactions sent to a single node may not be broadcasted. The rebroadcasting mechanism solves such problems by periodically checking the relevance of the transactions in a node UTX.

This mechanism checks all the transactions in the UTX once in a period of time set in the `interval` parameter; it then resends to its peers those transaction whose creation date differs from the current date by more than the period set in the `threshold` parameter.

The UTX parameters are set in the `utx` section of the node configuration file:

```
utx {
   memory-limit=100Mb
   rebroadcast-threshold=5m
   rebroadcast-interval=5m
   }
```

- `memory-limit` – the maximum UTX pool size; when calculating the UTX pool size, only the serialized form, not the total size of transactions in memory, is taken into account;

- **rebroadcast-threshold** – after the transaction is created, when the time period specified in the parameter elapses, the transaction is considered "old" and must be rebroadcast; the default value of the parameter is **5m**;

- **rebroadcast-interval** – task interval for re-broadcasting the "old" transactions; the default value of the parameter is **5m**.

See also

    Deployment of the platform in a private network

    General platform configuration: consensus algorithm

    General platform configuration: execution of smart contracts

    Waves-NG blockchain protocol

Step 2. Precise platform configuration

This step configures the node's gRPC and REST API tools, their authorization, confidential data access groups, etc. You may need these settings if you change the pre-set settings for your hardware or software configuration.

All necessary parameters are also located in the **node.conf** node configuration file. The **api-key-hash.conf** file is also used to configure authorization, which is necessary when selecting the authorization method by a given *api-key* string hash.

Precise platform configuration: gRPC and REST API authorization

Authorization is required to provide access to the *gRPC* and *REST* node API tools.

The Waves Enterprise blockchain platform supports two types of authorization for gRPC and REST APIs:

- **api-key** string hash authorization;

- JWT token (**oAuth 2**) authorization.

> **Attention:** Authorization by **api-key** hash is a simple means of accessing a node, but the security level of this authorization method is relatively low. An attacker may gain access to a node if the **api-key** string is exposed. If you want to improve security on your network, we recommend using JWT token authentication via *authorization service*.

The **auth** section of the node configuration file is used to configure authorization.

> **Hint:** The REST and gRPC API interfaces use the same **api-key** for authorization by key string and **public-key** for authorization by JWT-token.

type = "api-key"

Authorization by hash of the key string `api-key` is used in the default node. When selecting the authorization method by hash of the key string `api-key` the `auth` section contains the following parameters:

```
auth {
  type = "api-key"

  # Hash of API key string
  api-key-hash = "G3PZAsY6EA8esgpKxB2UYTQJZJPzc14gLnNbm2xvcDf6"

  # Hash of API key string for PrivacyApi routes
  privacy-api-key-hash = "G3PZAsY6EA8esgpKxB2UYTQJZJPzc14gLnNbm2xvcDf6"

  # Hash of API key string for Confidential Smart Contracts API
  confidential-contracts-api-key-hash = "G3PZAsY6EA8esgpKxB2UYTQJZJPzc14gLnNbm2xvcDf6"
}
```

- `api-key-hash` − hash from the REST API access key string;
- `privacy-api-key-hash` − the hash from the access key string to the *REST methods for handling confidential data and obtaining information about access groups (privacy)* and similar *gRPC methods*;
- `confidential-contracts-api-key-hash` − the hash from the access key string to the *REST methods of handling confidential smart contracts* and similar *gRPC methods*.

To fill in these parameters you will need the `ApiKeyHash` utility from the `generator-x.x.x.jar` package, which you can download from the official Waves Enterprise repository on GitHub, selecting the platform version you use.

Place this file in the root folder of the platform and also create a file `api-key-hash.conf`:

```
apikeyhash-generator {
  crypto {
    type = WAVES
  }
  api-key = "some string for api-key"
  file = ${user.home}"/apikeyhash.out"
}
```

In this file, enter the string that you want to hash and use for authorization in the `api-key` parameter.

You can use the ''file'' parameter to specify the name of the file to which the hash will be saved. The parameter is optional. If it is not specified, the hash is output to the console.

Note: The `waves-crypto` field with `yes` and `no` values is still supported, but it will be deprecated in the platform future versions. Instead, use the `type` field in the `crypto` section.

Enter the prepared `api-key-hash.conf` file as an argument when you run the `ApiKeyHash` utility from the `generator` package:

```
java -jar generator-x.x.x.jar ApiKeyHash api-key-hash.conf
```

Output example:

```
Api key: some string for api-key
Api key hash: G3PZAsY6EA8esgpKxB2UYTQJZJPzc14gLnNbm2xvcDf6

2021-02-11 16:31:21,586 INFO  [main] c.w.g.ApiKeyHashGenerator$ - Generator done
```

Specify the resulting `Api key hash` value in the `api-key-hash`, `privacy-api-key-hash` and `confidential-contracts-api-key-hash` parameters in the `auth` section of the node configuration file as indicated above.

type = "oauth2"

When you select authorization by JWT-token, the `auth` section of the node configuration file looks as follows:

```
auth {
  type: "oauth2"
  public-key: "AuthorizationServicePublicKeyInBase64"
}
```

The public key for oAuth is generated during the initial deployment of the node. The key is located in the `./auth-service-keys/jwtRS256.key.pub` file.

Copy the line between `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` and paste it as the `public-key` parameter in the `auth` section of the node configuration file.

OAuth2 authorization roles

A number of *REST API methods* and *gRPC API methods* can only be invoked by users with specific authorization roles.

When registering a new user in *Waves Enterprise Client* the user is assigned the *user* role. Further the administrator of the *authorization service* can change the list of roles assigned to the user.

The user role is encrypted in the JWT token.

The tables below show the methods and the roles required to invoke them that are used in the Waves Enterprise Mainnet blockchain network.

The list of REST methods and the roles that have access to them in Mainnet

REST methods group	REST method	No role	user	admin	privacy	C…
activation			*	*	*	*
addresses			*	*	*	*
alias			*	*	*	*
anchoring			*	*	*	*
assets			*	*	*	*
blocks			*	*	*	*
consensus			*	*	*	*
	/consensus/algo is not available for user role			*		
contracts			*	*	*	*
confidential-contracts				*		*
crypto			*	*	*	*

Table 1 – continued from previous page

REST methods group	REST method	No role	user	admin	privacy	C...
debug						
	/debug/validate		*	*	*	*
	/debug/blocks/{howMany}			*		
	/debug/cleanState			*		
	/debug/configInfo			*		
	/debug/createGrpcAuth			*		
	/debug/freeze			*		
	/debug/historyInfo			*		
	/debug/info			*		
	/debug/minerInfo			*		
	/debug/portfolios/{address}			*		
	/debug/print			*		
	/debug/rollback			*		
	/debug/rollback-to/{signature}			*		
	/debug/state			*		
	/debug/stateWE/{height}			*		
	/debug/threadDump			*		
	/debug/utx-rebroadcast			*		
leasing			*	*	*	*
node						
	/node/status	*	*	*	*	*
	/node/version	*	*	*	*	*
	/node/healthcheck	*	*	*	*	*
	/node/owner		*	*	*	*
	/node/config		*	*	*	*
	get /node/logging		*	*	*	*
	get /node/metrics		*	*	*	*
	/node/stop			*		
	post /node/logging			*		
	post /node/metrics			*		
peers						
	/peers/all		*	*	*	*
	/peers/connected		*	*	*	*
	/peers/suspended		*	*	*	*
	/peers/allowedNodes			*		
	/peers/connect			*		
	/peers/hostname/{address}			*		
permissions			*	*	*	*
privacy						
	/privacy/{policyId}/recipients		*	*	*	*
	/privacy/{policyId}/owners		*	*	*	*
	/privacy/{policyId}/hashes		*	*	*	*
	/privacy/{policyId}/transactions		*	*	*	*
	/privacy/{policyId}/getData/{policyItemHash}				*	
	/privacy/{policyId}/getLargeData/{policyItemHash}				*	
	/privacy/{policyId}/getInfo/{policyItemHash}				*	
	/privacy/getInfos				*	
	/privacy/sendData				*	
	/privacy/sendDataV2				*	
	/privacy/sendLargeData				*	

REST methods group	REST method	No role	user	admin	privacy	Co
	/privacy/forceSync				*	
transactions			*	*	*	*
snapshot						
	/snapshot/status		*	*	*	*
	/snapshot/genesisConfig		*	*	*	*
	/snapshot/swapState			*		
utils			*	*	*	*

The list of gRPC methods and the roles that have access to them in Mainnet

gRPC service	gRPC method	No role	user	ad-min	pri-vacy	ConfidentialCon-tractUser
TransactionPublic-Service						
	grpc-tx		*	*	*	*
	UtxInfo	*	*	*	*	*
	TransactionInfo		*	*	*	*
Blockchain-EventsService						
	SubscribeOn		*	*	*	*
PrivacyEventsSer-vice						
	SubscribeOn		*	*	*	*
PrivacyPublicSer-vice						
	GetPolicyItemData				*	
	GetPolicyItemInfo				*	
	PolicyItemDataEx-ists				*	
	SendData				*	
ContractStatusSer-vice						
	ContractExecution-Statuses		*	*	*	*
	ContractsExecution-Events		*	*	*	*
NodeInfoService						
	NodeConfig	*	*	*	*	*
ContractPublicSer-vice						
	ConfidentialCall				*	

See also

> *Deployment of the platform in a private network*
>
> *Precise platform configuration: node gRPC and REST API configuration*
>
> *Precise platform configuration: confidential data groups configuration*
>
> *Precise platform configuration: TLS*
>
> *PrivacyEventsService and PrivacyPublicService methods authorization*
>
> **Privacy group methods authorization**
>
> *Authorization service*
>
> *Authorization service: authorization variants*

Precise platform configuration: node gRPC and REST API configuration

The gRPC and REST API parameters for each node are in the `api` section of the configuration file:

```
api {
  rest {
    # Enable/disable REST API
    enable = yes

    # Network address to bind to
    bind-address = "0.0.0.0"

    # Port to listen to REST API requests
    port = 6862

    # Enable/disable TLS for REST
    tls = no

    # Enable/disable CORS support
    cors = yes

    # Max number of transactions
    # returned by /transactions/address/{address}/limit/{limit}
    transactions-by-address-limit = 10000

    distribution-address-limit = 1000
  }

  grpc {
    # Enable/disable gRPC API
    enable = yes

    # Network address to bind to
    bind-address = "0.0.0.0"

    # Port to listen to gRPC API requests
    port = 6865
```

```
    # Enable/disable TLS for GRPC
    tls = no

    # Parameters for internal gRPC services. Recommended to be left as is.
    services {
            blockchain-events {
              max-connections = 5
              history-events-buffer {
                enable: false
                size-in-bytes: 50MB
              }
            }

            privacy-events {
              max-connections = 5
              history-events-buffer {
                enable: false
                size-in-bytes: 50MB
              }
            }

            contract-status-events {
              max-connections = 5
            }
          }
    }
```

rest – ″ block

The `rest { }` block is used for setting of the REST API interface. It includes following parameters:

- `enable` – activation of the node REST API. Enabling – `yes`, disabling – `no`.

- `bind-address` – network address of the node where the REST API interface will be available.

- `port` – the listening port of the REST API requests.

- `tls` – enable or disable TLS for REST API requests. Specify `yes` to enable the option , or ``no``enable to disable it. This option requires *the node TLS setup*.

Note: The TLS protocol is not available in the *opensource* version of the platform.

- `cors` – support of cross-domain requests to REST API. Enable – `yes`, disable – `no`.

- `transactions-by-address-limit` – maximum number of transactions returned by the `GET / transactions/address/{address}/limit/{limit}` method.

- `distribution-address-limit` – the maximum number of addresses specified in the `limit` field and returned by the `GET /assets/{assetId}/distribution/{height}/limit/{limit}` method.

grpc − ″ block

The `grpc { }` block is used to configure the gRPC toolkit of a node. It includes the following parameters:

- `enable` − activation of the node gRPC interface.
- `bind-address` − the network address of the node where the gRPC interface will be available.
- `port` − the listening port of the gRPC requests.
- `tls` − enable or disable TLS for gRPC requests. Specify `yes` to enable the option , or `no` to disable it. This option requires *the node TLS setup*.

Note: The TLS protocol is not available in the *opensource* version of the platform.

The `services{ }` section contains parameters of public gRPC services that collect data from the platform components:

- `blockchain-events` − service for collecting data on events in the blockchain network;
- `privacy-events` − service for collecting data on events related to privacy groups;
- `contract-status-events` − service for collecting data on statuses of smart contracts.

In this section, we recommend to use the default parameters mentioned in the example.

See also

> *Deployment of the platform in a private network*
>
> *Precise platform configuration: gRPC and REST API authorization*
>
> *Precise platform configuration: confidential data groups configuration*
>
> *Precise platform configuration: TLS*

Precise platform configuration: TLS

To work with smart contracts, the node uses two connection types, for each of which you can configure TLS: *docker-TLS and API connection*.

Note: The TLS protocol is not available in the *opensource* version of the platform.

You can configure TLS for gRPC and REST API for each node using the gRPC and REST API operation parameters in the `api` section of the node configuration file. To configure TLS, use the `TLS` parameter in the *rest block* and in the *grpc block*.

To work with TLS for API:

1. *enable TLS in the node.api.grpc section of the node configuration file*;
2. obtain TLS artefacts:
   - obtain keystore file named `we.jks`;
   - issue `we.cert` client certificate;

- import the client certificate into the trusted certificates storage.

An example of the preparation of these artifacts is given in the following section:

Example of how to prepare artefacts for TLS

If you plan to *use TLS*, you must configure the TLS settings as part of the infrastructure setup.

To work with TLS for API you need to get a keystore file. Here is an example of using the standard **keytool** utility for this purpose:

```
keytool \
-keystore we.jks -storepass 123456 -keypass 123456 \
-genkey -alias we -keyalg RSA -validity 9999 \
-dname "CN=Waves Enterprise,OU=security,O=WE,C=RU" \
-ext "SAN=DNS:welocal.dev,DNS:localhost,IP:51.210.211.61,IP:127.0.0.1"
```

- `keystore` – keystore file name;
- `storepass` – keystore password, which should be stated in the `keystore-password` section of the node configuration file;
- `keypass` – private key password, which should be stated in the `private-key-password` section of the config file;
- `alias` – an alias name (upon a user decision);
- `keyalg` – keypair generation algorithm;
- `validity` – keypair validity time in days;
- `dname` – distinguished name according to the X.500 standard, connected with the keystore alias;
- `ext` – extensions that are used for key generation, all possible host names and IP addresses should be stated for work in different networks.

As a result of the keytool utility execution, the **we.jks** keystore file will be obtained. In order to connect to the node operating with the TLS, a user should also generate a client certificate:

```
keytool -export -keystore we.jks -alias we -file we.cert
```

The obtained certificate file `we.cert` should be imported into the trusted certificate storage. If a node is located in the same network with a user, it will be enough to state a relative path to the `we.jks` file in the node config file, as demonstrated above.

In case the node is located in another network, the `we.cert` certificate file should be imported into the keystore:

```
keytool -importcert -alias we -file we.cert -keystore we.jks
```

See also

> *Precise platform configuration: TLS*
>
> *Deployment of the platform in a private network*
>
> *Precise platform configuration: gRPC and REST API authorization*
>
> *Precise platform configuration: node gRPC and REST API configuration*
>
> *Precise platform configuration: confidential data groups configuration*

3. specify the relative path to the `we.jks` keystore file in the `tls` section of the node configuration file.

You will need the **keytool** utility included in the Java SDK or JRE to configure TLS.

tls section of the node configuration file

The `tls` section contains the following parameters:

```
tls {
  type = EMBEDDED
  keystore-path = ${node.directory}"/we_tls.jks"
  keystore-password = ${TLS_KEYSTORE_PASSWORD}
  private-key-password = ${TLS_PRIVATE_KEY_PASSWORD}
}
```

- `type` – TLS mode. Possible options:
  - `DISABLED` – disabled, in this case other options should be excluded or commented out and
  - `EMBEDDED` – enabled, the certificate is signed by a node provider and packed within a JKS file (keystore); the certificate directory and keystore access parameters should be stated by a user in the fields below.
- `keystore-path` – keystore relative path within the node directory: `${node.directory}"/we_tls.jks"`.
- `keystore-password` – password for the node keystore. Specify the password you set earlier with the **storepass** flag for the **keytool** utility.
- `private-key-password` – password for the private key. Specify the password you set earlier with the **keypass** flag for the **keytool** utility.

See also

> *Deployment of the platform in a private network*
>
> *Example of how to prepare artefacts for TLS*
>
> *Precise platform configuration: gRPC and REST API authorization*
>
> *Precise platform configuration: node gRPC and REST API configuration*
>
> *Precise platform configuration: confidential data groups configuration*

Precise platform configuration: confidential data groups configuration

If you use **privacy** API-methods to manage *confidential data*, configure the data access parameters in the node configuration file `privacy` section.

gRPC API methods of the **privacy** group are described in the *gRPC: handling confidential data* section. REST API methods of the **privacy** group are described in the *REST API: confidential data exchange and obtaining of information about confidential data groups* section.

Important: **privacy** group API-methods can be used only in PKI test mode, i.e. when the *node.crypto.pki.mode* node configuration file parameter is set to `TEST`, or when PKI is disabled (`node.crypto.pki.mode = OFF`).

Below is an example of configuration using the PostgreSQL database:

Example with the PostgreSQL database used

```
privacy {

  replier {
    parallelism = 10
    stream-timeout = 1 minute
    stream-chunk-size = 1MiB
  }

  synchronizer {
    request-timeout = 2 minute
    init-retry-delay = 5 seconds
    inventory-stream-timeout = 15 seconds
    inventory-request-delay = 3 seconds
    inventory-timestamp-threshold = 10 minutes
    crawling-parallelism = 100
    max-attempt-count = 24
    lost-data-processing-delay = 10 minutes
    network-stream-buffer-size = 10
  }

  inventory-handler {
    max-buffer-time = 500ms
    max-buffer-size = 100
    max-cache-size = 100000
    expiration-time = 5m
    replier-parallelism = 10
  }

  cache {
    max-size = 100
    expire-after = 10m
  }

  storage {
    vendor = postgres
```

(continued from previous page)

```
    schema = "public"
    migration-dir = "db/migration"
    profile = "slick.jdbc.PostgresProfile$"
    upload-chunk-size = 1MiB
    jdbc-config {
      url = "jdbc:postgresql://postgres:5432/node-1"
      driver = "org.postgresql.Driver"
      user = postgres
      password = wenterprise
      connectionPool = HikariCP
      connectionTimeout = 5000
      connectionTestQuery = "SELECT 1"
      queueSize = 10000
      numThreads = 20
    }
  }

  service {
    request-buffer-size = 10MiB
    meta-data-accumulation-timeout = 3s
  }
}
```

Choosing the database

Before changing the node configuration file, decide on the database that you plan to use to store confidential data. The Waves Enterprise blockchain platform supports interaction with PostgreSQL database or Amazon S3.

PostgreSQL

During the installation of a database running under PostgreSQL, you will create an account to access the database. The username and password you set for this account must then be specified in the node configuration file (in the `user` and `password` fields of the `storage` block of the `privacy` section, see the *vendor = postgres* section for details).

To use PostgreSQL DBMS, you will need to install the JDBC interface (Java DataBase Connectivity). When installing JDBC, set the profile name. This name must then be specified in the node configuration file (in the `profile` field of the `storage` block of the `privacy` section, see the *vendor = postgres* section for details).

For optimization purposes, connection to PostgreSQL can be done through the pgBouncer tool. In this case, pgBouncer requires special configuration, which is described below in the *storage-pgBouncer* section.

Amazon S3

When using Amazon S3, the information must be stored on the Minio server. During the Minio server installation, you will be prompted for a login and password to access the data. These login and password must then be specified in the node configuration file (in the `access-key-id` and `secret-access-key` fields, see *vendor = s3* section for details).

After installing the DBMS appropriate for your project, adjust the `storage` block of the `privacy` section in the node configuration file as specified below.

storage block

Specify the DBMS you are using in the `vendor` parameter in the `storage` block of the `privacy` section:

- `postgres` – for PostgreSQL;

- `s3` – for Amazon S3.

Important: If you do not use the **privacy** API methods, specify `none` in the `vendor` parameter and comment out or delete the rest of the parameters in the `privacy` section.

---

vendor = postgres

When using the PostgreSQL DBMS, the `storage` block of the `privacy` section looks like this:

```
storage {
  vendor = postgres
  schema = "public"
  migration-dir = "db/migration"
  profile = "slick.jdbc.PostgresProfile$"
  upload-chunk-size = 1MiB
  jdbc-config {
    url = "jdbc:postgresql://postgres:5432/node-1"
    driver = "org.postgresql.Driver"
    user = postgres
    password = wenterprise
    connectionPool = HikariCP
    connectionTimeout = 5000
    connectionTestQuery = "SELECT 1"
    queueSize = 10000
    numThreads = 20
  }
}
```

The block must contain the following parameters:

- `schema` – the used scheme of interaction between elements within the database. By default, the `public` scheme is used, but if your database provides another scheme, specify its name;

- `migration-dir` – directory for data migration;

- `profile` – profile name for JDBC access, set during JDBC installation (see the *PostgreSQL* section);

- **upload-chunk-size** – the size of the data fragment uploaded using *POST /privacy/sendLargeData* REST API method or *SendLargeData* gRPC API method;

- **url** – the PostgreSQL database address (see the *url field* section for details);

- **driver** – the name of the JDBC driver that allows Java applications to communicate with the database;

- **user** – user name to access the database; specify the login of the account you created to access the database under *PostgreSQL*;

- **password** – the password to access the database; specify the password of the account you created to access the database under *PostgreSQL*;

- **connectionPool** – the connection pool name, **HikariCP** by default;

- **connectionTimeout** – time of connection inactivity before it is broken (in milliseconds);

- **connectionTestQuery** – a test query to test the connection to the database; for PostgreSQL, it is recommended to send **SELECT 1**;

- **queueSize** – the size of the query queue;

- **numThreads** – the number of simultaneous connections to the database.

url field

In the **url** field, specify the address of the database you are using.

More info on url field

Use the following format:

```
jdbc:postgresql://<POSTGRES_ADDRESS>:<POSTGRES_PORT>/<POSTGRES_DB>
```

, where

- **POSTGRES_ADDRESS** – PostgreSQL host address;

- **POSTGRES_PORT** – PostgreSQL host port number;

- **POSTGRES_DB** – the PostgreSQL database name.

You can specify the database address along with the account data using the **user** and **password** parameters:

```
privacy {
  storage {
    ...
    url = "jdbc:postgresql://yourpostgres.com:5432/privacy_node_0?user=user_
↪privacy_node_0@company&password=7nZL7Jr41qOWUHz5qKdypA&sslmode=require"
    ...
  }
}
```

In this example, **user_privacy_node_0@company** is the username, **7nZL7Jr41qOWUHz5qKdypA** is its password.

You can also use the **sslmode=require** command to require ssl usage when authorizing.

pgBouncer

To optimize interoperation with the PostgreSQL database you can use **pgBouncer** − the tool to connect to the PostgreSQL database.

More info on pgBouncer

pgBouncer is configured in a separate configuration file − **pgbouncer.ini**.

We recommend to use `pool_mode` with `session` mode in **pgbouncer.ini** settings file to prevent data loss, as `pool_mode = transaction` mode in pgBouncer configuration does not support prepared server-side statements. When using session mode you should set the `server_reset_query` parameter to `DISCARD ALL`.

```
[pgbouncer]
pool_mode = session
server_reset_query = DISCARD ALL
```

More information about how session mode with prepared operators works can be found in the official documentation for pgBouncer.

vendor = s3

When using the Amazon S3 DBMS, the `storage` block of the `privacy` section looks like this:

```
storage {
  vendor = s3
  url = "http://localhost:9000/"
  bucket = "privacy"
  region = "aws-global"
  access-key-id = "minio"
  secret-access-key = "minio123"
  path-style-access-enabled = true
  connection-timeout = 30s
  connection-acquisition-timeout = 10s
  max-concurrency = 200
  read-timeout = 0s
  upload-chunk-size = 5MiB
}
```

- `url` − address of the Minio server to store data; by default, Minio uses the 9000 port;

- `bucket` − name of the S3 database table to store data;

- `region` − name of the S3 region, the parameter value is `aws-global`;

- `access-key-id` − identifier of the data access key; specify the data access login that you set during the Minio server installation (see *Amazon S3*);

- `secret-access-key` − data access key in the S3 repository; specify the data access password that you set during the Minio server installation (see *Amazon S3*);

- `path-style-access-enabled = true` − the path to S3 table; unchangeable parameter;

- `connection-timeout` − period of inactivity before the connection is broken (in seconds);

- `connection-acquisition-timeout` − period of inactivity when establishing a connection (in seconds);

- `max-concurrency` – the maximum number of concurrent accesses to the storage;

- `read-timeout` – period of inactivity when reading data (in seconds);

- `upload-chunk-size` – the size of the data fragment uploaded using *POST /privacy/sendLargeData* REST API method or *SendLargeData* gRPC API method.

replier block

Use the `replier` block in the `privacy` section to specify confidential data streaming parameters:

```
replier {
  parallelism = 10
  stream-timeout = 1 minute
  stream-chunk-size = 1MiB
}
```

The block must contain the following parameters:

- `parallelism` – the maximum number of parallel tasks for processing privacy data requests;

- `stream-timeout` – the maximum time the read operation on the stream should perform;

- `stream-chunk-size` – the size of one partition when transferring data as a stream.

inventory-handler block

Use the `inventory-handler` block in the `privacy` section to specify policies inventory data aggregation parameters:

```
inventory-handler {
  max-buffer-time = 500ms
  max-buffer-size = 100
  max-cache-size = 100000
  expiration-time = 5m
  replier-parallelism = 10
}
```

The block must contain the following parameters:

- `max-buffer-time` – the maximum time for buffer; when the specified time elapses, the node processes all inventories in batch;

- `max-buffer-size` – the maximum number of inventories in buffer; when the limit is reached, the node processes all inventories in batch;

- `max-cache-size` – the maximum size of inventories cache; using this cache the node selects only new inventories;

- `expiration-time` – expiration time for cache items (inventories);

- `replier-parallelism` – the maximum parallel tasks for processing inventory requests.

cache block

Use the `cache` block in the `privacy` section to specify policy data responses cache parameters:

```
cache {
  max-size = 100
  expire-after = 10m
}
```

Note: Large files (files uploaded using *POST /privacy/sendLargeData* REST API method or *SendLargeData* gRPC API method) are not cached.

The block must contain the following cache parameters:

- `max-size` – the maximum count of elements;

- `expire-after` – the time to expire for element if it hasn't got access during this time.

synchronizer block

Use the `synchronizer` block in the `privacy` section to specify private data synchronization parameters:

```
synchronizer {
  request-timeout = 2 minute
  init-retry-delay = 5 seconds
  inventory-stream-timeout = 15 seconds
  inventory-request-delay = 3 seconds
  inventory-timestamp-threshold = 10 minutes
  crawling-parallelism = 100
  max-attempt-count = 24
  lost-data-processing-delay = 10 minutes
  network-stream-buffer-size = 10
}
```

The block must contain the following parameters:

- `request-timeout` – maximum response waiting time after a data request; the default value is `2 minute`;

- `init-retry-delay` – first delay after an unsuccessful attempt; with each attempt, the delay increases by 4/3; the default value is `5 seconds`;

- `inventory-stream-timeout` – the maximum time the node waits for a network message with the inventory information, i.e. confirmation from a particular node that it has certain data and can provide it for downloading. When this timeout expires, the node sends inventory-request to all the peers to see if they have the necessary data for downloading; the default value is `15 seconds`;

- `inventory-request-delay` – delay after requesting peers data inventory (inventory-request); the default value is – `3 seconds`;

- `inventory-timestamp-threshold` – time threshold for inventory broadcast; inventory broadcast is used for new transactions to speed up the privacy subsystem; the parameter is used to decide whether to send PrivacyInventory message when the data is synchronized (downloaded) successfully; the default value is *10 minutes*`;

- `crawling-parallelism` – the maximum parallel *crawling* tasks count; the default value is 100;

- `max-attempt-count` – the number of attempts that *the crawler* will take before the data is marked as lost; the default value is 24;

- `lost-data-processing-delay` – the delay between the attempts to process the lost items queue; the default value is 10 `minutes`;

- `network-stream-buffer-size` – the maximum count of the data chunks in the buffer; when the limit is reached, back pressure is activated; the default value is 10.

inventory-timestamp-threshold field

A node sends a PrivacyInventory message to peers after it has inserted data into its private storage by a certain data hash. A cache is used to store the PrivacyInventory, which is limited by the number of objects and their time in the cache. Depending on the value of the `inventory-timestamp-threshold` parameter, the data insertion event handler decides whether the PrivacyInventory message should be sent when the data is inserted. The handler compares the transaction timestamp, which corresponds to the given data hash, and the current time on the node. If the difference exceeds the value of the `inventory-timestamp-threshold` parameter, the PrivacyInventory messages are not sent. By adjusting the value of the `inventory-timestamp-threshold` parameter, you can avoid the situation where a node which synchronizes the state with the network clogs the network with unnecessary PrivacyInventory messages.

service block

In the `service` block of the `privacy` section, specify the *SendLargeData* gRPC method and *POST /privacy/sendLargeData* REST method parameters to send a stream of confidential data.

```
service {
  request-buffer-size = 10MiB
  meta-data-accumulation-timeout = 3s
}
```

The block must contain the following parameters:

- `request-buffer-size` – the maximum request buffer size; when the specified size is reached, the back pressure is activated;

- `meta-data-accumulation-timeout` – the maximum time of metadata entity accumulation when sending data via *POST /privacy/sendLargeData* REST API method.

See also

    Deployment of the platform in a private network

    Precise platform configuration: gRPC and REST API authorization

    Precise platform configuration: node gRPC and REST API configuration

    Precise platform configuration: TLS

    Confidential data exchange

Precise platform configuration: logging

The total logging level in a node is set in the `logging-level` parameter in the `node` section of the node configuration file. The specified value will be valid for all the loggers. You can override the logging level for the loggers you list in the `node.loggers` configuration file section. For example:

```
node {
  ...
  # Application logging level. Could be DEBUG | INFO | WARN | ERROR. Default␣
→value is INFO.
  logging-level = DEBUG
  loggers {
       "com.wavesplatform.mining": "TRACE"
  }
}
```

You can set the following logging levels:

- `ERROR` – error logging;

- `WARN` – warning logging;

- `INFO` – node events logging; this is the default value;

- `DEBUG` – extended information about the events for each running node module: a record of events that occurred and actions performed;

- `TRACE` – detailed information about the events of the `DEBUG` level;

- `ALL` – displaying data from all logging levels.

Note: A very detailed logging level on a node can degrade performance, so it is recommended to use the `INFO` level (at most the `DEBUG` level) for the node as a whole (`logging-level` parameter), and configure more detailed logging only for individual loggers.

Log storage

All the node logs are written to the `/node/data/log/we.log` file on the node. To work with this file, you need to enter the node container.

Logging management

The following REST API methods are provided to manage the logging levels of the node:

- *GET /node/logging*
- *POST /node/logging*

List of loggers

Below is a list of loggers available on the node.

List of loggers

- ROOT-DEBUG
- akka-DEBUG
- akka.actor-DEBUG
- akka.actor.LocalActorRef-DEBUG
- akka.event-DEBUG
- akka.event.slf4j-DEBUG
- akka.event.slf4j.Slf4jLogger-DEBUG
- com-DEBUG
- com.github-DEBUG
- com.github.dockerjava-DEBUG
- com.github.dockerjava.api-DEBUG
- com.github.dockerjava.api.async-DEBUG
- com.github.dockerjava.api.async.ResultCallbackTemplate-DEBUG
- com.github.dockerjava.api.command-DEBUG
- com.github.dockerjava.api.command.PullImageResultCallback-DEBUG
- com.github.dockerjava.core-DEBUG
- com.github.dockerjava.core.command-DEBUG
- com.github.dockerjava.core.command.AbstrDockerCmd-DEBUG
- com.github.dockerjava.core.exec-DEBUG
- com.github.dockerjava.core.exec.AuthCmdExec-DEBUG
- com.github.dockerjava.core.exec.CreateContainerCmdExec-DEBUG
- com.github.dockerjava.core.exec.InspectImageCmdExec-DEBUG
- com.github.dockerjava.core.exec.PingCmdExec-DEBUG
- com.github.dockerjava.core.exec.PullImageCmdExec-DEBUG
- com.github.dockerjava.core.exec.RemoveContainerCmdExec-DEBUG
- com.github.dockerjava.core.exec.StartContainerCmdExec-DEBUG
- com.github.dockerjava.jaxrs-DEBUG
- com.github.dockerjava.jaxrs.JerseyDockerHttpClient-DEBUG
- com.github.dockerjava.jaxrs.JerseyDockerHttpClient$1-DEBUG
- com.github.dockerjava.jaxrs.filter-DEBUG
- com.github.dockerjava.jaxrs.filter.LoggingFilter-DEBUG

- com.github.dockerjava.jaxrs.filter.ResponseStatusExceptionFilter-DEBUG

- com.wavesenterprise-DEBUG

- com.wavesenterprise.AppSchedulers-DEBUG

- com.wavesenterprise.AppSchedulers$-DEBUG

- com.wavesenterprise.CorporateAppSchedulers-DEBUG

- com.wavesenterprise.CorporateApplication-DEBUG

- com.wavesenterprise.CorporateApplication$-DEBUG

- com.wavesenterprise.CorporateApplication$$anon-DEBUG

- com.wavesenterprise.CorporateApplication$$anon$1-DEBUG

- com.wavesenterprise.ResourceAvailability-DEBUG

- com.wavesenterprise.ResourceAvailability$-DEBUG

- com.wavesenterprise.api-DEBUG

- com.wavesenterprise.api.grpc-DEBUG

- com.wavesenterprise.api.grpc.CorporateCompositeGrpcService-DEBUG

- com.wavesenterprise.api.grpc.service-DEBUG

- com.wavesenterprise.api.grpc.service.BlockchainEventsServiceImpl-DEBUG

- com.wavesenterprise.api.http-DEBUG

- com.wavesenterprise.api.http.CorporateCompositeHttpService-DEBUG

- com.wavesenterprise.api.http.CorporateTransactionsApiRoute-DEBUG

- com.wavesenterprise.api.http.service-DEBUG

- com.wavesenterprise.api.http.service.PrivacyApiService-DEBUG

- com.wavesenterprise.consensus-DEBUG

- com.wavesenterprise.consensus.MinerBanHistoryV2-DEBUG

- com.wavesenterprise.consensus.PoAConsensus-DEBUG

- com.wavesenterprise.consensus.WarnFaultyMiners-DEBUG

- com.wavesenterprise.crypto-DEBUG

- com.wavesenterprise.crypto.internals-DEBUG

- com.wavesenterprise.crypto.internals.gost-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostAlgorithms-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostCryptoContext-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostCryptoContext$-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostCryptoContext$$anon-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostCryptoContext$$anon$1-DEBUG

- com.wavesenterprise.crypto.internals.gost.GostCryptoTools-DEBUG

- com.wavesenterprise.database-DEBUG

- com.wavesenterprise.database.migration-DEBUG

- com.wavesenterprise.database.migration.SchemaManager-DEBUG

- com.wavesenterprise.database.rocksdb-DEBUG

- com.wavesenterprise.database.rocksdb.Listeners-DEBUG

- com.wavesenterprise.database.rocksdb.Listeners$-DEBUG

- com.wavesenterprise.database.rocksdb.RocksDBStorage-DEBUG

- com.wavesenterprise.database.rocksdb.RocksDBStorage$-DEBUG

- com.wavesenterprise.database.rocksdb.RocksDBWriter-DEBUG

- com.wavesenterprise.docker-DEBUG

- com.wavesenterprise.docker.CorporateGrpcContractExecutor-DEBUG

- com.wavesenterprise.docker.DockerEngineImpl-DEBUG

- com.wavesenterprise.docker.MinerTransactionsExecutor-DEBUG

- com.wavesenterprise.docker.grpc-DEBUG

- com.wavesenterprise.docker.grpc.service-DEBUG

- com.wavesenterprise.docker.grpc.service.ContractServiceImpl-DEBUG

- com.wavesenterprise.docker.validator-DEBUG

- com.wavesenterprise.docker.validator.ExecutableTransactionsValidator-DEBUG

- com.wavesenterprise.http-DEBUG

- com.wavesenterprise.http.HealthCheckerStateful-DEBUG

- com.wavesenterprise.license-DEBUG

- com.wavesenterprise.license.LicenseChecker-DEBUG

- com.wavesenterprise.metrics-DEBUG

- com.wavesenterprise.metrics.Metrics-DEBUG

- com.wavesenterprise.metrics.Metrics$-DEBUG

- com.wavesenterprise.mining-DEBUG

- com.wavesenterprise.mining.CorporateMiner-DEBUG

- com.wavesenterprise.mining.CorporateMiner$-DEBUG

- com.wavesenterprise.mining.CorporateMiner$$anon-DEBUG

- com.wavesenterprise.mining.CorporateMiner$$anon$2-DEBUG

- com.wavesenterprise.mining.CorporateMinerTransactionsConfirmatory-DEBUG

- com.wavesenterprise.mining.CorporateTransactionsAccumulator-DEBUG

- com.wavesenterprise.network-DEBUG

- com.wavesenterprise.network.Attributes-DEBUG

- com.wavesenterprise.network.Attributes$-DEBUG

- com.wavesenterprise.network.BlockLoader-DEBUG

- com.wavesenterprise.network.CorporateHistoryReplier-DEBUG

- com.wavesenterprise.network.CorporateInitialSyncNetworkClient-DEBUG

- com.wavesenterprise.network.CorporateMicroBlockLoader-DEBUG
- com.wavesenterprise.network.CorporateNetworkInitialSync-DEBUG
- com.wavesenterprise.network.CorporateNetworkServer-DEBUG
- com.wavesenterprise.network.EnabledTxBroadcaster-DEBUG
- com.wavesenterprise.network.FatalErrorHandler-DEBUG
- com.wavesenterprise.network.IdleConnectionDetector-DEBUG
- com.wavesenterprise.network.NodeAttributesHandler-DEBUG
- com.wavesenterprise.network.NodeAttributesSender-DEBUG
- com.waveenterprise.network.P2PNetwork-DEBUG
- com.wavesenterprise.network.P2PNetwork$-DEBUG
- com.wavesenterprise.network.ScoringSyncChannelSelector-DEBUG
- com.wavesenterprise.network.TrafficLogger-DEBUG
- com.wavesenterprise.network.WriteErrorHandler-DEBUG
- com.wavesenterprise.network.handshake-DEBUG
- com.wavesenterprise.network.handshake.CorporateHandshakeHandler-DEBUG
- com.wavesenterprise.network.handshake.CorporateHandshakeHandler$Client-DEBUG
- com.wavesenterprise.network.handshake.CorporateHandshakeHandler$Server-DEBUG
- com.wavesenterprise.network.handshake.HandshakeDecoder-DEBUG
- com.wavesenterprise.network.handshake.HandshakeTimeoutHandler-DEBUG
- com.wavesenterprise.network.netty-DEBUG
- com.wavesenterprise.network.netty.handler-DEBUG
- com.wavesenterprise.network.netty.handler.stream-DEBUG
- com.wavesenterprise.network.netty.handler.stream.ChunkedWriteHandler-DEBUG
- com.wavesenterprise.network.package-DEBUG
- com.wavesenterprise.network.package$-DEBUG
- com.wavesenterprise.network.peers-DEBUG
- com.wavesenterprise.network.peers.PeerDatabaseImpl-DEBUG
- com.wavesenterprise.network.peers.PeerSynchronizer-DEBUG
- com.wavesenterprise.network.privacy-DEBUG
- com.wavesenterprise.network.privacy.EnablePolicyDataReplier-DEBUG
- com.wavesenterprise.network.privacy.EnablePolicyDataSynchronizer-DEBUG
- com.wavesenterprise.network.privacy.EnablePolicyDataSynchronizer$-DEBUG
- com.wavesenterprise.network.privacy.EnabledPrivacyMicroBlockHandler-DEBUG
- com.wavesenterprise.network.privacy.PrivacyInventoryHandler-DEBUG
- com.wavesenterprise.privacy-DEBUG
- com.wavesenterprise.privacy.PolicyStorage-DEBUG

- com.wavesenterprise.privacy.PolicyStorage$-DEBUG

- com.wavesenterprise.privacy.db-DEBUG

- com.wavesenterprise.privacy.db.PolicyPostgresStorageService-DEBUG

- com.wavesenterprise.privacy.db.PostgresPolicyDao-DEBUG

- com.wavesenterprise.privacy.db.SchemaMigration-DEBUG

- com.wavesenterprise.privacy.db.SchemaMigration$-DEBUG

- com.wavesenterprise.settings-DEBUG

- com.wavesenterprise.settings.Gost-DEBUG

- com.wavesenterprise.settings.Gost$-DEBUG

- com.wavesenterprise.settings.Gost$$anon-DEBUG

- com.wavesenterprise.settings.Gost$$anon$1-DEBUG

- com.wavesenterprise.state-DEBUG

- com.wavesenterprise.state.CorporateBlockchainUpdaterImpl-DEBUG

- com.wavesenterprise.state.appender-DEBUG

- com.wavesenterprise.state.appender.BaseAppender-DEBUG

- com.wavesenterprise.state.appender.BaseAppender$-DEBUG

- com.wavesenterprise.state.appender.CorporateBaseAppender-DEBUG

- com.wavesenterprise.state.appender.CorporateBlockAppender-DEBUG

- com.wavesenterprise.state.appender.MicroBlockAppender-DEBUG

- com.wavesenterprise.transaction-DEBUG

- com.wavesenterprise.transaction.TransactionFactory-DEBUG

- com.wavesenterprise.transaction.TransactionFactory$-DEBUG

- com.wavesenterprise.transaction.smart-INFO

- com.wavesenterprise.utils-DEBUG

- com.wavesenterprise.utils.NTP-DEBUG

- com.wavesenterprise.utx-DEBUG

- com.wavesenterprise.utx.CorporateUtxPool-DEBUG

- com.wavesenterprise.wallet-DEBUG

- com.wavesenterprise.wallet.WalletImpl-DEBUG

- com.zaxxer-DEBUG

- com.zaxxer.hikari-DEBUG

- com.zaxxer.hikari.HikariConfig-DEBUG

- com.zaxxer.hikari.HikariDataSource-DEBUG

- com.zaxxer.hikari.pool-DEBUG

- com.zaxxer.hikari.pool.HikariPool-DEBUG

- com.zaxxer.hikari.pool.PoolBase-DEBUG

- com.zaxxer.hikari.pool.PoolEntry-DEBUG

- com.zaxxer.hikari.pool.ProxyConnection-DEBUG

- com.zaxxer.hikari.pool.ProxyLeakTask-DEBUG

- com.zaxxer.hikari.util-DEBUG

- com.zaxxer.hikari.util.ConcurrentBag-DEBUG

- com.zaxxer.hikari.util.DriverDataSource-DEBUG

- com.zaxxer.hikari.util.PropertyElf-DEBUG

- io-DEBUG

- io.netty-INFO

- io.netty.bootstrap-INFO

- io.netty.bootstrap.Bootstrap-INFO

- io.netty.bootstrap.ServerBootstrap-INFO

- io.netty.buffer-INFO

- io.netty.buffer.AbstractByteBuf-INFO

- io.netty.buffer.ByteBufUtil-INFO

- io.netty.buffer.PoolThreadCache-INFO

- io.netty.buffer.PooledByteBufAllocator-INFO

- io.netty.channel-INFO

- io.netty.channel.AbstractChannel-INFO

- io.netty.channel.AbstractChannelHandlerContext-INFO

- io.netty.channel.ChannelHandlerMask-INFO

- io.netty.channel.ChannelInitializer-INFO

- io.netty.channel.ChannelOutboundBuffer-INFO

- io.netty.channel.DefaultChannelId-INFO

- io.netty.channel.DefaultChannelPipeline-INFO

- io.netty.channel.MultithreadEventLoopGroup-INFO

- io.netty.channel.nio-INFO

- io.netty.channel.nio.AbstractNioChannel-INFO

- io.netty.channel.nio.NioEventLoop-INFO

- io.netty.channel.socket-INFO

- io.netty.channel.socket.nio-INFO

- io.netty.channel.socket.nio.NioServerSocketChannel-INFO

- io.netty.channel.socket.nio.NioSocketChannel-INFO

- io.netty.handler-INFO

- .netty.handler.flow-INFO

- io.netty.handler.flow.FlowControlHandler-INFO

- io.netty.resolver-INFO

- io.netty.resolver.AddressResolverGroup-INFO

- io.netty.util-INFO

- io.netty.util.NetUtil-INFO

- io.netty.util.NetUtilInitializations-INFO

- io.netty.util.Recycler-INFO

- io.netty.util.ReferenceCountUtil-INFO

- io.netty.util.ResourceLeakDetector-INFO

- io.netty.util.ResourceLeakDetectorFactory-INFO

- io.netty.util.concurrent-INFO

- io.netty.util.concurrent.AbstractEventExecutor-INFO

- io.netty.util.concurrent.DefaultPromise-INFO

- io.netty.util.concurrent.DefaultPromise.rejectedExecution-INFO

- io.netty.util.concurrent.GlobalEventExecutor-INFO

- io.netty.util.concurrent.SingleThreadEventExecutor-INFO

- io.netty.util.internal-INFO

- io.netty.util.internal.CleanerJava9-INFO

- io.netty.util.internal.InternalThreadLocalMap-INFO

- io.netty.util.internal.MacAddressUtil-INFO

- io.netty.util.internal.PlatformDependent-INFO

- io.netty.util.internal.PlatformDependent0-INFO

- io.netty.util.internal.SystemPropertyUtil-INFO

- io.netty.util.internal.logging-INFO

- io.netty.util.internal.logging.InternalLoggerFactory-INFO

- io.swagger-INFO

- javax-DEBUG

- javax.management-INFO

- kamon-DEBUG

- kamon.Kamon-DEBUG

- kamon.ReporterRegistry-DEBUG

- kamon.ReporterRegistry$Default-DEBUG

- kamon.ReporterRegistry$Default$MetricReporterTicker-DEBUG

- kamon.context-DEBUG

- kamon.context.Codecs-DEBUG

- kamon.context.Codecs$Binary-DEBUG

- kamon.context.Codecs$HttpHeaders-DEBUG

- kamon.influxdb-DEBUG

- kamon.influxdb.InfluxDBReporter-DEBUG

- kamon.metric-DEBUG

- kamon.metric.MetricRegistry-DEBUG

- kamon.metric.RangeSamplerMetric-DEBUG

- kamon.metrics-DEBUG

- kamon.metrics.SystemMetrics-DEBUG

- kamon.sigar-DEBUG

- kamon.sigar.SigarProvisioner-DEBUG

- kamon.trace-DEBUG

- kamon.trace.Tracer-DEBUG

- org-DEBUG

- org.apache-DEBUG

- org.apache.http-INFO

- org.aspectj-INFO

- org.asynchttpclient-INFO

- org.flywaydb-INFO

- org.flywaydb.core-INFO

- org.flywaydb.core.Flyway-INFO

- org.flywaydb.core.api-INFO

- org.flywaydb.core.api.configuration-INFO

- org.flywaydb.core.api.configuration.ClassicConfiguration-INFO

- org.flywaydb.core.internal-INFO

- org.flywaydb.core.internal.callback-INFO

- org.flywaydb.core.internal.callback.SqlScriptCallbackFactory-INFO

- org.flywaydb.core.internal.command-INFO

- org.flywaydb.core.internal.command.DbMigrate-INFO

- org.flywaydb.core.internal.command.DbSchemas-INFO

- org.flywaydb.core.internal.command.DbValidate-INFO

- org.flywaydb.core.internal.database-INFO

- org.flywaydb.core.internal.database.DatabaseFactory-INFO

- org.flywaydb.core.internal.database.base-INFO

- org.flywaydb.core.internal.database.base.Database-INFO

- org.flywaydb.core.internal.database.base.Table-INFO

- org.flywaydb.core.internal.database.postgresql-INFO

- org.flywaydb.core.internal.database.postgresql.PostgreSQLAdvisoryLockTemplate-INFO

- org.flywaydb.core.internal.jdbc-INFO
- org.flywaydb.core.internal.jdbc.JdbcUtils-INFO
- org.flywaydb.core.internal.jdbc.TransactionTemplate-INFO
- org.flywaydb.core.internal.license-INFO
- org.flywaydb.core.internal.license.VersionPrinter-INFO
- org.flywaydb.core.internal.resolver-INFO
- org.flywaydb.core.internal.resolver.AbstractJavaMigrationResolver-INFO
- org.flywaydb.core.internal.scanner-INFO
- org.flywaydb.core.internal.scanner.Scanner-INFO
- org.flywaydb.core.internal.scanner.classpath-INFO
- org.flywaydb.core.internal.scanner.classpath.ClassPathScanner-INFO
- org.flywaydb.core.internal.scanner.classpath.JarFileClassPathLocationScanner-INFO
- org.flywaydb.core.internal.scanner.filesystem-INFO
- org.flywaydb.core.internal.scanner.filesystem.FileSystemScanner-INFO
- org.flywaydb.core.internal.schemahistory-INFO
- org.flywaydb.core.internal.schemahistory.JdbcTableSchemaHistory-INFO
- org.flywaydb.core.internal.sqlscript-INFO
- org.flywaydb.core.internal.sqlscript.SqlScript-INFO
- org.flywaydb.core.internal.util-INFO
- org.flywaydb.core.internal.util.ClassUtils-INFO
- org.flywaydb.core.internal.util.FeatureDetector-INFO
- org.flywaydb.core.internal.util.Locations-INFO
- org.glassfish-DEBUG
- org.glassfish.jersey-DEBUG
- org.glassfish.jersey.client-DEBUG
- org.glassfish.jersey.client.ClientExecutorProvidersConfigurator-INFO
- org.glassfish.jersey.inject-DEBUG
- org.glassfish.jersey.inject.hk2-DEBUG
- org.glassfish.jersey.inject.hk2.AbstractHk2InjectionManager-DEBUG
- org.glassfish.jersey.internal-DEBUG
- org.glassfish.jersey.internal.ServiceFinder-DEBUG
- org.glassfish.jersey.internal.util-DEBUG
- org.glassfish.jersey.internal.util.ReflectionHelper-INFO
- org.glassfish.jersey.process-DEBUG
- org.glassfish.jersey.process.internal-DEBUG
- org.glassfish.jersey.process.internal.ExecutorProviders-DEBUG

- org.influxdb-DEBUG

- org.influxdb.impl-DEBUG

- org.influxdb.impl.BatchProcessor-DEBUG

- org.postgresql-DEBUG

- org.postgresql.Driver-INFO

- org.postgresql.core-DEBUG

- org.postgresql.core.v3-DEBUG

- org.postgresql.core.v3.ConnectionFactoryImpl-DEBUG

- org.postgresql.jdbc-DEBUG

- org.postgresql.jdbc.PgConnection-DEBUG

- org.postgresql.ssl-DEBUG

- org.postgresql.ssl.MakeSSL-DEBUG

- org.reflections-DEBUG

- org.reflections.Reflections-DEBUG

- ru-DEBUG

- ru.CryptoPro-INFO

- ru.CryptoPro.JCP-INFO

- ru.CryptoPro.JCP.tools-INFO

- ru.CryptoPro.JCP.tools.JCPLogger-INFO

- ru.CryptoPro.JCSP-INFO

- ru.CryptoPro.JCSP.JCSPLogger-INFO

- slick-INFO

- slick.basic-INFO

- slick.basic.BasicBackend-INFO

- slick.basic.BasicBackend.action-INFO

- slick.compiler-INFO

- slick.compiler.AssignUniqueSymbols-INFO

- slick.compiler.CodeGen-INFO

- slick.compiler.CreateResultSetMapping-INFO

- slick.compiler.ExpandSums-INFO

- slick.compiler.ExpandTables-INFO

- slick.compiler.FlattenProjections-INFO

- slick.compiler.HoistClientOps-INFO

- slick.compiler.MergeToComprehensions-INFO

- slick.compiler.PruneProjections-INFO

- slick.compiler.QueryCompiler-INFO

- slick.compiler.QueryCompilerBenchmark-INFO
- slick.compiler.RemoveFieldNames-INFO
- slick.jdbc-INFO
- slick.jdbc.JdbcBackend-INFO
- slick.jdbc.JdbcBackend.benchmark-INFO
- slick.jdbc.JdbcBackend.parameter-INFO
- slick.jdbc.JdbcBackend.statement-INFO
- slick.jdbc.JdbcBackend.statementAndParameter-INFO
- slick.jdbc.StatementInvoker-INFO
- slick.jdbc.StatementInvoker.result-INFO
- slick.relational-INFO
- slick.relational.ResultConverterCompiler-INFO
- slick.util-INFO
- slick.util.ManagedArrayBlockingQueue-INFO
- sun-DEBUG
- sun.net-DEBUG
- sun.net.www-DEBUG
- sun.net.www.protocol-DEBUG
- sun.net.www.protocol.http-DEBUG
- sun.net.www.protocol.http.HttpURLConnection-DEBUG
- sun.rmi-INFO

See also

> *Deployment of the platform in a private network*
>
> *GET /node/logging*
>
> *POST /node/logging*

Precise platform configuration: anchoring

If you plan to use the data *anchoring* from your network to a larger network, configure the data transfer settings in the `anchoring` block of the node's configuration file. In the terminology of the configuration file, `targetnet` is the blockchain to which your node will perform anchoring transactions from the current network.

```
anchoring {
  enable = yes
  height-range = 30
  height-above = 8
  threshold = 20
  tx-mining-check-delay = 5 seconds
```

(continues on next page)

```
    tx-mining-check-count = 20

    targetnet-authorization {
      type = "oauth2" # "api-key" or "oauth2"
      authorization-token = ""
      authorization-service-url = "https://client.wavesenterprise.com/
↪authServiceAddress/v1/auth/token"
      token-update-interval = "60s"
      # api-key-hash = ""
      # privacy-api-key-hash = ""
    }

    targetnet-scheme-byte = "V"
    targetnet-node-address = "https://client.wavesenterprise.com:6862/
↪NodeAddress"
    targetnet-node-recipient-address = ""
    targetnet-private-key-password = ""

    wallet {
      file = "node-1_mainnet-wallet.dat"
      password = "small"
    }

    targetnet-fee = 10000000
    sidechain-fee = 5000000
}
```

Anchoring parameters

- `enable` – enable or disable anchoring (*yes / no*);

- `height-range` – the block interval, after which the private blockchain node sends transactions to the Targetnet for anchoring;

- `height-above` – the number of blocks in Targetnet, after which the private blockchain node creates a confirmation anchoring transaction with the data of the first transaction. It is recommended to set the value not exceeding the maximum value of rollback of blocks in Targetnet (`max-rollback`);

- `threshold` – the number of blocks that is subtracted from the current height of the private blockchain. Anchoring transaction sent to Targetnet will receive information from the block at `current-height` − `threshold`. If the value **0** is set, the block value at the current block height is written to the anchoring transaction. It is recommended to set the value close to the maximum rollback value in the private blockchain (`max-rollback`);

- `tx-mining-check-delay` – the wait time between checks of transaction availability for anchoring in Targetnet;

- `tx-mining-check-count` – the maximum number of checks for transaction availability for anchoring in the Targetnet, after completion of which the transaction is not considered to enter the network.

Depending on the mining settings on the `Targetnet`, the distance between anchoring transactions may vary. The set value of `height-range` defines the approximate interval between anchoring transactions. The actual time for anchoring transactions to hit a mined block on the Targetnet network may be longer than the time it takes to mine the number of `height-range` blocks on the Targetnet network.

Authorization parameters for anchoring

- `type` – type of authorization when using anchoring:

    - `api-key` – authorization by an `api-key-hash`;

    - `auth-service` – authorization by a JWT-token through *authorization service*.

If you choose authorization by `api-key-hash`, it is sufficient to specify the key value in the `api-key` parameter. If you choose authorization by a JWT-token, you must specify `type = "auth-service"` and uncomment and fill in the parameters below:

- `authorization-token` – permanent authorization token;

- `authorization-service-url` – URL of the authorization service;

- `token-update-interval` – interval for authorization token update.

Targetnet access parameters

A separate `keystore.dat` file is generated for the node that will send anchoring transactions to the Targetnet with the key pair for access to the Targetnet.

- `targetnet-scheme-byte` – Targetnet network (Waves Enterprise Mainnet - **V**);

- `targetnet-node-address` – full network address of the node together with the port number in the Targetnet network to which transactions will be sent for anchoring. The address must be specified together with the connection type (http/https), port number and parameter `NodeAddress`: `http://node.weservices.com:6862/NodeAddress`;

- `targetnet-node-recipient-address` – the address of the node in the Targetnet network, to which the transactions for anchoring will be written, signed by the key pair of this address;

- `targetnet-private-key-password` – node private key password to sign anchoring transactions.

The network address and port for anchoring to the Targetnet network can be obtained from Waves Enterprise technical support specialists. If you use multiple private blockchains with mutual anchoring, use the appropriate private network settings.

Key pair file parameters for signing anchoring transactions in Targetnet (wallet section)

- `file` – file name and path to the file storage directory with the key pair for signing anchoring transactions in the Targetnet network. The file is located on the private network node;

- `password` – the password of the key pair file.

Fee parameters

- `targetnet-fee` – a fee for issuing a transaction for anchoring in the Targetnet network;

- `sidechain-fee` – a fee for issuing a transaction in the current private blockchain.

See also

Deployment of the platform in a private network

Precise platform configuration: node gRPC and REST API configuration

Precise platform configuration: confidential data groups configuration

Precise platform configuration: TLS

Precise platform configuration: snapshot

The `node.consensual-snapshot` block of the node configuration file is used for the *snapshot mechanism* configuration:

```
node.consensual-snapshot {
  enable = yes
  snapshot-directory = ${node.data-directory}"/snapshot"
  snapshot-height = 12000000
  wait-blocks-count = 10
  back-off {
    max-retries = 3
    delay = 10m
  }
  consensus-type = CFT
}
```

This block includes following parameters:

- `snapshot-directory` – directory on a hard drive to save snapshot data. By default, it is the `snapshot` subdirectory in the directory with node data;

- `snapshot-height` – height of the blockchain at which the data snapshot will be created;

- `wait-blocks-count` – number of blocks after data snapshot creation is finished, after which the node sends a message to its peers (addresses from the `peers` list in the node configuration file) that the data snapshot is ready;

- `back-off` – settings section for retries to create a data snapshot in case of errors:

  - `max-retries` – total number of retries,

  - `delay` – interval between retries (in minutes);

- `consensus-type` – consensus type of the genesis block of the new network. Possible values: `POA`, `CFT`.

See also

Deployment of the platform in a private network

Snapshooting

Precise platform configuration: snapshot

Precise platform configuration: node in the watcher mode

The blockchain node can be configured for operation in the watcher mode.

In this mode, the node functions as follows:

- The watcher node does not obtain or send unconfirmed transactions.

- The watcher node does not create new blocks.

- The watcher node does not upload or execute smart contracts.

- The UTX pool of the watcher node does not synchronize with other nodes.

- The watcher node obtains data of microblocks, blocks and transactions for updating its state.

This mode allows to create nodes that are able to obtain the actual blockchain state, but do not participate in mining and do not overflow the network with corresponding messages.

Configuration

To set the node in the watcher mode, change the `mode` parameter in the `node.network` section of the configuration file:

```
node {
...
  network {
    # ENUM: default or watcher
    mode = default
    ...
  }
}
```

- `default` - the standard operational mode;
- `watcher` - the watcher mode.

See also

> *Deployment of the platform in a private network*
>
> *Precise platform configuration: gRPC and REST API authorization*
>
> *Precise platform configuration: node gRPC and REST API configuration*
>
> *Precise platform configuration: confidential data groups configuration*

Precise platform configuration: setting the fee for the transactions sent to the blockchain

Waves Enterprise Mainnet users are charged Mainnet *transaction fees* for each *transaction* on the blockchain network. On a private blockchain network, you can customize the amount of transaction fees.

In the `node.blockchain.fees` section of the node configuration file, you can set the minimum (base) and additional fees for each type of transaction. The default fee settings are as follows:

```
blockchain {
    ...
    ...

    fees {
      base {
        issue = 1 WEST
        transfer = 0.01 WEST
        reissue = 1 WEST
        burn = 0.05 WEST
        exchange = 0.005 WEST
        lease = 0.01 WEST
        lease-cancel = 0.01 WEST
        create-alias = 1 WEST
        mass-transfer = 0.05 WEST
        data = 0.05 WEST
        set-script = 0.5 WEST
        sponsor-fee = 1 WEST
        set-asset-script = 1 WEST
        permit = 0.01 WEST
        create-contract = 1 WEST
        call-contract = 0.1 WEST
        disable-contract = 0.01 WEST
        update-contract = 1 WEST
        register-node = 0.01 WEST
        create-policy = 1 WEST
        update-policy = 0.5 WEST
        policy-data-hash = 0.05 WEST
      }

      additional {
        mass-transfer = 0.01 WEST
        data = 0.01 WEST
      }
    }
  }
```

Important: It is not recommended to change the fee while the network is running, especially to reduce them, as this will cause transaction validation problems on the nodes that will validate the state when synchronizing from 0 height (mined transactions will have unacceptably small fees).

Zero fee

If you want to build a network where no fees are charged for sending transactions, assign `false` to the `fees.enabled` parameter in the `node.blockchain` section of the node configuration file.

If the `fees.enabled` parameter is set to `false`, the node will be able to send transactions for which the `fee` is set to `0` to the blockchain. Also, when synchronizing the state the node will recognize as valid the transactions of other nodes, in which the `fee` field is set to `0`.

Note: If the `fees.enabled` parameter is set to `false`, then both zero fee and fee greater than zero are allowed in transactions.

See also

> *Deployment of the platform in a private network*
>
> *Transactions of the blockchain platform*

Full examples of configuration files to configure each node are given by *here*.

1.5.3 Obtaining a private network license and associated files

To deploy the platform on a private network, you need to get the kind of license that suits your purposes: *trial, commercial or non-commercial*.

Note: The opensource version of the Waves Enterprise blockchain platform does not require a license.

The license to run a node is tied to the node owner's key. The license contains the address of the node for which the license is issued.

To discuss the details of your license, contact Waves Enterprise Sales at sales@wavesenterprise.com.

After that, you will be sent the license file. Place the file into the folder whose path is specified in the `license-file` parameter of the node configuration file.

Before deployment, read the blockchain platform *system requirements*.

1.5.4 Genesis block signing

After configuring your network's nodes, you must create a genesis block, the first private blockchain block which contains the transactions that determine a node's initial balance and permissions.

A genesis block is signed by the *GenesisBlockGenerator* utility included in the **generator** package. It uses the `node.conf` node configuration file that you set up as an argument:

```
java -jar generator-x.x.x.jar GenesisBlockGenerator node.conf
```

As a result, the utility fills the `genesis-public-key-base-58` and `signature` fields located in the `genesis` block of the `blockchain` section in the node configuration file with the generated values of the public key and signature of the genesis block.

Example:

```
genesis-public-key-base-58: "4ozcAj...penxrm"
signature: "5QNVGF...7Bj4Pc"
```

1.5.5 Launching the network

After signing the genesis block, the platform is fully configured and ready to run the network.

You can launch the network using one of the following docker container administration tools:

- Docker Compose
- Kubernetes

If you plan to use Docker Compose to launch the network, follow the same process you use for *the trial mode*.

If you plan to use Kubernetes to launch the network, launch it according to the instructions received from the Waves Enterprise Technical Support Team.

1.5.6 Attaching the Client application to the private network

Once the network is up and running, attach the *Waves Enterprise Client application* to it: with this, network users can send transactions to the blockchain, as well as broadcast and call smart contracts.

1. Open your browser and enter the network address of your computer with the deployed node software in the address bar.

2. Register to the web client using any valid email address and log in to the web client.

3. Open the **Select address -> Create address** page. To open the menu after the first login, you must enter the password that you entered when you registered your account.

4. Select **Add address from the node keystore** and click **Continue**.

5. Fill in the fields below. The required values are given in the `credentials.txt` file for the first node in the working directory.

- Address name – specify the name of the node;

- Node URL – specify the `http://<computer network address>/<node address>` value;

- Type of authorization on the node – select the authorization type you configured earlier: by JWT-token or by `api-key`;

- Blockchain address – specify the address of your node;

- Key pair password – specify the password to the node key pair if you have set it up while generating the account.

Client description is provided in the *Client* article.

See also

> *Examples of node configuration files*
>
> *Generators*
>
> *Licenses of the Waves Enterprise blockchain platform*

Installation and usage of the platform

1.6 Examples of node configuration files

1.6.1 node.conf

This configuration example:

- uses the PoA consensus algorithm;
- uses the second genesis version;
- enables the **sender** permission for the network participants (see *Permissions*);
- enables mining for three nodes;
- disables *TLS*;
- enables the gRPC and REST API tools without TLS, as well as execution of smart contracts;
- enables api-key hash authorization for gRPC and REST API;
- uses **privacy** methods with a PostgreSQL database for confidential data storage;
- the function of periodic deletion of invalid transactions from the UTX pool of a non-miner is configured.
- the delay of checking the UTX pool (whether there are transactions in the pool or it is empty) by the miner is set.

Fields whose values you get when using the **generators** package or set yourself based on your hardware and software configuration are marked as /FILL/.

Each section is provided with an additional comment.

node.conf:

```
node {
  # Type of cryptography. The field is deprecated since v1.9.0, use 'node.crypto.type =␣
␣waves' instead.
  waves-crypto = yes

  crypto {
    # Possible values: [WAVES]
    type = WAVES
  }

  # Node owner address
  owner-address = " /FILL/ "

  # NTP settings
```

(continues on next page)

```
  ntp.fatal-timeout = 5 minutes

  # Node "home" and data directories to store the state
  directory = "/node"
  data-directory = "/node/data"

  # Location and name of a license file
  # license.file = ${node.directory}"/node.license"

wallet {
  # Path to keystore.
  file = "/node/keystore.dat"

  # Access password
  password = " /FILL/ "
}

# Blockchain settings
blockchain {
  type = CUSTOM
  fees.enabled = false
  consensus {
    type = "poa"
    round-duration = "17s"
    sync-duration = "3s"
    ban-duration-blocks = 100
    warnings-for-ban = 3
    max-bans-percentage = 40
  }
  custom {
    address-scheme-character = "E"
    functionality {
      feature-check-blocks-period = 1500
      blocks-for-feature-activation = 1000
      pre-activated-features = { 2 = 0, 3 = 0, 4 = 0, 5 = 0, 6 = 0, 7 = 0, 9 = 0, 10 = 0,
→ 100 = 0, 101 = 0 }
    }

    # Mainnet genesis settings
    genesis {
      version: 2
      sender-role-enabled: true
      average-block-delay: 60s
      initial-base-target: 153722867

      # Filled by GenesisBlockGenerator
      block-timestamp: 1573472578702

      initial-balance: 16250000 WEST

      # Filled by GenesisBlockGenerator
      genesis-public-key-base-58: ""
```

```
      # Filled by GenesisBlockGenerator
      signature: ""

      transactions = [
          # Initial token distribution:
          # - recipient: target's blockchain address (base58 string)
          # - amount: amount of tokens, multiplied by 10e8 (integer)
          #
          #     Example: { recipient: "3HQSr3VFCiE6JcWwV1yX8xttYbAGKTLV3Gz", amount:␣
→30000000 WEST }
          #
          # Note:
          #   Sum of amounts must be equal to initial-balance above.
          #
        { recipient: " /FILL/ ", amount: 1000000 WEST },
        { recipient: " /FILL/ ", amount: 1500000 WEST },
        { recipient: " /FILL/ ", amount: 500000 WEST },
      ]
      network-participants = [
          # Initial participants and role distribution
          # - public-key: participant's base58 encoded public key;
          # - roles: list of roles to be granted;
          #
          #     Example: {public-key: "EPxkVA9iQejsjQikovyxkkY8iHnbXsR3wjgkgE7ZW1Tt",␣
→roles: [permissioner, miner, connection_manager, contract_developer, issuer]}
          #
          # Note:
          #   There has to be at least one miner, one permissioner and one connection_
→manager for the network to start correctly.
          #   Participants are granted access to the network via␣
→GenesisRegisterNodeTransaction.
          #   Role list could be empty, then given public-key will only be granted␣
→access to the network.
          #
        { public-key: " /FILL/ ", roles: [permissioner, sender, miner, connection_
→manager, contract_developer, issuer]},
        { public-key: " /FILL/ ", roles: [miner, sender]},
        { public-key: " /FILL/ ", roles: []},
      ]
    }
  }
}

# Application logging level. Could be DEBUG | INFO | WARN | ERROR. Default value is INFO.
logging-level = DEBUG

tls {
  # Supported TLS types:
  # ● EMBEDDED: Certificate is signed by node's provider and packed into JKS Keystore.␣
→The same file is used as a Truststore.
  #            Has to be manually imported into system by user to avoid certificate␣
```

```
→warnings.
  # ● DISABLED: TLS is fully disabled
  type = DISABLED

  # type = EMBEDDED
  # keystore-path = ${node.directory}"/we_tls.jks"
  # keystore-password = ${TLS_KEYSTORE_PASSWORD}
  # private-key-password = ${TLS_PRIVATE_KEY_PASSWORD}
}

# P2P Network settings
network {
  # Network address
  bind-address = "0.0.0.0"
  # Port number
  port = 6864
  # Enable/disable network TLS
  tls = no

  # ENUM: regular or watcher
  mode = regular

  # Peers network addresses and ports
  #   Example: known-peers = ["node-1.com:6864", "node-2.com:6864"]
  known-peers = [  /FILL/  ]

  # Node name to send during handshake. Comment this string out to set random node name.
  #   Example: node-name = "your-we-node-name"
  node-name = " /FILL/ "

  # How long the information about peer stays in database after the last communication␣
→with it
  peers-data-residence-time = 2h

  # String with IP address and port to send as external address during handshake. Could␣
→be set automatically if uPnP is enabled.
  #   Example: declared-address = "your-node-address.com:6864"
  declared-address = "0.0.0.0:6864"

  # Delay between attempts to connect to a peer
  attempt-connection-delay = 5s
}

# New blocks generator settings
miner {
  enable = yes
  # Important: use quorum = 0 only for testing purposes, while running a single-node␣
→network;
  # In other cases always set quorum > 0
  quorum = 2
  interval-after-last-block-then-generation-is-allowed = 10d
  micro-block-interval = 5s
```

```
  min-micro-block-age = 3s
  max-transactions-in-micro-block = 500
  minimal-block-generation-offset = 200ms
  utx-check-delay = 100ms
}

# Nodes REST API settings
api {
  rest {
    # Enable/disable REST API
    enable = yes

    # Network address to bind to
    bind-address = "0.0.0.0"

    # Port to listen to REST API requests
    port = 6862

    # Enable/disable TLS for REST
    tls = no
  }

  grpc {
    # Enable/disable gRPC API
    enable = yes

    # Network address to bind to
    bind-address = "0.0.0.0"

    # Port to listen to gRPC API requests
    port = 6865

    # Enable/disable TLS for gRPC
    tls = no
  }
}

  auth {
    type: "api-key"

    # Hash of API key string
    # You can obtain hashes by running ApiKeyHash generator
    api-key-hash: " /FILL/ "

    # Hash of API key string for PrivacyApi routes
    privacy-api-key-hash: " /FILL/ "
  }
}

#Settings for Privacy Data Exchange
privacy {
```

```
replier {
  parallelism = 10
  stream-timeout = 1 minute
  stream-chunk-size = 1MiB
}

# Syncs private data.
synchronizer {
  request-timeout = 2 minute
  init-retry-delay = 5 seconds
  inventory-stream-timeout = 15 seconds
  inventory-request-delay = 3 seconds
  inventory-timestamp-threshold = 10 minutes
  crawling-parallelism = 100
  max-attempt-count = 24
  lost-data-processing-delay = 10 minutes
  network-stream-buffer-size = 10
}

inventory-handler {
  max-buffer-time = 500ms
  max-buffer-size = 100
  max-cache-size = 100000
  expiration-time = 5m
  replier-parallelism = 10
}

cache {
  max-size = 100
  expire-after = 10m
}

storage {
  vendor = postgres

  # for postgres vendor:
  schema = "public"
  migration-dir = "db/migration"
  profile = "slick.jdbc.PostgresProfile$"
  upload-chunk-size = 1MiB
  jdbc-config {
                url = "jdbc:postgresql://postgres:5432/node-1"
                driver = "org.postgresql.Driver"
            user = postgres
                password = wenterprise
                connectionPool = HikariCP
                connectionTimeout = 5000
                connectionTestQuery = "SELECT 1"
                queueSize = 10000
                numThreads = 20
            }
```

```
    # for s3 vendor:
    # url = "http://localhost:9000/"
    # bucket = "privacy"
    # region = "aws-global"
    # access-key-id = "minio"
                   # secret-access-key = "minio123"
                   # path-style-access-enabled = true
                   # connection-timeout = 30s
                   # connection-acquisition-timeout = 10s
                   # max-concurrency = 200
                   # read-timeout = 0s
    # upload-chunk-size = 5MiB
  }

  service {
    request-buffer-size = 10MiB
    meta-data-accumulation-timeout = 3s
  }
}


# Docker smart contracts settings
docker-engine {
  # Docker smart contracts enabled flag
  enable = yes

  # For starting contracts in a local docker
  use-node-docker-host = yes

  default-registry-domain = "registry.wavesenterprise.com/waves-enterprise-public"
  # Basic auth credentials for docker host
   #docker-auth {
   #  username = "some user"
   #  password = "some password"
   #}

  # Optional connection string to docker host
   docker-host = "unix:///var/run/docker.sock"

  # Optional string to node REST API if we use remote docker host
  # node-rest-api = "node-0"

  # Execution settings
  execution-limits {
    # Contract execution timeout
    timeout = 10s
    # Memory limit in Megabytes
    memory = 512
    # Memory swap value in Megabytes (see https://docs.docker.com/config/containers/
→resource_constraints/)
    memory-swap = 0
  }
```

```
  # Reuse once created container on subsequent executions
  reuse-containers = yes

  # Remove container with contract after specified duration passed
  remove-container-after = 10m

  # Remote registries auth information
  remote-registries = []

  # Check registry auth on node startup
  check-registry-auth-on-startup = yes

  # Contract execution messages cache settings
  contract-execution-messages-cache {
    # Time to expire for messages in cache
    expire-after = 60m
    # Max number of messages in buffer. When the limit is reached, the node processes␣
↪all messages in batch
    max-buffer-size = 10
    # Max time for buffer. When time is out, the node processes all messages in batch
    max-buffer-time = 100ms
    #The interval after which invalid transactions (with Error status) are removed from␣
↪the UTX pool of a non-miner node
    utx-cleanup-interval = 1m
    #The minimum number of transaction Error statuses received from other nodes, after␣
↪which the transaction is removed from the UTX pool of a non-miner node
    contract-error-quorum = 2
  }
 }
}
```

1.6.2 accounts.conf

In this example, Waves Crypto encryption is enabled, the standard network identification byte is used and the keystore node update option for generating 1 key pair is disabled.

Password which you have to enter by yourself is marked as /FILL/.

accounts.conf:

```
accounts-generator {
  crypto {
    type = WAVES
    pki {
      mode = OFF
      required-oids = []
    }
  }
  chain-id = T
```

```
  amount = 5
  wallet = ${user.home}"/node/wallet/wallet1.dat"
  wallet-password = "/FILL/"
  reload-node-wallet {
    enabled = false
    url = "http://localhost:6869/utils/reload-wallet"
  }
}
```

1.6.3 api-key-hash.conf

In this example, Waves encryption is enabled.

api-key-hash.conf:

```
apikeyhash-generator {
  crypto {
    type = Waves
  }
  api-key = "some string for api-key"
}
```

1.6.4 Additional examples

For more examples of configuration files with comments, see the official Waves Enterprise GitHub repository.

See also

> Deployment of the platform in a private network
>
> Generators

1.7 System errors

Below is a list of error codes for the Waves Enterprise blockchain platform.

0-10 ˘ Swagger/API Specific Errs

Table 2: 0-10 - Swagger and API Errors

Node level errors	HTT code	API level code	Node level code	Message	Context	Condition
		No	No	Transaction is not in blockchain	When querying a transaction by id	The transaction is not in the blockchain
	400	1	WrongJson	Failed to parse json message	Relevant for requests via Swagger	
	403	2	ApiKeyNotValid	Provided API key is not correct	Relevant only for signing transactions on the node, because the key is not needed when transmitting signed transactions	An invalid or empty key was passed in the request
TooBigArray	400	10	TooBigArrayAllocation	Too big sequences requested	When querying through Swagger	The request contains too many values

101-111 ˘ TxValidation Errs

Table 3: 101-111 − Transaction validation errors

Node level errors	HTTP code	API level code	Node level code	Message	Context	Condition
In-validSig-na-ture(_, _) In-validRec nature	400	101	In-validS na-ture	invalid signature	Blockchain event during blocks validation (not displayed in the Client)	Incorrect transaction id Incorrect block signature in request An error is displayed in the Client interface when attempting to rollback the blockchain for the block with the specified signature
Inval-idAd-dress	400	102	In-val-i-dAd-dress	invalid address. Logged as: 1. Bad public key string lenght. 2. Unable to decode base58: ${ex.getMessage}" 3. "Unable to create public key: ${ex.getMessage}"	When validating any field containing the address, alias, and private key of either sender or recipient on the node. If the client part does not check the validity of the address	An invalid address, alias, or public key was passed in the request
	400	106	In-valid-Send	invalid sender	When forming Diff from ExecutedContract Transaction	If the creator of the Executed Contract Transaction is not the block miner
	400	108	In-valid-Pub-licKe	invalid public key	When checking if the passed string is a public key	GET /addresses/ publicKey/ {publicKey}
	400	110	In-validl sage	invalid message	When verifying the signature of a message on a node. The transaction signature does not match the public key.	POST / addresses/ verify/ {address} POST / addresses/ verifyText/ {address}
Nega-tiveAm of)	400	111	Neg-a-tiveA (s"$x of $of")	negative amount: $msg	When creating a transfer, mass transfer, leasing, issue and reissue transaction, the user the user enters a negative number in the "Amount" field	A negative value is passed. In the Client interface, when trying to specify a negative number, the input field goes to error state and displays the "Enter a positive number" error

112 ˘ StateCheckFailed Errs ˘ StateCheckFailed(tx:Transaction, err: String)

This section describes the TransactionValidationError node level error. It corresponds to the 400 HTTP code and the 112 API level code.

Table 4: 112 −StateCheckFailed(tx: Transaction, err: "State check
failed. Reason: PermissionError")

Message	Context	Condition
State check failed. Reason: $err ("error id", "message", tx.json())	112 StateCheckFailed error is returned with an attachment, that contains the code and description of the error	Validation prior to UTX
Script doesn't exist and proof doesn't validate as signature for $pt	The sender's public key does not match the transaction signature	
Transactions from non-scripted accounts must have exactly 1 proof	When sending a transaction with more than one signature to an account without a script	
${tx.getClass.getSimpleName} transaction has not been activated yet	The option used in the transaction is not enabled on the validator node	
Transaction $tx is already in the state on a height of $txHeight	A transaction with this id already exists in the blockchain	
Attempt to transfer unavailable funds: Transaction application leads to " +s"negative WEST balance to (at least) temporary negative state, current balance equals $oldWestBalance, " +s"spends equals ${spendings.balance}, result is $newWestBalance)	There are insufficient funds in the balance for transfer transactions to one or more recipients of the primary WEST token. The Client interface displays the error: "Failed to execute transaction (%Transaction type%)" Not enough tokens to pay the fee. Refill the balance and try again."	
Attempt to transfer unavailable funds: Transaction application leads to negative asset " + s"'$aid' balance to (at least) temporary negative state, current balance is availableBalance, " + s"spends equals $delta, result is ${availableBalance + delta}	На балансе недостаточно средств для транзакций перевода одному или нескольким получателям ассета В интерфейсе клиента отображается ошибка: "Не удалось выполнить операцию (%Тип транзакции%) The calculated commission amount is not enough to pay for the transaction. Edit the fee amount and try again."	
s"Fee in ${feeAssetId.fold ("WEST")(_.toString)} for ${tx.builder.classTag} does not exceed minimal value of $minimumFee WEST: $feeAmount"	The fee for a standard transaction (without scripts) is less than required	
This transaction with a smart token requires ${-restFeeAmount} additional fee	Комиссии недостаточно из-за смарт токена. В интерфейсе клиента отображается ошибка: "Failed to execute operation (%Transaction type%)" Рассчитанной суммы комиссии недостаточно для оплаты транзакции"	
Scripted account requires ${-restFeeAmount} additional fee for this transaction	Комиссии недостаточно из-за смарт аккаунта. В интерфейсе клиента отображается ошибка: "Не удалось выполнить операцию (%Тип транзакции%). Рассчитанной суммы комиссии недостаточно для оплаты транзакции"	

112 ˘ StateCheckFailed Errs ˘ StateCheckFailed(tx: Transaction, err: ɪState check failed. Reason: PermissionErrorɟ)

В данном разделе описана ошибка уровня ноды TransactionValidationError. Ей соответствует HTTP-код 400, код уровня API 112 и ошибка уровня API `StateCheckFailed(tx: Transaction, err: "State check failed. Reason: PermissionError")`.

Table 5: 112 − StateCheckFailed(tx: Transaction, err: "State check failed. Reason: PermissionError")

Message	Context	Condition
Genesis Permissioner role cannot be removed	Role recall	Отсутствует разрешение на действие или пользователь в списке banned
Sender ${senderAccount.address} permission validation failed: ${permErr.err}		
Transaction ${ unauthorizedTx.id ()} is unauthorized, permission validation impossible		
Address ${address.address} is not a miner		
Blockchain error during permission validation: $ex		
Doesn't have any of required roles: ${roles.map(_.prefixS).mkString(", ")}"		
Required ${roles.map(_.prefixS).mkStrin ")} roles, missing {missingRoles.map(_.prefixS).mkString(", ")} roles		
Has no active $role role		
Sender is $role		

See also

- *Transfer transaction*
- *Mass-transfer transaction*
- *Tokens of the Waves Enterprise blockchain platform*

113-117 ˘ TxValidation Errs

Node node level	HTT code	API level code	Node level code	Message	Context	Condition
Over-flow-Er-ror	400	113	Over-flow-Er-ror	overflow error	If the sum of transaction and fee exceeds the permissible value: the total amount in the mass transfer is greater than the long (technically unlikely)	Long overflow
Neg-a-tiveM-in-Fee(x of)	400	114	Neg-a-tiveM-inFee (s "$x per $of")	negative fee per: $msg	Relevant only for Sponsorship transactions	An error is returned on transaction validation, if a negative value is passed
Miss-ingSe Pri-vateK	400	115	Miss-ingSen Pri-vateKe	no private key for sender address in wallet or provided password is incorrect	When trying to sign a transaction on a node (not on the Client), the key for this public key is not found in the key store	No private (for signing) key was found, or the key pair password is incorrect. A message is displayed in the Client: "No matching private key was found in the keystore, or the key pair password is entered incorrectly
In-valid-Name	400	116	In-valid-Name	invalid name	Incorrect asset name in transaction	the length of the asset name is out of bounds in an issue transaction
		117		"Trying to revoke role '$role' from it's last owner: '$ad-dress'"	There are too few participants with the given role left in the network	

See also

Sponsorship Transaction

Permissions (roles)

199 ˘ CustomValidationError

В данном разделе описана ошибка уровня ноды CustomValidationError. Ей соответствует HTTP-код 400 и код уровня API 199.

Table 6: 199 – Ошибки валидации

Node level errors	API level errors	Message	Context	Condition
	Custom Validation-Error	Одно из перечисленных ниже сообщений	Возвращается ошибка 119 Custom ValidationError со вложением, которое содержит название и текст ошибки	Условия описаны ниже
Generic-Error(ge)	Custom Validation-Error(err)	err: String (Throwables. getStack Trace AsString(ex)) - место для новых ошибок		В интерфейсе клиента отображается ошибка: "Нам не удалось определить причину сбоя. Скопируйте код ошибки и отправьте его в поддержку"
Generic-Error ("Alias already claimed"	Custom Validation-Error ("Alias already claimed")	Alias already claimed	При создании псевдонима на этапе регистрации пользователя	У другого адреса в блокчейне уже есть такой псевдоним. В интерфейсе клиента отображается ошибка: "Не удаётся продолжить регистрацию. Указанный вами псевдоним уже занят. Придумайте и укажите новый, чтобы продолжить регистрацию.
Un-supported Trans-action-Type	Custom Validation-Error ("Unsupported Trans-action-Type")	Unsupported TransactionType (version: Int)	Клиент работает только с актуальными типами	Transaction type not supported
Mi-croBlock Ap-pen-dError	Custom Validation-Error (er-ror.toStrin	MicroBlockAp-pend Error($err, ${microBlock. totalResBlock-Sig} ˜> ${mi-croBlock. pre-vResBlockSig. trim}])		Microblock validation error
Ac-count-Bal-ance Error	Custom Validation-Error (errs.value: mkString (", "))	errs: Map[Address, String]	Verifying that there are enough funds in the ac-count balance	The balance does not allow the transac-tion The Client interface displays the error: "Failed to execute transaction (%Trans-action type%)" Недостаточно токенов для оплаты комиссии. Пополните баланс и повторите попытку.
Block-From-Future	Custom Validation-Error (er-ror.toStrin	ts: Long	Не клиентская ошибка, ошибка внутри ноды	Некорректная временная метка создания блока (timestamp)
Un-sup	Custom Vali-	version: Int	Некорректное значение версии	Неподдерживаемая версия транзакции

301-304 ˇ TxValidation Errs

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

Node node level	HTTP code	API level code	Node level code	Message	Context	Condition

305-307, 309-310, 600-605 ˇ RIDE and Docker Contract Errs

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

606-629, 636 ˇ Privacy, Auth, PKI, Contracts

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

631-635 ˇ License Errs

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

640 ˇ Health check

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

641-643 ˇ gRPC specific

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

700-799 ˇ Snapshot

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition

800 ˇ ForbiddenDuePkiModeError

Node level errors	HTTP-code	API level code	API level errors	Message	Context	Condition
	++	""	•	—		

See also

> Deployment of the platform in a private network
>
> Deploying the platform in the trial mode (Sandbox)
>
> Deploying a platform with connection to Mainnet

1.8 gRPC tools

The Waves Enterprise blockchain platform provides the ability to interact with the blockchain using a gRPC interface.

gRPC is a high-performance Remote Procedure Call (RPC) framework developed by Google Corporation. The framework works via the HTTP/2. The **protobuf** serialization format is used to transfer data between the client and the server. The format describes the data types used.

Officially, gRPC supports 10 programming languages. A list of supported languages is available in the official gRPC documentation.

Some services are available in two versions: for external integration (public services) and for smart contracts (contract services). Use public services for WE integration. Contract services are not intended to be called by an external user, they have a different authorization and behavior. The contract services are packaged in protobuf files placed in the **contract** directory and are described in the *gRPC services used by Docker smart contracts* section. When used in smart contracts, these methods require authorization.

1.8.1 Preconfiguring the gRPC interface

Before using the gRPC interface:

1. decide on the programming language you will use to interact with the node;

2. install the gRPC framework for your programming language according to the official gRPC documentation;

3. download and unpack the protobuf package we-proto-x.x.x.zip for the platform version you are using and the protoc plugin to compile the protobuf files;

4. make sure that the gRPC interface *is started and configured in the configuration file of the node*, with which data will be exchanged.

To communicate with the node via the gRPC interface, the default port is **6865**.

1.8.2 What the gRPC interface is for

You can use the gRPC interface of each node for the following tasks:

gRPC: monitoring of blockchain events

The gRPC interface provides the ability to track certain groups of events occurring in the blockchain. Information about the selected groups of events is collected in streams, which are sent to the gRPC interface of the node.

A set of fields for serializing and transmitting blockchain event data are given in the files that are located in the **messagebroker** directory of the `we-proto-x.x.x.zip` package:

- `messagebroker_blockchain_events_service.proto` – main protobuf file;

- `messagebroker_blockchain_event.proto` – a file that contains response fields with event group data and error messages.

To track a specific group of events on the blockchain, send the `SubscribeOn(startFrom, transactionTypeFilter)` query that initializes a subscription to the selected event group.

Important: The field data types for the request and response are specified in the protobuf files.

Query parameters:

- **startFrom** – the moment when the event tracking starts; the following values are available:

- `CurrentEvent` – start tracking from the current event;

- `GenesisBlock` – getting all events of the selected group, starting from the genesis block;

- `BlockSignature` – the start of tracking from the specified block.

- **transactionTypeFilter** – filter output events by transactions that are produced during these events:

- `Any` – output events with all types of transactions;

- `Filter` – output events with transaction types specified as a list;

- `FilterNot` – display events with all transactions except those specified in this parameter as a list.

- **connectionId** – optional parameter which can be sent in order to alleviate identification of the request in the node logs.

Together with the `SubscribeOnRequest` query, authorization data is sent: the JWT token or the `api-key` passphrase, depending on the authorization method used.

Query examples:

SubscribeOn:

Send messages starting from the first block:

```
{"genesis_block": {}}
```

Send messages starting from the current moment:

```
{"current_event": {}}
```

Send messages starting from the block spacified:

```
{
    "block_signature": {
        "last_block_signature": {
            "value": "G4gTl/
→5fA2g2YAFCjCGu+tXJVqvQCLNM8CxzT6Nfc3KSRg3egAY8Mb4df5tufIf9Tv2xfCPQQ5m7X4MoPBvnBg=="
        }
    }
}
```

Note: Various information about blocks can also be obtained using the REST methods of the *blocks* group.

Information about events

After a successful request is sent to the gRPC interface, the following groups of events will receive data:

1. **MicroBlockAppended** – successful microblock mining:

   • `transactions` – full transaction bodies from the received microblock.

2. **BlockAppended** – successful completion of a mining round with a block formation:

   • `block_signature` – signature of an obtained block;

   • `reference` – signature of a previous block;

   • `tx_ids` – list of transaction IDs from the received block;

   • `miner_address` – miner address;

   • `height` – height at which the resulting block is located;

   • `version` – version of the block;

   • `timestamp` – time of block formation;

   • `fee` – total fee amount for transactions within the block;

   • `block_size` – size of a block (in bytes);

   • `features` – list of blockchain soft-forks that the miner voted for during the round.

---

3. **RollbackCompleted** – block rollback:

- `return_to_block_signature` – signature of the block to which the rollback occurred;

- `rollback_tx_ids` – list of IDs of transactions that will be deleted from the blockchain.

4. **AppendedBlockHistory** – информация о транзакциях сформированного блока. Данный тип событий поступает на gRPC-интерфейс до достижения текущей высоты блокчейна, если в запросе в качестве отправной точки для получения событий указаны `GenesisBlock` или `BlockSignature`. После достижения текущей высоты начинают выводиться текущие события по заданным фильтрам.

Response data:

- `signature` – block signature;

- `reference` – signature of a previous block;

- `transactions` – full transaction bodies from the microblock;

- `miner_address` – miner address;

- `height` – height at which the resulting block is located;

- `version` – version of the block;

- `timestamp` – time of block formation;

- `fee` – total fee amount for transactions within the block;

- `block_size` – size of a block (in bytes);

- `features` – list of blockchain soft-forks that the miner voted for during the round.

Information about errors

The ErrorEvent message with the following error options is provided to display information about errors during blockchain event tracking:

- `GenericError` – a general or unknown error with the message text;

- `MissingRequiredRequestField` – the required field is not filled in when forming a SubscribeOnRequest query;

- `BlockSignatureNotFoundError` – the signature of the requested block is missing in the blockchain;

- `MissingAuthorizationMetadata` – no authorization data was entered when forming the `SubscribeOn` query;

- `InvalidApiKey` – wrong passphrase when authorizing by api-key;

- `InvalidToken` – wrong JWT-token when authorizing by OAuth.

See also

gRPC tools

gRPC: obtaining node information

The **NodeInfoService** gRPC service is provided to obtain node configuration parameters and data about its owner.

The **NodeInfoService** has the following methods described in the **util_node_info_service.proto** protobuf file:

- **NodeConfig**;
- **NodeOwner**.

Important: The field data types for the request and response are specified in the protobuf files.

Note: The same data as the NodeConfig and NodeOwner gRPC methods return, can be obtained using the REST methods of the *node* group.

---

gRPC: obtaining node configuration parameters

Use the `NodeConfig` method to retrieve node configuration parameters. The `NodeConfig` method does not require any additional query parameters. The following configuration parameters for the node that was queried are displayed in the response:

- `version` – version of the blockchain platform in use;
- `crypto_type` – cryptographic algorithm in use;
- `chain_id` – identifying byte of the network;
- `consensus` – consensus algorithm in use;
- `minimum_fee` – minimum transaction fee;
- `minimum_fee` – additional transaction fee;
- `max-transactions-in-micro-block` – the maximum number of transactions in a microblock;
- `min_micro_block_age` – the minimum time of microblock existence (in seconds);
- `micro-block-interval` – the interval between microblocks (in seconds);
- `pos_round info`– when using the PoS consensus algorithm, the value of the `average_block_delay` parameter is displayed (the average block creation delay time, in seconds); this parameter is set in *the node configuration file*;
- `poa_round_info` – when using the PoA consensus algorithm, the following parameters are displayed:
    - `round_duration` – block mining round length, in seconds and
    - `sync_duration` – block mining synchronization period, in seconds.
- `crlChecksEnabled` – the certificate revocation list (CRL) check mode during certificate validation.

Note: The same data as the NodeConfig gRPC method returns, can be obtained using the REST methods of the *node* group.

---

gRPC: retrieving data about the node owner

Use the `NodeOwner` method to get data about the owner of a node. The method `NodeOwner` does not require any additional query parameters. The following information on the node that was queried is displayed in the response:

- `address` – node address;

- `public_key` – node public key.

Note: The same data as the NodeOwner gRPC method returns, can be retrieved using the *GET /node/owner* REST method.

See also

> *gRPC tools*

gRPC: obtaining information on the results of the execution of a smart contract call

To obtain information on the results of a smart contract call use the **ContractStatusService** gRPC service.

The service has two methods described in the **util_contract_status_service.proto** protobuf file:

- **ContractExecutionStatuses**,

- **ContractsExecutionEvents**.

Important: The field data types for the request and response are specified in the protobuf file.

Use the **ContractExecutionStatuses** method to retrieve information on the execution results of a particular smart contract call. The method accepts the **ContractExecutionRequest** query that requires the `tx_id` parameter – the ID of the transaction that calls the smart contract whose status information you want to retrieve.

Use the **ContractsExecutionEvents** method to subscribe to a stream with the results of all the smart contracts calls execution. The method requires no input parameters.

Important: In addition to the methods of the ContractStatusService, the following methods of the ContractService can be used to obtain information about the results of the smart contract call execution:

- *GetContractKeys*,

- *GetContractKey*,

- *GetContractBalances*,

- *CalculateAssetId*.

These methods can be used as both contract and public methods.

Information on the results of the execution of a smart contract call

Both **ContractExecutionStatuses** and **ContractsExecutionEvents** methods output the following smart contract data in the response to the query:

- `senderAddress` – the address of the participant who sent the smart contract to the blockchain;
- `senderPublicKey` – the public key of the participant who sent the smart contract to the blockchain;
- `tx_id` – smart contract call transaction ID;
- `Status` – smart contract state:
    - `0` – successfully executed (SUCCESS);
    - `1` – business error, the contract is not executed, the call is rejected (ERROR);
    - `2` – system error during the execution of the smart contract (FAILURE).
- `code` – code of an error that occurred during the smart contract execution (if any);
- `message` – transaction status message; contains additional information about the status specified in the `status` field, for example,

    ```
    "message": "Smart contract transaction successfully mined";
    ```

- `timestamp` – the *Unix Timestamp* (in milliseconds), marking the time of the smart contract call;
- `signature` – transaction signature.

Note: *GET /contracts/status/{id}* REST method returns the same information as ContractExecutionStatuses gRPC method.

See also

>   *gRPC tools*

gRPC: obtaining information about UTX pool size

The `UtxInfo` query about the *UTX pool* size is sent as a subscription: after it is sent, the response from the node comes once per second.

This request requires no additional parameters and is described in the **transaction_public_service.proto** file.

In response to the query the `UtxSize` message is returned, which contains two parameters:

- `size` – UTX pool size in kilobytes;
- `size_in_bytes` – UTX pool size in bytes.

Important: The field data types for the request and response are specified in the protobuf files.

Note: The number of transactions in UTX pool can be retrieved using the *GET /transactions/unconfirmed/size* REST method.

See also

   gRPC tools

gRPC: retrieving certificates

To request a certificate from a node certificate store use the **PkiPublicService** service methods. The methods are described in the **pki_public_service.proto** file.

Note: **PkiPublicService** methods cannot be used in the *opensource* version of the platform.

With these methods you can retrieve a certificate by different fields:

- *GetCertificateByDn(CertByDNRequest)* – retrieve a certificate by its `DN` (distinguished name),
- *GetCertificateByDnHash(CertByDNHashRequest)* – retrieve a certificate by its `DN Hash`,
- *GetCertificateByPublicKey(CertByPublicKeyRequest)* – retrieve a certificate by its `publicKey`,
- *GetCertificateByFingerprint(CertByFingerprintRequest)* – retrieve a certificate by its `fingerprint`.

In the request, these methods take the value of the corresponding certificate field and, optionally, the `plainText` parameter, which determines the format of the response.

If the certificate exists, the node returns the certificate in DER format (as it is recorded in the node certificate store) in the response of each of these methods. If the `plainText` parameter in the method request is set to `true`, then the certificate is returned in plainText format.

If no such certificate exists, then each of these methods returns an error.

Note: The same data as the gRPC PkiPublicService methods return, can be retrieved using the REST methods of the */pki/certificate* group.

Authorization of methods for obtaining certificates

In case of API-KEY, authorization is not required.

In case of OAuth2 authorization, the `user` role in JWT token is required.

Retrieving a certificate by its DN

The **GetCertificateByDn(CertByDNRequest)** method returns the certificate by its distinguished name stored in the `DN` field.

Note: The same data as the GetCertificateByDn(CertByDNRequest) gRPC method returns, can be retrieved using the *GET /pki/certificate/by-dn/%percent-encoded-DN%* REST method.

Retrieving a certificate by its DN hash

The **GetCertificateByDnHash(CertByDNHashRequest)** method returns the certificate by SHA-1 hash (Keccak) from the `DN` certificate field.

Note: The same data as the GetCertificateByDnHash(CertByDNHashRequest) gRPC method returns, can be retrieved using the *GET /pki/certificate/by-dn-hash/%DN-hash-string%* REST method.

Retrieving a certificate by its public key

The **GetCertificateByPublicKey(CertByPublicKeyRequest)** method returns the certificate by its public key stored in the `publicKey` field.

Note: The same data as the GetCertificateByPublicKey(CertByPublicKeyRequest) gRPC method returns, can be retrieved using the *GET /pki/certificate/by-public-key/%public-key-base58%* REST method.

Retrieving a certificate by its fingerprint

The **GetCertificateByFingerprint(CertByFingerprintRequest)** method returns the certificate by its SHA-1 fingerprint stored in its `fingerprint` field.

Note: The same data as the GetCertificateByFingerprint(CertByFingerprintRequest) gRPC method returns, can be retrieved using the *GET /pki/certificate/by-fingerprint/%fingerprint-base64%* REST method.

See also

> *gRPC tools*
>
> *REST API: retrieving certificates*

gRPC: handling transactions

Use the **TransactionPublicService** gRPC service to handle transactions.

The **TransactionPublicService** service has the following methods, described in the **transaction_public_service.proto** protobuf file:

- *Broadcast*;
- *BroadcastWithCerts*;
- *UtxInfo*;
- *TransactionInfo*;
- *UnconfirmedTransactionInfo*.

Important: The field data types for the request and response are specified in the protobuf files.

Sending transactions into the blockchain

Choose the appropriate method for your task to send transactions to the blockchain:

- `BroadcastWithCerts` – to send the *RegisterNode* transaction;
- `Broadcast` – to send all other transactions.

Note: You can also use the *POST /transactions/broadcast* and *POST /transactions/signAndBroadcast* REST methods to send transactions to the blockchain.

Broadcast

The method requires the following query parameters:

- `version` – transaction version;
- `transaction` – the name of the transaction along with the set of parameters intended for it.

For each transaction, there is a separate protobuf file describing the request and response fields. These fields are universal for gRPC and REST API queries and are given in the *Transactions of the blockchain platform* article.

Note: You can also use the *POST /transactions/broadcast* and *POST /transactions/signAndBroadcast* REST methods to send transactions to the blockchain.

BroadcastWithCerts

The method is used to broadcast the *RegisterNode* transaction and requires the same set of incoming parameters as the *Broadcast* method.

The `certificates` field is mandatory and must contain the certificates chain that corresponds to the public key in the transaction `target` field.

Note: You can also use the *POST /transactions/broadcast* and *POST /transactions/signAndBroadcast* REST methods to send transactions to the blockchain.

Retrieving data from a transaction

Use the `TransactionInfo` method to retrieve transaction data.

The method requires the following query parameter:

- `tx_id` – ID of the transaction on which information is being requested.

The `TransactionInfo` method response contains the following transaction data:

- `height` – the blockchain height on which the transaction was made;
- `transaction` – transaction name;

as well as the transaction data similar to that in the `Broadcast` method response.

Retrieving data from a transaction that is in the UTX pool

Use the `UnconfirmedTransactionInfo` method to retrieve data of the transaction held in the UTX pool. The method response contains transaction data similar to the `Broadcast` method response.

See also

gRPC tools

Description of transactions

Mainnet fees

gRPC: handling confidential data

Use the **PrivacyEventsService** and **PrivacyPublicService** gRPC services to handle *confidential data (privacy)*.

Note: You can also use REST methods of the *Privacy* group to handle confidential data.

PrivacyEventsService and PrivacyPublicService methods authorization

PrivacyEventsService and PrivacyPublicService methods authorization:

To use the methods of gRPC API **PrivacyEventsService** and **PrivacyPublicService** services, authorization by api-key or JWT-token is required. Authorization of the methods is implemented as follows:

- in case of api-key authorization, PrivacyApiKey is required;

- in case of OAuth2 authorization, the Privacy role in the JWT token is required.

For each of the methods, you must provide the following data:

- Recipients — userAuth;

- Owners — userAuth;

- Hashes — userAuth;

- GetPolicyItemData — privacyAuth;

- GetPolicyItemInfo — privacyAuth;

- SendData — privacyAuth;

- SendLargeData — privacyAuth,

- forceSync — privacyAuth.

  where

- userAuth — user api-key passed in the 'X-Api-Key' request header OR a JWT token with the *user role* in the 'Authorization' header;

- privacyAuth — privacy user api-key in the 'X-Api-Key' request header OR a JWT token with the *privacy role* in the 'Authorization' header.

On top of that gRPC and REST API authorization is configured in the `auth` section of the *node configuration file*.

PrivacyEventsService

The **PrivacyEventsService** has one method **SubscribeOn**, described in the **privacy_events_service.proto** protobuf file. Use this method to get the stream of events related to receiving or deleting confidential data that comes to the node gRPC interface. To do this, send a `SubscribeOn (SubscribeOnRequest)` request that initializes the subscription to the stream.

Information on receiving or deleting confidential data

After a successful request is sent to the gRPC interface, the following data will be received:

- `policy_id` – confidential data group id;
- `data_hash` – confidential data identifying hash;
- `event_type` – event type; the following types are available:
    - DATA_ACQUIRED – the data is saved in the database;
    - DATA_INVALIDATED – the data is marked for deletion due to lack of activity on it or at rollback.

PrivacyPublicService

The **PrivacyPublicService** service has the following methods, described in the protobuf file **privacy_public_service.proto**:

- *GetPolicyItemData*;
- *GetDataLarge*;
- *GetPolicyItemInfo*;
- *PolicyItemDataExists*;
- *SendData*;
- *SendLargeData*;
- *Recipients*;
- *Owners*;
- *Hashes*;
- *forceSync*.

Important: The field data types for the request and response are specified in the protobuf file.

Note: You can also use REST methods of the *Privacy* group to handle confidential data.

Retrieving confidential data hash sum

Use the **GetPolicyItemData** method to retrieve the policy's confidential data package by the identifying hash. The method requires the following query parameters: `policy_id` (confidential data group id) and `data_hash` (identifying hash). After the request is successfully sent to the gRPC-interface, the hash sum of confidential data is returned.

Note: You can also use the REST *GET /privacy/{policy-id}/hashes* method to retrieve an array of identification hashes of data that are bound to the access group `{policy-id}`.

Downloading big data from a node

Use the **GetDataLarge** method to download big data uploaded using *SendLargeData* from a node. The method requires the following query parameters: `policy_id` (confidential data group id) and `data_hash` (identifying hash). After the successful request the PolicyItemDataResponse data stream is received by the gRPC interface.

Retrieving metadata for a confidential data package

Use the **GetPolicyItemInfo** method to retrieve metadata for the policy's confidential data package by identifying hash. The method requires the following query parameters: `policy_id` (confidential data group id) and `data_hash` (identifying hash). After the successful request the following data will be sent to the gRPC interface:

- `sender` – confidential data sender address;
- `policy_id` – a confidential data group identifier;
- `type` – confidential data type (`file`);
- `info` – file data array:
    - `filename` – name of a file;
    - `size` – file size;
    - `timestamp` – the file placement *Unix Timestamp* (in milliseconds);
    - `author` – file author;
    - `comment` – optional comment to the file;
- `hash` – confidential data identifying hash.

Confidential data package existence verification

Use the **PolicyItemDataExists** method to get information about the presence of the policy's confidential data package packet by the identifying hash. The method requires the following query parameters: `policy_id` (confidential data group id) and `data_hash` (identifying hash). After the successful request, `true` will be sent to the gRPC interface in case the data exists, or `false` will be sent, if the data does not exist.

Sending confidential data to the blockchain

Use the **SendData** method to send *confidential data* (that will be available only to the policy members defined for that data) to the blockchain.

Note: Use the *SendLargeData* method to send data larger than 20 MB.

Note: Use the *SendLargeData* method to send sensitive data stream to the blockchain.

The method requires the following query parameters:
- `sender_address` – blockchain address from which the data should be sent (corresponds to the value of the "privacy.owner-address" parameter in the node configuration file);
- `policy_id` – identifier of the confidential data access group that will have access to the data being sent;
- `data_hash` – identifying sha256-hash of confidential data in base58 format;
- `info` – information about data being sent:
    - `filename` – name of a file,
    - `size` – file size,
    - `timestamp` – data timestamp,
    - `author` – email of the data author,
    - `comment` – optional comment to the file.
- `fee` – transaction fees;
- `fee_asset_id` – optional field used for smart contracts only;
- `atomic_badge` – a label field indicating that the transaction is supported by the atomic transaction;
- `password` – password to access the private key in the node keystore;
- `broadcast_tx` – if `true` is passed, the created *PolicyDataHash* transaction is sent to the blockchain, if `false` is passed, the transaction and Privacy Inventory are not sent; see *below* for details;
- `data` – string containing data in **base64** format.

Note: When sending files via Amazon S3/Minio, the fields `comment`, `author`, `filename` must have ascii characters. This is a Java SDK AWS limitation.

After a successful request is sent to the gRPC interface, the following data will be received:
- `tx_version` – transaction version;
- `tx` – the created PolicyDataHash transaction.

broadcast_tx parameter

To reduce the probability of data delivery errors, it is recommended to set the `broadcast_tx` parameter to `false` if after sending data using **SendData** API method an atomic transaction which contains a *CreatePolicy* and a *PolicyDataHash* is sent.

Note: You can also use the *POST /privacy/sendData* REST method to send confidential data to the blockchain.

Sending confidential data stream to the blockchain

Use the **SendLargeData** method to send a stream of confidential data to the blockchain. The data will only be available to the members of the confidential data access group defined for that data.

Note: Use the *SendData* method to send data smaller than 20 MB.

The method accepts a data stream in the following format as the request:

- `metadata` – metadata for the confidential data package, similar to the input data of the *SendData* method;
- `content` – an array of bytes representing a confidential data package.

After a successful request is sent, the gRPC interface will receive the same data as from the *SendData* method.

Note: You can also use the *POST /privacy/sendDataV2* and *POST /privacy/sendLargeData* REST methods to send confidential data stream or data larger that 20 MB to the blockchain.

Obtaining the addresses of all the members of a confidential data access group

Use the **Recipients** method to get the addresses of all the members of a confidential data access group. The method requires the `policy_id` query parameter – access group identifier. In response, the method returns an array of strings with addresses of the access group members.

Note: You can also use the *GET /privacy/{policy-id}/recipients* REST method to retrieve the addresses of the confidential data access group members.

Obtaining the addresses of the owners of a confidential data access group

Use the **Owners** method to get the addresses of confidential data access group owners. The method requires the `policy_id` query parameter (access group identifier). In response, the method returns an array of strings with addresses of access group owners.

Note: You can also use the *GET /privacy/{policy-id}/owners* REST method to retrieve the addresses of the confidential data access group owners.

Obtaining an array of identification hashes

Use the **Hashes** method to get an array of identification hashes of data that are bound to a confidential data access group. The method requires entering the `policy_id` query parameter (access group identifier). In response, the method returns an array of strings with identity hashes of access group data.

Synchronization of data on the specified confidential data access group

Use the **forceSync** method to synchronize data on the specified sensitive data access group. The method requires the `policy_id` query parameter – access group identifier.

As a result of the method execution, the node starts the synchronization process and returns the size of confidential data in MB. If synchronization fails to start, the node returns an error description.

Note: You can also use *POST /privacy/forceSync* and *GET /privacy/forceSync/{policyId}* REST methods to force a confidential data package retrieval.

See also

> *gRPC tools*
>
> *Precise platform configuration: gRPC and REST API authorization*

gRPC: transfer of confidential smart contract data

The gRPC **ConfidentialContractService** is used to transfer *confidential smart contracts* data.

The following service methods are described in the **confidential_contract_service.proto** protobuf file:

- **ConfidentialCall**,
- **ConfidentialExecutedTxByExecutableTxId**,
- **GetContractKeys**.

Important: The field data types for requests and responses are specified in the protobuf file.

Important: The ConfidentialCall, ConfidentialExecutedTxByExecutableTxId, GetContractKeys methods call is only available when you use an oAuth token with **ConfidentialContractUser** *role* or a special api-key.

ConfidentialCall

The **ConfidentialCall** method accepts the **ConfidentialCallRequest** containing the following fields:

- `broadcast` – the flag that marks the need to broadcast the generated *CallContract* transaction; the field defaults to `true`; the `false` value is used to form an atomic container;

- `commitmentVerification` – the flag that marks the necessity of input data commitment reconciliation and the need for the user to provide a key for commitment generation; the field defaults to `false`; if `false` is set, the node generates the key randomly and calculates the commitment;

- `sender` – the address of the sender of the confidential smart contract data;

- `contractId` – the confidential smart contract identifier;

- `contractVersion` – the confidential smart contract version;

- `params` – when working with *CallContract* transaction, the field holds input data of the confidential smart contract, represented as an array of objects; data is entered using the following fields:

    - `key` – the parameter key;

    - `type` – the parameter data type;

    - `value` – the parameter value.

- `timestamp` – a timestamp in Unix Timestamp format (in milliseconds) marking the time when the smart contract was invoked;

- `atomicBadge` – the flag that marks if the transaction can be included in an *atomic* transaction;

- `fee` – the transaction fee;

- `feeAssetId` – fee token identifier;

- `commitment` – commitment;

- `commitmentKey` – the commitment key.

The ConfidentialCall method takes all the data needed to send a CallContract transaction, sends it, and returns a protobuf that includes the CallContract version 6 transaction and the confidentialInput of the confidential smart contract in the response.

Note: The *POST /confidential-contracts/call* REST method is similar to the ConfidentialCall gRPC method.

ConfidentialExecutedTxByExecutableTxId

The ConfidentialExecutedTxByExecutableTxId method accepts a request containing the `transaction_id` transaction identifier of the confidential smart contract call transaction.

The method returns the transaction of writing the execution result of a confidential smart contract to its state (*105.ExecutedContract* version 4), the confidential input data to run the contract (*ConfidentialInput*) and the confidential results of the contract execution (*ConfidentialOutput*) to the participants of the corresponding *policy* (authorization group).

In turn, 105.ExecutedContract transaction contains all the fields of *103. CreateContract*, *104. CallContract*, *107. UpdateContract* transactions of the smart contract.

Note: The *GET /confidential-contracts/tx/{executable-tx-id}* REST method is similar to the ConfidentialExecutedTxByExecutableTxId gRPC method.

Note: There is a REST method similar to ConfidentialExecutedTxByExecutableTxId, but for regular (non-confidential) smart contracts: *GET /contracts/executed-tx-for/{id}*.

GetContractKeys

The method returns the values of the selected keys from the confidential smart contract state to the members of the corresponding *policy* (authorization group). The result of the smart contract execution is returned as key-value pairs as an array of objects with data type indication.

Note: There is a REST method similar to the GetContractKeys method: *GET /confidential-contracts/{contractId}*.

Note: There is a gRPC method similar to the GetContractKeys method but for regular (non-confidential) smart contracts: GetContractKeys.

See also

gRPC tools

Confidential smart contracts

gRPC: retrieving auxiliary information

Use the **UtilPublicService** service to retrieve auxiliary information.

Obtaining the current node time

The **UtilPublicService** has one method: **GetNodeTime**, described in the **util_public_service.proto** protobuf file. Use this method to get the current node time. The method does not require any additional query parameters.

Important: The field data types for the response are specified in the protobuf files.

The method returns the current node time in two formats:

- `system` – the system time of the node PC;

- `ntp` – network time.

See also

> *gRPC tools* *Auxiliary queries*

gRPC: information about network participants

Use **AddressPublicService** and **AliasPublicService** gRPC services to obtain information about network participants.

gRPC: information about the network members' addresses

Use **AddressPublicService** service to obtain information about the network members' addresses.

The **AddressPublicService** service has the following methods, described in the **address_public_service.proto** protobuf file:

- **GetAddresses**;

- **GetAddressData**;

- **GetAddressDataByKey**.

Important: The field data types for the request and response are specified in the protobuf file.

Note: You can also use the REST methods of the *addresses* group to get information about the network members' addresses.

Retrieving all participants' addresses

Use the **GetAddresses** method to retrieve all the addresses of the participants whose key pairs are stored in the node keystore. The method does not require additional query parameters.

The method returns an array of participants' addresses.

Note: You can also use *GET /addresses* and *GET /addresses/seq/{from}/{to}* REST methods to retrieve the addresses of the network members whose key pairs are stored in the node keystore.

Retrieving data from a specified address

Use the **GetAddressData** method to obtain data written to the specified address using the *transaction 12*. The method requires the following query parameters:

- `address` – address of a node;
- `limit` – the maximum number of records that the method will return;
- `offset` – the number of the records at the given address that the method will skip.

The method returns the data written to the specified address.

Note: You can also use the *GET /addresses/data/{address}* REST method to retrieve data recorded at the specified address using Data Transaction.

Retrieving data from a specified address by a key

Use the **GetAddressDataByKey** method to retrieve data written to the specified address using the *transaction 12*. This key is specified in transaction 12 in the `data.key` field. The method requires the following query parameters:

- `address` – address of a node;
- `key` – key.

The method returns data recorded at the specified address with key `{key}`.

Note: You can also use the *GET /addresses/data/{address}/{key}* REST method to retrieve data recorded at the specified address using Data Transaction with *{key}* key.

gRPC: retrieving information about network participants by alias

Use the **AliasPublicService** service to obtain information about a network participant by his alias.

The **AliasPublicService** service has the following methods, described in the protobuf file **alias_public_service.proto**:

- **AddressByAlias**;
- **AliasesByAddress**.

Note: You can also use the REST methods of the *alias* group to get information about network members by alias.

Retrieving an address by alias

Use the **AddressByAlias** method to retrieve the address by alias. The method requires entering a single query parameter:

- `alias` – network participant alias.

The method returns the participant's address.

Note: You can also use the *GET /alias/by-alias/{alias}* REST method to get the address of a network member by his alias.

Retrieving alias by address

Use the **AliasesByAddress** method to retrieve alias by address. The method requires the address of the network participant in the query.

The method returns all the aliases of the network participant.

Note: You can also use the *GET /alias/by-address/{address}* REST method to get the alias of a network member by his address.

See also

gRPC tools

GET /addresses

alias group

Each of these tasks has its own set of methods packed in the corresponding protobuf files. You can find a detailed description of each set of methods in the articles referenced above.

The node gRPC methods authorization is configured in the `auth` section of the *node configuration file*.

See also

gRPC services used by Docker smart contracts

1.9 REST API methods

The REST API allows users to remotely interact with a node via JSON requests and responses. The API is accessed via the https protocol. The Swagger framework is used as an interface to the REST API.

1.9.1 REST API usage

All REST API method calls are HTTP GET, POST or DELETE requests to URL `https://yournetwork.com/node-N`, containing the corresponding parameter sets.

The platform also provides access to the Swagger `https://yournetwork.com/node-N/api-docs/index.html` interface, which allows you to make and send HTTP requests to the node through the web interface. The desired groups of requests can be selected in the Swagger interface by selecting routes – the URLs to individual REST API methods.

At the end of each route there is an endpoint – a reference to the method.

Example of a UTX pool size query:

The port for listening to the REST API requests is specified in the `api.rest.port` parameter of the *node configuration file*; generally the `6862` port is used.

Almost all REST API methods require authorization by `api-key` or `JWT token` (oAuth2 authorization). The authorization method is set in the *auth* section of the node configuration file.

When authorizing by `api-key`, specify the value of the selected key phrase, and when authorizing by `JWT-token` specify the **access**-token value.

When `JWT token` authorization is used, the user is assigned a role (or several roles), which determines certain REST API methods accessibility. For more details see *Authorization service* and *OAuth2 authorization roles* sections.

1.9.2 What the platform REST API is for

You can use the REST API to perform the following tasks:

- *REST API methods to work with data snapshots*

Each article contains a table with the addresses of the methods as well as the query and response fields of each method.

If the described REST API methods require authorization, there is an icon at the beginning of the article.

If authorization is not required, you will see an icon.

See also

Precise platform configuration: node gRPC and REST API configuration

1.10 Development and usage of smart contracts

The definition and general description of how smart contracts work on the Waves Enterprise blockchain platform are provided in the *Smart contracts* article.

The following are examples of *Docker smart contracts* and *WASM smart contracts* development.

1.10.1 Development and usage of Docker smart contracts

Preparation for development

Before you start developing a smart contract, make sure that you have the Docker containerization software package installed on your machine. The principles of working with Docker are described in the official documentation.

Also make sure that the node you are using is configured for *smart contract execution* . If your node is running in the Mainnet, it is by default configured to install smart contacts from the open repository and has the recommended settings to ensure optimal smart contact execution.

If you are developing a smart contract to run on a private network, deploy your own registry for Docker images and specify its address and credentials on your server in the `remote-registries` block of the *node configuration file*. You can specify multiple repositories in this block if you need to define multiple storage locations for different smart contracts. Also you can configure repository authorization in the node configuration file.

You can also load a Docker contract image from a repository not specified in the node configuration file using 103 CreateContract transaction, which initiates the creation of a smart contract. For more information, see *Development and installation of a smart contract* and *description of the 103 transaction*.

When working in the Mainnet, the Waves Enterprise open registry is pre-installed in the configuration file.

Smart contract development

Waves Enterprise blockchain platform smart contracts can be developed in any programming language you prefer and implement any algorithms. The finished smart contract code is packaged in a Docker image with used **protobuf** files.

Examples of Python smart contract code using gRPC API methods to exchange data with a node, as well as a step-by-step guide on how to create the corresponding Docker images are given in the following articles:

Example of a Docker smart contract with gRPC

This section describes an example of creating a simple Docker smart contract in Python. The smart contract uses gRPC interface to exchange data with a node.

Before you start, make sure that the utilities from the **grpcio** package for Python are installed on your machine:

```
pip3 install grpcio
```

To install and use the gRPC utilities for other available programming languages, see the official gRPC website.

Program description and listing

When a smart contract is initialized using the 103 transaction, the `sum` integer parameter with a value of `0` is set for it.

Whenever a smart contract is called using transaction 104, it returns an increment of the `sum` parameter (`sum + 1`).

Program listing:

```python
import grpc
import os
import sys

from protobuf import common_pb2, contract_pb2, contract_pb2_grpc

CreateContractTransactionType = 103
CallContractTransactionType = 104

AUTH_METADATA_KEY = "authorization"

class ContractHandler:
    def __init__(self, stub, connection_id):
        self.client = stub
        self.connection_id = connection_id
        return

    def start(self, connection_token):
```

(continues on next page)

```python
        self.__connect(connection_token)

    def __connect(self, connection_token):
        request = contract_pb2.ConnectionRequest(
            connection_id=self.connection_id
        )
        metadata = [(AUTH_METADATA_KEY, connection_token)]
        for contract_transaction_response in self.client.Connect(request=request,
↪ metadata=metadata):
            self.__process_connect_response(contract_transaction_response)

    def __process_connect_response(self, contract_transaction_response):
        print("receive: {}".format(contract_transaction_response))
        contract_transaction = contract_transaction_response.transaction
        if contract_transaction.type == CreateContractTransactionType:
            self.__handle_create_transaction(contract_transaction_response)
        elif contract_transaction.type == CallContractTransactionType:
            self.__handle_call_transaction(contract_transaction_response)
        else:
            print("Error: unknown transaction type '{}'".format(contract_
↪transaction.type), file=sys.stderr)

    def __handle_create_transaction(self, contract_transaction_response):
        create_transaction = contract_transaction_response.transaction
        request = contract_pb2.ExecutionSuccessRequest(
            tx_id=create_transaction.id,
        r   esults=[common_pb2.DataEntry(
                    key="sum",
                    int_value=0)]
        )
        metadata = [(AUTH_METADATA_KEY, contract_transaction_response.auth_
↪token)]
        response = self.client.CommitExecutionSuccess(request=request,␣
↪metadata=metadata)
        print("in create tx response '{}'".format(response))

    def __handle_call_transaction(self, contract_transaction_response):
        call_transaction = contract_transaction_response.transaction
        metadata = [(AUTH_METADATA_KEY, contract_transaction_response.auth_
↪token)]

        contract_key_request = contract_pb2.ContractKeyRequest(
            contract_id=call_transaction.contract_id,
            key="sum"
        )
        contract_key = self.client.GetContractKey(request=contract_key_request,
↪ metadata=metadata)
        old_value = contract_key.entry.int_value

        request = contract_pb2.ExecutionSuccessRequest(
            tx_id=call_transaction.id,
            results=[common_pb2.DataEntry(
```

```
                    key="sum",
                    int_value=old_value + 1)]
        )
        response = self.client.CommitExecutionSuccess(request=request,␣
→metadata=metadata)
        print("in call tx response '{}'".format(response))

def run(connection_id, node_host, node_port, connection_token):
    # NOTE(gRPC Python Team): .close() is possible on a channel and should be
    # used in circumstances in which the with statement does not fit the needs
    # of the code.
    with grpc.insecure_channel('{}:{}'.format(node_host, node_port)) as␣
→channel:
        stub = contract_pb2_grpc.ContractServiceStub(channel)
        handler = ContractHandler(stub, connection_id)
        handler.start(connection_token)

CONNECTION_ID_KEY = 'CONNECTION_ID'
CONNECTION_TOKEN_KEY = 'CONNECTION_TOKEN'
NODE_KEY = 'NODE'
NODE_PORT_KEY = 'NODE_PORT'

if __name__ == '__main__':
    if CONNECTION_ID_KEY not in os.environ:
        sys.exit("Connection id is not set")
    if CONNECTION_TOKEN_KEY not in os.environ:
        sys.exit("Connection token is not set")
    if NODE_KEY not in os.environ:
        sys.exit("Node host is not set")
    if NODE_PORT_KEY not in os.environ:
        sys.exit("Node port is not set")

    connection_id = os.environ['CONNECTION_ID']
    connection_token = os.environ['CONNECTION_TOKEN']
    node_host = os.environ['NODE']
    node_port = os.environ['NODE_PORT']

    run(connection_id, node_host, node_port, connection_token)
```

If you want transactions calling your contract to be able to be processed simultaneously, you must pass the `async-factor` parameter in the contract code itself. The contract passes the value of the `async-factor` parameter as part of the `ConnectionRequest` gRPC message defined in the `contract_contract_service.proto` file:

```
message ConnectionRequest {
string connection_id = 1;
int32 async_factor = 2;
}
```

Detailed information about parallel execution of smart contracts.

Authorization of a Docker smart contract with gRPC

To work with *gRPC*, a smart contract needs authorization. For the smart contract to work correctly with API methods, the following steps are performed:

1. The following parameters must be defined in the environment variables of the smart contract:

   - `CONNECTION_ID` - connection identifier passed by the contract when connecting to a node;

   - `CONNECTION_TOKEN` - authorization token passed by the contract when connecting to a node;

   - `NODE` - IP address or domain name of the node;

   - `NODE_PORT` - port of the gRPC service deployed on the node.

The values of the `NODE` and `NODE_PORT` variables are taken from the node configuration file of the *docker-engine.grpc-server* section. The other variables are generated by the node and passed to the container when the smart contract is created.

Development of a Docker smart contract

1. In the directory that will contain your smart contract files, create an ``src`` subdirectory and place the file **contract.py** with the smart contract code in it.

2. In the `src` directory, create a `protobuf` directory and put the following **protobuf** files in it:

   - `contract_contract_service.proto`

   - `data_entry.proto`

These files are placed in the we-proto-x.x.x.zip archive, which can be downloaded from the official GitHub repository of Waves Enterprise.

3. Generate the code of the gRPC methods in Python based on the `contract_contract_service.proto` file:

```
python3 -m grpc.tools.protoc -I. --python_out=. --grpc_python_out=. contract_contract_
→service.proto
```

As a result, two files will be created:

   - `contract_contract_service_pb2.py`

   - `contract_contract_service_pb2_grpc.py`

In the `contract_contract_service_pb2.py` file, change the line `import data_entry_pb2 as data__entry__pb2` as follows:

```
import protobuf.data_entry_pb2 as data__entry__pb2
```

In the same way, change the line `import contract_contract_service_pb2 as contract__contract__service__pb2` in the file `contract_contract_service_pb2_grpc.py`:

```
import protobuf.contract_contract_service_pb2 as contract__contract__service__pb2
```

Then generate an auxiliary file `data_entry_pb2.py` based on the `data_entry.proto`:

```
python3 -m grpc.tools.protoc -I. --python_out=. data_entry.proto
```

All three resulting files must be in the **protobuf** directory along with the source files.

4. Create a **run.sh** shell script, which will run the smart contract code in the container:

```sh
#!/bin/sh

eval $SET_ENV_CMD
python contract.py
```

Place the **run.sh** file in the root directory of your smart contract.

5. Create a **Dockerfile** script file to build and control the startup of your smart contract. When developing in Python, the basis for your smart contract image can be the official Python `python:3.8-slim-buster''` image. Note that the packages ``dnsutils` and `grpcio-tools` must be installed in the Docker container to make the smart contract work.

Dockerfile example:

```dockerfile
FROM python:3.8-slim-buster
RUN apt update && apt install -yq dnsutils
RUN pip3 install grpcio-tools
ADD src/contract.py /
ADD src/protobuf/common_pb2.py /protobuf/
ADD src/protobuf/contract_pb2.py /protobuf/
ADD src/protobuf/contract_pb2_grpc.py /protobuf/
ADD run.sh /
RUN chmod +x run.sh
ENTRYPOINT ["/run.sh"]
```

Place the **Dockerfile** in the root directory of your smart contract.

6. In case you are working in the Waves Enterprise Mainnet, contact the Technical Support team to place your smart contract in the public repository.

If you work on a private network, *build your smart contract yourself and place it in your own registry*.

How a Docker smart contract with gRPC works

Once called, a Docker smart contract with gRPC works as follows:

1. After the program starts, the presence of environment variables is checked.

2. Using the values of the `NODE` and `NODE_PORT` environment variables, the contract creates a gRPC connection with a node.

3. Next, the `Connect` stream method of the gRPC `ContractService` is called. The method receives a `ConnectionRequest` gRPC message, which specifies the connection identifier (obtained from the `CONNECTION_ID` environment variable). The method metadata contains the `authorization` header with the value of the authorization token (obtained from the `CONNECTION_TOKEN` environment variable).

4. If the method is called successfully, a gRPC stream is returned with objects of type `ContractTransactionResponse` for execution. The object `ContractTransactionResponse` contains two fields:

   - `transaction` - a transaction to create or call a contract;

   - `auth_token` - authorization token specified in the `authorization` metadata header of the called method of gRPC services.

If `transaction` contains a *103* transaction, the initial state is initialized for the contract. If `transaction` contains a call transaction (the *104* transaction), the following actions are performed:

- the value of `sum` key (`GetContractKey` method of the `ContractService`) is requested from the node;

- the key value is incremented by one, i.e. `sum = sum + 1`;

- The new key value is saved on the node (`CommitExecutionSuccess` method of the `ContractService`), i.e. the contract state is updated.

See also

Development and usage of smart contracts

gRPC tools

You can use JS Contract SDK Toolkit and Java/Kotlin Contract SDK Toolkit to develop, test and deploy smart contracts in Waves Enterprise public blockchain networks. These toolkits are described in the following sections:

Constructing smart contracts with JS Contract SDK

This section describes **JS Contract SDK Toolkit** – a toolkit for development, testing and deploying smart contracts in Waves Enterprise public blockchain networks. Use the toolkit to fast take off with the Waves Enterprise ecosystem using programming languages such as JavaScript or TypeScript, since smart contracts are deployed in a Docker container.

Smart contracts are often deployed into different environments and networks. For example, you can scaffold local environment based on a sandbox node and deploy contracts to this network for test use-cases.

Deploy your smart contract to different environments using **WE Contract Command line interface (CLI)**.

Requirements

Before you start, make sure that the following software is installed:

- Docker
- Node.js (LTS)

Quickstart

Run the following command in command line to scaffold your new project:

Using npm npx

```
npx create-we-contract YourContractName -t path-to-contract -n package-name
```

or

```
npm create we-contract YourContractName -t path-to-contract -n package-name
```

or using yarn

```
yarn create we-contract YourContractName -t path-to-contract -n package-name
```

This creates your first smart-contract that is ready for development and deployment to the Waves Enterprise blockchain. Now you can run the following command to initialize dependencies and start to develop your project:

```
npm i // or yarn
```

Configuration

The configuration file is used to set up the image name and the contract name to be displayed in the explorer. You can also set the image tag (the `name` property) which will be used to send the contract to the registry in the configuration file.

Add the `contract.config.js` file to the root directory of your project to initialize your contract configuration.

If you scaffolded the project with the `create-we-contract` command as described above in the *Quickstart* section, the configuration is set by default.

Default configuration

An example of default configuration is given below:

```
module.exports = {
  image: "my-contract",
  name: 'My Contract Name',
  version: '1.0.1',
  networks: {
  /// ...
  }
}
```

Network configuration

In the `networks` section, provide specific configuration for your network:

```
module.exports = {
  networks: {
    "sandbox": {
        seed: "#your secret seed phrase" // or get it from env process.env.MY_SECRET_SEED

        // also you can provide
        registry: 'localhost:5000',
        nodeAddress: 'http://localhost:6862',
        params: {
          init: () => ({
            paramName: 'paramValue'
          })
        }
    }
  }
}
```

- **seed** – if you are going to deploy a contract to the sandbox network, provide the contract initiator seed phrase;

- **registry** – if you used a specific Docker registry, provide the registry name;

- **nodeAddress** – provide specific **nodeAddress** to deploy to.

- **params.init** – to specify initialization parameters, set a function.

> **Caution:** DO NOT publish your secret phrases in public repositories.

Deploy contract

Smart contracts are executed once they are deployed in the blockchain. To deploy a contract run the `deploy` command in WE Contract CLI:

```
we-toolkit deploy -n testnet
```

where testnet is the name of the network specified in the configuration file. For example, to deploy a contract to the sandbox network run the following command:

```
we-toolkit deploy -n sandbox
```

Contract SDK Toolkit

Core concepts

The basics of making a contract class is to specify class annotations per method. The most important annotations are:

- **Contract** – register a class as a contract;

- **Action** – register action handler of the contract;

- **State** – decorate the class property to access the contract state;

- **Param** – decorator that maps transaction parameters to the contract class action parameters.

The SDK provides contract templates to which you can add your business logic:

```
@Contract
export class ExampleContract {
    @State state: ContractState;

    @Action
    greeting(@Param('name') name: string) {
        this.state.set('Greeting', `Hello, ${name}`);
    }
}
```

Methods

Methods to manage smart contract state

`ContractState` class exposes useful methods to write to contract state. You can find the list of data types currently available in contract state in the node documentation. Contract SDK supports all the data types currently available in the contract state.

Write

The easiest way to write the state is to use `set` method. This method automatically casts data type.

```
this.state.set('key', 'value')
```

For explicit type casting use the methods in the examples below:

```
// for binary
this.state.setBinary('binary', Buffer.from('example', 'base64'));

// for boolean
this.state.setBool('boolean', true);

// for integer
this.state.setInt('integer', 102);

// for string
this.state.setString('string', 'example');
```

Read

Reading the state is currently asynchronous, and reading behavior depends on the contract configuration.

```
@Contract
export class ExampleContract {
    @State state: ContractState;

    @Action
    async exampleAction(@Param('name') name: string) {
        const stateValue: string = await this.state.get('value', 'default-value');
    }
}
```

> **Caution:**   state.get `` method has no information about the internal state type in runtime. To explicitly cast types use the following methods:  ``getBinary,   getString, getBoo`l, getNum.

Write Actions

The key decorators are `Action` and `Param`.

Init Actions

To describe create contract action set the `onInit` action decorator parameter to `true`.

```
@Contract
export class ExampleContract {
    @State state: ContractState;

    @Action({onInit: true})
    exampleAction(@Param('name') name: string) {

        this.state.set('state-initial-value', 'initialized')
    }
}
```

By default `action` is used as the name of contract method. To set a different action name, assign it to the
`name` parameter of the decorator.

```
@Contract
export class ExampleContract {
    @State state: ContractState;

    @Action({name: 'specificActionName'})
    exampleAction() {
        // Your code
    }
}
```

ontract version update

Use the `update` method to update contract version. The method updates the last deployed contract version.
If no contract is deployed, the method performs no updates.

```
we-cli update -n, --network <char>
```

See also

Development and usage of smart contracts

Constructing smart contracts with Java/Kotlin Contract SDK

Smart contracts

Constructing smart contracts with Java/Kotlin Contract SDK

This section describes **Java/Kotlin Contract SDK Toolkit** – a toolkit for development, testing and deploying Docker smart contracts in Waves Enterprise public blockchain networks. Use the toolkit to fast take off with the Waves Enterprise ecosystem using any of the JVM programming languages, since smart contracts are deployed in a Docker container. You can create a smart contract using any of the JVM languages, such as Java.

Smart contracts are often deployed into different environments and networks. For example, you can scaffold local environment based on a sandbox node and deploy contracts to this network for test use-cases.

All the transaction handling is done via methods of a single class marked with `@ContractHandler` annotation. The methods which implement handling logic are marked with `@ContractInit` (for CreateContractTx) and `@ContractAction` (for CallContractTx).

To deploy your contract, issue *103* и *104* transactions.

Requirements

Before you start developing your smart contracts, make sure that the following software is installed:

- Docker
- JDK 8 or higher

To deploy your smart contracts, the following software should be installed:

- Docker
- JRE 8 or higher

Dependencies

Maven

```
<dependency>
  <groupId>com.wavesenterprise</groupId>
  <artifactId>we-contract-sdk-grpc</artifactId>
  <version>1.0.0</version>
</dependency>
```

Gradle

```
dependencies {
    implementation("com.wavesenterprise:we-contract-sdk-grpc:1.0.0")
}
```

Quickstart

Take the following steps to create your new contract:

Note: All examples below are taken from the Samples .

1. Create contract handler

```
@ContractHandler
public class SampleContractHandler {

    private final ContractState contractState;
    private final ContractTransaction tx;

    private final Mapping<List<MySampleContractDto>> mapping;

    public SampleContractHandler(ContractState contractState, ContractTransaction tx) {
        this.contractState = contractState;
        mapping = contractState.getMapping(
          new TypeReference<List<MySampleContractDto>>() {
          }, "SOME_PREFIX");
        this.tx = tx;
    }
}
```

2. Add @ContractInit and @ContractAction methods to handle contract transactions

```
public class SampleContractHandler {

  // ...

  @ContractInit
  public void createContract(String initialParam) {
    contractState.put("INITIAL_PARAM", initialParam);
  }

  @ContractAction
  public void doSomeAction(String dtoId) {
      contractState.put("INITIAL_PARAM", Instant.ofEpochMilli(tx.getTimestamp().
→getUtcTimestampMillis()));

      if (mapping.has(dtoId)) {
          throw new IllegalArgumentException("Already has " + dtoId + " on state");
      }
      mapping.put(dtoId,
              Arrays.asList(
                      new MySampleContractDto("john", 18),
                      new MySampleContractDto("harry", 54)
```

```
            ));
    }
}
```

3. Dispatch the contract with the specified contract handler and settings

```
public class MainDispatch {
  public static void main(String[] args) {
      ContractDispatcher contractDispatcher = GrpcJacksonContractDispatcherBuilder.
→builder()
              .contractHandlerType(SampleContractHandler.class)
              .objectMapper(getObjectMapper())
              .build();

      contractDispatcher.dispatch();
  }

  private static ObjectMapper getObjectMapper() {
      ObjectMapper objectMapper = new ObjectMapper();
      objectMapper.registerModule(new JavaTimeModule());
      return objectMapper;
  }
}
```

4. Create Docker image

```
FROM openjdk:8-alpine
MAINTAINER Waves Enterprise <>

ENV JAVA_MEM="-Xmx256M"
ENV JAVA_OPTS=""

ADD build/libs/*-all.jar app.jar

RUN chmod +x app.jar
RUN eval $SET_ENV_CMD
CMD ["/bin/sh", "-c", "eval ${SET_ENV_CMD} ; java $JAVA_MEM $JAVA_OPTS -jar app.jar"]
```

5. Push the image to the Docker repository used by WE node mining contract transactions

Publish the image to the registry used by the Waves Enterprise blockchain node. For convenience, you can use the build_and_push_to_docker.sh bash script which builds your smart contract image, pushes it to the specified registry and displays `image` and `imageHash`.

```
./build_and_push_to_docker.sh my.registry.com/contracts/my-awesome-docker-contract:1.0.0
```

6. Sign and broadcast transactions for creating and invoking the published contract

You will need `image` and `imageHash` of the published contract to create it.

CreateContractTx example:

```
{
  "image": "my.registry.com/contracts/my-awesome-docker-contract:1.0.0",
  "fee": 0,
  "imageHash": "d17f6c1823176aa56e0e8184f9c45bc852ee9b076b06a586e40c23abde4d7dfa",
  "type": 103,
  "params": [
      {
         "type": "string",
         "value": "createContract",
         "key": "action"
      },
      {
         "type": "string",
         "value": "initialValue",
         "key": "createContract"
      }
  ],
  "version": 2,
  "sender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX",
  "feeAssetId": null,
  "contractName": "myAwesomeContract"
}
```

To call contract you will need `contractId = CreateContractTx.id`.

CallContractTx example:

```
{
   "contractId": "7sVc6ybnqZr523xWK5Sg7xADsX597qga8iQNAS9f1D3c",
   "fee": 0,
   "type": 104,
   "params": [
     {
     "type": "string",
     "value": "doSomeAction",
     "key": "action"
     },
     {
        "type": "string",
        "value": "someValue",
        "key": "createContract"
     }
   ],
   "version": 2,
   "sender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX",
   "feeAssetId": null,
   "contractVersion": 1
}
```

Notes on usage

Usage with Java 11 and higher

The library has been tested against Java 8, 11 and 17. When using with Java 11 and higher, additional Java options should be specified for the io.grpc to enable optimizations:

```
--add-opens java.base/jdk.internal.misc=ALL-UNNAMED --add-opens=java.base/java.nio=ALL-
↪UNNAMED -Dio.netty.tryReflectionSetAccessible=true
```

A complete example can be found in Dockerfile for Java 17.

See also

Development and usage of smart contracts-
Constructing smart contracts with JS Contract SDK
Smart contracts

WE Contract SDK (Java/Kotlin Contract SDK) Client

This section describes the **WE Contract SDK Client**.The client is used to interact with the contract from the backend code of Java/Kotlin applications.

Main abstractions

- ContractBlockingClientFactory – Factory to create a contract client.
- NodeBlockingServiceFactory – A factory that creates services for interacting with a node.
- TxService – Interface for working with transactions on the node.
- TxSigner – Interface for signing transactions on the node.
- ConverterFactory – Factory for creating services for converting values when working with a state.
- ContractToDataValueConverter – Interface for converting values to DataValue objects.
- ContractFromDataEntryConverter – Interface for converting Data Entry values from a state.
- ContractClientParams – Class intended for the settings of the client being created.
- ContractSignRequestBuilder – The SignRequest(transaction) builder that creates a contract creation object (103rd transaction) or a contract invocation object (104th transaction).

Quickstart

Follow these steps to create WE contract SDK client.

Note: All the examples below are taken from the Waves Enterprise GitHub. In addition, the Waves Enterprise `GitHub repository <https://github.com/waves-enterprise/we-contract-sdk/tree/master/we-contract-sdk-client>`_ provides more examples.

1. Create and configure services to work with the node:

```
val objectMapper = ObjectMapper()
 .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
 .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
 .registerModule(JavaTimeModule())
 .registerModule(
     KotlinModule.Builder()
         .configure(KotlinFeature.NullIsSameAsDefault, true)
         .build()
)
val converterFactory = JacksonConverterFactory(objectMapper)
val feignNodeClientParams = FeignNodeClientParams(
    url = "{node.url}",
    decode404 = true,
    connectTimeout = 5000L,
    readTimeout = 3000L,
    loggerLevel = Logger.Level.FULL,
)
val feignTxService = FeignTxService(
    weTxApiFeign = FeignWeApiFactory.createClient(
        clientClass = WeTxApiFeign::class.java,
        feignProperties = feignNodeClientParams,
    )
)
val feignNodeServiceFactory = FeignNodeServiceFactory(
    params = feignNodeClientParams
)
val contractProperties = ContractProperties(
    senderAddress = "",
    fee = 0L,
    contractId = "contractId",
    contractVersion = 1,
    version = 1,
    image = "image",
    imageHash = "imageHash",
    contractName = "contractName",
)
val contractClientParams = ContractClientParams(localValidationEnabled = true)
val contractSignRequestBuilder = ContractSignRequestBuilder()
    .senderAddress(Address.fromBase58(contractProperties.senderAddress))
    .fee(Fee(0L))
```

(continues on next page)

```
        .contractId(ContractId.fromBase58(contractProperties.contractId))
        .contractVersion(ContractVersion(contractProperties.contractVersion))
        .version(TxVersion(contractProperties.version))
        .image(ContractImage(contractProperties.image))
        .imageHash(Hash.fromHexString(contractProperties.imageHash))
        .contractName(ContractName(contractProperties.contractName))
val contractClientParams = ContractClientParams(localValidationEnabled = true)
```

2. Form transaction data:

```
 val contractSignRequestBuilder = ContractSignRequestBuilder()
.senderAddress(Address.fromBase58(contractProperties.senderAddress))
.fee(Fee(0L))
.contractId(ContractId.fromBase58(contractProperties.contractId))
.contractVersion(ContractVersion(contractProperties.contractVersion))
.version(TxVersion(contractProperties.version))
.image(ContractImage(contractProperties.image))
.imageHash(Hash.fromHexString(contractProperties.imageHash))
.contractName(ContractName(contractProperties.contractName))
```

3. Create a client contract factory and configure it:

```
val contractFactory = ContractBlockingClientFactory(
    contractClass = TestContractImpl::class.java,
    contractInterface = TestContract::class.java,
    converterFactory = converterFactory,
    contractClientProperties = contractClientParams,
    contractSignRequestBuilder = contractSignRequestBuilder,
    nodeBlockingServiceFactory = nodeBlockingServiceFactory,
)
```

4. Create TxSigner:

```
val txServiceTxSigner = TxServiceTxSignerFactory(
  txService = feignTxService,
)
```

5. Create and invoke client methods:

```
val executionContext: ExecutionContext = contractFactory.executeContract(
txSigner = txSigner) { contract ->
    contract.create()
}
```

See also

Constructing smart contracts with Java/Kotlin Contract SDK

Development and usage of smart contracts

Constructing smart contracts with JS Contract SDK

Smart contracts

Uploading a smart contract into a registry

If you work in the Waves Enterprise Mainnet, contact the Waves Enterprise Technical Support team to place your smart contract into the open repository.

When working on a private network, upload the Docker image of the smart contract into your own Docker registry as described below.

Uploading a smart contract into a registry on a private network

1. Start your registry in a container:

```
docker run -d -p 5000:5000 --name my-registry-container my-registry:2
```

2. Navigate to the directory containing the smart contract files and the Dockerfile script file with commands for building the image.

3. Build the image of your smart contract:

```
docker build -t my-contract .
```

4. Specify the image name and its location address in the repository:

```
docker image tag my-contract my-registry:5000/my-contract
```

5. Run the repository container you created:

```
docker start my-registry-container
```

6. Upload your smart contract to the repository:

```
docker push my-registry:5000/my-contract
```

7. Get information about the smart contract. To do this, display the information about the container:

```
docker image ls|grep 'my-node:5000/my-contract'
```

This will give you the ID of the container. Output the information about it with the `docker inspect` command:

```
docker inspect my-contract-id
```

Response example:

```
{
"Id": "sha256:57c2c2d2643da042ef8dd80010632ffdd11e3d2e3f85c20c31dce838073614dd
↪",
"RepoTags": [
    "wenode:latest"
],
"RepoDigests": [],
"Parent":
↪"sha256:d91d2307057bf3bb5bd9d364f16cd3d7eda3b58edf2686e1944bcc7133f07913",
"Comment": "",
"Created": "2019-10-25T14:15:03.856072509Z",
"Container": "",
"ContainerConfig": {
    "Hostname": "",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
```

The `Id` field is the identifier of the Docker image of the smart contract, which is entered in the `ImageHash` field of the 103 transaction when creating the smart contract.

Installing a smart contract into the blockchain

After uploading the smart contract to the repository, publish the contract on the network using the *103. CreateContract* transaction.

To do this, sign the transaction via the blockchain platform *client*, the *sign* REST API method or the *JavaScript SDK* method.

The data returned in the method's response is fed into transaction 103 when it is published.

Below, you will see the examples of signing and sending a transaction using the `sign` and `broadcast` methods. In the examples, the transactions are signed with the key stored in the keystore of the node.

Curl-query to sign transaction 103:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
↪application/json' --header 'X-Contract-Api-Token' -d '    { "
      "fee": 100000000, "
      "image": "my-contract:latest", "
      "imageHash":
↪"7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65", "
      "contractName": "my-contract", "
      "sender": "3PudkbvjV1nPj1TkuuRahh4sGdgfr4YAUV2", "
      "password": "", "
      "params": [], "
      "type": 103, "
      "version": 1 "
  }' 'http://my-node:6862/transactions/sign'
```

The response of the `sign` method, which is passed to the `broadcast` method:

```
{
  "type": 103,
  "id": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky",
  "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
  "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
  "fee": 100000000,
  "timestamp": 1550591678479,
  "proofs": [
↪"yecRFZm9iBLyDy93bDVaNo1PR5Qkkic7196GAgUt9TNH1cnQphq4yGQQ8Fxj4BYA4TaqYVw5qxtWzGMPQyVeKYv
↪" ],
  "version": 1,
  "image": "my-contract:latest",
  "imageHash":
↪"7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65",
  "contractName": "my-contract",
  "params": [],
  "height": 1619
  }
```

Curl-response to sign transaction 103:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
↪application/json' --header 'X-Contract-Api-Token' -d '{ "
{
    "type": 103, \
    "id": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky", \
    "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew", \
    "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M", \
    "fee": 500000, \
    "timestamp": 1550591678479, \
    "proofs": [
↪"yecRFZm9iBLyDy93bDVaNo1PR5Qkkic7196GAgUt9TNH1cnQphq4yGQQ8Fxj4BYA4TaqYVw5qxtWzGMPQyVeKYv
↪" ], \
    "version": 1, \
```

```
    "image": "my-contract:latest", \
    "imageHash":
↪"7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65", \
    "contractName": "my-contract", \
    "params": [], \
    "height": 1619 \
}
}' 'http://my-node:6862/transactions/broadcast'
```

After the *103. CreateContract* transaction, which specifies a reference to the smart contract in the repository, is published, i.e., written to a block of the blockchain during a mining round, network users will be able to invoke that smart contract.

Note: If the smart contract code is updated in the future, the contract will need to be republished. To do this, use the *107. UpdateContract Transaction*.

Important: The smart contract itself is not placed into the blockchain; the blockchain receives the transaction with the hash of the Docker image in which the smart contract code is packaged. Thus the smart contract Docker image hash is stored on all the nodes of the blockchain, but the smart contract itself is in the Docker registry outside the blockchain network.

Smart contract execution

Once a smart contract is installed in the blockchain, it can be invoked with a *104 CallContract Transaction*.

This transaction can also be signed and sent to the blockchain via the blockchain platform client, the `sign` REST API method or the *JavaScript SDK* method. When signing a transaction 104, specify the ID of the 103 transaction for the called smart contract in the `contractId` field (the `id` field of the `sign` method response).

Examples of signing and sending a transaction using the `sign` and `broadcast` methods using a key stored in the keystore of a node:

Curl-query to sign transaction 104:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
↪application/json' --header 'X-Contract-Api-Token' -d '{ "
"contractId": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky", "
"fee": 10, "
"sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew", "
"password": "", "
"type": 104, "
"version": 1, "
"params": [ "
    { "
        "type": "integer", "
        "key": "a", "
```

```
        "value": 1 "
    } "
] "
}' 'http://my-node:6862/transactions/sign'
```

The response of the `sign` method, which is passed to the `broadcast` method:

```
{
"type": 104,
"id": "9fBrL2n5TN473g1gNfoZqaAqAsAJCuHRHYxZpLexL3VP",
"sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
"senderPublicKey": "2YvzcVLrqLCqouVrFZynjfotEuPNV9GrdauNpgdWXLsq",
"fee": 10,
"timestamp": 1549365736923,
"proofs": [

→"2q4cTBhDkEDkFxr7iYaHPAv1dzaKo5rDaTxPF5VHryyYTXxTPvN9Wb3YrsDYixKiUPXBnAyXzEcnKPFRCW9xVp4v
→"
],
"version": 1,
"contractId": "2sqPS2VAKmK77FoNakw1VtDTCbDSa7nqh5wTXvJeYGo2",
"params": [
    {
    "key": "a",
    "type": "integer",
    "value": 1
    }
    ]
  }
```

Curl-query to broadcast the transaction 104:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
→application/json' --header 'X-Contract-Api-Token' -d '{ "
"type": 104, "
"id": "9fBrL2n5TN473g1gNfoZqaAqAsAJCuHRHYxZpLexL3VP", "
"sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58", "
"senderPublicKey": "2YvzcVLrqLCqouVrFZynjfotEuPNV9GrdauNpgdWXLsq", "
"fee": 10, "
"timestamp": 1549365736923, "
"proofs": [ "

→"2q4cTBhDkEDkFxr7iYaHPAv1dzaKo5rDaTxPF5VHryyYTXxTPvN9Wb3YrsDYixKiUPXBnAyXzEcnKPFRCW9xVp4v
→" "
], "
"version": 1, "
"contractId": "2sqPS2VAKmK77FoNakw1VtDTCbDSa7nqh5wTXvJeYGo2", "
"params": [ "
    { "
    "key": "a", "
    "type": "integer", "
    "value": 1 "
```

```
    } "
] "
}' 'http://my-node:6862/transactions/broadcast'
```

1.10.2 Development and usage of WASM smart contracts

This section provides an example of the development of a WASM smart contract using Rust CDK. The Rust CDK is a set of libraries and utilities that provide an eDSL for writing smart contracts in the Rust language.

Preparation for development

To get started, you need to have Rust and Cargo installed on your system.

Cargo-we installation

To install *cargo-we*, run the following command:

```
cargo install --git https://github.com/waves-enterprise/we-cdk.git --force
```

Use *–force* to install the latest version of the utility.

Creating a project

To create a project, use the `cargo we new <NAME>` command, for example:

```
cargo we new flipper
```

This command will create a `flipper` folder in your working directory. The following files will be created in the folder:

- Cargo.toml – a file containing the project metadata needed for the build;
- lib.rs – the source code file for the contract;
- .gitignore – a file, that specifies file ignore for git.

An example contract, Flipper, will be created in the lib.rs file.

Building the project

To build the project, run the following command:

```
cargo we build
```

WASM smart contract example ˘ Flipper

Flipper is a simple smart contract containing only one `bool` value. The contract provides a method that changes its value from `true` to `false` and vice versa. Below is the code of the contract using CDK.

```
use we_cdk::*;

// Объявление функции, доступной для вызова.
// Для этого используется ключевое слово - #[action].
// _constructor - обязательный метод, который вызывается при CreateContract Transaction.
#[action]
fn _constructor(init_value: Boolean) {
    // Данная функция устанавливает значение, полученное аргументом функции, по ключу
→"value".
    set_storage!(boolean :: "value" => init_value);
}

#[action]
fn flip() {
    // Читаем значение по ключу.
    let value: Boolean = get_storage!(boolean :: "value");
    // Записываем значение обратное полученному.
    set_storage!(boolean :: "value" => !value);
}
```

CDK fundamentals

Types

CDK uses types similar to the types available for storage in the contract state:

- Integer

- Boolean

- Binary

- String

Called functions

To make a function available for external call, specify the `action` attribute:

```
#[action]
fn flip() {
...
```

Called functions must not return values. By default, all functions are not accessible for external call.

Contract constructor

Any contract must have a contract constructor function. This function is called in CreateContract Transaction. The function must be named `_constructor`.

```
#[action]
fn _constructor() {
...
```

This method is used to initialize the contract when it is placed in the network. Most often – to set starting values, roles and so on.

The function must also be marked with the `action` attribute. The presence or absence of arguments depends on the logic of your constructor.

Main components of the we-cdk

crates/cargo-we

The utility is designed to work with smart contracts: project creation and biuld, utilities for WASM and WAT.

crates/cdk

Rust library to write WASM smart contracts.

crates/codegen

WEVM bindings and algorithms for intermediate representation.

crates/proc-macro

Procedure macros to generate code for WASM contracts.

examples

Contract examples.

See also

Smart contracts

General platform configuration: execution of smart contracts

gRPC services used by Docker smart contracts

1.11 JavaScript SDK

JavaScript SDK is an application integration library for the Waves Enterprise platform. It solves a wide range of tasks related to signing and sending transactions to the blockchain.

JavaScript SDK supports:

- operation in a browser, as well as in the Node.js environment;

- signing all types of Waves Enterprise platform transactions;

- operations with seed phrases: creating a new phrase, creating from an existing phrase, encryption;

- client implementation of the node `crypto/encryptCommon`, `crypto/encryptSeparate`, `crypto/decrypt` methods.

The JavaScript SDK uses the *node REST API methods* to work with the blockchain. However, applications written with this library do not interact with the blockchain directly, but sign transactions locally – in a browser or in the Node.js environment. After local signing, the transactions are sent to the network. This way of interaction allows the development of multilayer applications and services that interact with the blockchain.

Data from the application is transmitted and received in *json* format over the HTTPS protocol.

The general chart of JavaScript SDK operation:
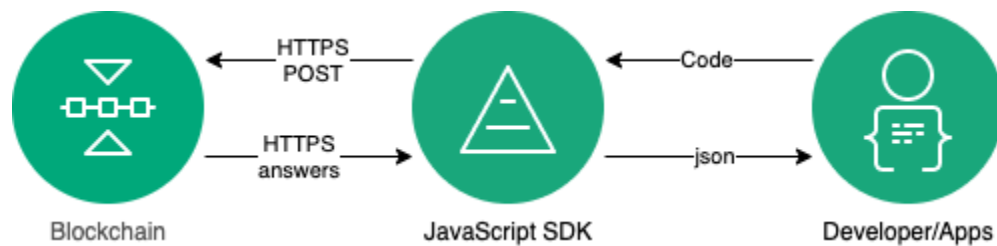
The JavaScript SDK package, as well as the instructions for installing it, are available at the Waves Enterprise GitHub repository.

JavaScript SDK installation and operation are described in more detail in the following sections:

1.11.1 How the JavaScript SDK works

Authorization in the blockchain

In order for an application user to interact with the blockchain, the user must be authorized on the network. To do this, the JavaScript SDK provides authorization service REST API methods that allow you to make a multi-level algorithm with all possible types of queries related to user authorization in the blockchain.

Authorization can be done both in the browser and in the Node.js environment.

When authorizing in a browser, the **Fetch API** interface is used.

For authorization via Node.js, the **Axios** HTTP client is used.

If the blockchain node used by the application uses the oAuth authorization method, it is recommended to use the **api-token-refresher** library for its authorization. This library automatically updates access tokens when their usage time expires. For more information about the oAuth authorization and the api-token-refresher library, see the *Using the JS SDK in a node with oAuth authorization* section.

Seed phrase generation

The JS SDK-based application can work with seed phrases in the following variants:

- create a new randomized seed phrase;

- create a seed phrase from an existing phrase;

- encrypt the seed phrase with a password or decrypt it.

Examples of how the JS SDK works with seed phrases are given in the *Variants of generation of a seed phrase and work with it in the JS SDK* section.

Signing and sending transactions

For JS SDK-based applications, any platform transactions can be signed and sent to the blockchain. A list of all transactions is given in the *Description of transactions*.

The process of signing and sending transactions to the network is as follows:

1. The application initiates generation of a transaction.

2. All transaction fields are serialized into bytecode using the transactions-factory auxiliary component of the JS SDK.

3. The transaction is then signed using the signature-generator component with the user's private key in the browser or in the Node.js environment. The transaction is signed using a POST request `/transactions/sign`.

4. The JavaScript SDK sends a transaction to the blockchain using the POST request `/transactions/broadcast`.

5. The application gets a response in the form of a transaction hash to a POST request.

Examples of signing and sending different types of transactions are given in the *Creating and sending transactions with the use of the JS SDK* section.

Cryptographic node methods used by the JavaScript SDK

Three cryptographic methods are available for the JavaScript SDK:

- `crypto/encryptCommon` – data encryption with a single CEK key for all recipients, which in turn is wrapped by unique KEK keys for each recipient;

- `crypto/encryptSeparate` – separate text encryption with a unique key for each recipient;

- `crypto/decrypt` – data decryption, provided that the key of the message recipient is in the keystore of the node.

See also

> *JavaScript SDK*
>
> *Description of transactions*
>
> *REST API: encryption and decryption methods*

1.11.2 JS SDK installation and initialization

If you are going to use the JS SDK in a Node.js environment, install the Node.js package from the official website.

Install the **js-sdk** package using **npm**:

```
npm install @wavesenterprise/js-sdk --save
```

In the selected development environment, import the package containing the JS SDK library:

```
import WeSdk from '@wavesenterprise/js-sdk'
```

In addition to importing a package, you can use the `require` function:

```
const WeSdk = require('@wavesenterprise/js-sdk');
```

Then initialize the library:

```
const config = {
    ...WeSdk.MAINNET_CONFIG,
    nodeAddress: 'https://hoover.welocal.dev/node-0',
    crypto: 'waves',
    networkByte: 'V'.charCodeAt(0)
}

const Waves = WeSdk.create({
    initialConfiguration: config,
    fetchInstance: window.fetch // Browser feature. For Node.js use node-fetch
});
```

When working in a browser, use the `window.fetch` function as `fetchInstance`. If you work in Node.js, use the module node-fetch.

Once the JavaScript SDK is initialized, you can start creating and sending transactions.

Below is a complete listing with the creation of a typical transaction:

```
import WeSdk from '@wavesenterprise/js-sdk'

const config = {
    ...WeSdk.MAINNET_CONFIG,
    nodeAddress: 'https://hoover.welocal.dev/node-0',
    crypto: 'waves',
    networkByte: 'V'.charCodeAt(0)
}

const Waves = WeSdk.create({
    initialConfiguration: config,
    fetchInstance: window.fetch
});
```

```
// Create a seed phrase from an existing one
const seed = Waves.Seed.fromExistingPhrase('examples seed phrase');

const txBody = {
    recipient: seed.address, // Send tokens to the same address
    assetId: '',
    amount: '10000',
    fee: '1000000',
    attachment: 'Examples transfer attachment',
    timestamp: Date.now()
};

const tx = Waves.API.Transactions.Transfer.V3(txBody);

await tx.broadcast(seed.keyPair)
```

A description of the transaction parameters, as well as examples, is available in the "Creating and sending transactions" section.

See also

> *JavaScript SDK*

1.11.3 Creating and sending transactions with the use of the JS SDK

Principles of transaction creation

Any transaction is called using the function `Waves.API.Transactions.<TRANSACTION_Name>.<TRANSACTION_VERSION>`.

For example, a transaction call for a version 3 token transfer transaction can be done as follows:

```
const tx = Waves.API.Transactions.Transfer.V3(txBody);
```

txBody – transaction body, which contains the necessary parameters. For example, for the above Transfer transaction it may look like this:

```
const tx = Waves.API.Transactions.Transfer.V3(txBody);

{
  "sender": "3M6dRZXaJY9oMA3fJKhMALyYKt13D1aimZX",
  "password": "",
  "recipient": "3M6dRZXaJY9oMA3fJKhMALyYKt13D1aimZX",
  "amount": 4000000000,
  "fee": 100000
}
```

You can leave the transaction body blank and fill in the necessary parameters later by accessing the variable where the result of the transaction call function is returned (in the example, the `tx` variable):

```
const tx = Waves.API.Transactions.Transfer.V3({});
tx.recipient = '12afdsdga243134';
tx.amount = 10000;
//...
tx.sender = "3M6dRZXaJY9oMA3fJKhMALyYKt13D1aimZX";
//...
tx.amount = 40000000000;
tx.fee = 10000;
```

This way of calling a transaction allows more flexibility in making numerical operations in the code and using separate functions to define certain parameters.

3, *13*, *14* and *112* transactions use the `description` text field, and *4* and *6* transactions use the `attachment` text field. Messages sent in these transaction fields need to be converted into **base58** format before being sent. There are two functions in the JS SDK for that:

- ``base58.encode'' – translates the text string into base58 format;
- `base58.decode` – reverse decode the base58 format string into text.

An example of a transaction body using `base58.encode`:

```
const txBody = {
  recipient: seed.address,
  assetId: '',
  amount: 10000,
  fee: minimumFee[4],
  attachment: Waves.tools.base58.encode('Examples transfer attachment'),
  timestamp: Date.now()
}

const tx = Waves.API.Transactions.Transfer.V3(txBody);
```

> **Attention:** When calling transactions with the use of JS SDK, you need to fill all necessary parameters of transaction body except `type`, `version`, `id`, `proofs` and `senderPublicKey`. These parameters are filled in automatically when the key pair is generated.

For a description of the parameters included in the body of each transaction, see *Transaction Description*.

Broadcasting a transaction

The `broadcast` method is used to broadcast a transaction to the network via the JS SDK:

```
await tx.broadcast(seed.keyPair);
```

This method is called after creating a transaction and filling its parameters. The result of its execution can be assigned to a variable to display the result of sending the transaction to the network (in the example, the `result` variable):

```
try {
    const result = await tx.broadcast(seed.keyPair);
    console.log('Broadcast PolicyCreate result: ', result)
} catch (err) {
```

```
      console.log('Broadcast error:', err)
}
```

Below is the full listing of the token transfer transaction call and its broadcasting:

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
  const headers = options.headers || {}
  return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key': 'wavesenterprise
→'} });
}

(async () => {
  const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();

  const wavesApiConfig = {
    ...MAINNET_CONFIG,
    nodeAddress,
    crypto: gostCrypto ? 'gost' : 'waves',
    networkByte: chainId.charCodeAt(0),
  };

  const Waves = createApiInstance({
    initialConfiguration: wavesApiConfig,
    fetchInstance: fetch
  });

  const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

  const txBody = {
    recipient: seed.address,
    assetId: '',
    amount: 10000,
    fee: minimumFee[4],
    attachment: Waves.tools.base58.encode('Examples transfer attachment'),
    timestamp: Date.now()
  }

  const tx = Waves.API.Transactions.Transfer.V3(txBody);

  try {
    const result = await tx.broadcast(seed.keyPair);
    console.log('Broadcast transfer result: ', result)
```

```
  } catch (err) {
    console.log('Broadcast error:', err)
  }

})();
```

For examples of calling and sending other transactions, see "Examples of JavaScript SDK usage" Additional methods available when creating and sending a transaction

In addition to the broadcast method, the following methods are available for debugging and defining transaction parameters:

- `isValid` – transaction body check, returns 0 or 1;

- `getErrors` – returns a string array containing the description of errors made when filling the fields;

- `getSignature` – returns a string with the key with which the transaction was signed;

- `getId` – returns a string with the ID of the transaction to be sent;

- `getBytes` – an internal method that returns an array of bytes to sign.

See also

> *JavaScript SDK*
>
> *Description of transactions*
>
> *Mainnet fees*

1.11.4 Examples of JavaScript SDK usage

Token transfer (4)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
  const headers = options.headers || {}
  return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key': 'wavesenterprise
→'} });
}

(async () => {
  const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();

  const wavesApiConfig = {
    ...MAINNET_CONFIG,
```

```
    nodeAddress,
    crypto: gostCrypto ? 'gost' : 'waves',
    networkByte: chainId.charCodeAt(0),
  };

  const Waves = createApiInstance({
    initialConfiguration: wavesApiConfig,
    fetchInstance: fetch
  });

  // Create Seed object from phrase
  const seed = Waves.Seed.fromExistingPhrase(seedPhrase);


  // see docs: https://docs.wavesenterprise.com/en/latest/how-the-platform-works/data-
→structures/transactions-structure.html#transfertransaction
  const txBody = {
    recipient: seed.address,
    assetId: '',
    amount: 10000,
    fee: minimumFee[4],
    attachment: Waves.tools.base58.encode('Examples transfer attachment'),
    timestamp: Date.now()
  }

  const tx = Waves.API.Transactions.Transfer.V3(txBody);

  try {
    const result = await tx.broadcast(seed.keyPair);
    console.log('Broadcast transfer result: ', result)
  } catch (err) {
    console.log('Broadcast error:', err)
  }

})();
```

Creation of a confidential data group (112)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key':
→'wavesenterprise'} });
```

```
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();

    const wavesApiConfig = {
        ...MAINNET_CONFIG,
        nodeAddress,
        crypto: gostCrypto ? 'gost' : 'waves',
        networkByte: chainId.charCodeAt(0),
    };

    const Waves = createApiInstance({
        initialConfiguration: wavesApiConfig,
        fetchInstance: fetch
    });

    // Create Seed object from phrase
    const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

    // Transaction data
    // https://docs.wavesenterprise.com/en/latest/how-the-platform-works/data-structures/
→transactions-structure.html#createpolicytransaction
    const txBody = {
        sender: seed.address,
        policyName: 'Example policy',
        description: 'Description for example policy',
        owners: [seed.address],
        recipients: [],
        fee: minimumFee[112],
        timestamp: Date.now(),
    }

    const tx = Waves.API.Transactions.CreatePolicy.V3(txBody);

    try {
        const result = await tx.broadcast(seed.keyPair);
        console.log('Broadcast PolicyCreate result: ', result)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

Permission adding and removing (102)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key':
→'wavesenterprise'} });
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();

    const wavesApiConfig = {
        ...MAINNET_CONFIG,
        nodeAddress,
        crypto: gostCrypto ? 'gost' : 'waves',
        networkByte: chainId.charCodeAt(0),
    };

    const Waves = createApiInstance({
        initialConfiguration: wavesApiConfig,
        fetchInstance: fetch
    });

    // Create Seed object from phrase
    const seed = Waves.Seed.fromExistingPhrase(seedPhrase);
    const targetSeed = Waves.Seed.create(15);

    // https://docs.wavesenterprise.com/en/latest/how-the-platform-works/data-structures/
→transactions-structure.html#permittransaction
    const txBody = {
        target: targetSeed.address,
        opType: 'add',
        role: 'issuer',
        fee: minimumFee[102],
        timestamp: Date.now(),
    }

    const permTx = Waves.API.Transactions.Permit.V2(txBody);

    try {
        const result = await permTx.broadcast(seed.keyPair);
        console.log('Broadcast ADD PERMIT: ', result)

        const waitTimeout = 30
```

(continues on next page)

```
        console.log(`Wait ${waitTimeout} seconds while tx is mining...`)

        await new Promise(resolve => {
            setTimeout(resolve, waitTimeout * 1000)
        })

        const removePermitBody = {
            ...txBody,
            opType: 'remove',
            timestamp: Date.now()
        }

        const removePermitTx = Waves.API.Transactions.Permit.V2(removePermitBody);

        const removePermitResult = await removePermitTx.broadcast(seed.keyPair);

        console.log('Broadcast REMOVE PERMIT: ', removePermitResult)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

Smart contract creation (103)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key':
↪'wavesenterprise'} });
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
↪config`)).json();

    const wavesApiConfig = {
        ...MAINNET_CONFIG,
        nodeAddress,
        crypto: gostCrypto ? 'gost' : 'waves',
        networkByte: chainId.charCodeAt(0),
    };
```

```
    const Waves = createApiInstance({
        initialConfiguration: wavesApiConfig,
        fetchInstance: fetch
    });

    // Create Seed object from phrase
    const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

    const timestamp  = Date.now();

    //body description: https://docs.wavesenterprise.com/en/latest/how-the-platform-
→works/data-structures/transactions-structure.html#createcontracttransaction
    const txBody = {
        senderPublicKey: seed.keyPair.publicKey,
        image: 'we-sc/grpc-contract-example:2.1',
        imageHash: '9fddd69022f6a47f39d692dfb19cf2bdb793d8af7b28b3d03e4d5d81f0aa9058',
        contractName: 'Sample GRPC contract',
        timestamp,
        params: [],
        fee: minimumFee[103]
    };

    const tx = Waves.API.Transactions.CreateContract.V3(txBody)

    try {
        const result = await tx.broadcast(seed.keyPair);
        console.log('Broadcast docker create result: ', result)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

Smart contract call (104)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key':
→'wavesenterprise'} });
}
```

```
(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
↪config`)).json();

    const wavesApiConfig = {
        ...MAINNET_CONFIG,
        nodeAddress,
        crypto: gostCrypto ? 'gost' : 'waves',
        networkByte: chainId.charCodeAt(0),
    };

    const Waves = createApiInstance({
        initialConfiguration: wavesApiConfig,
        fetchInstance: fetch
    });

    // Create Seed object from phrase
    const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

    const timestamp  = Date.now()

    //body description: https://docs.wavesenterprise.com/en/latest/how-the-platform-
↪works/data-structures/transactions-structure.html#callcontracttransaction
    const txBody = {
        authorPublicKey: seed.keyPair.publicKey,
        contractId: '4pSJoWsaYvT8iCSAxUYdc7LwznFexnBGPRoUJX7Lw3sh', // Predefined␣
↪contract
        contractVersion: 1,
        timestamp,
        params: [],
        fee: minimumFee[104]
    };

    const tx = Waves.API.Transactions.CallContract.V4(txBody)

    try {
        const result = await tx.broadcast(seed.keyPair);
        console.log('Broadcast docker call result: ', result)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

Atomic transaction (120)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
  const headers = options.headers || {}
  return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key': 'wavesenterprise
→'} });
}

(async () => {
  const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();

  const wavesApiConfig = {
    ...MAINNET_CONFIG,
    nodeAddress,
    crypto: gostCrypto ? 'gost' : 'waves',
    networkByte: chainId.charCodeAt(0),
  };

  const Waves = createApiInstance({
    initialConfiguration: wavesApiConfig,
    fetchInstance: fetch
  });

  // Create Seed object from phrase
  const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

  const transfer1Body = {
    recipient: seed.address,
    amount: 10000,
    fee: minimumFee[4],
    attachment: Waves.tools.base58.encode('Its beautiful!'),
    timestamp: Date.now(),
    atomicBadge: {
      trustedSender: seed.address
    }
  }

  const transfer1 = Waves.API.Transactions.Transfer.V3(transfer1Body);

  const transfer2Body = {
    recipient: seed.address,
    amount: 100000,
    fee: minimumFee[4],
    attachment: Waves.tools.base58.encode('Its beautiful!'),
```

```
    timestamp: Date.now(),
    atomicBadge: {
      trustedSender: seed.address
    }
  }
}

const transfer2 = Waves.API.Transactions.Transfer.V3(transfer2Body);

const dockerCall1Body = {
  authorPublicKey: seed.keyPair.publicKey,
  contractId: '4pSJoWsaYvT8iCSAxUYdc7LwznFexnBGPRoUJX7Lw3sh', // Predefined contract
  contractVersion: 1,
  timestamp: Date.now(),
  params: [],
  fee: minimumFee[104],
  atomicBadge: {
    trustedSender: seed.address
  }
}

const dockerCall1 = Waves.API.Transactions.CallContract.V4(dockerCall1Body);

const dockerCall2Body = {
  authorPublicKey: seed.keyPair.publicKey,
  contractId: '4pSJoWsaYvT8iCSAxUYdc7LwznFexnBGPRoUJX7Lw3sh',
  contractVersion: 1,
  timestamp: Date.now() + 1,
  params: [],
  fee: minimumFee[104],
  atomicBadge: {
    trustedSender: seed.address
  }
}

const dockerCall2 = Waves.API.Transactions.CallContract.V4(dockerCall1Body);

const policyDataText = `Some random text ${Date.now()}`
const uint8array = Waves.tools.convert.stringToByteArray(policyDataText);
const { base64Text, hash } = Waves.tools.encodePolicyData(uint8array)

const policyDataHashBody = {
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "policyId": "9QUUuQ5XetCe2wEyrSX95NEVzPw2bscfcFfAzVZL5ZJN",
  "type": "file",
  "data": base64Text,
  "hash": hash,
  "info": {
    "filename":"test-send1.txt",
    "size":1,
    "timestamp": Date.now(),
    "author":"temakolodko@gmail.com",
    "comment":""
```

```
    },
    "fee": 5000000,
    "password": "sfgKYBFCF@#$fsdf()*%",
    "timestamp": Date.now(),
    "version": 3,
    "apiKey": 'wavesenterprise',
  }
  const policyDataHashTxBody = {
    ...policyDataHashBody,
    atomicBadge: {
      trustedSender: seed.address
    }
  }

  const policyDataHashTx = Waves.API.Transactions.PolicyDataHash.
→V3(policyDataHashTxBody);

  try {
    const transactions = [transfer1, transfer2, policyDataHashTx]
    const broadcast = await Waves.API.Transactions.broadcastAtomic(
      Waves.API.Transactions.Atomic.V1({transactions}),
      seed.keyPair
    );
    console.log('Atomic broadcast successful, tx id:', broadcast.id)
  } catch (err) {
    console.log('Create atomic error:', err)
  }

})();
```

Token issue/burning (3 / 6)

```
const { create: createApiInstance, MAINNET_CONFIG } = require('..');
const nodeFetch = require('node-fetch');

const nodeAddress = 'https://hoover.welocal.dev/node-0';
const seedPhrase = 'examples seed phrase';

const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key':
→'wavesenterprise'} });
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
→config`)).json();
```

```
const wavesApiConfig = {
    ...MAINNET_CONFIG,
    nodeAddress,
    crypto: gostCrypto ? 'gost' : 'waves',
    networkByte: chainId.charCodeAt(0),
};

const Waves = createApiInstance({
    initialConfiguration: wavesApiConfig,
    fetchInstance: fetch
});

// Create Seed object from phrase
const seed = Waves.Seed.fromExistingPhrase(seedPhrase);

const quantity = 1000000

//https://docs.wavesenterprise.com/en/latest/how-the-platform-works/data-structures/
↪transactions-structure.html#issuetransaction
const issueBody = {
    name: 'Sample token',
    description: 'The best token ever made',
    quantity,
    decimals: 8,
    reissuable: false,
    chainId: Waves.config.getNetworkByte(),
    fee: minimumFee[3],
    timestamp: Date.now(),
    script: null
}

const issueTx = Waves.API.Transactions.Issue.V2(issueBody)
try {
    const result = await issueTx.broadcast(seed.keyPair);

    console.log('Broadcast ISSUE result: ', result)
    const waitTimeout = 30
    console.log(`Wait ${waitTimeout} seconds while tx is mining...`)

    await new Promise(resolve => {
        setTimeout(resolve, waitTimeout * 1000)
    })

    const burnBody = {
        assetId: result.assetId,
        amount: quantity,
        fee: minimumFee[6],
        chainId: Waves.config.getNetworkByte(),
        timestamp: Date.now()
    }

    const burnTx = Waves.API.Transactions.Burn.V2(burnBody)
```

```
        const burnResult = await burnTx.broadcast(seed.keyPair);
        console.log('Broadcast BURN result: ', burnResult)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

See also

JavaScript SDK

1.11.5 Using the JS SDK in a node with oAuth authorization

If the node uses the oAuth authorization, it is necessary to initialize the Waves API with the authorization headers for the call.

To automatically update tokens when developing applications with the JS SDK, we recommend using the external module **api-token-refresher**. However, you can use your solution instead.

To work with **api-token-refresher**, install dependencies using **npm**:

```
npm i @wavesenterprise/api-token-refresher@3.1.0 --save, axios --save-dev, cross-fetch --
↪save-dev, @wavesenterprise/js-sdk@3.1.1 --save
```

Initialize **api-token-refresher** as follows:

```
import { init: initRefresher } from '@wavesenterprise/api-token-refresher/dist/fetch'

const { fetch } = initRefresher({
  authorization: {
    access_token,
    refresh_token
  }
});

const Waves = WeSdk.create({
    initialConfiguration: config,
    fetchInstance: fetch
});
```

The `access_token` and `refresh_token` parameters are given in the authorization response to the `loginSecure` request, which is available in the browser.

The following listing contains the initialization of the library followed by the first block check:

```
const WeSdk = require('@wavesenterprise/js-sdk');
const { ApiTokenRefresher } = require('@wavesenterprise/api-token-refresher');

const apiTokenRefresher = new ApiTokenRefresher({
    authorization: {
        access_token: 'access_token',
        refresh_token: 'refresh_token'
    }
})

const { fetch } = apiTokenRefresher.init()

const Waves = WeSdk.create({
    initialConfiguration: {
        ...WeSdk.MAINNET_CONFIG,
        nodeAddress: 'https://hoover.welocal.dev/node-1',
        crypto: 'waves',
        networkByte: 'V'.charCodeAt(0)
    },
    fetchInstance: fetch
});

const testFirstBlock = async () => {
    const data = await Waves.API.Node.blocks.first()
    console.log('First block:', data)
}

testFirstBlock()
```

See also

JavaScript SDK

Authorization and data services

1.11.6 Variants of generation of a seed phrase and work with it in the JS SDK

1. Creating a new randomized seed phrase

```
const seed = Waves.Seed.create();

console.log(seed.phrase); // 'hole law front bottom then mobile fabric under horse drink␣
↪other member work twenty boss'
console.log(seed.address); // '3Mr5af3Y7r7gQej3tRtugYbKaPr5qYps2ei'
console.log(seed.keyPair); // { privateKey: 'HkFCbtBHX1ZUF42aNE4av52JvdDPWth2jbP88HPTDyp4
↪', publicKey: 'AF9HLq2Rsv2fVfLPtsWxT7Y3S9ZTv6Mw4ZTp8K8LNdEp' }
```

2. Creating a seed phrase from an existing one

```
const anotherSeed = Waves.Seed.fromExistingPhrase('a seed which was backed up some time␣
↪ago');

console.log(seed.phrase); // 'newly created seed'
console.log(seed.address); // '3N3dy1P8Dccup5WnYsrC6VmaGHF6wMxdLn4'
console.log(seed.keyPair); // { privateKey: '2gSboTPsiQfi1i3zNtFppVJVgjoCA9P4HE9K95y8yCMm
↪', publicKey: 'CFr94paUnDSTRk8jz6Ep3bzhXb9LKarNmLYXW6gqw6Y3' }
```

3. Encrypting the seed phrase with a password and decrypting it

Example of password encryption of a seed phrase:

```
const password = '0123456789';
const encrypted = seed.encrypt(password);

console.log(encrypted); // 'U2FsdGVkX1+5TpaxcK/
↪eJyjht7bSpjLYlSU8gVXNapU3MG8xgWm3uavW37aPz/
↪KTcROK7OjOA3dpCLXfZ4YjCV3OW2r1CCaUhOMPBCX64QA/iAlgPJNtfMvjLKTHZko/
↪JDgrxBHgQkz76apORWdKEQ=='
```

Example of seed phrase decryption with the use of a password:

```
const restoredPhrase = Waves.Seed.decryptSeedPhrase(encrypted, password);

console.log(restoredPhrase); // 'hole law front bottom then mobile fabric under horse␣
↪drink other member work twenty boss'
```

See also

>   *JavaScript SDK*

See also

>   *Cryptography*
>
>   *REST API: encryption and decryption methods*
>
>   *Transactions of the blockchain platform*

1.12 Confidential data exchange

The Waves Enterprise blockchain platform allows you to restrict access to certain data placed on the blockchain.

To do this, users are combined into groups that have access to sensitive data. One user can be a member of more than one such group. Any member of the group can distribute data to other members of the same group without the data being disclosed to the rest of the blockchain.

Confidential data is transmitted within a group on a peer-to-peer basis. It is not the data itself that is sent to the blockchain, but only the hash of the data. Confidential data is not stored in the blockchain state.

Important: If you are transferring confidential data on your private blockchain network, in order to upgrade from versions older than 1.7.2, you must first upgrade to version 1.7.2 and then to version 1.8 or higher. This requirement is related to the private data exchange protocol modification.

Important: You can also restrict access to data at the smart contract level. For this purpose, *confidential smart contracts* are implemented on the platform.

1.12.1 Creating a confidential data group

Any member of the network can create a confidential data access group (policy).

There are two roles in the group:

- *recipient* – is a member of the data exchange; he can read data from the group and send data to other members of the group;
- *owner* – the administrator of the group; in addition to accessing confidential data, he can change the composition of the group members.

Before you create an access group, decide on the list of members that will be part of it.

Then sign and submit the *112 CreatePolicy* transaction:

1. In the `recipients` field, enter the comma-separated addresses of participants who will have access to confidential data.

2. In the `owners` field, specify the addresses of the owners (administrators) of the access group, separated by commas.

For example:

```
policyName: "Private data exchange 1"
description: "This group is made to share private data..."
recipients: [
    "3AqTkL47j..."
    "5GdYrt9fD...."
]
owners: [
    "8FhBlR12g..."
]
fee: ...
timestamp: ...
```

When you send the transaction, you will receive the transaction ID of the signed CreatePolicy transaction; the same ID is the ID of the created access group (`policyId`). You will need it later to change the composition of the group members.

Once a transaction is sent to the blockchain, all participants registered in the created access group will have access to the confidential data sent to the network.

As the creator of the transaction, you will be able to change the composition of the group, as will the participants added to the `owners` field.

1.12.2 Updating a confidential data group

To change the membership of an access group, the group owner signs and submits the *113 UpdatePolicy* transaction:

1. In the `policyId` field, enter the identifier of the access group to be changed.

2. In the `opType` field, enter the action to be performed on the group:

   - `add` – add members;
   - `remove` – delete members.

3. If you want to add or remove members of an access group, type their public keys in the `recipients` field.

4. To add or remove access group owners, type their public keys in the `owners` field.

Access group information is updated after a transaction is sent to the blockchain.

Only the members of a confidential data group added to the `owners` field during the group creation, as well as its creator himself (**group owners**) can change the composition of the confidential data access group. If there is more than one owner in a group, each owner can change the group independently. That means one signature in the *113 UpdatePolicy* transaction is enough.

When a new member is added to an access group, he can request access to all of the confidential data sent to that group earlier.

1.12.3 Confidential data storage

To receive and send confidential data, you must configure the confidential data storage. Use the *privacy section of the node configuration file* for this purpose.

On the Waves Enterprise blockchain platform you can use the following types of storage for confidential data:

- PostgreSQL (version 8.2and higer)
- Amazon S3/MinIO

Note: Regardless of which storage type is selected, a single data format is used. Thus, members of the same group can use different types of storage.

Once the storage is set up and the group is created, you can send confidential data.

1.12.4 Sending confidential data into the network

Use the following methods to send confidential data into the network:

- gRPC methods
  - *SendData*,
  - *SendLargeData*.
- REST API methods
  - *POST /privacy/sendData*,
  - *POST /privacy/sendDataV2*,

– *POST /privacy/sendLargeData*.

With the **POST /privacy/sendData** and **POST /privacy/sendDataV2** methods, you can send data up to **20 megabytes**. Use the **POST /privacy/sendLargeData** method to send data larger than **20 megabytes**.

When confidential data is sent, its hash is sent to the blockchain network in a separate transaction. Group members can poll other members of the same group after receiving such a transaction.

These methods require authorization.

See also

Precise platform configuration: confidential data groups configuration

PrivacyPublicService

REST API: confidential data exchange and obtaining of information about confidential data groups

Description of transactions

1.13 Role management

All the permissions (roles) of the blockchain platform are described in the *Permissions* article. Permissions can be arbitrarily combined for any address; individual permissions can be revoked at any time.

102 Permission Transaction is provided to manage the participants' roles. The transaction can be signed using the REST API *sign method*, and broadcasted using the corresponding *gRPC* or *REST* API method. The response received from the `sign` method is sent to the `broadcast` method of gRPC or REST node API.

Only a participant with the **permissioner** role can send 102 transactions to the blockchain.

Regardless of the sending method used, the transaction includes the following fields:

- `type` – type of transaction for managing the participants' permissions (`type = 102`);

- `sender` – the address of the participant with authority to send transaction 102 (**permissioner** role);

- `password` – key pair password in the node keystore, optional field;

- `proofs` – transaction signature;

- `target` – the address of the participant whom you want to assign or remove permissions;

- `role` – the member's permission which you want to assign or remove; when sending the transaction via *broadcast* gRPC method, the field specifies the identifying byte of the role; valid values are listed in the *Roles Ids* table below;

- `opType` – type of operation:

  - `add` – add a permission or

  - `remove` – remove a permission;

- `dueTimestamp` – the date of the permission validity in the **Unix Timestamp** format (in milliseconds), optional field.

The following role IDs are used when broadcasting the 102 transaction via *broadcast* gRPC method:

Roles ids

Role	Byte id	prefixS
Miner	1	miner
Issuer	2	issuer
Permissioner	4	permissioner
Blacklister	5	blacklister
Banned	6	banned
ContractDeveloper	7	contract_developer
ConnectionManager	8	connection_manager
Sender	9	sender
ContractValidator	10	contract_validator

See also

Description of transactions

REST API: information about permissions of participants

1.14 Connecting and removing nodes

When working in Waves Enterprise Mainnet, member nodes are connected to the network and removed from it *with the help of Waves Enterprise specialists*.

In a private network, the connection and removal of new members is performed after manual configuration and the start of the first node.

1.14.1 Connecting a new node to a private network

To connect a new node, do the following:

1. Configure the node according to the instructions given in the article *Deploying the platform in a private network*.

2. Send the public key of the new node and its description to administrator of your network.

3. The network administrator (node with the **connection-manager** permission) uses the received public key and node description when creating a transaction *111 RegisterNode*. To register a node, the `opType` parameter, which defines the type of action to be performed, should be specified as `add` (add a new node).

4. The 111 transaction enters the block and then enters the network participants' node state. Thereafter, each member of the network must store the public key and the address of the new node.

5. If necessary, the network administrator can add additional roles to the new node with the *102* transaction. For more information about assigning member roles, see the *Participant role assignment* article.

6. Start the new node.

1.14.2 Removing node from a private network

To remove a node from the network, the network administrator sends a *111 RegisterNode* transaction to the blockchain. In this transaction, he specifies the public key of the node to be removed and the parameter `opType`: ``remove'' (remove the node from the network).

After a transaction is published to the blockchain, the node data is removed from the states of all participants.

See also

> *Description of transactions*
>
> *Role management*
>
> *Architecture*

1.15 Node start with a snapshot

In order to change the parameters of a private blockchain without losing the data stored in it, the Waves Enterprise blockchain platform has a *snapshot mechanism*. For example, the *consensus algorithm* used in the network can be changed using this mechanism. At the same time, addresses and current balances in the network remain unchanged. Also, when the network is restarted using the created data snapshot, the transaction history is reduced to the current state and, accordingly, the size of the state is reduced.

The snapshot mechanism is configured in the configuration file of the node (see the *Precise platform configuration: snapshot*).

After creating a snapshot in the private blockchain, you, as the network administrator, can change its parameters and restart it using the data stored in the snapshot.

To do this, carry out the following:

1. Use the *GET /snapshot/status* method to make sure that the data snapshot was received by your node and successfully verified;

2. Use the *GET /snapshot/genesis-config* method to request the configuration of the new genesis block and save it;

3. Use the *POST /snapshot/swap-state* method to replace the current network state with the data snapshot and wait for a successful response;

4. Prepare the node configuration files to restart:

  • change the genesis block parameters to those obtained in step 2;

  • disable the snapshot mechanism (`node.consensual-snapshot.enable = no`);

  • if necessary, change the parameters of the `blockchain.consensus` section of the node configuration file;

5. Restart the node.

After the node is restarted, a new genesis block of the network will be generated. The network is started with updated parameters and data recorded in the data snapshot.

See also

Snapshooting

Precise platform configuration: snapshot

REST API: information about configuration and state of the node, stopping the node

1.16 Architecture

1.16.1 Platform arrangement

The Waves Enterprise platform is based in the distributed ledger technology and built as a fractal network that consists of two elements:

- **Master blockchain** (Waves Enterprise Mainnet), which provides functioning of the overall network and acts as a global moderator for the basic network, as well as for many user networks;

- individual **sidechains** created for definite business tasks.

Interaction between the master blockchain and sidechains is provided by the anchoring mechanism which broadcasts cryptographic proofs of transaction into the basic blockchain network. The anchoring mechanism allows to freely configure sidechains and use any consensus algorithm without loss of connection with the master blockchain. For instance, the Waves Enterprise master blockchain is based on the Proof-of-Stake consensus algorithm, because it is supported by independent participants. At the same time, corporate sidechains that do not have to stimulate miners with transaction fees can use the Proof-of-Authority or Crash-Fault-Tolerance algorithms.

This two-part arrangement allows to optimize the network for high processing loads, increase information transmission rate, as well as to enhance concurrence and availability of data. Usage of the anchoring mechanism increases trust to data in sidechains, because they are validated in the master blockchain.

Platform architecture scheme:

1.16.2 Arrangement of nodes and auxiliary services

Each blockchain node is an independent network participant which has the software required for work with the network. Every node consists of the following components:

- **Consensus services and cryptolibraries** – components that are responsible for achievement of consensus between nodes and cryptographic algorithms.

- **Node API** – gRPC and REST API interfaces of the node that allow to receive data from the blockchain, sign and broadcast transactions, send confidential data, create and call smart contracts, etc.

- **Unconfirmed transaction pool (UTX pool)** – the component providing storage of unconfirmed transactions before their validation and broadcasting into the blockchain.

- **Miner** – the component responsible for creation of transaction blocks for adding into the blockchain, as well as for interaction with smart contracts.

- **Key store** – storage for key pairs of a node and users. All the keys are protected with the password.

- **Network layer** – the logic layer that provides interaction of nodes at the applied level via the network protocol over the TCP.

- **Node storage** – the system component based on RocksDB that provides storage of 'key-value' pairs for the entire set of confirmed transactions and blocks, as well as for the current blockchain state.
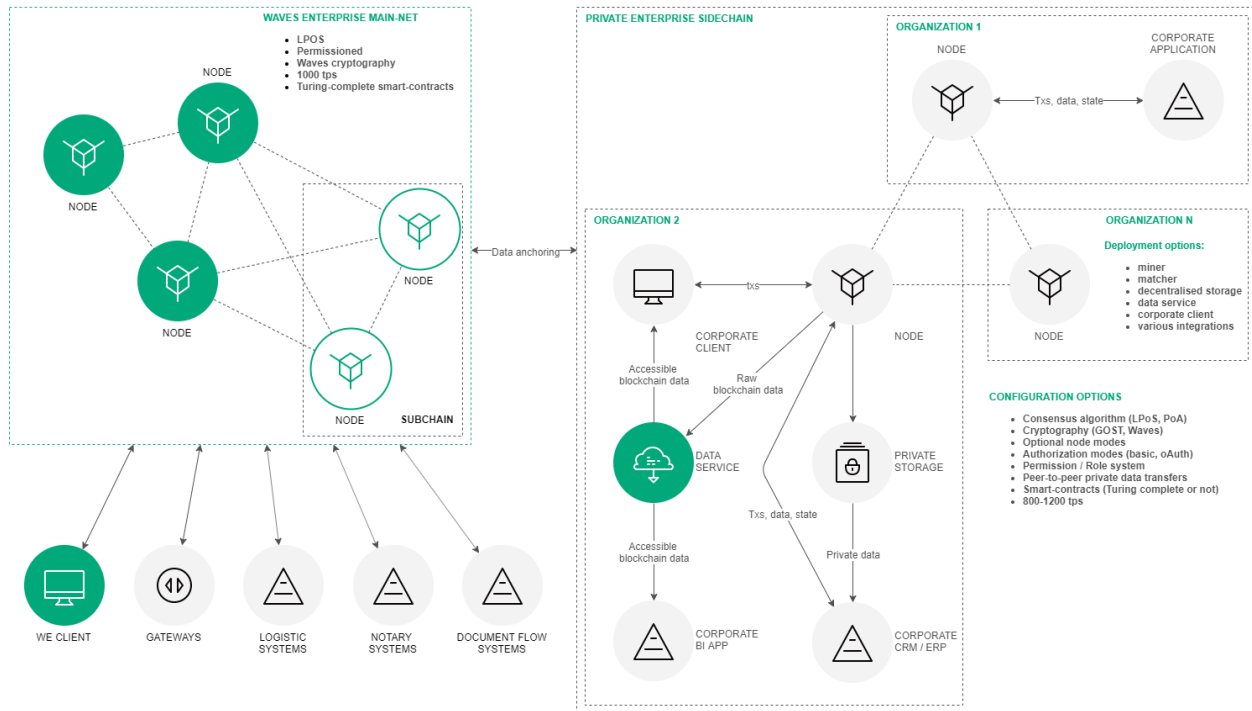
- **Validation logic** – the logic layer containing the rules of transaction validation, for instance, basic signature check and advanced check according to the script.

- **Configuration** – node configuration parameters that are set in the *node-name.conf* file.

Every node also contains a set of additional services:

- **Authorization service** – the service providing authorization of all components.

- **Data crawler** – the service for data extraction from a node and uploading of extracted data into the data service.

- **Generator** – the service for generation of key pairs for new accounts and creating of the `api-key-hash`.

- **Monitoring service** – the external service using the InfluxDB database for storage of time sequences with application data and metrics.

Installation of auxiliary services is not required, but they alleviate interaction of users with the blockchain network. Apart of ready-made services and depending on tasks, integration adapters can be developed for transit of transactions from client applications into the blockchain network, as well as for data exchange between a node and applied services of a customer.

Scheme of node and auxiliary services arrangement:

Waves Enterprise corporate client

Waves Enterprise Node

Waves Enterprise smart contracts

Blockchain explorer

Authorization Service

Refresh/Access tokens

Wallet and asset portfolio

Smart Contract toolkit

Transaction UI

User management

Data analysis and reporting

Monitoring dashboards

Refresh/Access tokens

New txs Blockchain data

Data service

Transforms raw blockchain data into prepared marts in PostgreSQL RDBMS

Prepared blockchain data

API

Data crawler

Data

Prepared blockchain data

Custom microservice plugins

Data

Node REST/gRPC API

Consensus

Cryptographic methods

Cryptographic methods

CryptoProvider

Consensus

Call&Execute contract

Validated unconfirmed txs

Miner

Key store

Unconfirmed transactions pool

Validated unconfirmed txs

Validated unconfirmed txs

Node storage | RocksDB

- Blocks
- Transactions
- State (indexes, balances)

Network layer

Confirmed txs blocks, balances scripts, data

Validated txs

Blocks

Scripts data

Validated blocks

New blocks new txs

Validation logic:
- consensus rules
- smart-accounts scripts
- privacy rules

New txs Private data

Private data

Private data hash txs

Privacy engine

P2P encrypted data exchange

Settings API-key

Private storage | PostgreSQL

- Private data

Consensus, cryptography settings

Network settings

Node and network configuration

Config

Monitoring events and settings

Monitoring service (influxDB, Grafana)

Integration with external systems

See also

1.17 Waves-NG blockchain protocol

Waves-NG is a blockchain protocol developed by Waves Enterprise on the basis of the Bitcoin-NG. The main concept of the protocol is a continuous generation of microblocks instead of one big block in each mining round. This approach allows to increase the blockchain operating speed, because microblocks are validated and transferred into the network much faster.

1.17.1 Description of a mining round

Each mining round consists of the following stages:

1. A used consensus algorithm defines a round miner and the time for generation of a **key block** which does not contain transactions.

2. The round miner generates a key block which contains only service information:

  • public key of the miner for validation of microblock signatures;

  • a miner fee for a previous block;

  • the miner signature;

  • a reference to a previous block.

3. After generating of a key block, the round miner generates a **liquid block**: each 5 seconds the miner generates microblocks with transactions and broadcasts them in the network . At this stage, microblocks are not validated by a consensus algorithm, which increases their generation speed. A first microblock refers to the key block, each subsequent microblock refers to a previous one.

4. The process of generation of microblocks within the liquid block continues up to generation of a next valid key block, which finishes the mining round. At the moment of generation of the next key block, the liquid block with all microblocks generated by the round miner is finalized as a next block of the blockchain.

1.17.2 Miner fee mechanism

The Waves-NG protocol supports financial motivation for miners. Each transaction in the Waves Enterprise blockchain requires a fee in WEST tokens. All fees for transactions in microblocks are summed up during a mining round. A total fee is distributed in the following way:

- a miner of the current round receives **40%** of the total fee for generation of the current block;

- a miner of the next round receives **60%** of the total fee.

The fee charging transaction is carried out for each 100 blocks in order to provide an additional checking interval:
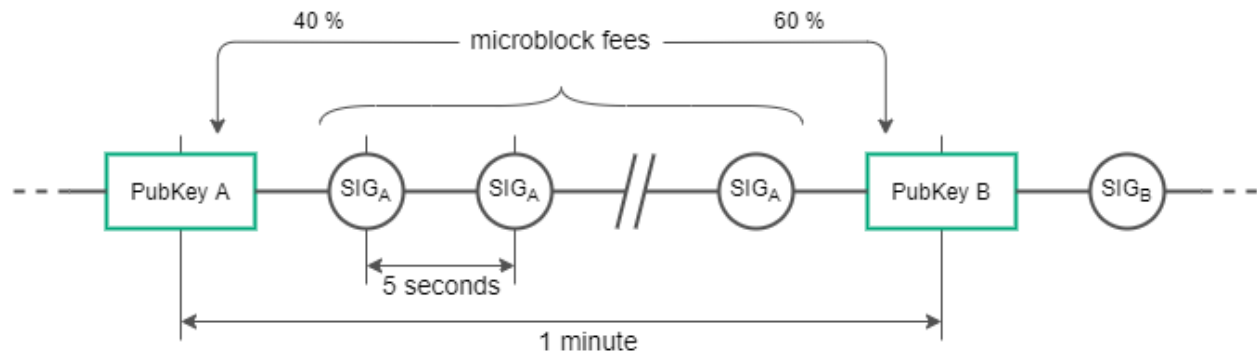
1.17.3 Smart contract validators fee mechanism

The Waves-NG protocol supports financial motivation for smart contract validators. Each *validated* smart contract execution transaction in the Waves Enterprise blockchain requires a fee in WEST tokens that is then transferred to miners and validators. A smart contract is validated if it uses the `Majority` or `MajorityWithOneOf` validation policy. The fee is distributed in the following way:

- **25%** of the smart contract execution transaction fee goes to validators. Remuneration is distributed among the validators in equal shares.

- **75%** of the smart contract execution transaction fee goes to the miners. This amount is distributed among the miners in the following way:

    – a miner of the current round receives 40% of 75%, i.e. **30%** of the total fee for the current block generation;

    – a miner of the next round receives 60% of 75%, i.e. **45%** of the total fee.

1.17.4 Conflict resolution while generating blocks

If a miner continues a previously created blockchain by generating two microblocks with the same parent block, an inconsistency of transaction occurs. It is detected by a blockchain node at the moment of generating of a next microblock, when a node accepts the received changes for its network state copy and synchronizes them with other nodes.

The Waves-NG protocol defines such situation as a fraud. A miner who has continued a foreign chain, is deprived of round transaction fees. A node that has detected an inconsistency receives a miner fee.

Generation and broadcasting of invalid blockchain blocks are also detected by the consensus algorithms.

See also

Architecture

Consensus algorithms

1.18 Data immutability in a blockchain

The blockchain process ensures that data cannot be deleted from the blockchain.

A user generates a transaction. Before sending the transaction, the user generates a digital signature for it using his account private key. This key is known only to the user. After signing, the transaction has a `proofs` field with an electronic signature. Now the body of the transaction is certified, its immutability and belonging to the author (public key) is confirmed.

The user uses POST /transactions/broadcast and POST /transactions/signAndbroadcast requests to send the signed transaction to the API of the node to which he has access.

The node checks the signature, transaction structure, contract, etc. If all checks are correct, the node accepts (validates) the transaction.

The validated transaction goes to the node's UTX pool. This node will then send information about the transaction to all other nodes with which it has a connection. Thus, every network node will have this transaction.

There are two options for a transaction in the UTX pool:

1. the transaction will be added to the block during the mining process, or

2. the transaction will be removed from the UTX pool and will not hit the block.

Each node on the blockchain knows the consensus parameters according to which it should release blocks. The node that is determined to be the leader (the round's miner) selects those transactions from the UTX pool that it is ready to release in a block, checks them again and releases the block.

When releasing a block, the node links the previous block, which is stored in its database, and the new block, including the transactions it contains. To do this, the node specifies in the body of the new released block the signatures of the previous block. Thus the signature of the new block is calculated from the data containing all the transactions of the current block and the signature of the previous block.

If an attacker tries to delete or modify the data of any transaction, the signature of the block it is part of will change. During node synchronization, the block will be sent out to other network members, fail verification and be rejected as invalid.

See also

Architecture

Connecting and removing nodes

1.19 Tokens of the Waves Enterprise blockchain platform

When you use the platform *connected to Mainnet* the WEST system token is used:

1. Each transaction in the Waves Enterprise Mainnet blockchain is charged *a fee in WEST*.

2. Miners and smart contract validators receive a *fee in WEST* for a block creation or a smart contract execution transaction, respectively.

In addition to the system token, you can create and use other tokens – so-called native tokens.

Unlike blockchain platforms where you need to publish a *ERC-20* standard smart contact to create a new token, the Waves Enterprise blockchain network provides the native way to issue tokens via a *token issue transaction*.

After a token issue transaction is accepted by the blockchain network, the issued token can be *transferred* to another network member or *mass transferred*) to multiple network members within a single transaction.

In addition, native tokens can be *reissued* after creation, if the `reissuable` parameter was set to `true` when they were released, and *burned*, which cannot be done with the WEST system token.

A native token can be *sponsored*, that is, provided with the system token. This allows you to pay fees for transactions on the network in native tokens, for example, for marketing purposes to attract new users.

Not only users can manage tokens, but also *smart contracts*.

See also

Description of transactions

1.20 Connection of a new node to blockchain network

The Waves Enterprise blockchain platform gives an opportunity to connect new nodes to a blockchain network at any moment.

Practical steps of node connection are stated in the article *Connection and removing of nodes*.

The general chart for connection of a new node is provided below:

1. A user of a new node passes the public key of the new node to the network administrator (node with the **connection-manager** permission).

2. The node with the **connection-manager** permission uses the received public key for creation of the *111 RegisterNode* transaction with the **"opType": "add"** parameter.

3. The 111 transaction gets to the block.

4. Further, information from the 111 transaction (sender address, new node name and public key) is transferred to states of participant nodes.

5. In case a new node key is absent in the list of nodes that have been registered in the network genesis block (Network Participants), a new node is initially synchronized. A new node sends the **PeerIdentityRequest** with its signature to all addresses from the peer list in its configuration file. Peers make sure that a node that has sent the **PeerIdentityRequest** has been registered in the network.

6. If the check is successful, peers send their public keys to the new node in response to the **PeerIdentityRequest**. The new node saves these public keys in its temporary address storage for primary connection with peers. After saving the addresses, the new node has an opportunity to validate network handshakes from its peers.
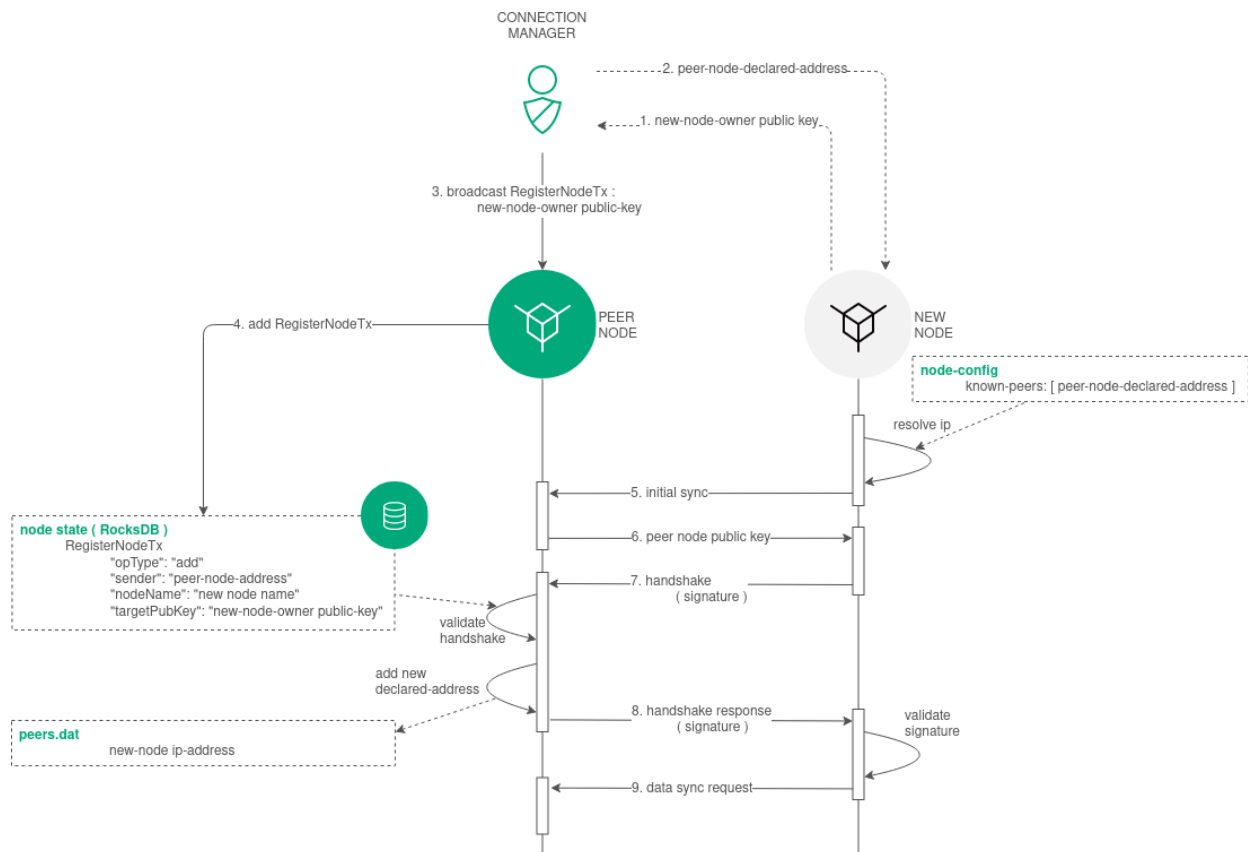
7. The new node sends handshake messages with its public key to network participants from the peer lists in its configuration file.

8. Peers compare the public key in the handshake message and the new node public key from the 111 transaction which has been sent by the node with the **connection-manager** permission. If the check is successful, peers send handshake responses with their signatures to the new node and send the **Peers Messages** to the network.

9. After successful connection, the new node performs synchronization with the network and receives the table with network participant addresses.

See also

Architecture

Connecting and removing nodes

Permissions

1.21 Activation of blockchain features

The Waves Enterprise blockchain platform supports activation of additional blockchain features by voting of nodes – in other words, the **soft fork mechanism**. Soft fork is an irreversible action, because the blockchain does not support a soft fork rollback.

Only nodes with the `miner` role can take part in the voting, because votes of each node are attached to a block created by this node.

1.21.1 Voting parameters

Identifiers of features supported by a node are stated in the `supported` string of the `features` block in the `node` section of the node configuration file:

```
features {
  supported = [100]
}
```

Voting parameters are defined in the `functionality` block of the node configuration file:

- `feature-check-blocks-period` – voting period (in blocks);
- `blocks-for-feature-activation` – number of blocks with a feature identifier required for activation of this feature.

By default, each node is set in a way that it votes for all supported features.

> **Attention:** Voting parameters of a node cannot be changed during blockchain operation: these parameters should be unified for the entire network in order to provide full synchronization of nodes.

1.21.2 Voting procedure

1. During a mining round, a miner node votes for features included in the `features.supported` block, if they have not been activated in the blockchain before: feature identifiers are put into the `features` field of each block during its creation. After that, created blocks are published in the blockchain. So, all the nodes with the `miner` role vote for their features during the `feature-check-blocks-period`.

2. After the `feature-check-blocks-period` elapses, the system counts the votes-identifiers of each feature in the created blocks.

3. If a voted feature collects a number of votes that is greater or equal to the `blocks-for-feature-activation` it gets an **APPROVED** status.

4. The approved feature is activated after the `feature-check-blocks-period` interval starting from a current blockchain height.

1.21.3 Usage of activated features

When activated, a new feature can be used by all blockchain nodes that support it. If any node does not support an activated feature, it will be disconnected from the blockchain in a moment of a first transaction using this unsupported feature.

When a new node is connected to the blockchain, it will automatically activate all previously voted and activated features. Activation is performed during synchronization of the node, if the node itself supports activated features.

1.21.4 Preliminary activation of features

All the features available for voting can also be forcibly activated while starting a new blockchain. Use the `pre-activated-features` block of the `blockchain` section in the node configuration file for this purpose:

```
pre-activated-features = {
  ...
  101 = 0
  }
```

Blockchain height for activation of a certain feature is stated after an equal mark in front of every feature.

1.21.5 List of available feature identifiers

Identifier	Description
100	Activation of the LPoS consensus algorithm
101	Support of gRPC by Docker smart contracts
119	Optimization of performance for the PoA consensus algorithm
120	Support of sponsored fees
130	Optimization of performance for miner ban history
140	Support of atomic transactions
160	Support of parallel creation of liquid blocks and microblocks
162	Validation of smart contracts in the blockchain
173	Support of micro-block inventory v2
180	Support of privacy large object subsystem
190	PKI support v1
1120	Support of *token operations for smart contracts*, PKI support v1 and REST-based smart contracts deprecation
1122	Atomic support for other transactions; see the *Atomic transactions* page for a complete list of transactions
1123	Support of *Lease/CancelLease* operations for smart-contracts
1130	Support of operating over *confidential data in smart contracts*
1140	*WASM smart contracts* support

See also

REST API: information about activation of the platform features

1.22 Anchoring

In a private blockchain, transactions are processed by a definite list of participants, each of participants is familiar for the network in advance. In comparison with the public network, private blockchains contain less participants, blocks and transactions, that can cause a threat of information replacement. This situation, in turn, creates a risk of blockchain override, especially in case the PoS consensus algorithm is used – because this algorithm is not protected from such occurrences.

In order to increase trust of private blockchain participants to the data broadcasted in it, the **anchoring** mechanism has been developed. Anchoring allows to check consistency of data. Consistency of data is guaranteed through broadcasting of data from a private blockchain into a larger network where data replacement is less possible because of larger number of participants and blocks. Block signatures and blockchain height are published from the private network. The mutual connectivity of two or more networks increases their resilience, since all connected networks must be attacked to forge or change data as a result of a long-range attack.

1.22.1 How the Waves Enterprise anchoring works
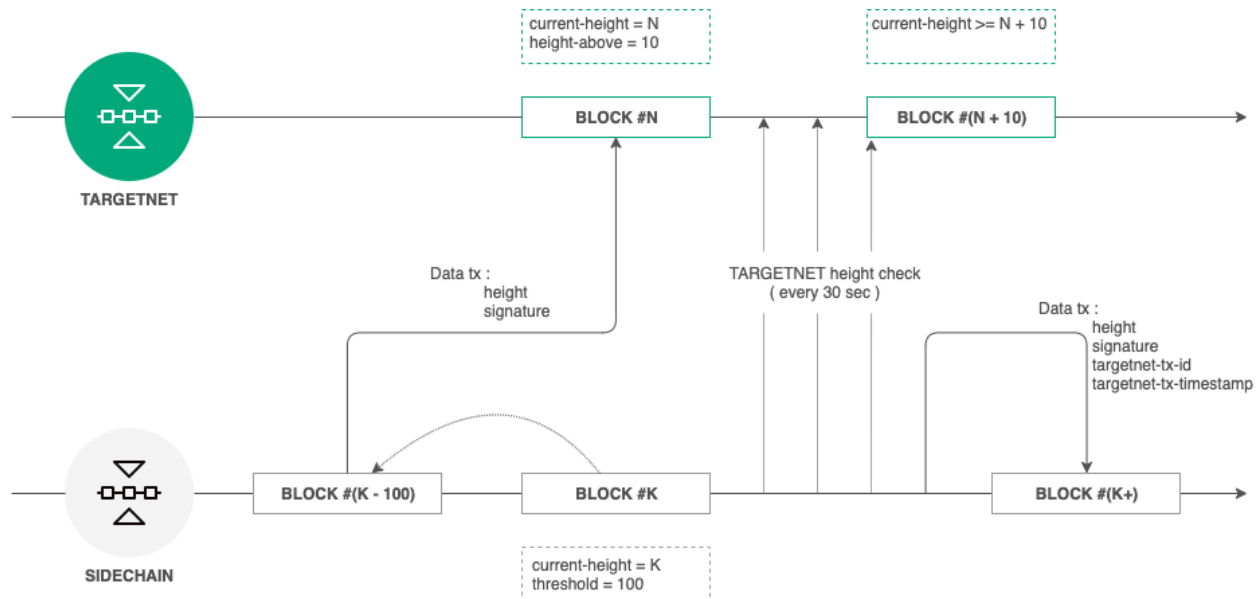
1. *Anchoring configuration* is performed in the private blockchain configuration file (set the corresponding parameters in accordance with the recommendations listed in the article in order to exclude complexities while working with anchoring);

2. After each configured number of blocks `height-range` the node saves information about the block at the `current-height - threshold` in the form of a transaction into the Targetnet. To do this, the *Data Transaction 12* containing the 'key-value' pairs is used. This pairs are described *below*;

3. After transaction broadcast, the node receives its height in the Targetnet;

4. The node checks the Targetnet blockchain each 30 seconds, until the height achieves the value **height of a created transaction** + `height-above`.

5. Upon achieving this Targetnet blockchain height and acknowledgement of presence of the first transaction in the blockchain, the node in the Targetnet creates a second transaction with data for anchoring in the private blockchain.

1.22.2 Anchoring data transaction structure

Transaction for broadcasting in a Targetnet contains following fields:

- `height` – height of a private blockchain block to be saved;
- `signature` – signature of a private blockchain block to be saved.

Transaction for a private blockchain contains following fields:

- `height` – height of a private blockchain block to be saved;
- `signature` – signature of a private blockchain block to be saved;
- `targetnet-tx-id` – identifier of a transaction for anchoring into the Targetnet;
- `targetnet-tx-timestamp` – date and time of creation of the Targetnet anchoring transaction.

1.22.3 Errors that can occur during anchoring

Anchoring errors can occur at any stage. In case of errors in a private blockchain, a *Data Transaction 12* with an error code and description is published. This transaction contains following fields:

- `height` – height of a private blockchain block to be saved;

- `signature` – signature of a private blockchain block to be saved;

- `error-code` – code of an error;

- `error-message` – description of an error.

Table 7: Anchoring error types

Code	Error message	Possible reason
0	`Unknown error`	Unknown error has occurred while broadcasting a transaction into a Targetnet
1	`failed to create a data transaction for targetnet`	Creating of a transaction for broadcasting into a Targetnet has not been completed and returned an error
2	`failed to send the transaction to targetnet`	Broadcasting of a transaction into a Targetnet has not been completed and returned an error (that can occur due to a JSON query error)
3	`invalid http status of response from targetnet transaction broadcast: $responseStatus`	Broadcasting of a transaction into the Targetnet returned a code other than 200
4	`failed to parse http body of response from targetnet transaction broadcast`	Broadcasting of a transaction into a Targetnet returned an unrecognizable JSON query
5	`targetnet returned transaction with id='$targetnetTxId', but it differs from the transaction that was sent (id='$sentTxId')`	Broadcasting of a transaction into a Targetnet returned an identifier that differs from the first transaction
6	`targetnet didn't respond to the transaction info request`	A Targetnet has not responded to a query on transaction information
7	`failed to get current height in targetnet`	The current height of a Targetnet has not been obtained
8	`anchoring transaction in targetnet disappeared after the height rose enough`	Anchoring transaction has not been found in a Targetnet after increase of height to the `height-above` value
9	`failed to create sidechain anchoring transaction`	Failed to broadcast an anchoring transaction in a private blockchain
10	`anchored block in sidechain was changed while waiting for targetnet height rise, looks like a rollback has happened`	While waiting for acknowledgement of a transaction in a Targetnet, a rollback occurred in a private blockchain, a transaction identifier has been changed

See also

   Precise platform configuration: anchoring

1.23 Snapshooting

Snapshooting is an auxiliary mechanism of the blockchain platform which allows to save the data of the working blockchain for a subsequent change of network configuration and starting of the network with the saved data.

The snapshooting mechanism allows to change the blockchain configuration parameters without loss of its data. The process of changing of the network configuration parameters with the use of a snapshot is called **migration**.

A snapshot includes the following data:

- states of network addresses: balances, permissions, keys;

- states of smart contracts created in the network: data received as a result of smart contract calls and attached to them with the use of *105 transactions*;

- data of miners of the previous rounds;

- information of *confidential data access groups*.

A snapshot does not include history of transactions, bans and network blocks.

In the process of migration, a snapshot becomes an initial state of the blockchain network with new parameters, and the network itself is restarted with generation of the new genesis block.

Snapshooting is enabled and configured in the section `node.consensual-snapshot` of the *node configuration file*.

1.23.1 Components of the snapshooting mechanism

SnapshotBroadcaster – the component for broadcasting of the `SnapshotNotification` messages, processing of requests for snapshot generation (`SnapshotRequest`) and subsequent transfer of a ready snapshot. As snapshots can have a large size, the SnapshotBroadcaster process not more than 2 requests simultaneously.

SnapshotLoader – the component that registrates incoming `SnapshotNotification` messages at a node, sends `SnapshotRequest` messages and loads snapshots. If a node receives the `SnapshotNotification` message, the sender address is added to the array of addresses that have the snapshot. After that, the notification is sent to other node peers.

The SnapshotLoader repeatedly checks the address array for presence of an address with a ready snapshot. If such address exists, as well as an open network channel with it, the node sends the `SnapshotRequest` message to this address for download of the snapshot. The response timeout for this message is 10 seconds. If a node with the snapshot does not respond within this timeout, it is excluded from the address array. In this case, the node picks a next address with a ready snapshot and sends a `SnapshotRequest` message to this address.

If the snapshot has been downloaded successfully, it is unpacked and verified with the node state. In case of a successful verification, the node which has received the snapshot sends the `SnapshotNotification` messages to its peers.

SnapshotApiRoute – the REST API interface for snapshot operations.

1.23.2 Generation and broadcasting of a snapshot in an operating blockchain

1. The node appointed for mining at the `snapshot-height` is also appointed for snapshot generation. Snapshot generation starts at the `snapshot-height + 1`, the generated snapshot is saved in the `snapshot-directory`. During the snapshot generation, entering of new transactions into the blockchain UTX pool is blocked. After successful generation of snapshot, the node creates an empty genesis block with the consensus algorithm of a new network (`consensus-type`) and saves it in the snapshot.

2. Upon achievement of the `snapshot-height + wait-blocks-count`, the node which has created the snapshot, archives it and sends the `SnapshotNotification` messages about readiness of the snapshot to its peers.

3. Upon receipt of the `SnapshotNotification`, the nodes initiate the `SnapshotRequest` messages to download a ready snapshot. In case of expiration of snapshot receiving timeout or an error while downloading it, the node picks another peer and requests a snapshot from it.

4. Each node that has received an archive with a snapshot, saves it in the `snapshot-directory`, unpacks it and checks its correctness: compares address balances and keys, checks smart contracts integrity, members and parameters of confidential data access groups, participants' permissions. If the snapshot verification is successful, the node sends the messages about availability of the snapshot (`SnapshotNotification`) to its peers. After this, peers of the node can send it a request for snapshot download.

As a result, the snapshot is downloaded by each node of the network, and verification on the level of each node excludes a possibility of snapshot data spoofing.

After generating of the snapshot, you can start your node with changed configuration parameters and the generated snapshot. Learn more about this in the article *Node start with a snapshot*.

If you connect to the node with an empty state (new node) to the network started from the snapshot, the process of snapshot download will be performed automatically: node automatically connects with peers for snapshot downloading and validation of its own configuration file. See the *Connection of a new node to blockchain network* section for description of the connection process.

1.23.3 Snapshot REST API methods

GET /**snapshot**/**status** – returns an actual snapshot status at the current node:

- `Exists` – the snapshot exists/has been downloaded;
- `NotExists` – the snapshot does not exists/has not been downloaded yet;
- `Failed` – failed to unpack or verify a snapshot;
- `Verified` – the snapshot has been verified successfully.

GET /**snapshot**/**genesis-config** – returns a configuration of a new network genesis block;

POST /**snapshot**/**swap-state** – freezes operation of the mode and switches its state with the snapshot. The query contains a `backupOldState` parameter, that defines if the current state should be saved or removed:

- `true` – save the current state in the``PreSnapshotBackup`` directory of the node;
- `false` – remove the current state.

1.23.4 Network messages

- `SnapshotNotification(sender)` – the message of a node about availability of a snapshot, is sent with a node public key;
- `SnapshotRequest(sender)` – request of a node for downloading of a snapshot, is also sent with a node public key.

See also

Node start with a snapshot

Precise platform configuration: snapshot

REST API methods

1.24 Smart contracts

Smart contract is a separate application which saves its entry data in the blockchain, as well as the output results of its algorithm. The Waves Enterprise blockchain platform supports development and usage of Turing complete smart contracts for creation of high-level business applications.

When a smart contract is launched on a blockchain network, nobody can arbitrarily change its code, spoof it or restrict its operation without interference with the entire network. This aspect allows to provide security of business applications.

Smart contracts can be developed in any programming language and do not have any restrictions for their internal logic.

The Waves Enterprise blockchain platform implements two types of smart contracts:

- *Docker Smart Contracts* – executed in a Docker container,
- *WASM Smart contracts* – executed on a WEVM virtual machine.

Smart contracts use node's API to access the node state for data exchange. Docker smart contracts use the *gRPC* API interface for this purpose. WASM smart contracts access the node's API directly.

Each smart contract has its own balance, which can hold *tokens*. For more details on managing tokens from a Docker smart contract *see below*. Managing tokens from a WASM smart contract is done in the same way.

Each network participant can create and call smart contracts.

A smart contract is created and updated by the same account. This may pose a threat to smart contract security: for example, a leaked mnemonic phrase from the account may allow an attacker to update the contract and withdraw funds. To prevent such situations, you can use the *Smart Account* technology as an additional security feature.

If you need to restrict access to a smart contract, create a *confidential smart contract*. Only the nodes you list when creating the smart contract will be able to call it and receive its result. You will be able to modify this list of nodes in the future. Learn more about confidential smart contracts:

1.24.1 Confidential smart contracts

In some business cases there is a requirement to restrict the use of a certain smart contract and make it available only to some nodes.

For example, an organization collaborates with multiple counterparties. The business logic of this cooperation is implemented as transactions within a single smart contract. However, some details of the smart contract call and its execution result should only be available to the organization and the selected counter-agents. That is, the data transmitted by such a smart contract should be available only to some participants of the network. Confidential smart contracts (CSCs) satisfy this requirement.

When you create a confidential smart contract, you define a group of nodes (policy) that will be able to call the smart contract and receive its results. All the rest network participants will not be able to call the contract and receive its results. The policy administrator can change the list of its members.

When included in an existing policy, a node synchronizes its state with other participants and receives preceding contract performance results. If a node is excluded from the policy, it stops receiving contract performance results.

Note: A node can be a member of any number of policies.

The creation and use of confidential smart contracts is possible starting with release 1.13 after the 1130 *feature activation*.

Note:

The same smart contract can be executed either confidentially or publicly. The confidentiality of the call is determined by the method by which the contract call transaction was sent – *signAnd-Broadcast* (for public smart contracts) or *POST /confidential-contracts/call* or *ConfidentialCall* (for confidential smart contracts).

Important: In addition to confidential smart contracts, the Waves Enterprise platform implements another confidential data exchange technology: *transaction-level access to confidential data*. When using this technology, only users who are members of a certain data access group can access data for certain transactions. Members of the same group can exchange data among themselves by sending it to the network via gRPC or REST APIs; the data will not be disclosed to the rest of the blockchain.

Confidential smart contract data

Confidential smart contracts receive and transmit the following data that require protection:

- ConfidentialInput – an object that describes the confidential input data to run the contract, as well as the key to generate the commitment. ConfidentialInput includes the following fields:
  - `txId` – identifier of the version 6 *104. CallContract* transaction to which the input data refers;
  - `commitmentKey` – the key to form commitment;
  - `param` – input data of the confidential smart contract, represented as an array of objects; entered using the following fields:
    * `key` – the parameter key;
    * `type` – the parameter data type;

* `value`– the parameter value.

- ConfidentialOutput – object describing confidential contract execution results. ConfidentialOutput includes the following fields:

    - `txId` – identifier of the executable transaction to which the output data refers;

    - `entries` – output data of the confidential smart contract, represented as an array of objects, each of which includes the following fields:

        * `key` – the key;

        * `type` – data type;

        * `value`– the entry value.

Policy participants exchange confidential data that is recorded in transactions in encrypted form.

Storage of confidential smart contract data

Confidential smart contract data is stored off-blockchain in a separate database. Only the logic and epy key names are available in the smart contract itself.

Control of confidential smart contract data integrity and its protection

To ensure the integrity and protection of the confidential smart contracts input and output data, an additional security mechanism – cryptographic commitment, or commitment scheme – is implemented. This mechanism includes hidden data transmission and data disclosure phases, and guarantees data consistency.

In addition, there is another mechanism to protect the confidential smart contracts data – non-disclosure to the miners. The node that mines in the current round creates a new block and, accordingly, learns about all new data, including smart contract transactions, before others. In order for a block containing a smart contract transaction to enter the blockchain, a quorum for that transaction must be assembled. In the case of a confidential smart contract, the validation of such transactions takes place within the policy. For this purpose, the policy must have at least three nodes with the `contract-validator` :ref:` role <permissions>`. Due to this requirement, the miner can make sure that the quorum for the transaction has been collected and the consensus is reached, but the data itself remains hidden from the miner.

Creation of a confidential smart contract

To register a confidential smart contract in the blockchain, version 6 of the *103. CreateContract* transaction is used.

When registering a confidential smart contract, you need to set its key parameters:

- set the `isConfidential` field to `true`, thus marking the new contract as confidential;

- define the policy, that is, the set of node addresses that will have access to sensitive data, in the `groupParticipants` field;

- define the policy administrators, that is, the set of node addresses that can modify the lists of participants and policy administrators (groupParticipants and groupOwners), in the `groupOwners` field.

When creating a confidential smart contract, the following conditions must be met:

- To create a confidential smart contract (`isConfidential` is set to `true`), you must specify three or more nodes with the `contract-validator` *role* in the `groupParticipants` field.

- A confidential smart contract (`isConfidential` is set to `true`) cannot work with native tokens, so the `payments` field cannot be used when creating such a contract.

- You cannot pass parameters in the `params` field for a confidential smart contract (`isConfidential` is set to `true`).

- If the `groupParticipants` or `groupOwners` field contains any nodes, the contract is confidential and the `isConfidential` field must be set to `true`.

- The size of the `groupParticipants` and `groupOwners` lists must not exceed 1024 participants.

Once the JSON representation of a CreateContract version 6 transaction has been generated, it must be signed and published in the same way as for a regular (public) smart contract, using one of the following methods:

- *POST /transactions/sign* and *POST /transactions/broadcast*, or

- *POST /transactions/signAndBroadcast*.

Confidential smart contract call

After a confidential smart contract is created and registered on the blockchain using transaction 103. CreateContract as described above, the member of the corresponding policy (node whose address is specified in the CreateContract transaction `groupParticipants` field) can call this smart contract using the *104. Call Contact* transaction and one of the following methods:

- *POST /confidential-contracts/call* REST method

- *ConfidentialCall* gRPC method

Updating confidential smart contracts

Use version 5 of the *107. UpdateContract* transaction to update confidential smart contracts.

When updating a confidential smart contract, a node whose address is specified in the `groupOwners` field can override the policy as follows:

- change the list of node addresses that will have access to confidential data in the `groupParticipants` field; after the list is updated, the `groupParticipants` field must contain at least three participants with the `contract-validator` *role*.

- change the list of policy administrators, that is, the nodes that will be able to modify the lists of participants and policy administrators (groupParticipants and groupOwners), in the `groupOwners` field.

Obtaining confidential smart contract result

You can get information about the confidential smart contract creation or update transaction by the transaction identifier {id} using the *GET /transactions/info/{id}* method. The transaction identifier is specified in the response of the *POST /transactions/sign* or *POST /transactions/signAndBroadcast* methods.

A policy member can retrieve the result of a confidential smart contract execution using the *GET /confidential-contracts/tx/{executable-tx-id}* method.

Note: If a user who is not a member of the policy, tries to get contract data after a confidential smart contract call transaction is mined using the /contracts/executed-tx-for/$txId method, the method response

---

will be missing `results`. Thus, the result of a confidential smart contract execution is hidden from nodes that are not included in the policy.

See also

Smart contracts

REST API: working with confidential smart contracts

gRPC: transfer of confidential smart contract data

Development and usage of smart contracts

General platform configuration: execution of smart contracts

Important: In releases 1.14.0 and 1.15.0, WASM smart contracts do not support *atomic transactions* and *confidential smart contracts*.

The node implements MVCC (Multiversion concurrency control) mechanism – *control of concurrent access to smart contracts state through multiversion*. With this feature the node can execute multiple transactions of any smart contracts in parallel. Data consistency is guaranteed. The MVCC mechanism is implemented in the same way for Docker and WASM smart contracts.

1.24.2 Docker Smart Contracts

A Docker smart contract is executed in a Docker container. This keeps the launch and execution of the smart contract separate from the blockchain platform itself.

A developed smart contract is packed in a Docker image which is stored in the open Waves Enterprise registry. This repository is based on the Docker Registry technology, every smart contract developer has an access to it.

In order to upload your smart contract into the registry, contact the Waves Enterprise technical support service. After your request is approved, your smart contract will be uploaded into the registry, and you will be able to call the smart contract in the platform Client or using a REST API query to your node.

If you are going to use Docker smart contracts in your own private blockchain network, you have to *create your own registry* for smart contract uploading and calling.

Docker smart contract operation chart

The general chart of Docker smart contract operation is provided below:
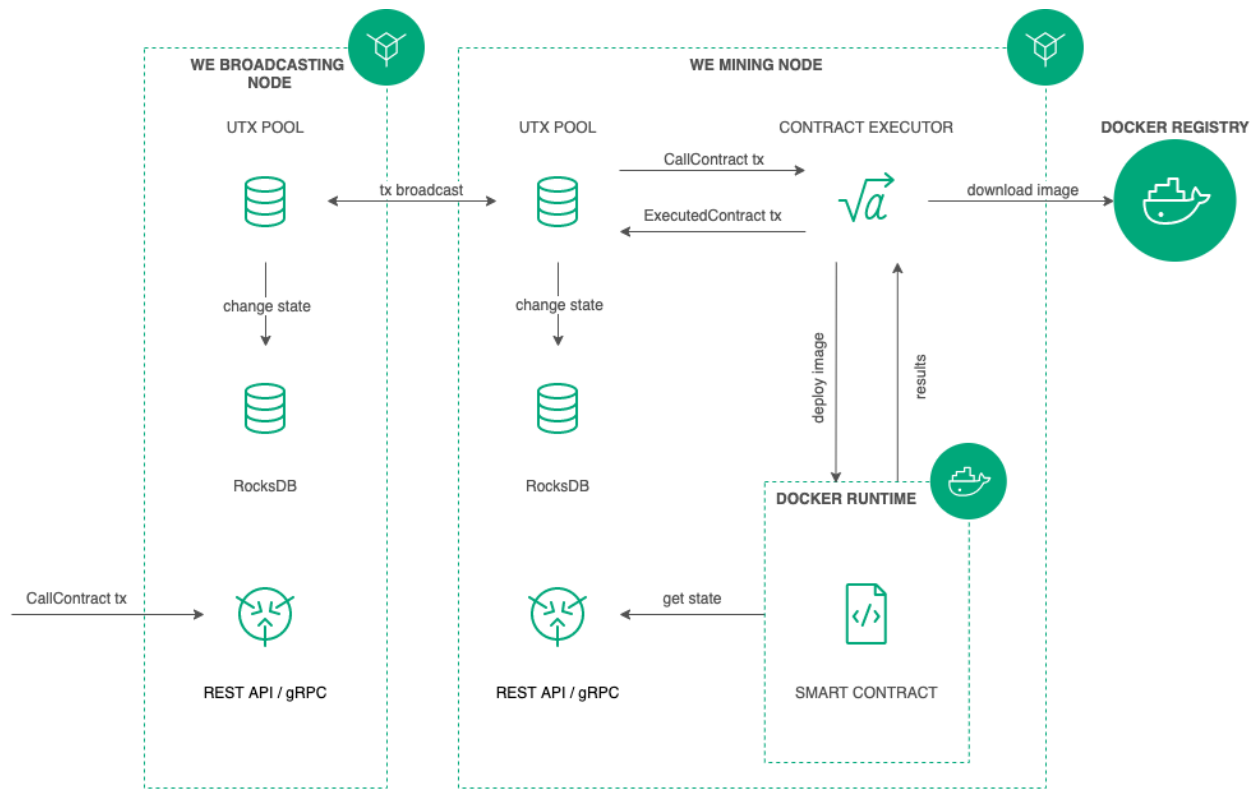
Handling tokens from a Docker smart contract

As of release 1.12, after the *1120 feature activation*, Waves Enterprise blockchain platform smart contracts have their own balance, which can store WEST system tokens as well as other *tokens*. For the smart contracts created on the previous versions, the balance of the WEST system tokens is set to zero.

Also, smart contracts can now perform basic operations with tokens:

- issue tokens,

- reissue tokens,

- burn the tokens on the smart contract balance,

- transfer tokens from the smart contract balance to a user's (users') balance by the user's (users') address.

These functions are implemented by the *CommitExecutionSuccess* gRPC API method.

With this functionality, smart contracts can change the states of assets and users (their balances). Users can also send tokens to a smart contract balance.

Development and installation of Docker smart contracts

Practical instructions on development of smart contracts, as well as an example of a smart contract in Python are listed in the article *Development and usage of smart contracts*.

A participant developing smart contracts should have the **contract_developer** *permission* in the network. This permission allows a participant to upload and call smart contracts, as well as to restrict operation of his own smart contracts and change their code.

Development of a smart contract starts with preparation of a Docker image which contains a ready smart contract code, its **Dockerfile** and, in case of usage of a smart contract with the gRPC interface for data exchange with the node, all required **protobuf** files.

The prepared image is built with the use of the **build** utility of the Docker package, and after this is upload into the registry.

In order to install and work with smart contracts, you have to set up the `docker-engine` section of the *node configuration file*. If your node works in the Waves Enterprise Mainnet, it already has the pre-set parameters for smart contracts installation from the open repository, as well as the recommended parameters for optimal operation of smart contracts.

Installation of smart contracts in the blockchain is performed with the use of the *103 CreateContract Transaction*. This transaction should contain a link to the image of the smart contract in the registry. It is recommended to send the *latest versions* of transactions while working with smart contracts.

In private networks, the 103 transaction allows to install Docker images of smart contracts not only from repositories stated in the `docker-engine` section of the node configuration file. If you need to install a smart contract from a registry not included in the list of the configuration file, type the full address of a smart contract in the registry you have created in the `name` field of the 103 transaction. An example of 103 transaction fields is provided in its *description*.

Upon receiving the 103 transaction, the node downloads the smart contract image specified in the `image` field. After that, the downloaded image is verified by the node and started in the Docker container.

Calling a Docker smart contract and saving the results of its operation

A smart contract is called for operation by a network participant using the *104 CallContract Transaction*.

This transaction transfers the smart contract Docker container id, as well as its input and output parameters in the form of key-value pairs.

The container is started if it has not been started before.

The smart contract is executed and sends the result via the gRPC API interface to the node that initiated the start of the smart contract. The node, in turn, generates a *105 ExecutedContract Transaction* with the result of the smart-contract execution. This way the result of the smart contract execution is recorded in its state by means of the 105 ExecutedContract transaction.

Validator nodes verify that everyone who executed that smart contract with that data got the same result. If the verification is successful, the miner node places the transactions into the block, and the result of the smart contract execution goes into the blockchain.

Restriction of Docker smart contract calls

To disable smart contract calls on the blockchain, submit the *106 DisableContract Transaction* specifying the `contractId` – contract identifier on the blockchain network. This transaction can only be sent by the creator of this smart contract with the non-expired **contract_developer** *permission*.

When disabled, a smart contract becomes unavailable for further calls. The data of disabled smart contracts is stored in the blockchain and can be obtained with the use of gRPC and REST API methods.

Updating Docker smart contracts

If you have changed the code of your smart contract, update it. To do this, upload your smart contract into the Waves Enterprise registry by sending a request for updating of your smart contract to the Waves Enterprise technical support service.

Then send the *107 UpdateContract Transaction* to your node. The contract to be updated should not be disabled with a 106 transaction.

After updating of the smart contract, mining nodes of the blockchain download it and check correctness of its operation. After that, information about update of the smart contract is included into its state with the use of the 105 transaction containing data of the corresponding 107 transaction.

Hint: Only the participant with the **contract_developer** *role* who created the *103 CreateContract Transaction* for this smart contract can modify the contract.

Docker smart contracts validation

The WE blockchain platform supports three smart contract validation policies to provide its additional integrity control. This opportunity is available under following conditions:

- The *162 soft fork* is activated in the network;

- The network includes one or more participants with the **contract_validator** permission;

- The version **4** (and higher) *103* and *107* transactions are used to create and update smart contracts.

The validation policy is configured with the use of the `validationPolicy.type` field of corresponding transaction.

Available validation policies:

- `any` - the general validation policy is kept in the network: to mine the updated smart contract, the miner signs the corresponding *105* transaction. Also, this parameter is set if there are no registered validators in the network.

- `majority` - a transaction is considered valid if it is confirmed by the majority of validators: **2/3** of the total number of registered addresses with the **contract_validator** permission.

- `majorityWithOneOf(List[Address])` - the transaction is considered valid if the majority of validators is collected, among which there is at least one of the addresses included in the parameter list. The addresses included in the list must have a valid **contract_validator** permission.

Warning: If the validation policy `majorityWithOneOf(List[Address])` is selected, the address list must contain at least one address; passing an empty list is not allowed.

Docker smart contracts parallel operation

The Waves Enterprise platform allows to run multiple smart contracts simultaneously. For this purpose, the node implements the MVCC (Multiversion concurrency control) mechanism – parallel access control through multiversion. The mechanism allows multiple transactions of containerized smart contracts to run in parallel and maintain data consistency.

All transactions are divided into two groups:

1. non-executable transactions – *atomic containers* and all the *classic transactions*: transfer transaction, data transaction, etc.;

2. executable transactions – transactions of all containerized smart contracts.

Transactions of the first group are always executed sequentially (the level of parallelism is 1). For the second group of transactions, the execution parallelism is determined by the value of the `node.docker-engine.contracts-parallelism` parameter in the *node configuration*:

```
node.docker-engine.contracts-parallelism = 8
```

The default value is 8. In this way, all smart contracts are executed in parallel, independently of the Docker image.

Note: There is competition between the two groups of transactions: if heterogeneous transactions accumulate in the UTX pool, concurrency may decrease. This behavior can be smoothed by increasing the pulling buffer size, but it cannot be completely eliminated.

The code logic of a smart contract, as well as its programming language, should take into account the peculiarities of parallel operation of smart contracts. For instance, if a smart contract containing a function of increment of a variable upon every smart contract call transaction operates simultaneously, its result will be incorrect, because a common authorization key is used for each smart contract call.

API methods available for Docker smart contracts

The platform provides the *gRPC* API methods to exchange data between a smart contract and a node. Use these methods to perform a wide range of operations with the blockchain.

Learn more:

gRPC services used by Docker smart contracts

The contract gRPC services described in this section are designed to exchange data between a Docker smart contract and a node. These services are only available to Docker smart contracts. An external user cannot call the contract services and use their functions.

Important: The following ContractService methods can be used as both contract and public methods:

- *GetContractKeys*,
- *GetContractKey*,
- *GetContractBalances*,
- *CalculateAssetId*.

General instructions on gRPC usage for Docker smart contracts development are provided in the *Example of a Docker smart contract with gRPC* article.

Versions of Docker smart contract API

gRPC methods (including the methods used by Docker smart contracts) constitute API defined by protobuf files.

To clearly define new methods and make changes to existing ones, API versioning is provided. Thanks to the version number assigned, a node determines the appropriate set of methods to use when executing a Docker smart contract.

The current version of the gRPC API for the blockchain platform version is contained in the **api_version.proto** file. Docker smart contracts that require an API version higher than that of the mining node are ignored during mining.

To create and update Docker smart contracts, *103 CreateContract Transaction* and *107 UpdateContract Transaction* (version 4 and higher) provide `apiVersion` field. This field specifies the API version used by the Docker smart contract for the mining node.

The table below provides API versions corresponding with the versions of the blockchain platform:

API version	Platform version
1.0	1.6.0 and 1.6.1
1.1	1.6.2
1.4	1.7.0
1.6	1.11.0
1.7	1.12.0
1.8	1.12.1
1.9	1.12.2
1.10	1.12.3 and 1.13.0

Protobuf files of the methods

Docker smart contracts that use gRPC to exchange data with a node can use the services listed in the protobuf files with names starting with `contract`:

protobuf	Methods
contract_address_service.	GetAddresses GetAddressData GetAssetBalance
contract_block_service.pr	GetBlockHeader
contract_contract_service	Connect CommitExecutionSuccess CommitExecutionError GetContractKeys GetContractKey GetContractBalances CalculateAssetId
contract_permission_serv	GetPermissions GetPermissionsForAddresses
contract_privacy_service.	GetPolicyRecipients GetPolicyOwners GetPolicyHashes
contract_transaction_serv	TransactionExists TransactionInfo
contract_util_service.prot	GetNodeTime

contract˙address˙service.proto

The **AddressService** methods are designed to retrieve participants' addresses from the node keystore and to retrieve the data stored at the address.

GetAddresses – the method for obtaining all the addresses of participants, whose key pairs are stored in the node keystore. The method returns the `addresses` string array.

GetAddressData – the method for obtaining all the data stored at a definite address with the use of the transaction *12*. The method query contains the following parameters:

- `address` – the address containing the data to be obtained;

- `limit` – the limit of number of data blocks to be obtained;

- `offset` – number of data blocks to be missed in the method response.

The method returns the `DataEntry` array containing the address data.

GetAssetBalance – the method for obtaining the current asset balance for a definite user. The method request includes the following parameters:

- `address` – the address the balance of which is to be displayed;

- `assetId` – asset identifier. This parameter is left blank for WEST.

contract˙block˙service.proto

The **BlockService** methods enable contracts to request information about a block from a node.

GetBlockHeader – method to get the block header by its signature (block ID) or by its height.

One of the following parameters is entered in the method request:

- `signature` – the signature of the requested block as a Base58-encoded string;
- `key` – height of the requested block.

The method returns the following information about the block header:

- `version` — block version;
- `height` – block height;
- `block_signature` — block signature (identifier) as a Base58-encoded string;
- `reference` — the signature of the previous block, to which the current block is referring, as a Base58-encoded string;
- `miner_address` — miner address, as a Base58-encoded string;
- `tx_count` — the number of transactions in the block;
- `timestamp` — block time.

If the block is not found, the method returns the `BlockDoesNotExist` error.

contract˙contract˙service.proto

The **ContractService** methods are designed to work with Docker smart contracts. These are service methods for contract execution, as well as methods for reading smart contract state information and for actions with assets.

Important: The following ContractService methods can be used as both contract and public methods:

- GetContractKeys,
- GetContractKey,
- GetContractBalances,
- CalculateAssetId.

Connect

Connect – the method for connecting a smart contract to a node.

> The method query should contain the following parameters:
>
> - `connection_id` – the identifier of the Docker smart contract connection (see *Authorization of smart contracts with gRPC*);
> - `async_factor` – the maximum number of simultaneously processed transactions of the smart contract (see *Parallel operation of smart contracts*).
>
> The method returns the following information about the transaction and the block:

- `transaction` — contract call transaction;

- `auth_token` — authorization token;

- `current_block_info` — information on the current block:

  - `height` – current height;

  - `timestamp` — block time;

  - `miner_address` — miner address, as a Base58-encoded string;

  - `reference` — the signature (identifier) of the previous block, to which the current block is referring, as a Base58-encoded string.

CommitExecutionSuccess

CommitExecutionSuccess – the method for transferring the result of a Docker smart contract successful execution to the node. With this method a *smart contract can send a sequence of operations on assets*.

The following data is passed in the method request:

- `tx_id` — the identifier of the contract call transaction to which the smart contract sends the result;

- `results` — an array of `key-value` values that the smart contract will write to its state as the execution result. If a key is returned that is already present in the state, its value is overwritten;

- `asset_operations` — an array of actions a smart contract performs on assets available to it, including issuing new asset, reissuing an asset, burning an asset or transferring an asset available to the contract to another user (`issue`, `reissue`, `burn`, `transfer`).

The method response is not provided.

CommitExecutionError

CommitExecutionError – the method for sending a smart contract execution error to the node.

GetContractKeys

GetContractKeys – the method for requesting values from the smart contract state by the key filter.

Important: The GetContractKeys method of the ContractService can be used as both a contract method and a public method.

The following data is passed in the method request:

- `contract_id` – smart contract identifier;

- `limit` – the limit of number of data blocks to be obtained;

- `offset` – number of data blocks to be missed in the method response;

- `matches` – an optional parameter for a regular expression for keys sorting.

The method returns a `DataEntry` array containing the requested keys with values from the current Docker smart contract state.

GetContractKey

GetContractKey – the method to get the value of a certain key from the Docker smart contract state.

Important: The GetContractKey method of the ContractService can be used as both a contract method and a public method.

The following data is passed in the method request:

- `contract_id` – Docker smart contract identifier;

- `key` – the required key.

The method returns `DataEntry` which matches the passed key from the Docker smart contract current state.

GetContractBalances

GetContractBalances – the method for obtaining the Docker smart contract balance(s) (in *system token or other tokens*).

Important: The GetContractBalances method of the ContractService can be used as both a contract method and a public method.

The list of asset identifiers is passed in the request (`assets_ids`); to get the balance of the WEST system token, an empty string should be passed in the list.

The method response displays the list of the balances for each of the requested assets.

CalculateAssetId

CalculateAssetId – the method to calculate assetId when a new token is issued by a smart contract based on the passed parameter:

- `nonce` — a one-time code that can only be used once; also, several assets with the same `nonce` cannot be released within the same contract call.

Important: The CalculateAssetId method of the ContractService can be used as both a contract method and a public method.

---

contract˙permission˙service.proto

The **PermissionService** methods are designed to retrieve information about participants` roles (permissions).

GetPermissions – the method returns a list of all the permissions of the participant whose address was specified, valid at the moment specified. The method query should contain the following parameters:

- `address` – the participant's address;

- `timestamp` – the *Unix Timestamp* (in milliseconds) for the moment of time when the active permissions were requested.

The response of the method returns the `roles` array containing permissions for the required address and the entered `timestamp`.

GetPermissionsForAddresses – the method returns a list of all the permissions of multiple participants whose addresses were specified, valid at the moment specified.

The following data is passed in the method request:

- `addresses` – a string array containing required addresses;
- `timestamp` – the *Unix Timestamp* (in milliseconds) for the moment of time when the active permissions were requested.

The method response returns an `address_to_roles` array containing permissions for each required address, as well as the entered `timestamp`.

contract˙pki˙service.proto

contract˙privacy˙service.proto

The **PrivacyService** methods are designed to obtain information about confidential data access groups and to work with confidential data.

Learn more about confidential data exchange and access groups in the article *Confidential data exchange*.

GetPolicyRecipients – the method for obtaining addresses of the confidential data group participants by the group `policy_id`. The method response returns a `recipients` string array which contains addresses of confidential data group participants.

GetPolicyOwners – the method for obtaining the addresses of the owners of a confidential data access group; the group identifier is passed in the method request as `policy_id`. The method returns the `owners` string array in the response, which contains addresses of confidential data access group owners.

GetPolicyHashes – the method for obtaining confidential data hashes from a confidential data access group; the group identifier is passed in the method request as `policy_id`. The method response outputs an array of `hashes` strings containing hashes from the access group.

contract˙transaction˙service.proto

The **TransactionService** methods are designed to retrieve information about transactions that have been sent to the blockchain.

Similar gRPC methods available to an external user are described in the *gRPC: handling transactions* section.

Unlike the *TransactionExists and TransactionInfo* methods available for external integration, contract methods return information not only about the transactions that have already been written to the block, but also about the transactions that are just getting ready to be packed into the block.

TransactionExists – the method for verifying if the transaction with the specified ID exists. The method returns `true` if a transaction with the specified ID exists, or `false` if it does not.

TransactionInfo – the method for obtaining the data on the transaction with the specified identifier: transaction name, transaction version, the blockchain height on which this transaction was made, other data about the transaction, depending on the type of this transaction.

contract ̇util ̇service.proto

The **GetNodeTime** method of the **UtilService** is designed to get the current node time.

The method returns the current node time in two formats:

- `system` – the system time of the node PC;

- `ntp` – network time.

See also

Smart contracts

Handling tokens from a Docker smart contract

Development and usage of smart contracts

General platform configuration: execution of smart contracts

1.24.3 WASM Smart contracts

Unlike *Docker smart contracts*, WASM smart contracts are a compiled byte code that is executed on a WEVM virtual machine. This approach improves the stability and speed of contract execution by eliminating additional components and communication between them, because the virtual machine is integrated inside the platform.

Important: WASM smart contracts can be used on the Waves Enterprise platform starting with release 1.14.0 after *1140* feature activation.

Key benefits of WASM:

- High performance and compatibility,

- Small size of WASM byte code,

- Isolated and platform-independent design,

- Extended language support.

Important: In releases 1.14.0 and 1.15.0, WASM smart contracts do not support *atomic transactions* and *confidential smart contracts*.

General WASM smart contract settings are specified in the `node.wasm` section of the *node configuration file*.

WASM Smart contract operation chart

A WASM smart contract is a base-64 WASM byte-code located inside ContractInfo.

The main difference between WASM smart contract implementation and Docker smart contract implementation is accessing the node API directly without using the gRPC API.

The general chart of WASM smart contract operation is provided below:

When a WASM smart contract is executed, a WASMService is created that accesses blockchain functionality (provides an interface to a DelegatedBlockchain instance such as TransactionsAccumulator).

WASMContractExecutor calls executeTransaction. Inside the executeTransaction, WASMExecutor calls the execution of the smart contract bytecode (runContract) and its validation (validateBytecode).

- In case of a **CreateContractTransaction**, the `runBytecode` method is called with the `_constructor` function passed in.

- In case of an **UpdateContractTransaction**, the `runBytecode` method is not called; `ContractExecutionSuccess` is returned with empty changes.

- In case of a **CallContractTransaction**, the `runBytecode` method is called with the `callFunc` function passed in *CallContractTransaction*. The function in CallContractTransaction cannot be `_constructor`.

When the next block is formed, the transaction is retrieved from the UTX pool, a one-time instance of the WASMService class is created, and the transaction is accumulated via WASMService.

WASM smart contract development

WASM smart contract can be developed in any programming language that supports source code compilation into WebAssembly bytecode. The virtual machine provides the necessary set of functions for writing contracts of any complexity. Before creating a WASM contract on the platform, all you need to do is compile it.

Practical instructions on WASM smart contracts development are listed in the *WASM smart contracts development and usage* article.

Creating and calling a WASM smart contract

The same transactions as for *Docker contracts* are used to create and call WASM smart contracts.

To create a contract, you must specify the byte code of the contract as a base64 string and its byte code hash in *103 CreateContract* transaction version 7.

Important: A participant developing smart contracts must have the **contract_developer** *permission* on the network. With this permission the participant can call smart contracts, as well as change his own smart contracts code and restrict their operation.

All WASM contracts are stored inside the blockchain state, so no additional downloads or third-party services are required when the contract is launched.

A WASM smart contract is called for operation by a network participant using the *104 CallContract Transaction* version 7. To call a WASM contract, you must specify the function to be called and the list of arguments required for the function.
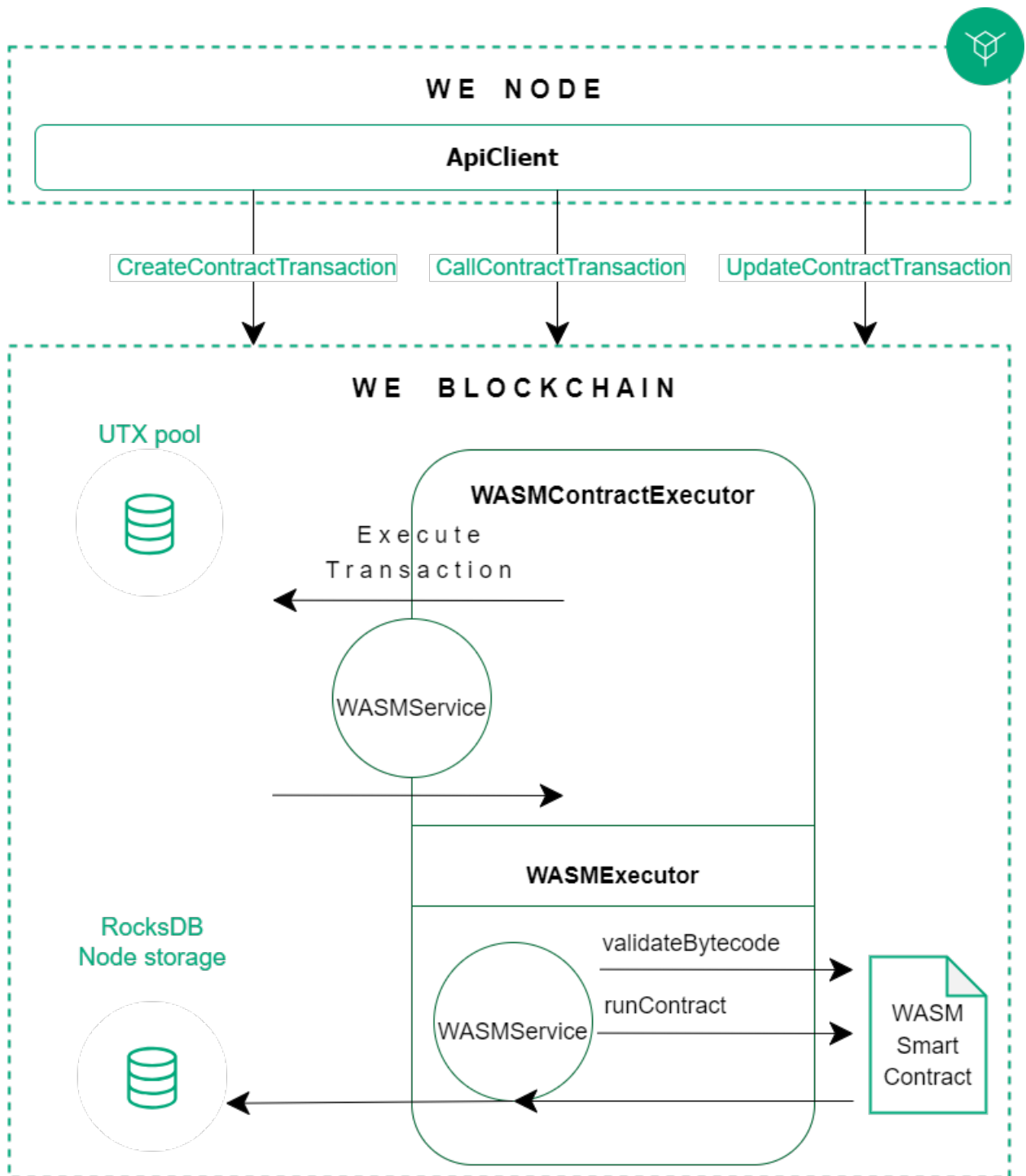
Capturing the results of WASM smart contract execution

The result of the contract execution is `0` if the contract executed successfully, or an error code:

Table 8: Error codes

Error code	Error name	Condition that returns the error
100	InvalidByte-code	Failed to recognize and validate the WASM bytecode
101	Construc-torNotFound	Failed to find the contract constructor
102	MemoryError	Error when working with virtual or linear memory
103	MemoryLimits	Memory size limit below u32::MAX
104	LinkerError	Error when working with Linker instances
105	Instantiate-Failed	Failed to create and run WASM bytecode
106	HeapBaseNot-Found	Unable to find Global heap base
107	FuncNotFound	Failed to find a function
108	InvalidNu-mArgs	Invalid number of arguments
109	FailedParse-FuncArgs	Failed to recognize function argument
110	FailedDeserial-izeDataEntry	Failed to recognize DataEntry arguments
111	FailedExec	Failure during execution
112	StackOverflow	Call stack overflow error
304	no data for this key: <key>	Attempt to access a key for which storage is not initialized
501	DataIsMissing	There is no contractKey or contractBytecode
502	InvalidArgu-ment	An invalid argument was passed from WEVM, e.g. invalid recipient address
503	InvalidTransfer	Transfer errors, e.g. when trying to pass a value less than 0 to Transfer or creating too many transfers to the contract balance (`payments` field)

If a contract makes any changes to its state, or performs an operation on assets, the WASMService captures such changes by calling external functions (Environment).

The smart contract is executed and sends the result via API to the node that initiated the smart contract launch. The node, in turn, generates a *105 ExecutedContract Transaction* with the result of the smart-contract execution. This way the result of the smart contract execution is fixed in its state using the 105 ExecutedContract transaction.

Validator nodes verify that everyone who executed that smart contract with that data got the same result. If the verification is successful, the miner node places the transactions into the block, and the result of the smart contract execution goes into the blockchain.

Prohibiting WASM smart contract calls

WASM smart contract can be disabled in the same way as *Docker smart contract*.

Updating WASM smart contracts

If you have changed your WASM smart contract code, update it using the *107 UpdateContract Transaction* version 7.

Important: You cannot update a smart contract that has been disabled by means of the *106 DisableContract Transaction*.

Hint: Only the participant with the **contract_developer** *role* who created the *103 CreateContract Transaction* for this smart contract can modify the contract.

WASM smart contracts validation

To provide additional integrity control for WASM smart contracts, the blockchain platform supports the same validation policies as for *Docker smart contracts*.

Parallel execution of WASM smart contracts

Multiple smart contracts can be run simultaneously on the Waves Enterprise platform. For this purpose, the node implements the MVCC (Multiversion concurrency control) mechanism – managing parallel access to the state of smart contracts through multiversionality. The mechanism allows multiple transactions of containerized smart contracts to run in parallel and maintain data consistency.

The MVCC mechanism for WASM smart contracts is similar to the mechanism for *Docker smart contracts*.

See also

> *Development and usage of smart contracts*
>
> *General platform configuration: execution of smart contracts*
>
> *WASM smart contracts execution configuration*

1.25 Smart Account

Every transaction on the Waves Enterprise blockchain platform is created on behalf of an account. Through the use of public and private keys, you can verify that a transaction issued from an account was actually sent from that account.

But the key pair may not be enough to secure transactions. For example, the account mnemonic phrase leakage would allow an attacker to send transactions to the blockchain on behalf of the account.

To increase the security of transactions, the Waves Enterprise blockchain platform implements Smart Account technology. A Smart Account is an account with a script that checks all the transactions sent by the account for compliance with the conditions specified in the script. This script allows outgoing transactions to be

---

validated, for example, for multisig. Below are some examples of parameters that the script can use to validate transactions:

- Transaction Type — only the transactions of the type specified in the script can be sent;

- Transaction Confirmation or Signature — you can set a rule that the `proofs` confirmation array in the transaction body must contain a specific transaction signature, multiple specific signatures, or other data;

- Current blockchain height — the account owner may set the rule that transactions can only be sent from his address if the blockchain height exceeds the number N specified in the script;

- Arbitrary data that exists on the blockchain — for example, oracles data.

You can also use the script to override all validations by setting the rule that all the transactions sent from the address must be considered valid.

Only the transactions that have been validated by the script can be sent from the smart account.

1.25.1 Creating an account script

The account owner creates an account script in the RIDE language.

Account script structure

An account script consists of a directive and an expression.

Directive

A directive is placed at the beginning of the script. For example:

```
{-# STDLIB_VERSION 2 #-}
{-# CONTENT_TYPE EXPRESSION #-}
{-# SCRIPT_TYPE ACCOUNT #-}
```

The above directive consists of three annotations and provides the compiler with the following information:

- the script uses version 2 of the library of standard functions,

- the type of script content is Expression,

- the script being created will be an account script.

Expression

The expression checks the transactions sent by the account for compliance with the specified conditions. If the conditions are not met, the transaction will not be sent. The following results of the expression execution are possible:

- `true` - the transaction is allowed,

- `false` - the transaction is prohibited,

- error.

1.25.2 Attaching an account script to an account

You can only install a script on an account using the *13. SetScript Transaction*. The account sending this transaction must have only the `contract_developer` *role*, or no role at all.

You can only attach one script to an account.

You can detach a script from a smart account or replace an old script with a new script unless the old script forbids it. To detach" a script or replace it with a new one, you will need to send a new SetScript transaction.

1.25.3 An example of creating and deploying an account script

This section provides an example of creating and deploying an account script manually without using client libraries and API libraries.

In this example we will create and deploy a simple account script that checks for transaction multiple signatures (two out of two).

Example Assumptions

In creating this account script, the following is implied:

1. You have your own node in the Waves Enterprise blockchain network.

2. You have three addresses generated on the Waves Enterprise blockchain network:

   - 3MxjWXEUcVCeiaEUqNcorB5HxSpLsgJCGxE – Alice's account,

   - 3MqGVvfgqdqqU6P9mTAsLSxyRoRjrHF18Mf – Bob's account,

   - 3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X – shared account.

Creating the account script

In this example, the following script is created:

- The first two lines of the script define 2 base58 encoded public keys for Alice's and Bob's addresses.

- Users then collect 2 public keys in the `proofs[0]` and `proofs[1]` fields.

  The account balance is funded by team members, then when 2 out of 2 team members decide to spend the money, they provide their signatures in a single transaction.

- The smart account script uses the sigVerify function to verify these signatures in `proofs`, and if two of the two signatures are valid, the transaction is also considered valid; otherwise, the transaction does not pass to the blockchain.

There is no directive in the script, so automatic values will be selected.

```
let alicePubKey    = base58'Ey6Z9XkWsvG8JZwyxhkTjydRcGp1wg6rbC3AYcxq7Efr'
let bobPubKey      = base58'5PvhyouzHn2Pcev56oBvwpnsGK5fEu1dA8fM2nJQM4HR'

let aliceSigned    = if(sigVerify(tx.bodyBytes, tx.proofs[0], alicePubKey)) then 1 else 0
let bobSigned      = if(sigVerify(tx.bodyBytes, tx.proofs[1], bobPubKey  )) then 1 else 0
aliceSigned + bobSigned == 2
```

Converting the script to Base64 format

Use the */utils/script/compile* method to compile the script and convert the script to Base64 format.

You can use Swagger for this purpose:

Or you can use curl to compile the script and convert it to Base64 format:

```
curl -X POST "http://localhost:6862/utils/script/compile" -H  "accept: application/json"↵
↪-H  "Content-Type: application/json" -d "let alicePubKey  = base58
↪'Ey6Z9XkWsvG8JZwyxhkTjydRcGp1wg6rbC3AYcxq7Efr'let bobPubKey    = base58
↪'5PvhyouzHn2Pcev56oBvwpnsGK5fEu1dA8fM2nJQM4HR'let aliceSigned  = if(sigVerify(tx.
↪bodyBytes, tx.proofs[0], alicePubKey)) then 1 else 0let bobSigned    = if(sigVerify(tx.
↪bodyBytes, tx.proofs[1], bobPubKey  )) then 1 else 0aliceSigned + bobSigned == 2"
```

Attaching the script to the account

To attach the Base64-encoded script to your account, follow these steps:

1. Prepare *13. SetScript Transaction* JSON for signing. Specify the shared account as the sender; set the account script in the `script` field:

```
{
  "type": 13,
  "version": 1,
  "sender": "3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X",
  "fee": 100000,
  "script": "<script>"
}
```

2. Sign the transaction using the */transactions/sign* method:

```
$ curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
↪application/json' \
--header 'X-API-Key: <it is a secret>' \
-d '{ "type": 13, "version": 1, "sender":
↪"3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X", "fee": 100000, \
"script": "<script>" }' 'https://example.org/transactions/sign'
```

The method will return JSON ready for broadcasting:

```
{
  "type": 13,
  "id": "8w7yauNiENsJP8oDUpVEfiAzyEzMKoXbJEqS26Ht99mg",
  "sender": "3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X",
  "senderPublicKey": "66xdGznqt2AVLMZRHme9vFPC6cvN4yV95wRWPfTus3Qe",
  "fee": 100000,
  "timestamp": 1525797758819,
  "proofs": [

↪"4Ro4e4UrsVkaFbHtu96qZwHAdf8N4TtpjSGik9kRusmmYKCxicdsEqcgQrYden36nurqhY9EBkTKwD499kAi5rxe
↪"
  ],
  "version": 1,
  "script": "<script>"
}
```

3. Broadcast the resulting JSON to the blockchain using the *POST /transactions/broadcast* method:

```
$ curl -X POST --header 'Content-Type: application/json' --header 'Accept:␣
↪application/json' \
--header 'X-API-Key: <it is a secret>' \
-d '{ "type": 13, "id": "8w7yauNiENsJP8oDUpVEfiAzyEzMKoXbJEqS26Ht99mg",
↪"sender": "3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X", \
"senderPublicKey": "66xdGznqt2AVLMZRHme9vFPC6cvN4yV95wRWPfTus3Qe", "fee":␣
↪100000, "timestamp": 1525797758819, \
"proofs": [
↪"4Ro4e4UrsVkaFbHtu96qZwHAdf8N4TtpjSGik9kRusmmYKCxicdsEqcgQrYden36nurqhY9EBkTKwD499kAi5rxe
↪" ], \
```

```
"version": 1, "script": "<script>" }' \
'https://example.org/transactions/broadcast'
```

4. Check to see if the script has been applied:

```
$ curl http://example.org/addresses/scriptInfo/
↪3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X
{
"address" : "3N7H4jTBMKtZfNCY86K2ND1rWcvFsGjDT3X",
"script" : "<script>",
"scriptText" : "<scriptText>",
"complexity" : 27,
"extraFee" : 400000
}
```

where `<scriptText>` is the string representation of the compiled `<script>` (expression tree).

Now when a transfer is sent from the shared account to another account using the *4. Transfer Transaction*, the script will verify if both signatures (of Alice's and Bob's accounts) are present in the `proofs` field. If there is no signature of at least one of the accounts, the script will return the error: "State check failed. Reason: TransactionNotAllowedByScript", and the transaction will not be published on the network.

See also

> *Smart contracts*
>
> *Description of permissions*
>
> *13. SetScript Transaction*

1.26 Transactions of the blockchain platform

Transaction is a separate operation in the blockchain changing the network state and performed on behalf of a participant. By sending a transaction, the participant sends a request to the network with the set of data needed for the corresponding change of the state.

1.26.1 Signing and sending of transactions

Prior to signing and sending of transactions, a participant generates a digital signature for it. To do this, he uses a private key of his account. Transaction signing can be done in three ways:

- with the use of the blockchain platform client;
- with the use of the REST API method (see *REST API: transactions*);
- with the use of the *JavaScript SDK*.

The transaction signature is inserted into the `proofs` field while sending the transaction into the blockchain. As a rule, this field contains one signature of the participant who sent the transaction. But this field supports up to 8 signatures: in case of transaction signing by a smart account, filling of an atomic transaction or smart contract broadcasting.

After signing, the transaction is sent into the blockchain. This can be done in three aforementioned ways, as well as with the use of the gRPC interface (see *gRPC: sending of transactions into the blockchain*)

1.26.2 Processing of transactions in the blockchain

After obtaining of a transaction, the node validates it in the following way:

1. Timestamp correspondence check: a transaction timestamp should derive from the current block timestamp for not more than 2 hours before or 1,5 hours after it.

2. Transaction type and version check: if support of such transactions type and versions has been activated in the blockchain (see *Activation of blockchain features*).

3. Correspondence of transaction fields with a defined data type;

4. Sender balance check: if balance is sufficient for fee payment;

5. Transaction signature check.

If a transaction is not validated, the node declines it. In case of successful validation, a transaction is added to the unconfirmed transaction (UTX) pool, where it is awaiting the next mining round for broadcasting in the blockchain. Together with transfer of this transaction into the UTX pool, the node sends it to other nodes of the network.

Each microblock has a limit of incoming transactions, each separate transaction can be transferred from the UTX pool not at once. During existence of a transaction in the UTX pool, a transaction can become invalid. For instance, its timestamp is not more corresponding with the current block timestamp, or a transaction transferred into the blockchain has decreased a sender balance and made it insufficient for payment of a transaction fee. In this case, a transactions will be declined and removed from the UTX pool.

After adding of a transaction into a block, the transaction changes the blockchain state. After this, transaction is considered executed.

Detailed information about transactions of the Waves Enterprise blockchain platform:

Description of transactions

The Waves Enterprise blockchain platform supports 28 types of transactions. Each of them contains its own set of data to be sent into the blockchain.

Requests and responses passed via the node *REST API* interface within the framework of each transaction, have JSON format. Requests and responses passed via the node *gRPC* interface, are defined by corresponding proto schemes. JSON models of transaction requests and responses are stated below.

Hint: In case you have protected the keypair of your node with a password while *generating the account*, set the password of your keypair in the `password` field of a transaction.

---

1. Genesis Transaction

First transaction of a new blockchain which performs first attachment of balance to addresses of created nodes.

This transaction does not require signing, that is why it is only broadcasted. Transaction has the only version.

Transaction data structures

Field	Data type	Description
type	Byte	Transaction number (**1**)
id	Byte	Transaction identifier
fee	Long	*WE Mainnet transaction fee Mainnet*
timestamp	Long	The **Unix Timestamp** of the transaction – in milliseconds
signature	ByteStr	Genesis block signature
recipient	ByteStr	Address of recipient of distributed tokens
amount	Long	Amount of tokens
height	Int	Height of transaction execution. For the first transaction – **1**

3. Issue Transaction

A transaction initiating the issue of *tokens*.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**3**)
version	Byte	Transaction version
name	Array[byte]	An arbitrary name of transaction
quantity	Long	Number of tokens to be issued
description	Array[byte]	An arbitrary description of the transaction (**in base58 format**)
sender	ByteStr	Address of sender of distributed tokens
password	String	Keypair password in the node keystore – *optional field*
decimals	Byte	Digit capacity of a token in use (WEST – **8**)
reissuable	Boolean	Re-issuability of a token
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number
id	Byte	Transaction identifier
sender	ByteStr	Address of the transaction sender
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*
proofs	List(ByteStr)	Array of transaction proofs
version	Byte	Transaction version
assetId	Byte	Identifier of an asset to be issued
name	Array[byte]	An arbitrary name of transaction
quantity	Long	Number of tokens to be issued
reissuable	Boolean	Re-issuability of a token
decimals	Byte	Digit capacity of a token in use (WAVES − **8**)
description	Array[byte]	An arbitrary description of the transaction
chainId	Byte	Identifying byte of the network (Mainnet - **87** or **V**)
script	Array[Byte]	Script for validation of the transaction an − *optional field*
height	Int	Height of transaction execution

Important: If the `reissuable` field is set to `False`, i.e. tokens are not allowed to be reissued, then it will be impossible to change this value in the future.

JSON:

Version 2

Signing:

```
{
    "type": 3,
    "version":2,
    "name": "Test Asset 1",
    "quantity": 100000000000,
    "description": "Some description",
    "sender": "3FSCKyfFo3566zwiJjSFLBwKvd826KXUaqR",
    "password": "",
    "decimals": 8,
    "reissuable": true,
    "fee": 100000000
}
```

Broadcasting:

```
{
    "type": 3,
    "id": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "sender": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
```

```
    "senderPublicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
    "fee": 100000000,
    "timestamp": 1549378509516,
    "proofs": [
→"NqZGcbcQ82FZrPh6aCEjuo9nNnkPTvyhrNq329YWydaYcZTywXUwDxFAknTMEGuFrEndCjXBtrueLWaqbJhpeiG
→" ],
    "version": 2,
    "assetId": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "name": "Token Name",
    "quantity": 10000,
    "reissuable": true,
    "decimals": 2,
    "description": "SmarToken",
    "chainId": 84,
    "script": "base64:AQa3b8tH",
    "height": 60719
}
```

Version 3

Signing:

```
{
  "type": 3,
  "version":3,
  "name": "Test Asset 1",
  "quantity": 100000000000,
  "description": "Some description",
  "sender": "3FSCKyfFo3566zwiJjSFLBwKvd826KXUaqR",
  "password": "",
  "decimals": 8,
  "reissuable": true,
  "fee": 100000000
  "atomicBadge":{
      "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
  }
}
```

Broadcasting:

```
{
    "type": 3,
    "id": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "sender": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
    "senderPublicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
    "fee": 100000000,
    "timestamp": 1549378509516,
    "proofs": [
→"NqZGcbcQ82FZrPh6aCEjuo9nNnkPTvyhrNq329YWydaYcZTywXUwDxFAknTMEGuFrEndCjXBtrueLWaqbJhpeiG
→" ],
    "version": 3,
```

```
    "assetId": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "name": "Token Name",
    "quantity": 10000,
    "reissuable": true,
    "decimals": 2,
    "description": "SmarToken",
    "chainId": 84,
    "script": "base64:AQa3b8tH",
    "height": 60719
}
```

4. Transfer Transaction

A transaction of *tokens* transfer from one address to another.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**4**)
version	Byte	Transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
recipient	ByteStr	Address of recipient of tokens
amount	Long	Amount of tokens
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
amount	Long	Amount of tokens
fee	Long	*WE Mainnet transaction fee*
type	Byte	Transaction number (**4**)
version	Byte	Transaction version
attachment	Byte	Comment to the transaction (**in base58** format) – *optional field*
sender	ByteStr	Address of the transaction sender
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
assetId	Byte	ID of a token to be transferred – *optional field*
recipient	ByteStr	Tokens recipient address
id	Byte	Transaction identifier
timestamp	Long	The **Unix Timestamp** of the transaction in milliseconds – *optional field*

JSON:

Version 2

Signing:

```
{
  "type": 4,
  "version": 2,
  "sender": "3M6dRZXaJY9oMA3fJKhMALyYKt13D1aimZX",
  "password": "",
  "recipient": "3M6dRZXaJY9oMA3fJKhMALyYKt13D1aimZX",
  "amount": 40000000000,
  "fee": 100000
}
```

Broadcasting:

```
{
    "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
    "amount": 200000000,
    "fee": 100000,
    "type": 4,
    "version": 2,
    "attachment": "3uaRTtZ3taQtRSmquqeC1DniK3Dv",
    "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
    "feeAssetId": null,
    "proofs": [
↪"2hRxJ2876CdJ498UCpErNfDSYdt2mTK4XUnmZNgZiq63RupJs5WTrAqR46c4rLQdq4toBZk2tSYCeAQWEQyi72U6
↪"
    ],
    "assetId": null,
    "recipient": "3GPtj5osoYqHpyfmsFv7BMiyKsVzbG1ykfL",
    "id": "757aQzJiQZRfVRuJNnP3L1d369H2oTjUEazwtYxGngCd",
    "timestamp": 1558952680800
}
```

Version 3

Signing:

```
{
  "type": 4,
  "version": 3,
  "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "password": "",
  "recipient": "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
  "amount": 40000000000,
  "fee": 10000000
  "atomicBadge" : {
    "trustedSender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
```

```
  },
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "amount" : 10,
  "fee" : 1000000,
  "type" : 4,
  "version" : 3,
  "atomicBadge" : {
    "trustedSender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  },
  "attachment" : "",
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
→"2vbAJmwzQw2FCtozcewxJVfxoHxf97BTNdGuaeSATV4vEHZ3XYA4Z7nXGsSnf18aesnAWTKWCfzwM5yGpWEyGM7f
→" ],
  "assetId" : null,
  "recipient" : "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
  "id" : "2wCEMREFbgk318hFFaNGsgFzyjZHuCrtwSnpK35qhiw4",
  "timestamp" : 1619186861204,
  "height" : 861644
}
```

5. Reissue Transaction

Transaction for native *token* re-issue.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**5**)
version	Byte	Transaction version
quantity	Long	Amount of tokens to be re-issued
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
assetId	Byte	ID of a token to be re-issued – *optional field*
reissuable	Boolean	Re-issuability of a token
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
quantity	Long	Amount of tokens to be re-issued
sender	ByteStr	Address of the transaction sender
chainId	Byte	Identifying byte of the network (Mainnet - **87** or **V**)
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
assetId	Byte	ID of a token to be re-issued – *optional field*
fee	Long	*WE Mainnet transaction fee*
id	Byte	Transaction identifier
type	Byte	Transaction number (**5**)
version	Byte	Transaction version
reissuable	Boolean	Re-issuability of a token
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
height	Int	Height of transaction execution

JSON:

Version 2

Signing:

```
{
  "type": 5,
  "version":2,
  "quantity": 556105,
  "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "password": "",
  "assetId": "6UAMZA6RshxyPvt9W7aoWiUiB6N73yLQMMfiRQYXdWZh",
  "reissuable": true,
  "fee": 100000000
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "quantity" : 556105,
  "fee" : 100000000,
  "type" : 5,
  "version" : 2,
  "reissuable" : true,
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "chainId" : 86,
  "proofs" : [
↪"5ahD78wciu8YTsLoxo1XRghJWAGG7At7ePiBWTNzdkvX7cViRCKRLjjjPTGCoAH2mdGQK9i1JiY1wh18eh4h7pGy
↪" ],
  "assetId" : "6UAMZA6RshxyPvt9W7aoWiUiB6N73yLQMMfiRQYXdWZh",
  "id" : "8T9jJUusN5KBexxDUX1XBjoDydXGP34zWH7Qvp5mnmES",
  "timestamp" : 1619187184206,
```

```
    "height" : 861645
}
```

Version 3

Signing:

```
{
    "type": 5,
    "version":3,
    "quantity": 556105,
    "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "password": "",
    "assetId": "6UAMZA6RshxyPvt9W7aoWiUiB6N73yLQMMfiRQYXdWZh",
    "reissuable": true,
    "fee": 100000000
    "atomicBadge":{
      "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "quantity" : 556105,
  "fee" : 100000000,
  "type" : 5,
  "version" : 3,
  "reissuable" : true,
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "chainId" : 86,
  "proofs" : [
↪"5ahD78wciu8YTsLoxo1XRghJWAGG7At7ePiBWTNzdkvX7cViRCKRLjjjPTGCoAH2mdGQK9i1JiY1wh18eh4h7pGy
↪" ],
  "assetId" : "6UAMZA6RshxyPvt9W7aoWiUiB6N73yLQMMfiRQYXdWZh",
  "id" : "8T9jJUusN5KBexxDUX1XBjoDydXGP34zWH7Qvp5mnmES",
  "timestamp" : 1619187184206,
  "height" : 861645
}
```

Important: If the `reissuable` field is set to `False`, i.e. tokens are not allowed to be reissued, then it will be impossible to change this value in the future.

---

6. Burn Transaction

Transaction for burning native *tokens*: decreases the amount of tokens at the sender's address, and, with this, decreases the total amount of tokens in the blockchain. The burned tokens cannot be restored.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**6**)
version	Byte	Transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
assetId	Byte	ID of a token to be burnt – *optional field*
quantity	Long	Number of tokens to be burnt
fee	Long	*WE Mainnet transaction fee*
attach-ment	Byte	Comment to the transaction (**in base58 format**) – *optional field*

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
amount	Long	Number of tokens to be burnt
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
assetId	Byte	ID of a token to be burnt – *optional field*
fee	Long	*WE Mainnet transaction fee*
id	Byte	Transaction identifier
type	Byte	Transaction number (**6**)
version	Byte	Transaction version
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
height	Int	Height of transaction execution

JSON:

Version 2

Signing:

```
{
    "type": 6,
    "version": 2,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "assetId": "7bE3JPwZC3QcN9edctFrLAKYysjfMEk1SDjZx5gitSGg",
    "quantity": 1000,
    "fee": 100000,
    "attachment": "string"
}
```

Broadcasting:

```
{
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "amount": 1000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "chainId": 84,
    "proofs": [
→"kzTwsNXjJkzk6dpFFZZXyeimYo6iLTVbCnCXBD4xBtyrNjysPqZfGKk9NdJUTP3xeAPhtEgU9hsdwzRVo1hKMgS
→" ],
    "assetId": "7bE3JPwZC3QcN9edctFrLAKYysjfMEk1SDjZx5gitSGg",
    "fee": 100000,
    "id": "3yd2HZq7sgun7GakisLH88UeKcpYMUEL4sy57aprAN5E",
    "type": 6,
    "version": 2,
    "timestamp": 1551448489758,
    "height": 1190
}
```

Version 3

Signing:

```
{
    "type": 6,
    "version": 3,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "assetId": "7bE3JPwZC3QcN9edctFrLAKYysjfMEk1SDjZx5gitSGg",
    "quantity": 1000,
    "fee": 100000,
    "attachment": "string"
    "atomicBadge":{
        "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "amount": 1000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "chainId": 84,
    "proofs": [
→"kzTwsNXjJkzk6dpFFZZXyeimYo6iLTVbCnCXBD4xBtyrNjysPqZfGKk9NdJUTP3xeAPhtEgU9hsdwzRVo1hKMgS
→" ],
    "assetId": "7bE3JPwZC3QcN9edctFrLAKYysjfMEk1SDjZx5gitSGg",
    "fee": 100000,
    "id": "3yd2HZq7sgun7GakisLH88UeKcpYMUEL4sy57aprAN5E",
    "type": 6,
    "version": 3,
    "timestamp": 1551448489758,
    "height": 1190
}
```

8. Lease Transaction

Leasing of *tokens* to another address. The tokens in leasing are taken into account in the generating balance of a recipient after 1000 blocks.

Leasing of tokens can be carried out for increasing of probability of node appointment as a next round miner. As a rule, a recipient shares his revenue for block generation with an address which has granted him tokens for leasing.

Tokens in leasing remain blocked at a sender address. Leasing can be cancelled with the use of leasing cancel transaction.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**8**)
version	Byte	Transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
recipient	ByteStr	Address of recipient of tokens
amount	Long	Number of tokens for leasing
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
amount	Long	Number of tokens for leasing
sender	ByteStr	Address of the transaction sender
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
fee	Long	*WE Mainnet transaction fee*
recipient	ByteStr	Address of recipient of tokens
id	Byte	Transaction identifier
type	Byte	Transaction number (**8**)
version	Byte	Transaction version
height	Int	Transaction version
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*

JSON:

Version 2

Signing:

```
{
    "type": 8,
    "version": 2,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "recipient": "3N1ksBqc6uSksdiYjCzMtvEpiHhS1JjkbPh",
    "amount": 1000,
    "fee": 100000
}
```

Broadcasting:

```
{
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "amount": 1000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "proofs": [
↪"5jvmWKmU89HnxXFXNAd9X41zmiB5fSGoXMirsaJ9tNeyiCAJmjm7MR48g789VucckQw2UExaVXfhsdEBuUrchvrq
↪" ],
    "fee": 100000,
    "recipient": "3N1ksBqc6uSksdiYjCzMtvEpiHhS1JjkbPh",
    "id": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp",
    "type": 8,
    "version": 2,
    "timestamp": 1551449299545,
    "height": 1190
}
```

Version 3

Signing:

```
{
    "type": 8,
    "version": 3,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "recipient": "3N1ksBqc6uSksdiYjCzMtvEpiHhS1JjkbPh",
    "amount": 1000,
    "fee": 100000
    "atomicBadge":{
        "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "amount": 1000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "proofs": [
→"5jvmWKmU89HnxXFXNAd9X41zmiB5fSGoXMirsaJ9tNeyiCAJmjm7MR48g789VucckQw2UExaVXfhsdEBuUrchvrq
→" ],
    "fee": 100000,
    "recipient": "3N1ksBqc6uSksdiYjCzMtvEpiHhS1JjkbPh",
    "id": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp",
    "type": 8,
    "version": 3,
    "timestamp": 1551449299545,
    "height": 1190
}
```

9. LeaseCancel Transaction

Cancelling of leasing of *tokens* that have been leased with the use of a transaction with a definite ID. The `lease` structure of this transaction is not filled: the node fills it automatically upon providing the transaction data.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**9**)
version	Byte	Transaction version
fee	Long	*WE Mainnet transaction fee*
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore − *optional field*
txId	Byte	ID of a leasing transaction

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
leaseId	Byte	ID of a leasing transaction
sender	ByteStr	Address of the transaction sender
chainId	Byte	Identifying byte of the network (Mainnet − **87** or **V**)
proofs	List(ByteStr)	Array of transaction proofs (in **base58** format)
fee	Long	*WE Mainnet transaction fee*
id	Byte	ID of a leasing cancel transaction
type	Byte	Transaction number (**9**)
version	Byte	Transaction version
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*
height	Int	Height of transaction execution

JSON:

Version 2

Signing:

```
{
    "type": 9,
    "version": 2,
    "fee": 100000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "txId": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp"
}
```

Broadcasting:

```
{
   "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "leaseId": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp",
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "chainId": 84,
    "proofs": [
↪"2Gns72hraH5yay3eiWeyHQEA1wTqiiAztaLjHinEYX91FEv62HFW38Hq89GnsEJFHUvo9KHYtBBrb8hgTA9wN7DM
↪" ],
```

```
    "fee": 100000,
    "id": "9vhxB2ZDQcqiumhQbCPnAoPBLuir727qgJhFeBNmPwmu",
    "type": 9,
    "version": 2,
    "timestamp": 1551449835205,
    "height": 1190
}
```

Version 3

Signing:

```
{
    "type": 9,
    "version": 3,
    "fee": 100000,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "txId": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp"
    "atomicBadge":{
        "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "leaseId": "6Tn7ir9MycHW6Gq2F2dGok2stokSwXJadPh4hW8eZ8Sp",
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "chainId": 84,
    "proofs": [
→"2Gns72hraH5yay3eiWeyHQEA1wTqiiAztaLjHinEYX91FEv62HFW38Hq89GnsEJFHUvo9KHYtBBrb8hgTA9wN7DM
→" ],
    "fee": 100000,
    "id": "9vhxB2ZDQcqiumhQbCPnAoPBLuir727qgJhFeBNmPwmu",
    "type": 9,
    "version": 3,
    "timestamp": 1551449835205,
    "height": 1190
}
```

10. CreateAlias Transaction

Creation of an alias for a sender address. An alias can be used in transactions as a recipient identifier.

The 3rd version of the transaction implemented the ability to pay fee in another token. The 4th version of the transaction features the ability to include the transaction in a *atomic transaction*.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**10**)
version	Byte	Transaction version
fee	Long	*WE Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
alias	Byte	An arbitrary alias

Broadcasting:

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**10**)
id	Byte	ID of a CreateAlias transaction
sender	ByteStr	Address of the transaction sender
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
alias	Byte	An arbitrary alias
height	Byte	Height of transaction execution

JSON:

Version 2

Signing:

```
{
    "type": 10,
    "version": 2,
    "fee": 100000000,
    "sender": "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
    "password": "",
    "alias": "1@k1_kv29"
}
```

Broadcasting:

```
{
  "senderPublicKey" : "C4eRfdUFaZMRkfUp91bYr7uMgdBRnUfAxuAjetxmK7KY",
  "sender" : "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
  "proofs" : [
→"3fhJztBNnTDjppmqgi4GugAYo1aS1mzZhVhPdnNsqYqCEyLLHfzgb75psRPntHD4uBZgk8jByFP9mwwx2Ezsdg59
→" ],
  "fee" : 100000000,
  "alias" : "1@k1_kv29",
  "id" : "AavgVzV7avPMpERro6YqikwFESAgG2wViprtPJUtXP6F",
  "type" : 10,
  "version" : 2,
  "timestamp" : 1608737444468,
  "height" : 595942
}
```

Version 3

Signing:

```
{
    "type": 10,
    "version": 3,
    "fee": 100000000,
    "feeAssetId": DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB,
    "sender": "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
    "password": "",
    "alias": "1@k1_kv29"
}
```

Broadcasting:

```
{
  "senderPublicKey" : "C4eRfdUFaZMRkfUp91bYr7uMgdBRnUfAxuAjetxmK7KY",
  "sender" : "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
  "proofs" : [
→"3fhJztBNnTDjppmqgi4GugAYo1aS1mzZhVhPdnNsqYqCEyLLHfzgb75psRPntHD4uBZgk8jByFP9mwwx2Ezsdg59
→" ],
  "fee" : 100000000,
```

```
    "feeAssetId": DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB,
    "alias" : "1@k1_kv29",
    "id" : "AavgVzV7avPMpERro6YqikwFESAgG2wViprtPJUtXP6F",
    "type" : 10,
    "version" : 3,
    "timestamp" : 1608737444468,
    "height" : 595942
}
```

Version 4

Signing:

```
{
    "type": 10,
    "version": 4,
    "fee": 100000000,
    "feeAssetId": DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB,
    "sender": "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
    "password": "",
    "alias": "1@k1_kv29"
    "atomicBadge":{
      "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
}
```

Broadcasting:

```
{
  "senderPublicKey" : "C4eRfdUFaZMRkfUp91bYr7uMgdBRnUfAxuAjetxmK7KY",
  "sender" : "3NwTvbW7TMckBc785XjtGTUfHmcesaWBe1A",
  "proofs" : [
→"3fhJztBNnTDjppmqgi4GugAYo1aS1mzZhVhPdnNsqYqCEyLLHfzgb75psRPntHD4uBZgk8jByFP9mwwx2Ezsdg59
→" ],
  "fee" : 100000000,
  "feeAssetId": DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB,
  "alias" : "1@k1_kv29",
  "id" : "AavgVzV7avPMpERro6YqikwFESAgG2wViprtPJUtXP6F",
  "type" : 10,
  "version" : 4,
  "timestamp" : 1608737444468,
  "height" : 595942
}
```

11. MassTransfer Transaction

Transfer of *tokens* to several recipients (1 to 100 addresses). The transaction fee depends on the number of addresses.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**11**)
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
fee	Long	*WE Mainnet transaction fee*
version	Byte	Transaction version
transfers	List	List of recipients with `recipient` and `amount` fields separated by a comma
recipient	ByteStr	Address of recipient of tokens
amount	Long	Number of tokens to be transferred to an address

Broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
type	Byte	Transaction number (**11**)
transferCount	Byte	Number of recipient addresses
version	Byte	Transaction version
totalAmount	Byte	Total number of tokens to be transferred
attachment	Byte	Comment to the transaction (**in base58 format**) – *optional field*
sender	ByteStr	Address of the transaction sender
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
assetId	Byte	ID of a token to be transferred – *optional field*
id	Byte	ID of a token transfer transaction
transfers	List	List of recipients with `recipient` and `amount` fields separated by a comma
transfers.recipient	ByteStr	Address of recipient of tokens
transfers.amount	Long	Number of tokens to be transferred to an address
height	Byte	Height of transaction execution

Example of the `transfers` field:

```
"transfers":
[
    { "recipient": "3MtHszoTn399NfsH3v5foeEXRRrchEVtTRB", "amount": 100000 },
    { "recipient": "3N7BA6J9VUBfBRutuMyjF4yKTUEtrRFfHMc", "amount": 100000 }
]
```

JSON:

Version 2

Signing:

```
{
    "type": 11,
    "sender": "3NydXoTq3UgUW5rxsNwEMs1iwbbvVEwxoHU",
    "password": "",
    "fee": 30000000,
    "version": 2,
    "transfers":
    [
        { "recipient": "3MtHszoTn399NfsH3v5foeEXRRrchEVtTRB", "amount": 100000 },
        { "recipient": "3N7BA6J9VUBfBRutuMyjF4yKTUEtrRFfHMc", "amount": 100000 }
    ]
}
```

Broadcasting:

```
{
  "senderPublicKey" : "AMhAY8RMy5QsPqj58xeMY3fJxTZKx71QztsjDzqWprHo",
  "fee" : 30000000,
  "type" : 11,
  "transferCount" : 4,
  "version" : 2,
  "totalAmount" : 400000000,
  "attachment" : "",
  "sender" : "3NydXoTq3UgUW5rxsNwEMs1iwbbvVEwxoHU",
  "feeAssetId" : "8bec1mhqTiveMeRTHgYr6az12XdqBBtpeV3ZpXMRHfSB",
  "proofs" : [
→"21hhAMmwze6nLLQ9K6AoU6scek9Sk5KabR4VggGfdTVFHonfMGwVTse6qL2f8zR8DRm7RckMaikiYRt5XxWEKWcA
→" ],
  "assetId" : "8bec1mhqTiveMeRTHgYr6az12XdqBBtpeV3ZpXMRHfSB",
  "transfers" : [ {
    "recipient" : "3NqEjAkFVzem9CGa3bEPhakQc1Sm2G8gAFU",
    "amount" : 100000000
  }, {
    "recipient" : "3NzkzibVRkKUzaRzjUxndpTPvoBzQ3iLng3",
    "amount" : 100000000
  }, {
    "recipient" : "3Nnx8cX3UiyfQeC3YQKVRqVr2ewSxrvaDyB",
    "amount" : 100000000
  }, {
    "recipient" : "3NzC4Ex91VBQKfJHPiGhuPEomLg48NMi2ZF",
```

(continues on next page)

```
        "amount" : 100000000
    } ],
    "id" : "EvnxFxdYhYxHgQSMhkyLaqgyUDZdnBknfAWEXyqEHt97",
    "timestamp" : 1627643861044,
    "height" : 1076874
}
```

Version 3

Signing:

```
{
    "type": 11,
    "sender": "3NydXoTq3UgUW5rxsNwEMs1iwbbvVEwxoHU",
    "password": "",
    "fee": 30000000,
    "version": 3,
    "transfers":
    [
        { "recipient": "3MtHszoTn399NfsH3v5foeEXRRrchEVtTRB", "amount": 100000 },
        { "recipient": "3N7BA6J9VUBfBRutuMyjF4yKTUEtrRFfHMc", "amount": 100000 }
    ]
    "atomicBadge":{
        "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "AMhAY8RMy5QsPqj58xeMY3fJxTZKx71QztsjDzqWprHo",
  "fee" : 30000000,
  "type" : 11,
  "transferCount" : 4,
  "version" : 3,
  "totalAmount" : 400000000,
  "attachment" : "",
  "sender" : "3NydXoTq3UgUW5rxsNwEMs1iwbbvVEwxoHU",
  "feeAssetId" : "8bec1mhqTiveMeRTHgYr6az12XdqBBtpeV3ZpXMRHfSB",
  "proofs" : [
↪"21hhAMmwze6nLLQ9K6AoU6scek9Sk5KabR4VggGfdTVFHonfMGwVTse6qL2f8zR8DRm7RckMaikiYRt5XxWEKWcA
↪" ],
  "assetId" : "8bec1mhqTiveMeRTHgYr6az12XdqBBtpeV3ZpXMRHfSB",
  "transfers" : [ {
    "recipient" : "3NqEjAkFVzem9CGa3bEPhakQc1Sm2G8gAFU",
    "amount" : 100000000
  }, {
    "recipient" : "3NzkzibVRkKUzaRzjUxndpTPvoBzQ3iLng3",
    "amount" : 100000000
  }, {
    "recipient" : "3Nnx8cX3UiyfQeC3YQKVRqVr2ewSxrvaDyB",
```

```
    "amount" : 100000000
  }, {
    "recipient" : "3NzC4Ex91VBQKfJHPiGhuPEomLg48NMi2ZF",
    "amount" : 100000000
  } ],
  "id" : "EvnxFxdYhYxHgQSMhkyLaqgyUDZdnBknfAWEXyqEHt97",
  "timestamp" : 1627643861044,
  "height" : 1076874
}
```

12. Data Transaction

Transaction for adding, editing and removing of entries in an address data storage. An address data storage contains data in the 'key:value' format.

The size of the address data repository is unlimited, but up to 100 new "key:value" pairs can be added with a single data transaction. Also the byte representation of the transaction after signing must not exceed **150 kilobytes**.

If the data author (the address in the `author` field) matches the transaction sender (the address in the `sender` field), the `senderPublicKey` parameter is not required when signing the transaction.

Data structure of a query for transaction signing:

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**12**)
version	Byte	Transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
sender-PublicKey	PublicKey-Account	Transaction sender public key
author	Byte	Author address for data to be entered
data	List	Data list with `key:`, `type:` and `value:` fields separated by commas
data.key	Byte	Record key
data.type	Byte	Record data type. Possible values: `binary bool integer string` and `null` (record deletion by its key)
data.value	Byte	Record value
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
sender-PublicKey	PublicKey-Account	Transaction sender public key
sender-PublicKey	PublicKey-Account	Data author public key
data	List	Data list with `key:` `type:` and `value:` fields separated by commas
data.key	Byte	Record key
data.type	Byte	Record data type. Possible values: `binary` `bool` `integer` `string` and `null` (record deletion by its key)
data.value	Byte	Record value
sender	ByteStr	Address of the transaction sender
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
author	Byte	Author address for data to be entered
fee	Long	*WE Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
id	Byte	Data transaction ID
type	Byte	Transaction number (**12**)
version	Byte	Transaction version
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*

Example of the `data` field:

```
"data": [
      {
       "key": "objectId",
       "type": "string",
       "value": "obj:123:1234"
      }, {...}
      ]
```

JSON:

Version 2

Signing:

```
{
    "type": 12,
    "version": 2,
    "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "password": "",
    "senderPublicKey": "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
    "author": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "data": [
      ...
    ],
    "fee": 150000000
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "data" : [
    ...
  ],
  "author" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "fee" : 150000000,
  "type" : 12,
  "version" : 2,
  "authorPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
→"4wFNmn32NZqGwP4D4aAxCMyigGEVZLWftqi919pHAK7mCj3sFw7Ekf76g2rr51PZuk5sLwzjkKiZArQvWY8uEGqk
→" ],
  "id" : "GcDy84oTFf5NQzDtixkfUqiFNZwMaN2vfXqxsbGxumfo",
  "timestamp" : 1619187166499,
  "height" : 861644
}
```

Version 3

Signing:

```
{
    "type": 12,
    "version": 3,
    "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "password": "",
    "senderPublicKey": "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
    "author": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "data": [
        ...
    ],
    "fee": 150000000
    "atomicBadge":{
        "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "data" : [
    ...
  ],
  "author" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "fee" : 150000000,
  "type" : 12,
  "version" : 3,
  "authorPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
```

(continued from previous page)
```
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
↪"4wFNmn32NZqGwP4D4aAxCMyigGEVZLWftqi9l9pHAK7mCj3sFw7Ekf76g2rr51PZuk5sLwzjkKiZArQvWY8uEGqk
↪" ],
  "id" : "GcDy84oTFf5NQzDtixkfUqiFNZwMaN2vfXqxsbGxumfo",
  "timestamp" : 1619187166499,
  "height" : 861644
}
```

13. SetScript Transaction

A transaction to attach a script to an account or detach a script from an account. An account with a script attached to it is called a *Smart Account*.

The script allows you to verify transactions transmitted on behalf of an account without using the blockchain transaction verification mechanism.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**13**)
version	Byte	Transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
fee	Long	*WE Mainnet transaction fee*
name	Array[Byte]	Script name
script	Array[Byte]	The compiled script in **base64** format. If you leave this field empty (`null`), the script will be detached from the account

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**13**)
id	Byte	ID of a script setting transaction
sender	ByteStr	Address of the transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), − *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
chainId	Byte	Identifying byte of the network (Mainnet - **87** or **V**)
version	Byte	Transaction version
script	Array[Byte]	Compiled script in **base64** format − *optional field*
name	Array[Byte]	Script name
description	Byte	Comment to the transaction (**in base58 format**) − *optional field*
height	Byte	Height of transaction execution

JSON:

Version 1

Signing:

```
{
    "type": 13,
    "version": 1,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "fee": 1000000,
    "name": "faucet",
    "script": "base64:AQQAAAAHJG1hdGNoMAUAAAACdHgG+RXSzQ=="
}
```

Broadcasting:

```
{
    "type": 13,
    "id": "HPDypnQJHJskN8kwszF8rck3E5tQiuiM1fEN42w6PLmt",
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "senderPublicKey": "Fbt5fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUopa6H3",
    "fee": 1000000,
    "timestamp": 1545986757233,
    "proofs": [
↪"2QiGYS2dqh8QyN7Vu2tAYaioX5WM6rTSDPGbt4zrWS7QKTzojmR2kjppvGNj4tDPsYPbcDunqBaqhaudLyMeGFgG
↪" ],
    "chainId": 84,
    "version": 1,
    "script": "base64:AQQAAAAHJG1hdGNoMAUAAAACdHgG+RXSzQ==",
    "name": "faucet",
    "description": "",
```

(continues on next page)

```
    "height": 3805
}
```

14. Sponsorship Transaction

A transaction that establishes or cancels a sponsorship.

The sponsoring mechanism allows addresses to pay fees for script call transactions and transfer transactions in the sponsor asset, replacing WEST.

Transaction data structures

Signing:

Field	Data type	Description
sender	ByteStr	Address of the transaction sender
assetId	Byte	Sponsorship asset (token) ID − *optional field*
fee	Long	*WE Mainnet transaction fee*
isEnabled	Bool	Set the sponsorship (`true`) or cancel it (`false`)
type	Byte	Transaction number (**14**)
password	String	Keypair password in the node keystore − *optional field*
version	Byte	Transaction version

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**14**)
id	Byte	Sponsorship transaction ID
sender	ByteStr	Address of the transaction sender
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
assetId	Byte	Sponsorship asset (token) ID − *optional field*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), − *optional field*
proofs	List (ByteStr)	Array of transaction proofs (**in base58 format**)
chainId	Byte	Identifying byte of the network (Mainnet - **87** or **V**)
version	Byte	Transaction version
isEnabled	Bool	Set the sponsorship (`true`) or cancel it (`false`)
height	Byte	Height of transaction execution

JSON:

Version 1

Signing:

```
{
    "sender": "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t",
    "assetId": "G16FvJk9vabwxjQswh9CQAhbZzn3QrwqWjwnZB3qNVox",
    "fee": 100000000,
    "isEnabled": false,
    "type": 14,
    "password": "1234",
    "version": 1
}
```

Broadcasting:

```
{
    "type": 14,
    "id": "Ht6kpnQJHJskN8kwszF8rck3E5tQiuiM1fEN42wGfdk7",
    "sender": "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t",
    "senderPublicKey": "Gt55fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUophy89",
    "fee": 100000000,
    "assetId": "G16FvJk9vabwxjQswh9CQAhbZzn3QrwqWjwnZB3qNVox",
    "timestamp": 1545986757233,
    "proofs": [
→"5TfgYS2dqh8QyN7Vu2tAYaioX5WM6rTSDPGbt4zrWS7QKTzojmR2kjppvGNj4tDPsYPbcDunqBaqhaudLyMeGFh7
→" ],
    "chainId": 84,
    "version": 1,
    "isEnabled": false,
    "height": 3865
}
```

Version 2

Signing:

```
{
    "sender": "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t",
    "assetId": "G16FvJk9vabwxjQswh9CQAhbZzn3QrwqWjwnZB3qNVox",
    "fee": 100000000,
    "isEnabled": false,
    "type": 14,
    "password": "1234",
    "version": 2,
    "atomicBadge":{
      "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
    }
}
```

Broadcasting:

```
{
    "type": 14,
    "id": "Ht6kpnQJHJskN8kwszF8rck3E5tQiuiM1fEN42wGfdk7",
    "sender": "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t",
    "senderPublicKey": "Gt55fKHesnQG2CXmsKf4TC8v9oB7bsy2AY56CUophy89",
    "fee": 100000000,
    "assetId": "G16FvJk9vabwxjQswh9CQAhbZzn3QrwqWjwnZB3qNVox",
    "timestamp": 1545986757233,
    "proofs": [
→"5TfgYS2dqh8QyN7Vu2tAYaioX5WM6rTSDPGbt4zrWS7QKTzojmR2kjppvGNj4tDPsYPbcDunqBaqhaudLyMeGFh7
→" ],
    "chainId": 84,
    "version": 2,
    "isEnabled": false,
    "height": 3865
}
```

15. SetAssetScript Transaction

A transaction to install or remove an asset script for an address. Asset script allows to verify transactions involving this or that asset (token) without using the blockchain transaction verification mechanism.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**15**)
version	Byte	Asset script transaction version
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
fee	Long	*WE Mainnet transaction fee*
script	Array[Byte]	Compiled script in **base64** format – *optional field*
assetId	Byte	Sponsorship asset (token) ID – *optional field*

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**15**)
id	Byte	Asset script transaction ID
sender	ByteStr	Address of the transaction sender
sender-Pub-licKey	PublicK-eyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*
proofs	List(ByteSt	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
chainId	Byte	Identifying byte of the network (Mainnet - **87** or **V**)
assetId	Byte	Sponsorship asset (token) ID – *optional field*
script	Ar-ray[Byte]	The compiled script is in **base64** format. If you leave this field empty (*null*), the script will be detached from the account
height	Byte	Height of transaction execution

JSON:

Version 1

Signing:

```
{
    "type": 15,
    "version": 1,
    "sender": "3N9vL3apA4j2L5PojHW8TYmfHx9Lo2ZaKPB",
    "password": "",
    "fee": 100000000,
    "script": "base64:AQQAAAAHJG1hdGNoMAUAAAACdHgG+RXSzQ==",
    "assetId": "7bE3JPwZC3QcN9edctFrLAKYysjfMEk1SDjZx5gitSGg"
}
```

Broadcasting:

```
{
    "type": 15,
    "id": "CQpEM9AEDvgxKfgWLH2HxE82iAzpXrtqsDDcgZGPAF9J",
    "sender": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
    "senderPublicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
    "fee": 100000000,
    "timestamp": 1549448710502,
    "proofs": [
→"64eodpuXQjaKQQ4GJBaBrqiBtmkjSxseKC97gn6EwB5kZtMr18mAUHPRkZaHJeJxaDyLzGEZKqhYoUknWfNhXnkf
→" ],
    "version": 1,
    "chainId": 84,
    "assetId": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "script": "base64:AQQAAAAHJG1hdGNoMAUAAAACdHgG+RXSzQ==",
```

(continues on next page)

```
    "height": 61895
}
```

101. GenesisPermission Transaction

A transaction to assign the first network administrator who distributes permissions to other participants.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**101**)
id	Byte	Transaction identifier
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*
signature	ByteStr	Transaction signature (**in base58 format**)
target	ByteStr	Address of a first administrator to be appointed
role	String	A permission to be assigned (for an administrator - `permissioner`)

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**101**)
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*
target	ByteStr	Address of a first administrator to be appointed
role	String	A permission to be assigned (for an administrator - `permissioner`)

102. Permission Transaction

Issuing or revoking a participant's role (permission). Only a participant with the *permissioner role* can send 102 transactions to the blockchain.

The permissions you can specify in the `role` field:

- permissioner
- sender
- blacklister
- miner
- issuer
- contract_developer

- connection_manager

- contract_validator

- banned

You can find the roles (permissions) description in the *Permissions* article.

The 102. Permission Transaction can be included in the *atomic transaction* starting from version 2.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**102**)
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore − *optional field*
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
target	ByteStr	Participant's address for the permission assignment
opType	String	Type of operation: `add` - add a permission; `remove` - remove a permission
dueTimes-tamp	Long	Role validity **Unix Timestamp** (in milliseconds) − *optional field*
version	Byte	Transaction version

Broadcasting:

Field	Data type	Description
senderPub-licKey	PublicKey-Account	Transaction sender public key
role	String	A permission to be assigned (for an administrator - `permissioner`)
sender	ByteStr	Address of the transaction sender
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
fee	Long	*WE Mainnet transaction fee*
opType	String	Type of operation: `add` - add a permission; `remove` - remove a permission
id	Byte	ID of a transaction for permission adding or removing
type	Byte	Transaction number (**102**)
dueTimes-tamp	Long	Role validity **Unix Timestamp** (in milliseconds) − *optional field*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), − *optional field*
target	ByteStr	Address of a first administrator to be appointed
atom-icBadge	Boolean	Flag that indicates whether the transaction can be included in an *atomic* transaction

JSON:

Version 1

Signing:

```
{
  "type": 102,
  "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
  "password": "",
  "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
  "fee": 0,
  "target": "3GPtj5osoYqHpyfmsFv7BMiyKsVzbG1ykfL",
  "opType": "add",
  "role": "contract_developer",
  "dueTimestamp": null,
  "version": 1
}
```

Broadcasting:

```
{
  "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
  "role": "contract_developer",
  "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
  "proofs": [

↪"5ABJCRTKGo6jmDZCRWcLQc257CCeczmcjmtfJmbBE7TP3KsVkwvisH9kEkfYPckVCzEMKZTCd3LKAPcN8o4Git3j
↪"
  ],
  "fee": 0,
  "opType": "add",
  "id": "8zVUH7nsDCcpwyfxiq8DCTgqL7Q23FW1KWepB9EZcFG6",
  "type": 102,
  "dueTimestamp": null,
  "timestamp": 1559048837487,
  "target": "3GPtj5osoYqHpyfmsFv7BMiyKsVzbG1ykfL"
  "version": 1
}
```

Version 2

Signing:

```
{
  "type": 102,
  "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
  "password": "",
  "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
  "fee": 0,
  "target": "3GPtj5osoYqHpyfmsFv7BMiyKsVzbG1ykfL",
  "opType": "add",
  "role": "contract_developer",
  "dueTimestamp": null,
```

(continues on next page)

```
  "version": 2
}
```

Broadcasting:

```
{
  "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
  "role": "contract_developer",
  "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
  "proofs": [

↪"5ABJCRTKGo6jmDZCRWcLQc257CCeczmcjmtfJmbBE7TP3KsVkwvisH9kEkfYPckVCzEMKZTCd3LKAPcN8o4Git3j
↪"
  ],
  "fee": 0,
  "opType": "add",
  "id": "8zVUH7nsDCcpwyfxiq8DCTgqL7Q23FW1KWepB9EZcFG6",
  "type": 102,
  "dueTimestamp": null,
  "timestamp": 1559048837487,
  "target": "3GPtj5osoYqHpyfmsFv7BMiyKsVzbG1ykfL"
  "version": 2
  "atomicBadge": null
}
```

103. CreateContract Transaction

Transaction creating a *smart contract*. The byte representation of this transaction after it is signed must not exceed **150 kilobytes**.

103 transaction can only be signed by a user with the **contract_developer** *permission*.

Transaction data structures

Signing:

Field	Data type	Description
fee	Long	*Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
image	Array[By	The name of the Docker image of the Docker smart contract; the field is used up to and including version 6 of the transaction; starting with version 7, the `storedContract.image` field is used instead
imageHash	Array[By	Docker image hash of the Docker smart contract; the field is used up to and including version 6 of the transaction; starting with version 7, the `storedContract.imageHash` field is used instead
contractName	Array[By	Smart contract name (if downloaded from a pre-installed repository) or its full address (if the smart contract repository is not specified in the node configuration file)
sender	ByteSt	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
params	List[Da	Input and output data of the smart contract; entered using the `type`, `value` and `key` fields separated by commas – *optional field*
params.ke	Byte	Parameter key
params.ty	Byte	Parameter type; possible values: `binary bool integer string`
params.va	Byte	Parameter value
type	Byte	Transaction number (**103**)
version	Byte	Transaction version
apiVersion	Byte	API version for the Docker smart contract gRPC methods (see *gRPC services used by Docker smart contracts*); the field is used up to and including version 6 of the transaction; from version 7 onwards, the `storedContract.apiVersion` field is used instead
validationPolicy.type	String	Smart contract validation policy type
payments.am		An integer that specifies the number of assets to be transferred to the contract; in the `amount` field the lower bits correspond to the fractional parts of the number of assets to be transferred, if its `decimals` is not zero – *the field is optional*
payments.ass		The identifier of the asset transferred to the contract; the `assetId` field must be empty for the WEST system token to be transferred; – *the field is optional*
atomicBadge	Boolea	Flag that indicates whether the transaction can be included in an *atomic* transaction
isConfidential	Boolea	A flag that indicates whether the contract will support *operation in confidential mode*
groupParticipants	Set[Ad	Addresses authorized to access *confidential data*
groupOwners	Set[Ad	Addresses that can modify the groupParticipants and groupOwners lists
storedContract.byte	Array[By	*WASM smart contract* bytecode; the field is used starting with transaction version 7
storedContract.byte	Array[By	*WASM smart contract* bytecode hash; the field is used starting with transaction version 7
storedContract.imag	Array[By	The name of the Docker image of the Docker smart contract; the field is used starting with version 7 of the transaction
storedContract.imag	Array[By	Docker image hash of the Docker smart contract; the field is used starting with version 7 of the transaction

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**103**)
id	Byte	ID of a CreateContract transaction
sender	ByteS	Address of the transaction sender
sender-Pub-licKey	Pub-licK-ey-Ac-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), – *optional field*
proofs	List(E	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
image	Ar-ray[B]	Docker smart contract name (if loaded from a pre-installed repository) or its full address (if the Docker smart contract repository is not specified in the node configuration file); the field is used up to and including version 6 of the transaction; starting with version 7, the `storedContract.image` field is used instead
image-Hash	Ar-ray[B]	Docker image hash of the Docker smart contract; the field is used up to and including version 6 of the transaction; starting with version 7, the `storedContract.imageHash` field is used instead
con-tract-Name	Ar-ray[B]	Smart contract name
params	List[D	Input and output data of the smart contract; entered using the `type`, `value` and `key` fields separated by commas – *optional field*
params.l	Byte	Parameter key
params.t	Byte	Parameter type; possible values: `binary bool integer string`
params.v	Byte	Parameter value
height	Byte	Height of transaction execution
apiVer-sion	Byte	API version for the Docker smart contract gRPC methods (see *gRPC services used by Docker smart contracts*); the field is used up to and including version 6 of the transaction; from version 7 onwards, the `storedContract.apiVersion` field is used instead
valida-tion-Pol-icy.type	String	Smart contract validation policy type
pay-ments.ar		An integer that specifies the number of assets to be transferred to the con-tract; in the `amount` field the lower bits correspond to the fractional parts of the number of assets to be transferred, if its `decimals` is not zero – *the field is optional*
pay-ments.as		The identifier of the asset transferred to the contract; the `assetId` field must be empty for the WEST system token to be transferred; – *the field is optional*
atom-icBadge	Boole:	Flag that indicates whether the transaction can be included in an *atomic* transaction
isCon-fiden-tial	Boole:	A flag that indicates whether the contract will support *operation in confi-dential mode*
group-Partic-ipants	Set[A	Addresses authorized to access *confidential data*
groupOw ers	Set[A	Addresses that can modify the groupParticipants and groupOwners lists
stored-Con-tract.byt	Ar-ray[B]	*WASM smart contract* bytecode; the field is used starting with transaction version 7

JSON:

Version 2

Signing:

```
{
  "type": 103,
  "version": 2,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "password": "signing-key-password",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null
}
```

Broadcasting:

```
{
  "id": "4WVhw3QdiinpE5QXDG7QfqLiLanM7ewBw4ChX4qyGjs2",
  "type": 103,
  "version": 2,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "senderPublicKey": "YNpp7chAaudMqEtSZZPyN4GYLJ5ZTXdjCXrQdszzuRp",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "proofs": [

↪"4vqLnpJRFpcDgM5vgi78DpZnVfqztsARHNb7Hbmq3mQBjS3SRnzFAiYjRvPazEVMhBM9cE4Rcp6H5K29kk75Uxyh
↪"
  ]
}
```

Version 3

Signing:

```
{
  "type": 103,
  "version": 3,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "password": "signing-key-password",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null
}
```

Broadcasting:

```
{
  "id": "4WVhw3QdiinpE5QXDG7QfqLiLanM7ewBw4ChX4qyGjs2",
  "type": 103,
  "version": 3,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "senderPublicKey": "YNpp7chAaudMqEtSZZPyN4GYLJ5ZTXdjCXrQdszzuRp",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null,
  "proofs": [

→"4vqLnpJRFpcDgM5vgi78DpZnVfqztsARHNb7Hbmq3mQBjS3SRnzFAiYjRvPazEVMhBM9cE4Rcp6H5K29kk75Uxyh
→"
  ]
}
```

Version 4

Signing:

```
{
  "type": 103,
  "version": 4,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "password": "signing-key-password",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null,
  "validationPolicy": {
    "type": "majority"
  },
  "apiVersion": "1.0"
}
```

Broadcasting:

```
{
  "id": "4WVhw3QdiinpE5QXDG7QfqLiLanM7ewBw4ChX4qyGjs2",
  "type": 103,
  "version": 4,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "senderPublicKey": "YNpp7chAaudMqEtSZZPyN4GYLJ5ZTXdjCXrQdszzuRp",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null,
  "proofs": [
```

```
↪"4vqLnpJRFpcDgM5vgi78DpZnVfqztsARHNb7Hbmq3mQBjS3SRnzFAiYjRvPazEVMhBM9cE4Rcp6H5K29kk75Uxyh
↪"
    ]
}
```

Version 5

Signing:

```
{
  "type": 103,
  "version": 5,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "password": "signing-key-password",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null,
  "validationPolicy": {
    "type": "majority"
  },
  "apiVersion": "1.0"
}
```

Broadcasting:

```
{
  "id": "4WVhw3QdiinpE5QXDG7QfqLiLanM7ewBw4ChX4qyGjs2",
  "type": 103,
  "version": 5,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "senderPublicKey": "YNpp7chAaudMqEtSZZPyN4GYLJ5ZTXdjCXrQdszzuRp",
  "contractName": "SOME_CONTRACT_NAME",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "key": "int",
      "type": "integer",
      "value": 24
    },
```

```
    {
      "key": "bool",
      "type": "boolean",
      "value": true
    },
    {
      "key": "blob",
      "type": "binary",
      "value": "base64:YWxpY2U="
    }
  ],
  "fee": 0,
  "timestamp": 1665267880,
  "feeAssetId": null,
  "atomicBadge": {
    "trustedSender": "SOME_SENDER_ACCOUNT_ADDRESS"
  },
  "proofs": [

→"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
→"
  ],
  "validationPolicy": {
    "type": "SOME_VALIDATION_POLICY_NAME"
  },
  "apiVersion": "SOME_API_VERSION",
  "payments": [
    {
      "amount": 100
    },
    {
      "assetId": "SOME_ASSET_ID",
      "amount": 100
    }
  ]
}
```

Version 6

Signing:

```
{
  "type": 103,
  "version": 6,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "password": "signing-key-password",
  "contractName": "Your contract name",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
```

```
      "type": "string",
      "key": "test_key",
      "value": "test_value"
    }
  ],
  "fee": 100000000,
  "timestamp": 1651487626477,
  "feeAssetId": null,
  "atomicBadge": null,
  "validationPolicy": {
    "type": "majority"
  },
  "apiVersion": "1.0"
  "isConfidential": true
  "groupParticipants" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
→"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
  "groupOwners" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
→"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"]
}
```

Broadcasting:

```
{
  "id": "4WVhw3QdiinpE5QXDG7QfqLiLanM7ewBw4ChX4qyGjs2",
  "type": 103,
  "version": 6,
  "sender": "3NpN3HyHzGj7Ny1k5F9zMMQ2n54TZg86G9D",
  "senderPublicKey": "YNpp7chAaudMqEtSZZPyN4GYLJ5ZTXdjCXrQdszzuRp",
  "contractName": "SOME_CONTRACT_NAME",
  "image": "registry.yourdomain.com/test-docker-repo/contract:v1.0.0",
  "imageHash": "573387bbf50cfdeda462054b8d85d6c24007f91044501250877392e43ff5ed50",
  "params": [
    {
      "key": "int",
      "type": "integer",
      "value": 24
    },
    {
      "key": "bool",
      "type": "boolean",
      "value": true
    },
    {
      "key": "blob",
      "type": "binary",
      "value": "base64:YWxpY2U="
    }
  ],
  "fee": 0,
  "timestamp": 1665267880,
  "feeAssetId": null,
  "atomicBadge": {
```

```
    "trustedSender": "SOME_SENDER_ACCOUNT_ADDRESS"
  },
  "proofs": [

↪"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
↪"
  ],
  "validationPolicy": {
    "type": "SOME_VALIDATION_POLICY_NAME"
  },
  "apiVersion": "SOME_API_VERSION",
  "payments": [
    {
      "amount": 100
    },
    {
      "assetId": "SOME_ASSET_ID",
      "amount": 100
    }
  ]
  "isConfidential": true
  "groupParticipants" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
↪"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
  "groupOwners" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
↪"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"]
}
```

Version 7

Signing:

```
{
  "type": 103,
  "version": 7,
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "contractName": " not world",
  "params": [
    {
      "type": "integer",
      "value": 0,
      "key": "count"
    }
  ],
  "fee": 100000000,
  "feeAssetId": null,
  "validationPolicy":
    {
      "type": "any"
    },
  "payments": [],
```

```
  "isConfidential": false,
  "groupParticipants": [],
  "groupOwners": [],
  "storedContract":
    {
      "bytecode": "AGFzbQEAAAABNwdgA39/fgF/YAADf39/YAR/f39/AX9gBH9/f38Cf35gBH9/f38Df39/
→YAR/f39/An9/
→YAF+AX8CnAEHA2VudGtZW1vcnkCAQIQBGVudjAPc2V0X3Nb3JhZ2VfaW50AAAEZW52MA1nZXRfdHhfc2VuZGVyAAEEZW52MBJzZ
→AUEQC38AQRoLfwBBIAsHRwUMX2NvbnN0cnVjdG9yAAYHY291bnRlcgAHD3Jlc3RvcmVfY291bnRlcgAICl9fZGF0YV9lbmQDAQtfX
→AkBBAEEAQZWAgIAAQQUQhICAgAAhAiEBIgMNABCBgICAABohBCIDDQAQgYCAgAAhBRoiAw0AIAIAEAUhYCAgAAhASIDDQBBr
→",
      "bytecodeHash": "083b7d0cb08b4a30f3d9f96a30ede04680623f73527432f947d5d5880e670625"
    }
}
```

Broadcasting:

```
{
  "senderPublicKey" :
→"5oKuxwiRmqHnr7vCAHK3VRJBhg9andjskfX11HpmJcYp8JifBXisz4KEKFD3pbRum3PWHDf4ZKkoCAgrrsLbp8HH
→",
  "isConfidential" : false,
  "fee" : 100000000,
  "payments" : [ ],
  "groupOwners" : [ ],
  "type" : 103,
  "params" : [ ],
  "version" : 7,
  "atomicBadge" : null,
  "groupParticipants" : [ ],
  "sender" : "3Hakpx6EE4fDb7Vd7EaWMG1HT9UJezLeVcG",
  "feeAssetId" : null,
  "storedContract" : {
    "bytecode" : "AGFzbQEAAAABJQVgBH9/f38Df39/YAN/f34Bf2AEf39/fwJ/fmAAAX9gAn9/
→AX8CSgQDZW52Bm1lbW9yeQIBAhAEZW52MARqb2luAAAEZW52MA9zZXRfc3RvcmFnZV9pbnQAAQRlbnYwD2dldF9zdG9yYWdlX2lud
→AUEQC38AQSALfwBBIAsHOQQMX2NvbnN0cnVjdG9yAAMLaW5jcmVtZW50XzEABApfX2RhdGFfZW5kAwELX19oZWFwX2Jhc2UDAgrXA
→QQAhAANAAkAgAEEKRw0AQQQAPCwJAQZqAgIAAQQBBmoCAgABBBhCAgICAACECIQEiAw0AIAEgAiAAQZCAgIAAakEBEICAgIAAIQIhAS
→",
    "bytecodeHash" : "c2f116a528291d6cbcadc308edd8a1f294c4656009705916f3f0929150838388"
  },
  "contractName" : "name",
  "id" : "GQ4CaT9vyKUsK7tFrg2F7bfqEspwVeaPFtE2tpAwAtye",
  "validationPolicy" : {
    "type" : "any"
  },
  "timestamp" : 1704962852721
}
```

Version 4

The 4th version of this transaction configures validation of the execution results of the updated smart contract using the `validationPolicy.type` field (see section *Validation of smart contracts*).

Variants of validation policies:

- `any` - the general validation policy is kept in the network: to mine the updated smart contract, the miner signs the corresponding *105* transaction. Also, this parameter is set if there are no registered validators in the network.

- `majority` - a transaction is considered valid if it is confirmed by the majority of validators: **2/3** of the total number of registered addresses with the **contract_validator** permission.

- `majorityWithOneOf(List[Address])` - the transaction is considered valid if the majority of validators is collected, among which there is at least one of the addresses included in the parameter list. The addresses included in the list must have a valid **contract_validator** permission.

> **Warning:** If you select the `majorityWithOneOf(List[Address])` validation policy, fill the address list. Passing an empty list is not allowed.

Version 5

In the version **5** of this transaction a user can transfer his assets to the balance of a contract. To do this, an array of assets and their number are specified in the `payments` field. Both the system WEST token and any other asset created in the network can be transferred. Version 5 of this transaction can be used starting from release 1.12 after the 1120 *feature activation*.

In private networks, the 103 transaction allows to install Docker images of smart contracts from repositories other than those stated in the `docker-engine` section of the node configuration file.

If you need to install a smart contract from a registry not included in the list in the node configuration file, specify the full address of the smart contract in the registry you have created in the `name` field of the 103 transaction.

Example of a request to publish a smart contract from a repository not specified in the node configuration file:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "image": "customregistry.com:5000/stateful-increment-contract:latest",
  "fee" : 100000000,
  "imageHash" :
↪"ad6d0f8a61222794da15571749bc9db08e76b6a120fc1db90e393fc0ee9540d8",
  "type" : 103,
  "params" : [ {
    "type" : "string",
    "value" : "Value_here",
    "key" : "data"
  }, {
```

(continues on next page)

```
      "type" : "integer",
      "value" : 500,
      "key" : "length"
  } ],
  "version" : 5,
  "atomicBadge" : null,
  "apiVersion" : "1.0",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : null,
  "proofs" : [
→"L521YncSMJDPqwBjQyS7m7Q6tseAw51nYE8iiPChEALx7S2WvpSosCVtWkXxh2ZqJ6LHkCvjVjRVuVs793kzjw8
→" ],
  "contractName" : "grpc_validatable_statefull here_often",
  "id" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "validationPolicy" : {
    "type" : "any"
  },
  "timestamp" : 1625732696641,
  "height" : 1028130
}
```

Version 6

Version **6** of this transaction implements support for *confidential smart contracts*. When registering a contract using the 6th version of the transaction, you can specify if the contract is confidential, and define a set of node addresses that have access to the smart contract confidential data. The following fields are implemented for this purpose:

Fields for working with confidential smart contracts:

The following fields are implemented for this purpose:

- the `isConfidential` flag determines whether the smart contract is *confidential* or not;

- the `groupParticipants` field specifies the group (policy) member nodes who have access to confidential smart-contract data; the maximum group size is 1024 members;

- the `groupOwners` field specifies the nodes that can modify the `groupParticipants` and `groupOwners` lists using the *UpdateContract* transaction; no more than 1024 nodes can be specified in this field.

Important: You cannot assign `true` to the `isConfidential` field when creating a contract if the `groupParticipants` field contains less than three participants with the `contract-validator` *role*.

You cannot assign `false` to the `isConfidential` field when creating a contract if the `groupParticipants` and `groupOwners` fields are not empty.

You cannot assign any value to the `payments` field when creating a contract if the `isConfidential` field is set to `true`.

You cannot pass parameters to the `params` field when creating a contract if the `isConfidential` field is set to `true`.

You can use the **6** th version of this transaction starting with 1.13 release after the **1130** *feature activation*.

Version 7 ˘ Create WASM Contract Transaction

In version **7** of this transaction, support for *WASM smart contracts* is implemented. For this purpose `image`, `imageHash`, `apiVersion` fields are removed and `storedContract` field is implemented. Depending on the type of smart contract, this field has the following structure:

- for Docker smart contracts:

```
"storedContract" : {
    "image" : <image docker> in Array[Byte]
    "imageHash": ssh256 from the image docker
    "apiVersion" : "1.10",
}
```

- for WASM smart contracts:

```
"storedContract" : {
    "bytecode" : <bytecode contracts> in Array[Byte]
    "bytecodeHash" : "Sha256 from the <bytecode contracts>"
}
```

You can use the version **7** of this transaction starting with release 1.14.0 after *1140 feature activation*.

Important: In release 1.14.0, WASM smart contracts do not support *atomic transactions* and *confidential smart contracts*. Therefore, the following fields must have the following values for a WASM smart contract:

```
...
"atomicBadge" : null,
...
"isConfidential": false,
"groupParticipants": [],
"groupOwners": [],
...
```

---

104. CallContract Transaction

Transaction calling a *smart contract* for execution. The byte representation of this transaction after it is signed must not exceed **150 kilobytes**.

Signing of the transaction is performed by the initiator of the contract execution.

The `contractVersion` field of the transaction specifies the contract version:

- 1 – for a new contract;
- 2 – for an updated contract.

This field is only available for the transaction of the second version and higher: if the `version` field of the smart contract call transaction is set to 2 or more. The contract is updated using the *107* transaction.

Prior to release 1.14.0, if a contract was not executed or was executed with an error, 103 and 104 transactions were deleted and not included into a block. Starting with release 1.14.0, execution of a smart contract in case of an error is also stored in the blockchain. For more details on the smart contract error handling algorithm, see *105. ExecutedContract Transaction − Version 5* .

Transaction data structures

Signing:

Field	Data type	Description
contractId	ByteStr	Smart contract ID
fee	Long	*WE Mainnet transaction fee*
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore − *optional field*
type	Byte	Transaction number (**104**)
params	List[Da	Input and output data of the smart contract; entered using the `type`, `value` and `key` fields separated by commas − *optional field*
params.l	Byte	Parameter key
params.t	Byte	Parameter type. Possible values: `binary`, `bool`, `integer`, `string`
params.v	Byte	Parameter value
version	Byte	Transaction version
inputCommitment	Commitment	The field is used in transaction version 6 to handle *confidential smart contracts* . Field length is a constant equal to hash length in current cryptography
inputCommitmentOpt	Commitment	The field is used in transaction version 7 and is similar to the `inputCommitment` field − *optional field*
contractVersion		Contract version: for a new contract the value 1 is specified, for an updated contract − 2
feeAssetId	Byte	Identifier of a token for fee payment − *optional field*
payments.ar		An integer that specifies the number of assets to be transferred to the contract; in the `amount` field the lower bits correspond to the fractional parts of the number of assets to be transferred, if its `decimals` is not zero − *the field is optional*
payments.as		The identifier of the asset transferred to the contract; the `assetId` field must be empty for the WEST system token to be transferred; − *the field is optional*
atomicBadge	Boolean	Flag that indicates whether the transaction can be included in an *atomic* transaction
callFunc		Name of contract function
contractEng		Called contract type: `docker` or `wasm`

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**104**)
id	Byte	Smart contract call transaction ID
sender	ByteStr	Address of the transaction sender
sender-Pub-licKey	Pub-licK-eyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
feeAs-setId	Byte	Identifier of a token for fee payment − *optional field*
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds), − *optional field*
proofs	List(By	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
con-tractId	ByteStr	Smart contract ID
params	List[Da	Input and output data of the smart contract; entered using the `type`, `value` and `key` fields separated by commas − *optional field*
params.l	Byte	Parameter key
params.t	Byte	Parameter type. Possible values: `binary`, `bool`, `integer`, `string`
params.v	Byte	Parameter value
con-tractVer-sion		Contract version: for a new contract the value 1 is specified, for an updated contract − 2
pay-ments.ar		An integer that specifies the number of assets to be transferred to the contract; in the `amount` field the lower bits correspond to the fractional parts of the number of assets to be transferred, if its `decimals` is not zero − *the field is optional*
pay-ments.as		The identifier of the asset transferred to the contract; the `assetId` field must be empty for the WEST system token to be transferred; − *the field is optional*
atom-icBadge	Boolean	Flag that indicates whether the transaction can be included in an *atomic* transaction
input-Com-mit-ment	Com-mit-ment	The field is used in transaction version 6 to handle *confidential smart contracts*. The field length is a constant and is equal to the hash length in the current cryptography
input-Com-mit-men-tOpt	Com-mit-ment	The field is used in transaction version 7 and is similar to the `inputCommitment` field − *optional field*
call-Func		Name of contract function
con-tractEng		Called contract type: `docker` or `wasm`

JSON:

Version 2

Signing:

```
{
    "contractId": "2sqPS2VAKmK77FoNakw1VtDTCbDSa7nqh5wTXvJeYGo2",
    "fee": 10,
    "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
    "password": "",
    "type": 104,
    "params":
    [
        {
            "type": "integer",
            "key": "a",
            "value": 1
        },
        {
            "type": "integer",
            "key": "b",
            "value": 100

        }
    ],
    "version": 2,
    "contractVersion": 1
}
```

Broadcasting:

```
{
    "type": 104,
    "id": "9fBrL2n5TN473g1gNfoZqaAqAsAJCuHRHYxZpLexL3VP",
    "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
    "senderPublicKey": "2YvzcVLrqLCqouVrFZynjfotEuPNV9GrdauNpgdWXLsq",
    "fee": 10,
    "timestamp": 1549365736923,
    "proofs": [
→"2q4cTBhDkEDkFxr7iYaHPAv1dzaKo5rDaTxPF5VHryyYTXxTPvN9Wb3YrsDYixKiUPXBnAyXzEcnKPFRCW9xVp4v
→" ],
    "version": 2,
    "contractVersion": 1,
    "contractId": "2sqPS2VAKmK77FoNakw1VtDTCbDSa7nqh5wTXvJeYGo2",
    "params":
    [
        {
        "key": "a",
        "type": "integer",
        "value": 1
        },
        {
        "key": "b",
        "type": "integer",
```

```
        "value": 100
        }
    ]
}
```

Version 3

Signing:

```
{
    "contractId": "Dgk1hR7xRnDT1KJreaXCVtZLrnd5LJ8uUYtoZyQrV1LJ",
    "fee": 10000000,
    "sender": "3NpkC1FSW9xNfmAMuhRSRArLgnfyGyEry7w",
    "password": "",
    "type": 104,
    "params":
    [ {
      "type" : "string",
      "value" : "value",
      "key" : "data"
    }, {
      "type" : "integer",
      "value" : 500,
      "key" : "length"
    } ],
    "version": 3,
    "contractVersion": 1,
}
```

Broadcasting:

```
{
  "senderPublicKey" : "9Kgnqqxr5MU3PNrLgf1dkZL2HH6LBktB5Pv9L1cVELi1",
  "fee" : 10000000,
  "type" : 104,
  "params" : [ {
    "type" : "string",
    "value" : "data_response",
    "key" : "action"
  }, {
    "type" : "string",
    "value" : "000008_regular_data_request_2m3SgcnQz9LXVi9ETy3CFHVGM1EyiqJi3vvRRQUM3oPp",
    "key" : "request_id"
  }, {
    "type" : "string",
    "value" : "76.33",
    "key" : "value"
  }, {
    "type" : "string",
    "value" : "1627678789267",
    "key" : "timestamp"
```

```
  } ],
  "version" : 3,
  "contractVersion" : 1,
  "sender" : "3NpkC1FSW9xNfmAMuhRSRArLgnfyGyEry7w",
  "feeAssetId" : null,
  "proofs" : [
↪"4aanqYjaTVNot8Fbz5ixjwKSdqS5x3DdvzxQ4WsTaPcftYdoFx99xwLC3UPN91VAtez4RTMzaYb1TECaVxHHT9AH
↪" ],
  "contractId" : "Dgk1hR7xRnDT1KJreaXCVtZLrnd5LJ8uUYtoZyQrV1LJ",
  "id" : "55imLuEXyVpBXb1S64R5PRx9acQQHaEATPwYwUVpqjAT",
  "timestamp" : 1627678789267,
  "height" : 1076064
}
```

Version 4

Signing:

```
{
    "contractId": "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
    "fee": 10000000,
    "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
    "password": "",
    "type": 104,
    "params":
    [ {
      "type" : "string",
      "value" : "value",
      "key" : "data"
    }, {
      "type" : "integer",
      "value" : 500,
      "key" : "length"
    } ],
    "version": 4,
    "contractVersion": 3,
    "atomicBadge" : null
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "fee" : 10000000,
  "type" : 104,
  "params" : [ {
    "type" : "string",
    "value" : "value",
    "key" : "data"
  }, {
    "type" : "integer",
```

```
    "value" : 500,
    "key" : "length"
  } ],
  "version" : 4,
  "contractVersion" : 3,
  "atomicBadge" : null,
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : null,
  "proofs" : [
↪"2bpALen4diR7DTFhNQCrZKPueCPds2gFFPxe1KVzQwfRuGaK6QfvtpN8oqaZMsStoEHAa5DrTkKM8AuzHPYyMPVP
↪" ],
  "contractId" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "id" : "GBfibn8VjGmDS9ex4Nd4JNRLvDyvJjj8jLUUcbYwFTCf",
  "timestamp" : 1625732766458,
  "height" : 1028132
}
```

Version 5

Signing:

```
{
  "contractId": "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "fee": 10000000,
  "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
  "password": "",
  "type": 104,
  "params": [
    {
      "type" : "string",
      "value" : "value",
      "key" : "data"
    },
    {
      "type" : "integer",
      "value" : 500,
      "key" : "length"
    }
  ],
  "version": 5,
  "contractVersion": 3,
  "atomicBadge" : null
}
```

Broadcasting:

```
{
  "senderPublicKey": "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
    "fee": 0,
    "type": 104,
    "params": [
```

```
      {
        "key": "int",
        "type": "integer",
        "value": 24
      },
      {
        "key": "bool",
        "type": "boolean",
        "value": true
      },
      {
        "key": "blob",
        "type": "binary",
        "value": "base64:YWxpY2U="
      }
    ],
    "version": 5,
    "contractVersion": "3",
    "atomicBadge": {
      "trustedSender": "SOME_SENDER_ACCOUNT_ADDRESS"
    },
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "feeAssetId": null,
    "proofs": [

↪"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
↪"
    ],
    "contractId": "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
    "id": "GBfibn8VjGmDS9ex4Nd4JNRLvDyvJjj8jLUUcbYwFTCf",
    "timestamp": 1665267880,
    "payments": [
      {
        "amount": 100
      },
      {
        "assetId": "SOME_ASSET_ID",
        "amount": 100
      }
    ]
  }
```

Version 6

Signing:

```
{
  "contractId": "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "fee": 10000000,
  "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
  "password": "",
  "type": 104,
  "params": [
    {
      "type" : "string",
      "value" : "value",
      "key" : "data"
    },
    {
      "type" : "integer",
      "value" : 500,
      "key" : "length"
    }
  ],
  "version": 6,
  "contractVersion": 3,
  "atomicBadge" : null
  "inputCommitment" : "SOME_COMMITMENT"
}
```

Broadcasting:

```
{
  "senderPublicKey": "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "fee": 0,
  "type": 104,
  "params": [
    {
      "key": "int",
      "type": "integer",
      "value": 24
    },
    {
      "key": "bool",
      "type": "boolean",
      "value": true
    },
    {
      "key": "blob",
      "type": "binary",
      "value": "base64:YWxpY2U="
    }
  ],
  "version": 6,
  "contractVersion": "3",
```

```
  "atomicBadge": {
    "trustedSender": "SOME_SENDER_ACCOUNT_ADDRESS"
  },
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId": null,
  "proofs": [

→"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
→"
  ],
  "contractId": "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "id": "GBfibn8VjGmDS9ex4Nd4JNRLvDyvJjj8jLUUcbYwFTCf",
  "timestamp": 1665267880,
  "payments": [
    {
      "amount": 100
    },
    {
      "assetId": "SOME_ASSET_ID",
      "amount": 100
    }
  ]
  "inputCommitment" : "SOME_COMMITMENT"
}
```

Version 7

Signing:

```
{
  "contractId": "2TUhT1eRvpKcwxUfpUEB7BHEiXNVZnEbEMgZKnxuPXHJ",
  "fee": 10000000,
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "type": 104,
  "params": [
    {
      "type": "integer",
      "value": 7,
      "key": "count"
    }
  ],
  "version": 7,
  "contractVersion": 1,
  "feeAssetId": null,
  "payments": [],
  "callFunc": "restore_counter",
  "contractEngine": "wasm"
}
```

Broadcasting:

```
{
  "senderPublicKey" :
↪"5oKuxwiRmqHnr7vCAHK3VRJBhg9andjskfX11HpmJcYp8JifBXisz4KEKFD3pbRum3PWHDf4ZKkoCAgrrsLbp8HH
↪",
  "fee" : 100000000,
  "type" : 104,
  "params" : [
    {
      "type" : "boolean",
      "value" : true,
      "key" : "bool_value"
    },
    {
      "type" : "integer",
      "value" : 100000000,
      "key" : "int_value"
    },
    {
      "type" : "string",
      "value" : "Hello World",
      "key" : "string_value"
    },
    {
      "type" : "string",
      "value" : "3Hakpx6EE4fDb7Vd7EaWMG1HT9UJezLeVcG",
      "key" : "address_value"
    }
  ],
  "version" : 7,
  "contractVersion" : 1,
  "atomicBadge" : null,
  "sender" : "3Hakpx6EE4fDb7Vd7EaWMG1HT9UJezLeVcG",
  "feeAssetId" : null,
  "contractId" : "GyDvD8r2yXE1Kdu31TMkYtSW9i7F4qXXkfWpfKgkYxX",
  "id" : "3bGqThohX5KX79k9snWENduwgkmpfKDMjqtK3QGMH1me",
  "timestamp" : 1704963915571
  "payments" : [ ],
  "callFunc" : "update_storage",
  "contractEngine" : "wasm"
}
```

Version 5

In the version **5** of this transaction a user can transfer his assets to the balance of a contract. To do this, an array of assets and their number are specified in the `payments` field. Both the system WEST token and any other asset created in the network can be transferred. Version 5 of this transaction can be used starting from release 1.12 after the 1120 *feature activation*.

Version 6

Version **6** of this transaction implements the field required to handle *confidential smart contracts*.

The use of version 6 of this transaction is possible starting from release 1.13 after 1130 *feature activation*. After the 1130 feature activation, only version 6 of the transaction is used in the network.

Version 7

Version **7** of this transaction supports *WASM smart contracts*. The following fields are implemented for this purpose:

- `contractEngine` – defines the type of contract to be called; available values:

    - `wasm`

    - `docker`

- `callFunc` – the name of the WASM smart contract function to be called; the `_constructor` function cannot be specified in this field, as it is only called when the contract is created; for a Docker smart contract, the field must be left empty: `callFunc:    null`.

> **Attention:** The order and type of parameters passed to the function is important: in the `params` field of the JSON to be signed, the parameters must be specified exactly in the order in which the function specified in the `callFunc` field expects them.

You can use the version **7** of this transaction starting with release 1.14.0 after *1140 feature activation*.

Important: In release 1.14.0, WASM smart contracts do not support *atomic transactions* and *confidential smart contracts*. Therefore, for a WASM smart contract, the `inputCommitmentOpt` field must be absent and the `atomicBadge` field must be absent or set to `null` in the JSON to be signed.

---

105. ExecutedContract Transaction

Transaction writing *smart contract* execution result to its state. The byte representation of this transaction after signing must not exceed **150 kilobytes**.

105 Transaction contains all the fields (body) of the *103. CreateContract*, *104. CallContract*, *107. UpdateContract* transaction of the smart contract whose execution result must be written to its state (the `tx` field). The smart contract execution result is entered into its state from the corresponding parameters of the `params` field of 103 or 104 transaction.

The transaction is signed by the node that forms the block after sending the request to publish the transaction.

Data structure of a query for transaction broadcast

Field	Data type	Description
type	Byte	Transaction number (**105**)
id	Byte	ExecutedContract transaction ID
sender	ByteStr	Address of the transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
password	String	Keypair password in the node keystore − *optional field*
fee	Long	*Transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
tx	Array	Body of transaction 103 or 104 of an executed smart contract
results	List[DataEntry[_]]	List of possible results of smart contract execution; the field is used up to and including version 4 of the transaction; from version 5 onwards, the `resultsMap` field is used instead
resultsMap	Map[ByteStr, List]	A set of lists of possible results of smart contracts execution; the field is used since version 5; earlier versions use the `results` field instead
height	Byte	Height of the transaction execution − an *optional field*

Field	Data type	Description
assetOperations		A structured list of the smart contract actions with the assets available to it, including • issuing a new asset, • reissuing an asset, • burning an asset, • transferring an asset available to the contract to another user, • leasing an asset available to the contract to another user, • cancelling of leasing of tokens that have been leased before The field is used up to and including version 4 of the transaction; starting with version 5, the `assetOperationsMap` field is used instead

assetOperations.operationType		A service field that represents the operation type. The field can take the following values:
		`issue,`
		`reissue,` `burn,`
		`transfer, lease,`
		`cancel-lease`

assetOperations.version		Service field representing the object version

Table 9 – continued from previous page

Field	Data type	Description
assetOperations.assetId		When issuing assets, the field value is calculated using the gRPC method CalculateAssetId of the ContractService. When reissuing or burning an asset, the identifier determines which token is being reissued or burned. When transferring an asset, the identifier determines which asset is being transferred. In the case of sending the WEST system token, the assetId field must be omitted or equal to null.
assetOperations.name		Asset name
assetOperations.description		Asset description
assetOperations.quantity		When issuing an asset, the field specifies the total amount of the issued asset. When reissuing an asset – the amount of the reissued asset
assetOperations.decimals		When issuing assets, the field specifies the number of decimal places of the asset being issued
assetOperations.isReissuable		The flag indicating if the asset can be reissued

Table 9 – continued from previous page

Field	Data type	Description
assetOperations.nonce		When issuing asserts, the field value is used to calculate the assetId. It cannot be equal to 0. The range of acceptable values is from -128 to 127. Several assets with the same `nonce` cannot be released within one contract call
assetOperations.amount		When burning assets, the field specifies the amount of asset to be burned. When transferring assets, the field specifies the amount of the transferred asset
assetOperations.recipient		When transferring assets, the field specifies the address of the user to whom the contract transfers the assets
assetOperationsMap	Map[ByteStr, List]	The field is similar to the `assetOperations` field, and is used since version 5 of the transaction
readings	ReadDescriptor	The field is used to handle *confidential smart contracts* and describes the public data reading order on the contract side
readingsHash	ByteStr	The field is used to handle *confidential smart contracts* and represents a hash of readings and the results of the readings. The field has a fixed length
outputCommitment	Commitment	The field is used in transaction version 4 to handle *confidential smart contracts*. Field length is a constant equal to hash length in current cryptography

continues on next page

Table 9 – continued from previous page

Field	Data type	Description
outputCommitmentOpt	Commitment	The field is used in transaction version 5 and is similar to the `outputCommitment` field – *optional field*
statusCode	Integer	The execution status of the smart contract. The following values are available: 0, 1, 2
errorMessage	String	Error description with error code

JSON:

Version 2

Broadcasting:

```
{
    "type": 105,
    "version": 2,
    "id": "38GmSVC5s8Sjeybzfe9RQ6p1Mb6ajb8LYJDcep8G8Umj",
    "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
    "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
    "password": "",
    "fee": 500000,
    "timestamp": 1550591780234,
    "proofs": [
→"5whBipAWQgFvm3myNZe6GDd9Ky8199C9qNxLBHqDNmVAUJW9gLf7t9LBQDi68CKT57dzmnPJpJkrwKh2HBSwUer6
→" ],
    "tx":
        {
            "type": 103,
            "id": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky",
            "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
            "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
            "fee": 500000,
            "timestamp": 1550591678479,
            "proofs": [
→"yecRFZm9iBLyDy93bDVaNo1PR5Qkkic7196GAgUt9TNH1cnQphq4yGQQ8Fxj4BYA4TaqYVw5qxtWzGMPQyVeKYv
→" ],
            "version": 2,
            "image": "stateful-increment-contract:latest",
            "imageHash":
→"7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65",
            "contractName": "stateful-increment-contract",
            "params": [],
            "height": 1619
        },
    "results": [],
    "height": 1619,
```

(continues on next page)

```
    "atomicBadge" : null
}
```

Version 3

Broadcasting:

```
{
    "type": 105,
    "version": 3,
    "id": "SOME_TX_ID",
    "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
    "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
    "fee": 0,
    "timestamp": 1665267880,
    "proofs": [

→"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
↪"
    ],
    "tx": { // inner (executed) tx json-object
      "id": "SOME_INNER_TX_ID",
      // ...
        },
    "results": [
        {
          "key": "int",
          "type": "integer",
          "value": 24
        },
        {
          "key": "bool",
          "type": "boolean",
          "value": true
        },
        {
          "key": "blob",
          "type": "binary",
          "value": "base64:YWxpY2U="
        }
      ],
    "assetOperations": [
        {
          "operationType": "issue",
          "version": 1,
          "assetId": "SOME_ASSET_ID",
          "name": "Gigacoin",
          "description": "Gigacoin",
          "quantity": 10000000000,
          "decimals": 8,
          "isReissuable": true,
```

```
        "nonce": 1 // SOME_NONCE
      },
      {
        "operationType": "burn",
        "version": 1,
        "assetId": "SOME_ASSET_ID",
        "amount": 1000
      },
      {
        "operationType": "reissue",
        "version": 1,
        "assetId": "SOME_ASSET_ID",
        "quantity": 10000000000,
        "isReissuable": true
      },
      {
        "operationType": "transfer",
        "version": 1,
        "recipient": "SOME_RECIPIENT_ACCOUNT_ADDRESS",
        "assetId": "SOME_ASSET_ID",
        "amount": 1000
      }
      {
        "operationType": "lease",
        "leaseId":  "SOME_LEASE_ID",
        "nonce": 1,
        "recipient": "SOME_RECIPIENT_ACCOUNT_ADDRESS"
        "amount": 1000
      }
      {
        "operationType": "cancel-lease",
        "leaseId":  "SOME_LEASE_ID"
      }
    ]
  "resultsHash": "SOME_RESULTS_HASH",
  "validationProofs": [],
}
```

Version 4

Broadcasting:

```
{
  "type": 105,
  "version": 4,
  "id": "SOME_TX_ID",
  "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
  "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
  "fee": 0,
  "timestamp": 1665267880,
  "proofs": [
```

```
→"32mNYSefBTrkVngG5REkmmGAVv69ZvNhpbegmnqDReMTmXNyYqbECPgHgXrX2UwyKGLFS45j7xDFyPXjF8jcfw94
→"
  ],
  "tx": { // inner (executed) tx json-object
    "id": "SOME_INNER_TX_ID",
    // ...
      },
  "results": [
      {
        "key": "int",
        "type": "integer",
        "value": 24
      },
      {
        "key": "bool",
        "type": "boolean",
        "value": true
      },
      {
        "key": "blob",
        "type": "binary",
        "value": "base64:YWxpY2U="
      }
    ],
  "assetOperations": [
      {
        "operationType": "issue",
        "version": 1,
        "assetId": "SOME_ASSET_ID",
        "name": "Gigacoin",
        "description": "Gigacoin",
        "quantity": 1000000000,
        "decimals": 8,
        "isReissuable": true,
        "nonce": 1 // SOME_NONCE
      },
      {
        "operationType": "burn",
        "version": 1,
        "assetId": "SOME_ASSET_ID",
        "amount": 1000
      },
      {
        "operationType": "reissue",
        "version": 1,
        "assetId": "SOME_ASSET_ID",
        "quantity": 1000000000,
        "isReissuable": true
      },
      {
        "operationType": "transfer",
```

```
      "version": 1,
      "recipient": "SOME_RECIPIENT_ACCOUNT_ADDRESS",
      "assetId": "SOME_ASSET_ID",
      "amount": 1000
    }
    {
      "operationType": "lease",
      "leaseId":  "SOME_LEASE_ID",
      "nonce": 1,
      "recipient": "SOME_RECIPIENT_ACCOUNT_ADDRESS"
      "amount": 1000
    }
    {
      "operationType": "cancel-lease",
      "leaseId":  "SOME_LEASE_ID"
    }
  ]
  "resultsHash": "SOME_RESULTS_HASH",
  "validationProofs": [],
  "readings": [ReadDescriptor1, ..., ReadDescriptorN],
  "readingsHash" : "SOME_READINGS_HASH",
  "outputCommitment" : "SOME_COMMITMENT"
}
```

Version 5

Broadcasting:

```
{
  "type": 105,
  "version": 5,
  "id": "HydNFEUeCj5DXFfHm32CrpcohvRvTABqdoFERtosgf5a",
  "sender": "3NdJB3vGAAQm2xQc2SAEhGNqDtXpL7YCn3v",
  "senderPublicKey": "9e4poNdEc9KF1qRxRJLbhqx6hrcjieQP2YcPiBdd3fpT",
  "fee": 0,
  "timestamp": 1708355888775,
  "proofs": [

→"3VHTSQh5HKkt1KGwhZg39WhPVNbNE5GnmyAD82no92e8CbYthh1KepjECyAcXXVu8QPoduscdZnnnrPtyfHZYjSR
→"
  ],

  "tx":
      {
        "type": 104,
        "version": 7,
        "sender": "3Nremv58EXSYK2qa5bhMeGnm1f2pRqLnv34",
        "senderPublicKey": "4sCvMtLD9MJUaw6dQrjnzWhrM6D32nrQcgQk5ULtQUXw",
        "contractEngine": "docker",
        "callFunc": null,
        "fee": 10000000,
```

```
          "feeAssetId": null,
          "payments": [],
          "params": [
            {
              "type": "integer",
              "value": 1,
              "key": "error_code"
            }
          ],
          "contractVersion": 1,
          "atomicBadge": null,
          "proofs": [

→"emoXX9D1tknstbNjkxAdERQsVz59AM9XchH9fwfeyUYNdkwSmBEU1FRfH71gDyyPHs3t4e6hrXqNiNUTrLkQ7pc
→"
          ],
          "contractId": "4K6gRgAhnzzbHXaGSRbWnjtU2r4kYUw61uwPuKJq1ims",
          "id": "Ecectk1L6T6TFAtUcQH2XerXNGT4gm7tMKBf2NnNKBjK",
          "timestamp": 1708355888031
        },
   "resultsHash": "xyw95Bsby3s4mt6f4FmFDnFVpQBAeJxBFNGzu2cX4dM",
   "validationProofs": [],
   "readings": [],
   "readingsHash" : null,
   "resultsMap": {},
   "assetOperationsMap": {},
   "statusCode": 2,
   "errorMessage": "Rejected because the CircuitBreaker is in the Open state, attempting␣
→to close in 53 millis"
 }
```

Version 3

In the version **3** of this transaction you can pass a sequence of operations on assets, such as transferring tokens from the user balance to the contract balance, in the `assetOperations` field.

- Both the WEST system token and any other asset created on the network can be transferred using the `transfer` operation.

- The `issue`, `reissue`, `burn` operations can be performed with any token except the WEST system token.

- The `lease` and `cancel-lease` operations work only with the WEST system token.

Important: If the `isReissuable` field is set to `False`, i.e. tokens are not allowed to be reissued, then it will be impossible to change this value in the future.

The version 3 of this transaction can be used starting with the 1.12 release after the 1120 *feature activation*. After the 1120 feature activation, only version 3 of this transaction is used on the network.

You can use the `lease` and `cancel-lease` operations starting from release 1.12.3 after the 1123 *feature activation*.

Version 4

Version **4** of this transaction implements the fields required to handle *confidential smart contracts*.

The use of version 4 of this transaction is possible starting from release 1.13 after 1130 *feature activation*. After the 1130 feature activation, only version 4 of the transaction is used in the network.

Version 5

In version **5** of this transaction, the algorithm for handling smart contract errors has been changed:

- In case of successful execution, the node (as in the previous versions) generates and sends to the blockchain a 105 transaction, which contains the details of the change to the state within the original 103 or 104 transaction in the following format:

```
"results": [
  {
    "type": "string",
    "value": "[{""accountNumber"":""1119810""}]",
    "key": "accounts"
  }
]
```

- In case of an error, transaction 105 is also generated, and the smart contract execution with the error is stored in the blockchain when the transaction reaches a quorum.

This version of the transaction implements the fields required to capture smart contract execution statuses:

- `statusCode` – smart contract execution status; the following values are available:

    - 0 – Success – successful contract execution; the intended result was obtained;

    - 1 – Error – business logic error; the contract ended in error, but may end successfully in the future;

    - 2 – Failure – system error; a blockchain component has failed, such as in the case of a timeout.

- `errorMessage` – error description; in the case of a WASM smart contract, the field contains *error code*; in the case of a Docker smart contract, a user-defined error or other error is returned.

In addition, data from multiple smart contracts can now be transferred in a single transaction. Whereby the fields have changed as follows:

- the `results` field has been replaced by a `resultsMap` field, which can pass a list of results for each of the smart contracts;

- the `assetOperations` field is replaced by the `assetOperationsMap` field. A number of ordered lists of smart contract actions over the assets available to them can be passed in this field.

You can use version **5** of this transaction starting with release 1.14.0 after *1140 feature activation*.

106. DisableContract Transaction

Disabling a *smart contract*.

Important: The transaction is irreversible, that is, the disabled contract cannot be used thereafter under any conditions.

The byte representation of this transaction after it is signed must not exceed **150 kilobytes**.

Transaction 106 can only be signed by a user with the **contract_developer** *role*.

Transaction data structures

Signing:

Field	Data type	Description
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
contractId	ByteStr	Smart contract ID
fee	Long	*WE Mainnet transaction fee*
type	Byte	Transaction number (**106**)
version	Byte	Transaction version

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**106**)
id	Byte	DisableContract transaction ID
sender	ByteStr	Address of the transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
proofs	List (ByteStr)	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
contractId	ByteStr	Smart contract ID
height	Byte	Height of transaction execution

JSON:

Version 1

Signing:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "password": "",
    "contractId": "HKftkVDTcQp6kxdqVYNdzB9d4rhND4YRKxwJV1thMXcr",
    "fee": 1000000,
```

(continues on next page)

```
    "type": 106,
    "version": 1,
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "proofs" : [
→"3FKPGT8YbLVun5cffZi1sHkgr9JZVxkeN7z2kUqDVLfhB5CwMtCAfyStRz1tpZuriKsR3MaBqNfReGx5sM2qey8i
→" ],
  "fee" : 1000000,
  "contractId" : "HKftkVDTcQp6kxdqVYNdzB9d4rhND4YRKxwJV1thMXcr",
  "id" : "5hXuHs5HVhZSfek153t76HfW6egmCLdZmi5AeFzYBFN",
  "type" : 106,
  "version" : 1,
  "timestamp" : 1625648619321,
  "height" : 1025992
}
```

Version 2

Signing:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "password": "",
    "contractId": "HKftkVDTcQp6kxdqVYNdzB9d4rhND4YRKxwJV1thMXcr",
    "fee": 1000000,
    "type": 106,
    "version": 2,
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : "7QpXWLGuaspzrMsESRaHTgksndq5mcvfbVrqBTuLbxuy",
  "proofs" : [
→"3FKPGT8YbLVun5cffZi1sHkgr9JZVxkeN7z2kUqDVLfhB5CwMtCAfyStRz1tpZuriKsR3MaBqNfReGx5sM2qey8i
→" ],
  "fee" : 1000000,
  "contractId" : "HKftkVDTcQp6kxdqVYNdzB9d4rhND4YRKxwJV1thMXcr",
  "id" : "5hXuHs5HVhZSfek153t76HfW6egmCLdZmi5AeFzYBFN",
  "type" : 106,
  "version" : 2,
  "timestamp" : 1625648619321,
  "height" : 1025992
}
```

Version 3

Signing:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "password": "",
    "contractId": "75PumcfCVxzV3v7RAPYQUwCtSpU21hxfaWFhureCRTLM",
    "fee": 1000000,
    "type": 106,
    "version": 3,
  "atomicBadge" : {
    "trustedSender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "atomicBadge" : {
    "trustedSender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  },
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
↪"22tK24qHhgbTDjtRmR86z3WeLLqLnqPvhUhQrz8ohfbCwQ9nrwmHESuT9aFuwABeBRJ7MfVob1FiJnqg3y2PHLSj
↪" ],
  "fee" : 1000000,
  "contractId" : "75PumcfCVxzV3v7RAPYQUwCtSpU21hxfaWFhureCRTLM",
  "id" : "7opPrLd6x1hATRr9R5oXnEbYjYQzo5cn4Qpkiz12Mw9b",
  "type" : 106,
  "version" : 3,
  "timestamp" : 1619186857911,
  "height" : 861644
}
```

107. UpdateContract Transaction

Updating a *smart contract* code. The byte representation of this transaction after it is signed must not exceed **150 kilobytes**.

Transaction 107 signing as well as smart contract updating can only be done by the user with the **contract_developer** *permission*.

Transaction data structures

Signing:

Field	Data type	Description
image	Ar-ray[B	The name of the Docker image of the Docker smart contract; the field is used up to and including version 5 of the transaction; starting with version 6, the `storedContract.image` field is used instead
image-Hash	Ar-ray[B	Docker image hash of the Docker smart contract; the field is used up to and including version 5 of the transaction; starting with version 6, the `storedContract.imageHash` field is used instead
sender	ByteS	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
fee	Long	*WE Mainnet transaction fee*
contrac-tId	ByteS	Smart contract ID
type	Byte	Transaction number (**107**)
version	Byte	Transaction version
feeAs-setId	Byte	Identifier of a token for fee payment – *optional field*
apiVer-sion	Byte	API version for the smart contract gRPC methods (see *gRPC services used by smart contracts*); the field is used up to and including transaction version 5; starting with version 6, the `storedContract.apiVersion` field is used instead
valida-tionPol-icy.type	String	Smart contract *validation policy type*
group-Partici-pants	Set[A	Addresses authorized to access *confidential data*
groupOwn-ers	Set[A	Addresses that can modify the groupParticipants and groupOwners lists
stored-Con-tract.bytec	Ar-ray[B	*WASM smart contract* bytecode ; the field is used starting with transaction version 6
stored-Con-tract.bytec	Ar-ray[B	*WASM smart contract* bytecode hash; the field is used starting with transaction version 6
stored-Con-tract.image	Ar-ray[B	Name of the smart contract Docker image; the field is used starting with version 6 of the transaction
stored-Con-tract.image	Ar-ray[B	Docker image hash of the Docker smart contract; the field is used starting with version 6 of the transaction
stored-Con-tract.apiVe	Byte	API version for the Docker smart contract gRPC methods (see *gRPC services used by smart contracts*); the field is used starting with transaction version 6; the field is not used for WASM smart contracts

Broadcasting:

Field	Data type	Description
sender-Pub-licKey	Pub-licK-ey-Ac-count	Transaction sender public key
tx	Ar-ray	Body of 105 transaction of an executed smart contract
fee	Long	*Mainnet transaction fee*
feeAs-setId	Byte	Identifier of a token for fee payment − *optional field*
apiVer-sion	Byte	API version for the smart contract gRPC methods (see *gRPC services used by smart contracts*); the field is used up to and including transaction version 5; starting with version 6, the `storedContract.apiVersion` field is used instead
type	Byte	Transaction number (**107**)
ver-sion	Byte	Transaction version
image	Ar-ray[B]	Docker Smart contract name (if downloaded from a pre-installed repository) or its full address (if the Docker smart contract repository is not specified in the node configuration file); the field is used up to and including version 5 of the transaction; starting from version 6, the `storedContract.image` field is used instead
image-Hash	Ar-ray[B]	Docker image hash of the Docker smart contract; the field is used up to and including version 5 of the transaction; starting with version 6, the `storedContract.imageHash` field is used instead
sender	ByteS	Address of the transaction sender
proofs	List(E	Array of transaction proofs (**in base58 format**)
con-tractId	ByteS	Smart contract ID
id	Byte	contract update transaction ID
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*
atom-icBadge	Boole	Flag that indicates whether the transaction can be included in an *atomic* transaction
group-Partic-ipants	Set[A	Addresses authorized to access *confidential data*
groupOv ers	Set[A	Addresses that can modify the groupParticipants and groupOwners lists
valida-tion-Pol-icy.type	String	Smart contract *validation policy type*
stored-Con-tract.byt	Ar-ray[B]	*WASM smart contract* bytecode ; the field is used starting with transaction version 6
stored-Con-tract.byt	Ar-ray[B]	*WASM smart contract* bytecode hash; the field is used starting with transaction version 6
stored-Con-tract.ima	Ar-ray[B]	Name of the smart contract Docker image; the field is used starting with version 6 of the transaction
stored-Con-tract.ima	Ar-ray[B]	Docker image hash of the Docker smart contract; the field is used starting with version 6 of the transaction
stored-Con-	Byte	API version for the Docker smart contract gRPC methods (see *gRPC services used by smart contracts*); the field is used starting with transaction version

JSON:

Version 2

Signing:

```
{
  "image" : "we-sc/grpc-contract-example:2.2-test-update",
  "imageHash" : "075ad1607f193cc6fdb5e85c201f9ca3907c622718d75706bbc2a94a330de5b5",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "fee" : 100000000,
  "contractId" : "BWzX4mRBEnHKgn3HB78My5DZzDAqnCLWCCNpCuRkZrJA",
  "type" : 107,
  "version" : 2
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "image" : "we-sc/grpc-contract-example:2.2-test-update",
  "imageHash" : "075ad1607f193cc6fdb5e85c201f9ca3907c622718d75706bbc2a94a330de5b5",
  "fee" : 100000000,
  "type" : 107,
  "version" : 2,
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : null,
  "proofs" : [
→"RetQwzuWZWxpSNMqwB7k7o6hSm6nhFCc49zKUpwZEedzBYcohj9NVEPwAbKLW9RzRKX168xApV7Nu2qV2jaHAMg
→" ],
  "contractId" : "BWzX4mRBEnHKgn3HB78My5DZzDAqnCLWCCNpCuRkZrJA",
  "id" : "6oopqcEf4AF943SCAqkBPrghyeQhmwn64TrhtCZbAn3v",
  "timestamp" : 1625649822957,
  "height" : 1026022
}
```

Version 3

Signing:

```
{
  "image" : "registry.wavesenterpriseservices.com/we-sc/grpc-contract-example:2.2-test-
→update",
  "imageHash" : "075ad1607f193cc6fdb5e85c201f9ca3907c622718d75706bbc2a94a330de5b5",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "fee" : 100000000,
  "contractId" : "HTqdjXUPTHZqGen2KKUkEenTELAqQ8irN58LA8EcP17q",
  "type" : 107,
  "version" : 3,
```

```
    "atomicBadge" : null
}
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "image" : "registry.wavesenterpriseservices.com/we-sc/grpc-contract-example:2.2-test-
↪update",
  "imageHash" : "075ad1607f193cc6fdb5e85c201f9ca3907c622718d75706bbc2a94a330de5b5",
  "fee" : 100000000,
  "type" : 107,
  "version" : 3,
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
↪"3ncWfFPqBAdgh65YceCCvF2RhUWWokQc9MsnHk27YLrYmPj9gWgrbRcousymJVA7ARFSz5UJcdW4Sa62FFhR5en3
↪" ],
  "contractId" : "HTqdjXUPTHZqGen2KKUkEenTELAqQ8irN58LA8EcP17q",
  "id" : "B7qjgCa9N6M6FwV63PbLwvtVpFo4bzB5gRZzGjwJpKJV",
  "timestamp" : 1619187337697,
  "height" : 861650
  "atomicBadge" : null,
}
```

Version 4

Signing:

```
{
  "image" : "we-sc/grpc_validatable_stateless:0.1",
  "imageHash" : "bd98a7d3e55506ff936d8ea15e170a24d27662edd1b47e4fd20801d10655af8d",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "fee" : 100000000,
  "contractId" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "type" : 107,
  "version" : 4,
  "atomicBadge" : null
  "validationPolicy" : {
    "type" : "any"
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "image" : "we-sc/grpc_validatable_stateless:0.1",
  "imageHash" : "bd98a7d3e55506ff936d8ea15e170a24d27662edd1b47e4fd20801d10655af8d",
  "fee" : 100000000,
  "type" : 107,
```

```
  "version" : 4,
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : null,
  "proofs" : [
→"fZr9LpqSWbPcUzArSZxFDEuygN62hR63j2Cz1GyTFxPNRrNvVwkDhTDcC8zwRp235gA1gSM8fvPps9mvPTWDQ4p
→" ],
  "contractId" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "id" : "HWZy7219Nx5oxj2QnK3ReEuZiqsjoULbmfdQz8YysFSz",
  "timestamp" : 1625732772746,
  "height" : 1028132,
  "atomicBadge" : null,
  "apiVersion" : "1.0",
  "validationPolicy" : {
    "type" : "any"
  },
}
```

Version 5

Signing:

```
{
  "image" : "we-sc/grpc_validatable_stateless:0.1",
  "imageHash" : "bd98a7d3e55506ff936d8ea15e170a24d27662edd1b47e4fd20801d10655af8d",
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "fee" : 100000000,
  "contractId" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "type" : 107,
  "version" : 5,
  "atomicBadge" : null
  "groupParticipants" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
→"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
  "groupOwners" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
→"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
  "validationPolicy" : {
    "type" : "any"
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "image" : "we-sc/grpc_validatable_stateless:0.1",
  "imageHash" : "bd98a7d3e55506ff936d8ea15e170a24d27662edd1b47e4fd20801d10655af8d",
  "fee" : 100000000,
  "type" : 107,
  "version" : 5,
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : null,
  "proofs" : [
```

```
↪"fZr9LpqSWbPcUzArSZxFDEuygN62hR63j2Cz1GyTFxPNRrNvVwkDhTDcC8zwRp235gA1gSM8fvPps9mvPTWDQ4p
↪" ],
  "contractId" : "HSLdKYqLq4LcZpq9LPki8Yv4ZRkFapVyHEYw1vZW2MoG",
  "id" : "HWZy7219Nx5oxj2QnK3ReEuZiqsjoULbmfdQz8YysFSz",
  "timestamp" : 1625732772746,
  "height" : 1028132,
  "atomicBadge" : null,
  "apiVersion" : "1.0",
  "validationPolicy" : {
    "type" : "any"
  },
  "groupParticipants" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
↪"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
  "groupOwners" : [ "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
↪"3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY"],
}
```

Version 6

Signing:

```
{
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "",
  "fee": 100000000,
  "contractId": "HcJTMMpcLaMXme2hLzC7JqZ5Dn8ecfNeKkwHNYCVfdFZ",
  "type": 107,
  "version": 6,
  "feeAssetId": null,
  "groupParticipants": [],
  "groupOwners": [],
  "validationPolicy": {"type": "any"},
  "storedContract":
    {
      "bytecode": "AGFzbQEAAAABGgRgA39/fgF/YAR/f39/An9+YAABf2ABfgF/
↪Aj4DA2VudgZtZW1vcnkCAQIQBGVudjAPc2V0X3N0b3JhZ2VfaW50AAAEZW52MA9nZXRfc3RvcmFnZV9pbnQAAQMEAwICAwQA38BQI
↪AEEgCwdFBQxfY29uc3RydWN0b3IAAAgtpbmNyZW1lbnRfMQADCWluY3JlbWVudAQEC19fZGF0YV9lbmQDAQtfX2hlYXBfYmFzZQMCC
↪",
      "bytecodeHash": "2d52876455a4d8cd599c16fa0e0ad6f028b76b8494e40f6c4651598f29066013"
    }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "fee" : 100000000,
  "type" : 107,
  "version" : 6,
  "sender" : "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId" : None,
```

```
  'proofs': [
↪'5x2giusM21s5jgSzQDkZABpnRGNrPyDxpTFm2RS1znP9DSEHEuccyPMUmhpPq78U1bKksbemYeRo8mApVWuGybpb
↪'],
  "contractId" : "HcJTMMpcLaMXme2hLzC7JqZ5Dn8ecfNeKkwHNYCVfdFZ",
  "id" : "8PrdyjiM443kYSrBpeU6BVBKkCbnTKcMmAsWHieMvW8n",
  "timestamp" : 1708435405364
  "atomicBadge" : None,
  "groupOwners" : [ ],
  "groupParticipants" : [ ],
  "validationPolicy" : {
    "type" : "any"
  },
  "storedContract" : {
    "bytecode" : "AGFzbQEAAAABLwZgA39/fwF/YAN/f34Bf2AEf39/fwF/YAJ/fwN/f39gAAN/
↪f39gBn9+f39/fwF/
↪ApcBBwNlbnYGbWVtb3J5AgECEARlbnYwEHNldF9zdG9yYWdlX2Jvb2wAAARlbnYwD3NldF9zdG9yYWdlX2ludAABBGVudjASc2V0X
↪AEHBAAt/
↪AEHQAAsHPAQMX2NvbnN0cnVjdG9yAAYOdXBkYXRlX3N0b3JhZ2UABwpfX2RhdGFfZW5kAwELX19oZWFwX2Jhc2UDAgqqAgKrAQACQ
↪",
    "bytecodeHash" : "1afd57a7be47a0f762821bdbe2099c9590efd4471a716e5f0da60a06bc317ec6"
  },
}
```

Version 4

The 4th version of this transaction configures validation of the execution results of the updated smart contract using the `validationPolicy.type` field (see section *Validation of smart contracts*).

Fields to configure smart contract validation:

Variants of validation policies:

- `any` - the general validation policy is kept in the network: to mine the updated smart contract, the miner signs the corresponding *105* transaction. Also, this parameter is set if there are no registered validators in the network.

- `majority` - a transaction is considered valid if it is confirmed by the majority of validators: $2/3$ of the total number of registered addresses with the **contract_validator** permission.

- `majorityWithOneOf(List[Address])` - the transaction is considered valid if the majority of validators is collected, among which there is at least one of the addresses included in the parameter list. The addresses included in the list must have a valid **contract_validator** permission.

> **Warning:** In case of using the `majorityWithOneOf(List[Address])` validation policy, fill the address list, passing an empty list is not allowed.

Version 5

Version **5** of this transaction implements the fields required to handle *confidential smart contracts*.

Fields for working with confidential smart contracts:

- the `groupParticipants` field specifies the group (policy) member nodes who have access to confidential smart-contract data; the maximum group size is 1024 members;

- the `groupOwners` field specifies the nodes that can modify the `groupParticipants` and `groupOwners` lists using the *UpdateContract* transaction; no more than 1024 nodes can be specified in this field.

Only the sender of the UpdateContract transaction, whose address is specified in the `groupOwners` field, can change the values of the `groupParticipants` and `groupOwners` fields.

The contract creator can modify the contract itself (the `image`, `imageHash` and `apiVersion` fields), but he must have his address listed in the `groupOwners` field to be able to modify the `groupParticipants` and `groupOwners` fields.

The group owner (the address listed in the `groupOwners` field) cannot modify the contract, that is the `image`, `imageHash` and `apiVersion` fields.

The `groupParticipants` and `groupOwners` fields must be empty if the `isConfidential` parameter was set to `false` when the contract *was created*.

Important: You cannot update a contract if its `isConfidential` parameter was set to `true` when it was *created*, and the `groupParticipants` field contains less than 3 participants with the `contract-validator` *role*.

You can use the **5** th version of this transaction starting with 1.13 release after the **1130** *feature activation*.

Version 6

Support for *WASM smart contracts* is implemented in version **6** of this transaction. For this purpose `image`, `imageHash`, `apiVersion` fields are removed and `storedContract` field is implemented:

- for Docker smart contracts:

```
"storedContract" : {
    "image" : String
    "imageHash": String
    "apiVersion" : Byte,
}
```

- For WASM smart contracts:

```
"storedContract" : {
    "bytecode" : <bytecode contracts>
    "bytecodeHash" : "Sha256 от <bytecode contracts>"
}
```

Important: A Docker smart contract cannot be updated to a WASM smart contract.

You can use the version **6** of this transaction starting with release 1.14.0 after *1140 feature activation*.

Important: In release 1.14.0, WASM smart contracts do not support *atomic transactions* and *confidential smart contracts*. Therefore, for a WASM smart contract, the `atomicBadge` field must be absent from the JSON for signing, or have the value of `null`; also the following fields must have the following values:

```
...
"groupParticipants": [],
"groupOwners": [],
...
```

110. GenesisRegisterNode Transaction

Registration of a node in a network genesis block while starting the blockchain.

This transaction does not require signing.

Data structure of a query for transaction broadcast

Field	Data type	Description
type	Byte	Transaction number (**110**)
id	Byte	GenesisRegisterNode transaction ID
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) − *optional field*
signature	ByteStr	Transaction signature (**in base58 format**)
version	Byte	Transaction version
targetPub-Key	Byte	Public key of a node to be registered
height	Byte	Height of transaction execution

111. RegisterNode Transaction

Registration of a new node in the blockchain or its deletion.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**111**)
opType	String	Type of operation: `add` - add a node; `remove` - remove a node
sender	ByteStr	Address of the transaction sender
password	String	Keypair password in the node keystore – *optional field*
targetPubKey	Byte	Public key of a node to be removed
NodeName	Byte	Name of a node
fee	Long	*WE Mainnet transaction fee*

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**111**)
id	Byte	RegisterNode transaction ID
sender	ByteStr	Address of the transaction sender
senderPub- licKey	PublicKeyAc- count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
targetPub- Key	Byte	Public key of a node to be removed
NodeName	Byte	Name of a node
opType	String	Type of operation: `add` - add a node; `remove` - remove a node
height	Byte	Height of transaction execution
password	String	Keypair password in the node keystore – *optional field*

JSON:

Version 1

Signing:

```
{
  "type": 111,
  "opType": "add",
  "sender":"3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
  "password": "",
  "targetPubKey": "6caEKC1UBgRvgAe9A7L5PWcrawrnEZGxtsXynGESwSj7",
  "nodeName": "GATEs node",
  "fee": 1100000,
}
```

Broadcasting:

```
{
  "senderPublicKey" : "FWz5gZ2w2ZxXbKEiwhgEcZKT4we1Wneh9XqmCeGPsA4r",
  "nodeName" : "GATEs node",
  "fee" : 1100000,
  "opType" : "add",
  "type" : 111,
  "version" : 1,
  "target" : "3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY",
  "sender" : "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
  "proofs" : [
↪"FHEexr8MqMCkdqaVRrfxv7dnQFwo1VQxQFb4rW2VKh1NkuAhjhtzftKybBQCVbpKcCD1ZTRhwATpwERF9re2Viz
↪" ],
  "id" : "6WnDGkBDeSjg5y6QqVdy3BFHUy5nnr4QsxZCeNXZtZoq",
  "targetPubKey" : "6caEKC1UBgRvgAe9A7L5PWcrawrnEZGxtsXynGESwSj7",
  "timestamp" : 1619078302988,
  "height" : 858895
}
```

Version 2

Signing:

```
{
  "type": 111,
  "version" : 2,
  "opType": "add",
  "sender":"3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
  "password": "",
  "targetPubKey": "6caEKC1UBgRvgAe9A7L5PWcrawrnEZGxtsXynGESwSj7",
  "nodeName": "GATEs node",
  "fee": 1100000,
  "atomicBadge":{
    "trustedSender":"3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
  }
}
```

Broadcasting:

```
{
  "senderPublicKey" : "FWz5gZ2w2ZxXbKEiwhgEcZKT4we1Wneh9XqmCeGPsA4r",
  "nodeName" : "GATEs node",
  "fee" : 1100000,
  "opType" : "add",
  "type" : 111,
  "version" : 2,
  "target" : "3NtieMGjVAH1nDsvnSEJ37BSW3hpJV2CneY",
  "sender" : "3NgSJRdMYu4ZbNpSbyRNZLJDX926W7e1EKQ",
  "proofs" : [
↪"FHEexr8MqMCkdqaVRrfxv7dnQFwo1VQxQFb4rW2VKh1NkuAhjhtzftKybBQCVbpKcCD1ZTRhwATpwERF9re2Viz
↪" ],
  "id" : "6WnDGkBDeSjg5y6QqVdy3BFHUy5nnr4QsxZCeNXZtZoq",
  "targetPubKey" : "6caEKC1UBgRvgAe9A7L5PWcrawrnEZGxtsXynGESwSj7",
```

```
    "timestamp" : 1619078302988,
    "height" : 858895
}
```

112. CreatePolicy Transaction

Creation of a *confidential data group* consisting of the addresses stated in the transaction.

Transaction data structures

Signing:

Field	Data type	Description
sender	ByteStr	Address of the transaction sender
policy-Name	String	Name of an access group to be created
pass-word	String	Keypair password in the node keystore – *optional field*
recipi-ents	Ar-ray[Bytes]	Array of addresses of a group participants separated by commas
fee	Long	*WE Mainnet transaction fee*
de-scrip-tion	Ar-ray[byte]	An arbitrary description of the transaction (**in base58 format**)
owners	Ar-ray[Bytes]	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
type	Byte	Transaction number (**112**)
version	Byte	Transaction version

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**112**)
id	Byte	CreatePolicy transaction ID
sender	ByteStr	Address of the transaction sender
sender-Pub-licKey	PublicK-eyAccount	Transaction sender public key
policy-Name	String	Name of an access group to be created
recipi-ents	Ar-ray[Bytes]	Array of addresses of a group participants separated by commas
owners	Ar-ray[Bytes]	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
fee	Long	*WE Mainnet transaction fee*
feeAs-setId	Byte	Identifier of a token for fee payment – *optional field*
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
proofs	List(ByteSt	Array of transaction proofs (**in base58 format**)
height	Byte	Height of transaction execution
descrip-tion	Ar-ray[byte]	An arbitrary description of the transaction (**in base58 format**)
version	Byte	Transaction version

JSON:

Version 1

Signing:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "policyName": "Policy# 7777",
    "password":"sfgKYBFCF@#$fsdf()%",
    "recipients": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
      "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
      "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
    ],
    "fee": 15000000,
    "description": "Buy bitcoin by 1c",
    "owners": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
    ],
    "type": 112,
    "version": 1,
}
```

Broadcasting:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "policyName": "Policy# 7777",
    "password":"sfgKYBFCF@#$fsdf()%",
    "recipients": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
      "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
      "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
    ],
    "fee": 15000000,
    "description": "Buy bitcoin by 1c",
    "owners": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
    ],
    "type": 112,
    "version": 1,
}
```

Version 2

Signing:

```
{
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "policyName": "Policy# 7777",
  "password":"sfgKYBFCF@#$fsdf()%",
  "recipients": [
    "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
    "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
    "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
    "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  ],
  "fee": 15000000,
  "description": "Buy bitcoin by 1c",
  "owners": [
    "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
    "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
  ],
  "type": 112,
  "version": 2,
}
```

Broadcasting:

```
{
    "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "policyName": "Policy# 7777",
    "password":"sfgKYBFCF@#$fsdf()%",
    "recipients": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
      "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
      "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
    ],
    "fee": 15000000,
    "feeAssetId" : null,
    "description": "Buy bitcoin by 1c",
    "owners": [
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
    ],
    "type": 112,
    "version": 2,
}
```

Version 3

Signing:

```
{
  "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "policyName": "Policy_v3_for_demo_txs",
  "password":"sfgKYBFCF@#$fsdf()%",
  "recipients" : [
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
      "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
      "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
      "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
      "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn"
  ],
  "fee": 100000000,
  "description": "",
  "owners" : [
      "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
      "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
      "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  ],
  "type": 112,
  "version": 3
}
```

Broadcasting:

```
{
    "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
    "policyName" : "Policy_v3_for_demo_txs",
    "fee" : 100000000,
    "description" : "",
    "owners" : [
        "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
        "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
        "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
        "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
        "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn"
    ],
    "type" : 112,
    "version" : 3,
    "atomicBadge" : null,
    "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
    "feeAssetId" : null,
    "proofs" : [
→"4NccZyPCgchDjeMdMmFKu7kxyU8AFF4e9cWaPFTQVQyYU1ZCCu3QmtmkfJkrDpDwGs4eJhYUVh5TnwYvjZYKPhLp
→" ],
    "recipients" : [
        "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
        "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
        "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
        "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
        "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn"
    ],
    "id" : "5aYtmTr1AYYG8BrYvTTSqKzfJZxfgorx1BLGVwSAhwrz",
    "timestamp" : 1619186864092,
    "height" : 861637
}
```

113. UpdatePolicy Transaction

Updating a *confidential data group*.

Transaction data structures

Signing:

Field	Data type	Description
poli-cyId	String	Confidential data group identifier
pass-word	String	Keypair password in the node keystore – *optional field*
sender	ByteStr	Address of the transaction sender
recip-ients	Ar-ray[Bytes]	Array of addresses of a group participants separated by commas
fee	Long	*WE Mainnet transaction fee*
op-Type	String	Type of operation: `add` - add participants; `remove` - remove participants
own-ers	Ar-ray[Bytes]	Array of addresses of group administrators separated by commas: admin-istrators are entitled to change an access group
type	Byte	Transaction number (**113**)
ver-sion	Byte	Transaction version

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**113**)
id	Byte	UpdatePolicy transaction ID
sender	ByteStr	Address of the transaction sender
sender-Pub-licKey	PublicK-eyAccount	Transaction sender public key
policyId	String	Confidential data group identifier
recipi-ents	Ar-ray[Bytes]	Array of addresses for adding or removing of group participants sep-arated by commas
owners	Ar-ray[Bytes]	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
fee	Long	*WE Mainnet transaction fee*
feeAs-setId	Byte	Identifier of a token for fee payment – *optional field*
times-tamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
height	Byte	Height of transaction execution
opType	String	Type of operation: `add` - add a permission; `remove` - remove a per-mission
descrip-tion	Ar-ray[byte]	An arbitrary description of the transaction (**in base58 format**)
version	Byte	Transaction version

JSON:

Version 1

Signing:

```
{
  "policyId": "UkvoboGXiwWpASr1GLG9M1MUbhrEMo4NBS7kquxVMw5",
  "password":"sfgKYBFCF@#$fsdf()*%",
  "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
   "fee": 50000000,
   "opType": "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type": 113,
  "version": 1
 }
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "fee" : 50000000,
  "opType" : "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type" : 113,
  "version" : 1,
  "policyId" : "UkvoboGXiwWpASr1GLG9M1MUbhrEMo4NBS7kquxVMw5",
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "proofs" : [
→"2CKd57kU3wbxdrHxMPNbrWHptnf5ZcydYjqxMPk46miMcUUAxgFGXcy621cjYFXC3vjpKNNrB2QcgtKe1Yx9TcLY
→" ],
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "id" : "6o4azRwzmMg9SqWUq6rv6GAe5gzTYJvE5ek1v9VM3Mb",
  "timestamp" : 1619004195630,
  "height" : 856970
}
```

Version 2

Signing:

```
{
  "policyId": "UkvoboGXiwWpASr1GLG9M1MUbhrEMo4NBS7kquxVMw5",
  "password":"sfgKYBFCF@#$fsdf()*%",
  "sender": "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
   "fee": 50000000,
   "opType": "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type": 113,
  "version": 2
 }
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "fee" : 50000000,
  "opType" : "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type" : 113,
  "version" : 2,
  "policyId" : "UkvoboGXiwWpASr1GLG9M1MUbhrEMo4NBS7kquxVMw5",
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
→"2CKd57kU3wbxdrHxMPNbrWHptnf5ZcydYjqxMPk46miMcUUAxgFGXcy621cjYFXC3vjpKNNrB2QcgtKe1Yx9TcLY
→" ],
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "id" : "6o4azRwzmMg9SqWUq6rv6GAe5gzTYJvE5ek1v9VM3Mb",
  "timestamp" : 1619004195630,
  "height" : 856970
}
```

Version 3

Signing:

```
{
  "policyId": "5aYtmTr1AYYG8BrYvTTSqKzfJZxfgorx1BLGVwSAhwrz",
  "password":"sfgKYBFCF@#$fsdf()*%",
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
   "fee": 50000000,
   "opType": "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type": 113,
  "version": 3
 }
```

Broadcasting:

```
{
  "senderPublicKey" : "7GiFGcGaEN87ycK8v71Un6b7RUoeKBU4UvUHPYbeHaki",
  "fee" : 50000000,
  "opType" : "remove",
  "owners" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
  "type" : 113,
  "version" : 3,
  "atomicBadge" : null,
  "policyId" : "5aYtmTr1AYYG8BrYvTTSqKzfJZxfgorx1BLGVwSAhwrz",
  "sender" : "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx",
  "feeAssetId" : null,
  "proofs" : [
→"2QMGoz6rycNsDLhN3mDce2mqGRQQ8r26vDDw551pnYcAecpFBDA8j38FVqDjLTGuFHs6ScX32fsGcaemmptpCFHk
→" ],
  "recipients" : [ "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF" ],
```

```
  "id" : "Hwqf8LgpQfEcUYX9nMNG8uU2Cw1xSuGFqYxmuACpvU1L",
  "timestamp" : 1619187450552,
  "height" : 861653
}
```

114. PolicyDataHash Transaction

Sending *confidential data* hash into the network. This transaction is created automatically while sending confidential data into the network with the use of the `POST /privacy/sendData` *REST API method*.

This transaction does not require signing.

Data structure of a query for transaction broadcast

Field	Data type	Description
type	Byte	Transaction number (**114**)
id	Byte	Transaction identifier
sender	ByteStr	Address of the transaction sender
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
policyId	String	Name of an access group to be created
dataHash	String	Confidential data hash to be sent
fee	Long	*Mainnet transaction fee*
feeAssetId	Byte	Identifier of a token for fee payment – *optional field*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
height	Byte	Height of transaction execution
version	Byte	Transaction version

JSON:

Version 1

Broadcasting:

```
{
  "senderPublicKey":
→"4L4XEpNpesX9r6rVJ8hW1TrMiNCZ6SMvRuWPKB7T47wKfnp4D84XBUv7xsa36CGwoyK3fzfojivwonHNrsX2fLBL
→",
  "dataHash": "8GPtHQeLxhtt8HianM9c8otS2EeAHNVZCfpCRUmYbSFi",
  "fee": 0,
  "type": 114,
  "version": 1,
  "policyId": "75rGACZxkTE5x5seNjEzJUEe73fTzkQiBrr28hCjMMVq",
  "sender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX",
  "proofs": [
```

```
→"5uW8SeX4k3nb8esuMeRY27MyZ6dnWijwbGhSo53zSKY1FjjofWiE4mPfNwUhYKgqyAtHtUvwsdTMyL87LGNqwp5o
→"
  ],
  "id": "52zCNUhfne9HYfHr7sEYBGFHqnzHKBdkGbGnsYfCYXug",
  "timestamp": 1632916536685,
  "height": 1585580
}
```

Version 2

Broadcasting:

```
{
  "senderPublicKey":
→"4L4XEpNpesX9r6rVJ8hW1TrMiNCZ6SMvRuWPKB7T47wKfnp4D84XBUv7xsa36CGwoyK3fzfojivwonHNrsX2fLBL
→",
  "dataHash": "8GPtHQeLxhtt8HianM9c8otS2EeAHNVZCfpCRUmYbSFi",
  "fee": 0,
  "type": 114,
  "version": 2,
  "policyId": "75rGACZxkTE5x5seNjEzJUEe73fTzkQiBrr28hCjMMVq",
  "sender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX",
  "feeAssetId": null,
  "proofs": [


→"5uW8SeX4k3nb8esuMeRY27MyZ6dnWijwbGhSo53zSKY1FjjofWiE4mPfNwUhYKgqyAtHtUvwsdTMyL87LGNqwp5o
→"
  ],
  "id": "52zCNUhfne9HYfHr7sEYBGFHqnzHKBdkGbGnsYfCYXug",
  "timestamp": 1632916536685,
  "height": 1585580
}
```

Version 3

Broadcasting:

```
{
  "senderPublicKey":
→"4L4XEpNpesX9r6rVJ8hW1TrMiNCZ6SMvRuWPKB7T47wKfnp4D84XBUv7xsa36CGwoyK3fzfojivwonHNrsX2fLBL
→",
  "dataHash": "8GPtHQeLxhtt8HianM9c8otS2EeAHNVZCfpCRUmYbSFi",
  "fee": 0,
  "type": 114,
  "version": 3,
  "atomicBadge": {
    "trustedSender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX"
```

```
  },
  "policyId": "75rGACZxkTE5x5seNjEzJUEe73fTzkQiBrr28hCjMMVq",
  "sender": "3M3ybNZvLG7o7rnM4F7ViRPnDTfVggdfmRX",
  "feeAssetId": null,
  "proofs": [

↪"5uW8SeX4k3nb8esuMeRY27MyZ6dnWijwbGhSo53zSKY1FjjofWiE4mPfNwUhYKgqyAtHtUvwsdTMyL87LGNqwp5p
↪"
  ],
  "id": "52zCNUhfne9HYfHr7sEYBGFHqnzHKBdkGbGnsYfCYXug",
  "timestamp": 1632916536685,
  "height": 1585580
}
```

120. Atomic Transaction

Atomic transaction sets other transactions in a container for their atomic execution. This transaction can be executed only in full (no transactions have been declined), in other cases it will not be executed.

You can include two or more transactions into an atomic transaction. Transaction types and versions that can be included in an atomic transaction are listed in the *Atomic transactions* section; this article also provides more detailed information about processing this type of transactions.

An atomic transaction itself does not require a fee: its total fee is summed up from fee of transactions included into it.

Transaction data structures

Signing:

Field	Data type	Description
type	Byte	Transaction number (**120**)
sender	ByteStr	Address of the transaction sender
transactions	Array	Full bodies of transactions to be included
password	String	Keypair password in the node keystore – *optional field*
fee	Long	*WE Mainnet transaction fee*
version	Byte	Transaction version

Broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**114**)
id	Byte	Transaction identifier
sender	ByteStr	Address of the transaction sender
senderPub-licKey	PublicKeyAc-count	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The **Unix Timestamp** of the transaction (in milliseconds) – *optional field*
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
height	Byte	Height of transaction execution
transactions	Array	Full bodies of transactions to be included
miner	String	Block miner public key; filled during a mining round
password	String	Keypair password in the node keystore – *optional field*
version	Byte	Transaction version

JSON:

Version 1

Signing:

```
{
    "sender": sender_0,
    "transactions": [
        signed_transfer_tx,
        signed_transfer_tx2
    ],
    "type": 120,
    "version": 1,
    "password":"lskjbJJk$%^#298",
    "fee": 0,
}
```

Broadcasting:

```
{
  "sender": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
  "transactions": [
        signed_transfer_tx,
        signed_transfer_tx2
  ],
  "type": 120,
  "version": 1,
}
```

See also

Description of transactions

Mainnet fees

Actual versions of transactions

When sending transactions to Waves Enterprise Mainnet or a private network, it is recommended to use the current versions of the transactions. The version of the transaction is specified in the `version` field when signing and sending.

Transaction number	Transaction name	Actual version
1	*Genesis Transaction*	No version
3	*Issue Transaction*	3
4	*Transfer Transaction*	3
5	*Reissue Transaction*	3
6	*Burn Transaction*	3
8	*Lease Transaction*	3
9	*Lease Cancel Transaction*	3
10	*Create Alias Transaction*	4
11	*Mass Transfer Transaction*	3
12	*Data Transaction*	3
13	*Set Script Transaction*	1
14	*Sponsorship Transaction*	2
15	*Set Asset Script Transaction*	1
101	*Genesis Permission Transaction*	No version
102	*Permission Transaction*	2
103	*Create Contract Transaction*	7
104	*Call Contract Transaction*	7
105	*Executed Contract Transaction*	5
106	*Disable Contract Transaction*	3
107	*Update Contract Transaction*	6
110	*Genesis Resgister Node Transaction*	1
111	*Register Node Transaction*	2
112	*Create Policy Transaction*	3
113	*Update Policy Transaction*	3
114	*Policy Data Hash Transaction*	3
120	*Atomic Transaction*	1

See also

Transactions of the blockchain platform

Description of transactions

Mainnet fees

1.27 Atomic transactions

The Waves Enterprise platform supports atomic operations. Atomic operations consist of multiple actions: if any action cannot be finalized, other actions also will not be performed. Atomic operations are implemented through the *120 AtomicTransaction*, which is a container that holds two or more signed transactions.

Two and more transactions of the following types and versions can be included into an atomic transaction:

- *4. Transfer Transaction*, ver. **3**
- *102. Permission Transaction*, ver. **2**
- *103. CreateContract Transaction*, ver. **5**
- *104. CallContract Transaction*, ver. **6**
- *105. ExecutedContract Transaction*, ver. **3**
- *106. DisableContract Transaction*, ver. **3**
- *107. UpdateContract Transaction*, ver. **4**
- *112. CreatePolicy Transaction*, ver. **3**
- *113. UpdatePolicy Transaction*, ver. **3**
- *114. PolicyDataHash Transaction*, ver. **3**

Once the *blockchain feature1122* is activated, you can include transaction of the following types in an atomic transaction:

- *3. Issue Transaction*, ver.**3**
- *5. Reissue Transaction*, ver. **3**
- *6. Burn Transaction*, ver. **3**
- *8. Lease Transaction*, ver. **3**
- *9. LeaseCancel Transaction*, ver. **3**
- *10. CreateAlias Transaction*, ver. **4**
- *11. MassTransfer Transaction*, ver. **3**
- *12. Data Transaction*, ver. **3**
- *14. Sponsorship Transaction*, ver. **2**
- *111. RegisterNode Transaction*, ver. **2**

The key distinction of transaction versions that are supported by atomic transactions, is the presence of the `atomicBadge` tag field.

This field contains a *trustedSender* value: the trusted address of the transaction sender to include into the *120* transaction container.

If no sender address is specified, the address from which the *120* transaction is sent to the blockchain becomes the sender.

1.27.1 Processing of atomic transactions

Atomic transactions have two signatures. The first signature belongs to its sender and is used for broadcasting. The second signature is generated by a miner and is used to include the transaction into the blockchain. When an atomic transaction is added to the UTX pool, the node checks its own signature, as well as signatures of all the transactions included into the atomic container.

Validation of included transactions is performed as follows:

- There should be more than one included transactions.

- All the transactions should have different identifiers.

- An atomic transaction should contain only supported transaction types.

It is not allowed to include one atomic transaction into another.

There should be no executed transactions inside an atomic transaction that is sent into the UTX pool; the `miner` field should be empty. This field is filled when an atomic transaction is transferred to a block.

There should be no executable transactions in an atomic transaction which is in the UTX pool.

After execution of an atomic transaction, its 'copy' is included into a block. This 'copy' is generated as follows:

- The `miner` field is not engaged for transaction signing and is filled with a miner public key.

- A block miner generates a `proofs` array, the source of which are identifiers of transactions included into an atomic transaction. When included into a block, an atomic transaction has 2 signatures: a signature of a source transaction and a miner's signature.

- If executable transactions are included into an atomic transaction, they are substituted with executed transactions. While validating an atomic transaction in a block, both signatures are checked.

1.27.2 Creating an atomic transaction

To create an atomic transaction, access to the node *REST API* is required.

1. A user picks supported transactions that should be executed as an atomic operation.

2. Then correctly fills in the fields of all transactions and signs them.

3. Next, the user fills the atomic transaction `transactions` field with the data of signed but not broadcasted transactions.

4. After entering all the transactions data, the user signs and broadcasts the completed atomic transaction to the blockchain.

Data structures for signing and broadcasting an atomic transaction are listed in the *list of transactions*.

Attention: If you create an atomic transaction including a *114* transaction, set its `broadcast` field value to `false` while signing.

See also

Description of transactions

Mainnet fees

1.28 Consensus algorithms

Blockchain is a distributed system which does not have a unified process regulator. Decentralization prevents corruption inside the system but complicates decision making and organization of an overall workflow.

These problems are resolved by the **consensus** — an algorithm which coordinates work of the blockchain participants by means of a certain voting method. Voting in the blockchain is always performed in support of the majority: minority interests are not taken into account, and decisions that have been made become mandatory for all network participants. However, voting guarantees achievement of a consensus that will be profitable for the entire network.

The Waves Enterprise blockchain platform supports three consensus algorithms:

1.28.1 LPoS consensus algorithm

The PoS (**Proof of Stake**) consensus algorithm is based on proofing of an address token share, the LPoS (**Leased Proof of Stake**) also includes an opportunity to lease tokens. With these algorithms, generation of a block does not need energy consuming calculations, a miner should create a digital signature of a block.

Proof of Stake

In the Proof of Stake consensus algorithm, the right to generate a block is determined in a pseudo-random way: a next miner is identified on the basis of previous miner data and balances of all network users. This is possible due to a deterministic computation of a block's generating signature, which can be obtained by SHA256 hashing of the current block's generating signature and the account's public key. The first 8 bytes of the resulting hash are converted to a hit digit X_n of an account, this digit will be a pointer to the following miner. The time of block generation for an i account is calculated as:

$$T_i = T_{min} + C_1 \log(1 - C_2 \frac{\log \frac{X_n}{X_{max}}}{b_i A_n})$$

where:

- b_i – a balance stake of a participant in comparison with the network total balance;

- A_n – baseTarget, the adaptive ratio regulating the average time of issue of the block;

- X_n – a pointer to the next miner;

- T_{min} – a constant value defining a minimum time interval between blocks (**5 seconds**);

- C_1 – a constant value correcting the form of interval allocation between blocks (**70**);

- C_2 – a constant value that is equal to the BaseTarget (**5E17**) by default and intended for its correction.

Based on this formula, the probability of selecting the participant to be rewarded depends on the participant's stake of assets in the system. The bigger the stake, the higher the chance of reward. The minimum number of tokens needed for mining is **50000 WEST**.

BaseTarget is a parameter that maintains the block generation time within a given range. BaseTarget is calculated as follows:

$$(S > R_{max} \rightarrow T_b = T_p + max(1, \frac{T_p}{100})) \wedge (S < R_{min} \wedge \wedge T_b > 1 \rightarrow T_b = T_p - max(1, \frac{T_p}{100}))$$

where

- R_{max} — maximal decrease of complexity that is engaged when block generation time exceeds 40 seconds (**90**);

- R_{min} — minimal increase of complexity that is engaged when block generation time is less than 40 seconds (**30**);

- S — the average time of generation of at least three last blocks;

- T_p — the previous baseTarget value;

- T_b — a calculated baseTarget value.

A detailed description of the technical characteristics and developments of the classical PoS algorithm for the Waves Enterprise blockchain platform is stated in this article.

Advantages over the Proof of Work (PoW)

The absence of complex calculations allows PoS networks to lower the hardware requirements for the system participants, which reduces the cost of deploying private networks. No additional emission is required, which in PoW systems is used to reward miners for finding a new block. In PoS systems, a miner receives a reward in the form of fees for transactions which appeared in the miner's block.

Leased Proof of Stake

A user who has an insufficient stake for effective mining may transfer his balance for lease to another participant and receive a portion of the income from mining. Leasing is a completely safe operation, as tokens do not leave the user's wallet, but are delegated to another miner, which gives the miner a greater opportunity to earn mining rewards.

See also

General platform configuration: consensus algorithm

Consensus algorithms

PoA consensus algorithm

CFT consensus algorithm

1.28.2 PoA consensus algorithm

In a private blockchain, tokens are not always needed. For example, a blockchain can be used to store hashes of documents exchanged by organizations. In this case, in the absence of tokens and fees from transactions, a solution based on the PoS consensus algorithm is redundant. The Waves Enterprise blockchain platform offers the option of a Proof of Authority (PoA) consensus algorithm. Mining permission is issued centrally in the PoA algorithm, which simplifies the decision-making compared to the PoS algorithm. The PoA model is based on a limited number of block validators, which makes it scalable. Blocks and transactions are verified by pre-approved participants who act as moderators of the system.

Algorithm description

An algorithm determining the miner of the current block is formed on the basis of the parameters stated below. The parameters of the consensus are specified in the `consensus` block of the node configuration file.

- t - the duration of a round in seconds (the parameter of the node configuration file: `round-duration`).

- t_s - the duration of a synchronization period, calculated as t*0.1, but not more than 30 seconds (the parameter of the node configuration file: `sync-duration`).

- N_{ban} - a number of missed consecutive rounds for issuing the ban for the miner (the parameter of the node configuration file: `warnings-for-ban`);

- P_{ban} - a maximum percentage of banned miners, from 0 to 100 (the parameter of the node configuration file: `max-bans-percentage`);

- t_{ban} - the duration of the miner ban in blocks (the parameter of the node configuration file: `ban-duration-blocks`).

- T_0 - unix timestamp of genesis block creation.

- T_H - unix timestamp of creation of the H block — the NG key block.

- r - round number, calculated as $(T_{Current} - T_0)$ div $(t + t_s)$.

- A_r - the leader of the round r, who is entitled to create key blocks and microblocks for NG in the round r.

- H − blockchain height, at that the NG key block and microblocks are created. The A_r round leader is entitled to generate the block.

- M_H - the miner who creates a block at the H height.

- Q_H - the queue of active miners at the H height.

The Q_H queue consists of addresses that have the miner permission. At the same time, the miner permission should not be removed from the addresses before the H height or expiry before the T_H time.

The queue is sorted by the time stamp of the mining rights transaction. The node which was granted the rights earlier will be higher in the queue. To keep the network consistent, this queue will be the same on each node.

A new block is generated during each r round. A duration of a round is t seconds. Each round is followed with t_s seconds for network data synchronization. During the synchronization, microblocks and key blocks are not generated. Each round has a leader A_r, who is entitled to generate a block in this round. A leader can be defined at each network node with the same result.

The round leader is defined as follows:

1. The miner M_{H-1} is defined, who has created a previous block at the H-1 height.

2. The queue of active miners Q_H is calculated.

3. Inactive miners are excluded from the queue (see *Exclusion of inactive miners*).

4. If the miner of the H-1 (M_{H-1}) block is in the Q_H queue, a next miner in the queue becomes the leader of the A_r round.

5. If the miner H-1 (M_{H-1}) block is not in the Q_H queue, the miner next to the miner of the H-2 (M_{H-2}) block becomes a leader of the A_r round, and so on.

6. If the miners of the (H-1..1) blocks are not in the queue, the first miner in the queue becomes the round leader.

This algorithm identifies and checks the miner, who creates each block of the chain by calculating the list of authorized miners for each moment of time. If the block was not created by the designated leader within the allotted time, no blocks are generated within that round, and the round is skipped. Leaders who skip block generation are temporarily excluded from the queue by the algorithm described in the paragraph *Exclusion of inactive miners*.

The block generated by the leader A_r with the time of the block T_H from the half-interval $(T_0 +(r-1)*(t+t_s$); $T_0 +(r-1)*(t+t_s$)+t] is determined to be valid. The block created by the miner out of its turn or not in time is considered invalid. After a round of t duration, the network synchronizes the data for t_s . The leader A_r has t_s seconds to propagate the validation block over the network. If any node of the network during ts has not received a block from the leader A_r , this node recognizes the round as 'skipped' and expects a new H block in the next round r+1, from the following leader A_{r+1} .

The consensus parameters t and t_s are configured in the *node configuration file*. *The parameter T should be the same for all network participants*, otherwise the network will fork.

Synchronization of time between network hosts

Each host should synchronize the application time with a trusted NTP server at the beginning of each round. The server address and port are specified in the node configuration file. The server must be available to each network node.

Exclusion of inactive miners

If any miner misses generation of a block N_{ban} times in a row, this miner is excluded from the queue for t_{ban} of next blocks (the `ban-duration-blocks` parameter in the node configuration file). Each node excludes an inactive miner on its own based on the calculated queue Q_H and information about the H block and the M_H miner. The P_{ban} parameter specifies the maximum percentage of excluded miners in the network in comparison with all active miners at any moment. If the N_{ban} of misses is achieved by a miner, but at the same time the P_{ban} is also achieved, this miner will not be excluded from the queue.

Monitoring

The PoA consensus monitoring helps to identify how non-valid blocks are created and distributed, as well as how miners skip the queue. Network administrators perform additional troubleshooting and blocking of malicious nodes.

To monitor the process of generating blocks using the PoA algorithm, the following details are entered in InfluxDB:

- Active list of miners sorted by the timestamp of granting of mining rights.

- Scheduled round timestamp.

- Actual round timestamp.

- Current miner.

Changing consensus settings

The consensus parameters (round time and synchronization period) are changed on the basis of the node configuration file at the `from-height` of the blockchain. If any node does not specify new parameters, the blockchain will fork.

Configuration example:

```
// specifying inside of the blockchain parameter
consensus {
  type = poa
  sync-duration = 10s
  round-duration = 60s
  ban-duration-blocks = 100
  changes = [
    {
      from-height = 18345
      sync-duration = 5s
      round-duration = 60s
    },
    {
      from-height = 25000
      sync-duration = 10s
      round-duration = 30s
    }]
}
```

See also

General platform configuration: consensus algorithm

Consensus algorithms

LPoS consensus algorithm

CFT consensus algorithm

1.28.3 CFT consensus algorithm

When information is exchanged extensively in a corporate blockchain, coherence among the network elements that form a single blockchain is important. And the more participants are engaged in the exchange, the more likely it is that an error will occur: a hardware failure by one of the participants, network problems, and so on. This can lead to forks of the main blockchain and, as a consequence, rollback of a block that seems to be already formed and included in the blockchain. In this case, the blocks subject to the rollback begin to be mined again and become unavailable in the blockchain for some time. This, in turn, can affect the business processes that use the blockchain. The Crash Fault Tolerance (CFT) consensus algorithm is designed to prevent such situations.

Algorithm description

The CFT consensus algorithm is based on the *PoA* with an added phase for voting of **mining round validators**: network participants that are automatically appointed by the consensus algorithm. This approach guarantees the following:

- more than a half of participants (validators) are familiar with a definite block and have validated it;

- the block will not be rollbacked and will be published in the blockchain;

- there will be no parallel chain in the network.

This is achieved by the finalization of a produced block. The finalization itself is based on the consensus of majority of round validators (50% + 1 vote). In accordance with this consensus, the decision of block broadcasting is taken. If this majority has not been achieved, mining will be stopped until the network cohesion is restored.

Like the PoA, the CFT algorithm depends on the current time, starting and ending time of each mining round is calculated upon the basis of a *genesis block* timestamp. Basic parameters that form an algorithm that is used for appointment of a current block miner are also identical to the PoA parameters (see the *PoA consensus algorithm* section). For validation of blocks, the `consensus` block of the node configuration file has been expanded with three new parameters:

- **max-validators** – limit of validators participating in a current round.

- **finalization-timeout** – time period, during which a miner waits for finalization of the last block in a blockchain. After that time, the miner will return the transactions back to the UTX pool and start mining the round again.

- **full-vote-set-timeout** – optional parameter which defines, how long a miner will wait for the full set of votes from all validators after the end of the round (node configuration file parameter: `round-duration`).

The following terms are used for the following description of CFT functionality:

- t – round duration in seconds (parameter of the node configuration file: round-duration).

- t_{start} – round start time.

- t_{sync} – blockchain synchronization time (t_{start} + t).

- t_{end} – round end time.

- t_{fin} - time period during which a miner waits for the finalization of the last block (parameter of the node configuration file: `finalization-timeout`).

- V_{max} – the maximum number of validators taking part in voting (parameter of the node configuration file: `max-validators`).

Voting

The general scheme of the CFT consensus mining round:

Voting is performed in each round, nodes with the miner role can take part in it. Voting starts upon t_{sync} and ends by t_{end} + t_{fin} . Within each time period defined for voting, *voting of validators* and *voting of current round miner* are performed. Each validator of the round can send multiple votes, but a miner can vote only once for its last microblock.

For voting, instance of a vote is used, which includes following parameters:

- **senderPublicKey** – a public key of a validator which has formed a vote;

- **blockVotingHash** – hash of a *liquid block* with votes confirmed by a validator;
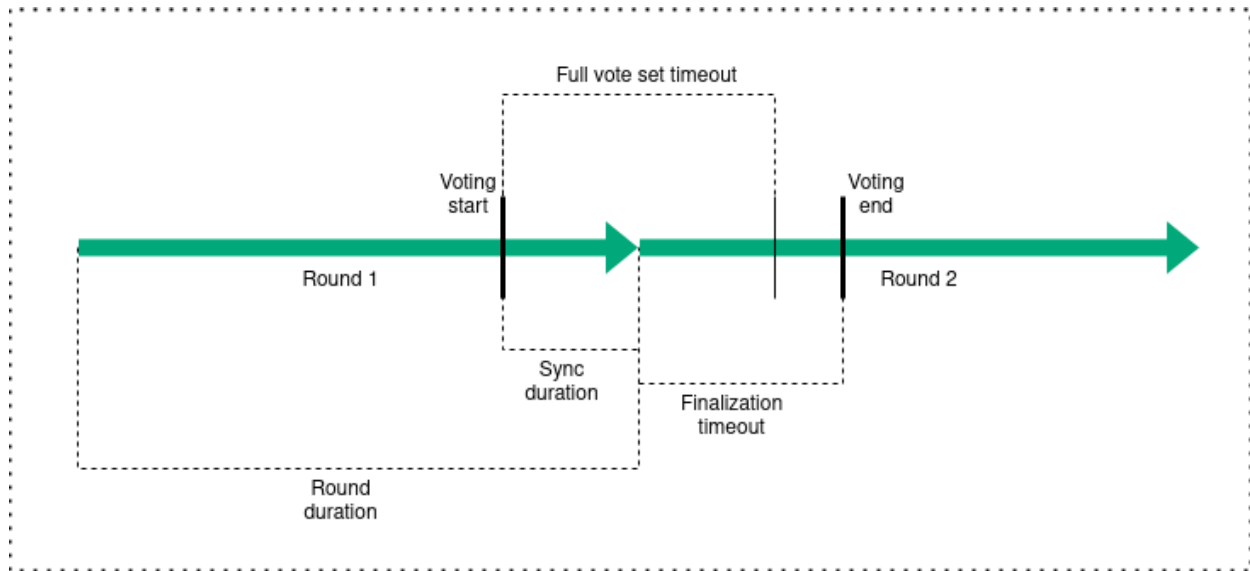
- **signature** – vote signature formed by a validator.

Defining of round validators and their voting

In order to define validators that can vote in a current round, a configurable node parameter max-validators (V_{max}) is used. If the number of active miners minus the current round miner does not exceed V_{max}, each of them can take part in voting. Otherwise, in order to define validators of a current round, the pseudorandom selection algorithm is used which allows to exclude the influence of a particular miner on choices of voters.

Voting of validators start under two preconditions:

- the next attempt to vote falls within the time interval required for voting;
- the address of the current node is one of the defined validators of the round for voting.

After the end of the round validators voting, the miner voting is started.

Voting of current round miners

The miner's vote is triggered under two conditions:

- the next attempt to vote falls within the time interval required for voting;
- the address of the current node is the miner of the round.

A vote is considered valid if it was issued by an address that is in a list of validators of the current round and has a correct signature. As soon as a miner gains the required number of votes, voting time slot is checked. Then the finalizing microblock with all votes is released. The block with votes is considered finalized.

Mining features

The basic rules of CFT consensus mining are identical to the PoA consensus rules. However, an additional mechanism has been introduced to ensure consensus fault tolerance.

With CFT consensus, another mining attempt is considered a failure in case the last received block has not been finalized – in other words, a microblock with valid votes has not been applied to the state. In this case, if the mining attempts exceed the $t_{start} + t_{fin}$, the node decides to return all transactions from the last block back to the UTX pool, after that the round starts mining again.

To avoid the possible return of your transactions into the UTX pool, it is highly recommended to work not with the current (liquid) block, but with a finalized one that has been already validated by the network participants.

Selecting a channel for synchronization

The PoS and PoA consensus algorithms use a module that selects the strongest chain for synchronization by comparing the data of the involved nodes. CFT uses a different selection mechanism, which also increases system fault tolerance: it selects a random channel from the channels that are active at the moment of synchronization. The list of active channels is constantly updated during the system operation, and the synchronization time with a particular channel is limited to distribute the load evenly over the network.

Changing consensus parameters

Like in the PoS and PoA consensus algorithms, the consensus parameters are configured in the node configuration file. The configuration example is stated below:

```
consensus {
  type: cft
  warnings-for-ban: 3
  ban-duration-blocks: 15
  max-bans-percentage: 33
  round-duration: 7s
  sync-duration: 2s
  max-validators: 7
  finalization-timeout: 4s
  full-vote-set-timeout: 4s
}
```

Recommendations for CFT configuration are stated in the *General platform configuration: consensus algorithm* section.

See also

> *Consensus algorithms*
>
> *Consensus algorithms*
>
> *LPoS consensus algorithm*
>
> *PoA consensus algorithm*

The Waves Enterprise Mainnet uses the **Leased Proof of Stake** consensus algorithm for the internal decision making. To support this, the **WEST** technical token has been developed, which serves as a proof of the right for mining, as well as a financial motivation for the participants.

Sidechains and private networks based on the Waves Enterprise blockchain platform can use any of three supported consensus algorithms, depending on needs of a certain project. A private network consensus algorithm is configured in the *node configuration file*.

See also

> *General platform configuration: consensus algorithm*

1.29 Cryptography

The Waves Enterprise platform uses the Waves cryptographic algorithm.

The table below lists the cryptographic functions used.

Table 10: **Cryptographic functions and algorithms used**

Functionality	Cryptographic functions and algorithms
Hash coding	**Blake2b256** and **Keccak256** functions consequently
Digital signature	Based on the **Curve25519** elliptic curve (ED25519 with X25519 keys)
Data encryption	AES symmetric data encryption
Confidential data encryption	TLS v1.2 with cryptoset TLS_RSA_WITH_AES_256_CBC_SHA

1.29.1 Hash coding

As indicated in the table above, in the Waves cryptography hash coding is performed consequently by the **Blake2b256** and **Keccak256** functions.

The size of an output data block is **256 bits**.

1.29.2 Electronic signature

As shown in the table above, in Waves cryptography, the algorithms for key generation, and digital signatures forming and verifying are implemented on the basis of the **Curve25519** elliptic curve (ED25519 with X25519 keys).

Learn more about the digital signatures generation and verification with the use of the gRPC and REST API methods in the following sections: *gRPC: generation and checking of data digital signatures* и *REST API: generation and checking of data digital signatures*.

1.29.3 Protecting confidential data

On the Waves Enterprise platform you can use the TLS protocol to protect data transmitted between nodes. The supported protocols are listed in the table above.

To enable TLS, set the `node.network.tls` parameter to `true` in the **node.conf** node configuration file.

If the TLS protocol is not used to create connections between nodes (the `node.network.tls` parameter is set to `false`), a TLS-like end-to-end encryption scheme using session keys based on the **Diffie-Hellman protocol** is used to protect transmitted confidential data (privacy). This protection will only be applied to confidential data when transmitted between nodes peer-to-peer, i.e. between two network members.

Here you can see the scheme of the text data encryption procedure based on the Diffie-Hellman protocol:
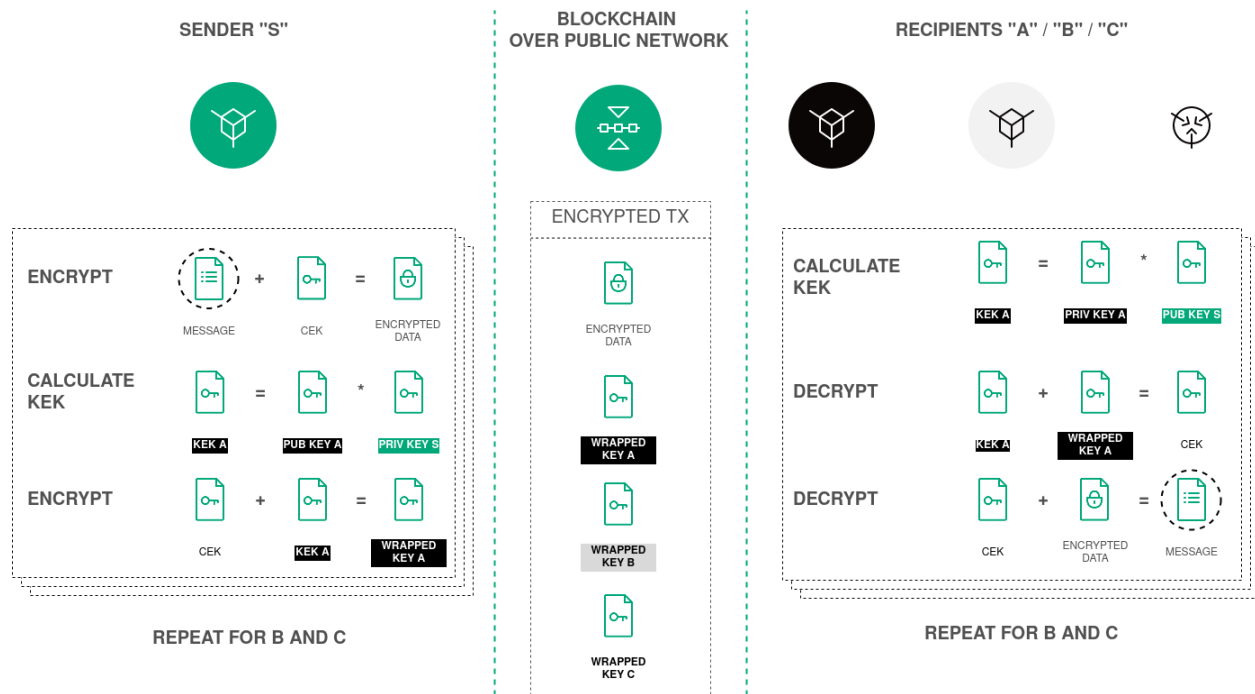
Note: The platform also uses theTLS protocol when working with smart contracts for the following connections:

- connection with the Docker-host (Docker-TLS);
- connection from a smart contract to a node via gRPC API.

TLS setup and usage in these cases are documented in the *General platform configuration: execution of smart contracts* section.

See also

General platform configuration: cryptography

Precise platform configuration: node gRPC and REST API configuration

Precise platform configuration: TLS

REST API: encryption and decryption methods

REST API: generation and checking of data digital signatures

contract_pki_service.proto

1.30 Permissions

The Waves Enterprise blockchain platform implements a permissioned blockchain model: only authorized participants can have access to it.

The platform also has a role (permission) model which allows to separate permissions of the network participants. Permission management is performed with the use of the *102 Permission Transaction*.

1.30.1 Description of permissions

permissioner

A participant with the `permissioner` role is a network administrator and is entitled to add or remove any roles of participants. The first `permissioner` is appointed upon the start of the blockchain network.

sender

A participant with the `sender` role is entitled to send transactions into the network.

This role can be enabled with the use of the `sender-role-enabled` parameter which is to be found in the `genesis` block of the *node configuration file*.

banned

The `banned` role temporarily or permanently limits the transaction sending from the participant to the blockchain. The address with the `banned` role is added to the **blacklist** of nodes – the list of addresses from which no transactions are accepted.

blacklister

A participant with the `blacklister` role is entitled to temporarily or constantly restrict the activity of other participants by adding the `banned` role to their accounts. To do this, a `blacklister` sends the *102* transaction with the corresponding parameters.

miner

A participant with the `miner` role can be chosen as a round miner. In this case he will be entitled to form a next blockchain block.

issuer

A participant with the `issuer` role is entitled to issue, reissue and burn tokens.

contract_developer

A participant with the `contract_developer` role is entitled to create smart contracts in the blockchain.

Learn more about smart contracts and usage of this role in the *Smart contracts* article.

contract_validator

A participant with the `contract_validator` role is entitled to validate smart contracts to be created or updated in the blockchain.

Learn more about smart contracts and usage of this role in the *Smart contract validation* article.

connection-manager

A participant with the `connection-manager` role is entitled to connect and disconnect network nodes. As a rule, network administrator is also appointed as a `connection-manager`.

Learn more about node connection and disconnection in the article *Connection and removing of nodes*.

1.30.2 Permission management

The permission list can be changed only by a node with the `permissioner` role. Roles are added and removed with the use of the *102 Permission Transaction*. You can sign the transaction using the *sign* REST API method, and broadcast it using the corresponding *gRPC* or *REST API* method.

The process of assigning a permission to and removal from a participant is described in the *Role management* article.

Prior to sending the 102 transaction, the node performs the following checks:

1. The sender of the 102 transaction is not included in the **blacklist**.

2. The sender address has the `permissioner` role.

3. The `permissioner` role of the address is active at the moment of transaction sending.

4. The role stated in the 102 transaction is not active in case it is added to the address and, vice versa, is active in case it is removed.

Adding and removing permissions is performed by broadcasting the corresponding transactions into the blockchain. Permissions can be arbitrarily combined for any address; a permission can be removed at any moment.

See also

> *REST API: information about permissions of participants*
>
> *Description of transactions*

1.31 Client

Waves Enterprise Client is a web application for interaction with the Waves Enterprise blockchain in the *Mainnet*.

The client consists of the following sections:

- *Network stats* – general information about the current state of the Waves Enterprise Mainnet, statistical data of the network and oracles;

- *Explorer* – information about transactions sent to the network;

- *Tokens* – issue, transfer and leasing of tokens;

- *Contracts* – smart contract broadcasting in the network;

- *Data transfer* – sending of data transactions and files, work with confidential data access groups;

- *Network settings* – information about network nodes, registration of new nodes and leasing calculation;

- *Write to us* – the Waves Enterprise support feedback form.

You can access settings of your profile in the upper right corner of the page by clicking on an icon with your e-mail address.

The *Address* button in the upper right corner of the page will direct you to the node address form or the form for creation of a new blockchain address and linkage of your profile to it. After setting up of the address, you will be able to list information about your account (public and private keys, seed phrase, current balance).

The 'Address' window also allows you to manage permissions of other blockchain network participants, if you have the *permissioner* permission.
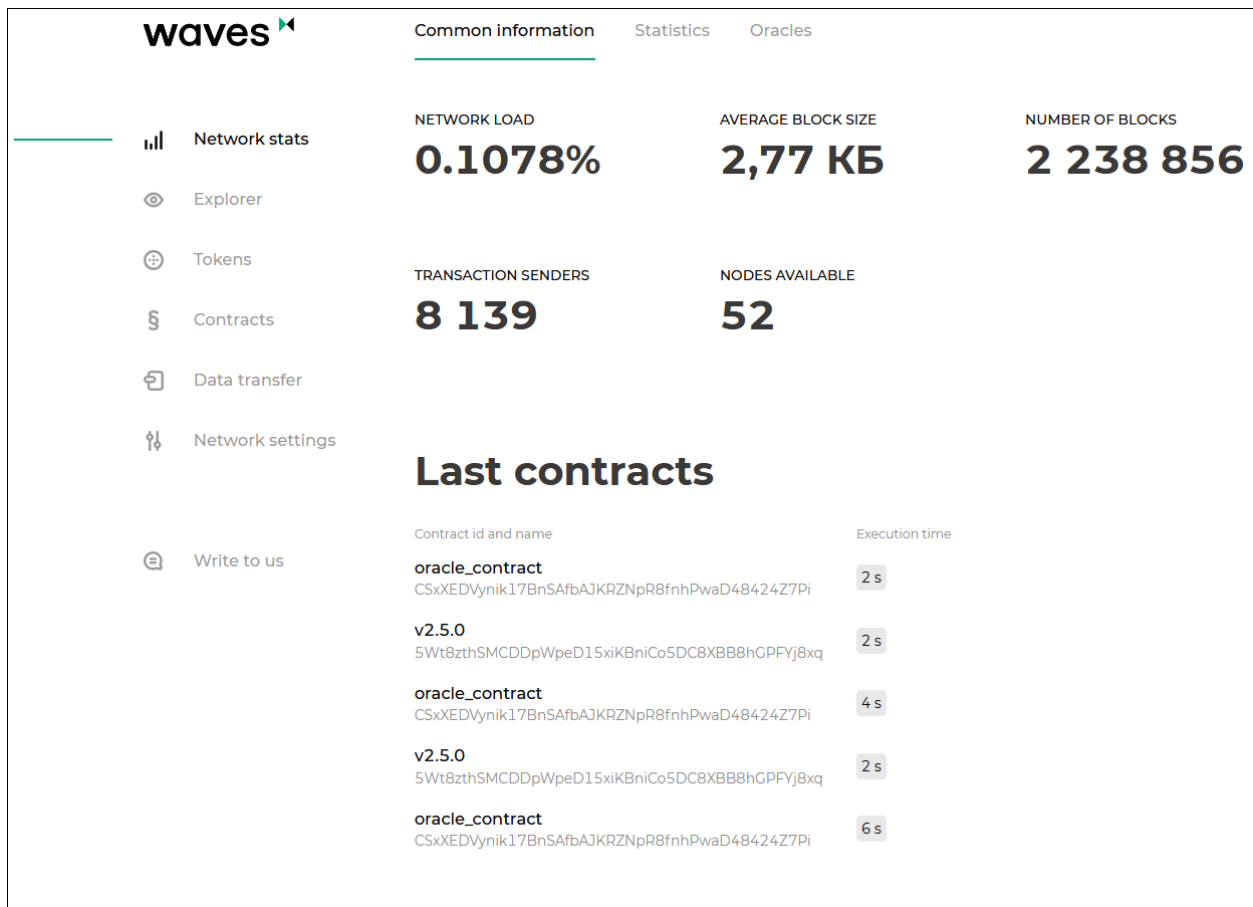
Working with **Ledger Nano** is described in the following section

1.31.1 Use Ledger Nano Devices with Waves Enterprise Client

Introduction

Ledger Nano is a hardware wallet for storing digital assets. Ledger Nano uses an offline (cold storage) method of private key generation, so it is considered one of the most reliable ways to store digital assets and many cryptocurrency users choose it. Below are the settings required to use Ledger Nano with the Waves Enterprise Client. The Waves Enterprise Client allows you to transfer tokens using the Ledger Nano device.

Prerequisites

1. You've initialized your **Ledger Nano** device.

   Note: Waves Enterprise supports Ledger Nano S, Ledger Nano S+ and Ledger Nano X models.

2. The latest firmware is installed.

3. Ledger Live is ready to use.

4. Google Chrome or Firefox browsers are installed.

Install Waves Enterprise App on Ledger Device

1. Open the Manager in Ledger Live.

2. Connect and unlock your Ledger Nano device.

3. If asked, allow the Manager on your device by pressing the right button.

4. In the Ledger Live catalog, find Waves Enterprise app and click `Install`.

   Note: The Waves Enterprise application requires about 40 kB to install. The exact size of the application is specified in the Ledger Live catalog.

The installation window will appear and your Ledger Nano device will display `Processing...`; then the app installation will be finished.

Open Waves Enterprise App on Your Ledger Nano Device

1. Once the Waves Enterprise application is installed, use the left or right button to find it on the dashboard.

2. Press both left and right buttons simultaneously to launch the app.

Use Ledger Nano Device with Waves Enterprise App

1. Make sure your Ledger Nano Device is connected, unlocked, other cryptocurrency apps are not running and not intercepting the connection between Ledger Nano and Waves Enterprise app.

2. Open Waves Enterprise Client in Google Chrome or Firefox browser.

3. Log in to your account and click the left button in the top menu to select or add address.

1. Click `Add address`.

2. Then select `Add address from ledger`.

3. On the next page enter account index (address id) or range of indexes then click the `Submit` button.

4. Select the address you need, name it and use as the current address.

Transferring tokens

At the moment, only transfer transactions are supported.

To transfer tokens:

1. In the Waves Enterprise Client, open the `Tokens` tab and click `Transfer token`.

2. On the next page enter the recipient's address, amount of tokens and the transfer description.

3. Verify data on you Ledger and sign the transaction.

Note: If you open the Waves Enterprise Client on a new machine or in a new browser, you have to validate it on the Ledger.

If you need help setting up work with Ledger Nano, contact us.

See also

Client

1.31.2 Network stats

The **General information** tab of the "Network stats" section shows the current state of the Waves Enterprise Mainnet:

- network load;

- average block size;

- total number of blocks in the network;

- number of nodes and transaction senders;

- last called smart contracts.

The **Stats** tab shows the basic metrics of the blockchain:

- Number of transactions in the network;

- Number of smart contract call transactions;

- Number of transactions for token operations;

- Number of other transactions;

- List of last called smart contracts;

- List of smart contract images being in use;

- Number of active addresses;

- Top 10 addresses according to number of sent transactions;
- Top 10 miner nodes;
- Token cycle stats.

The **Oracles** tab shows data obtained from external sources.
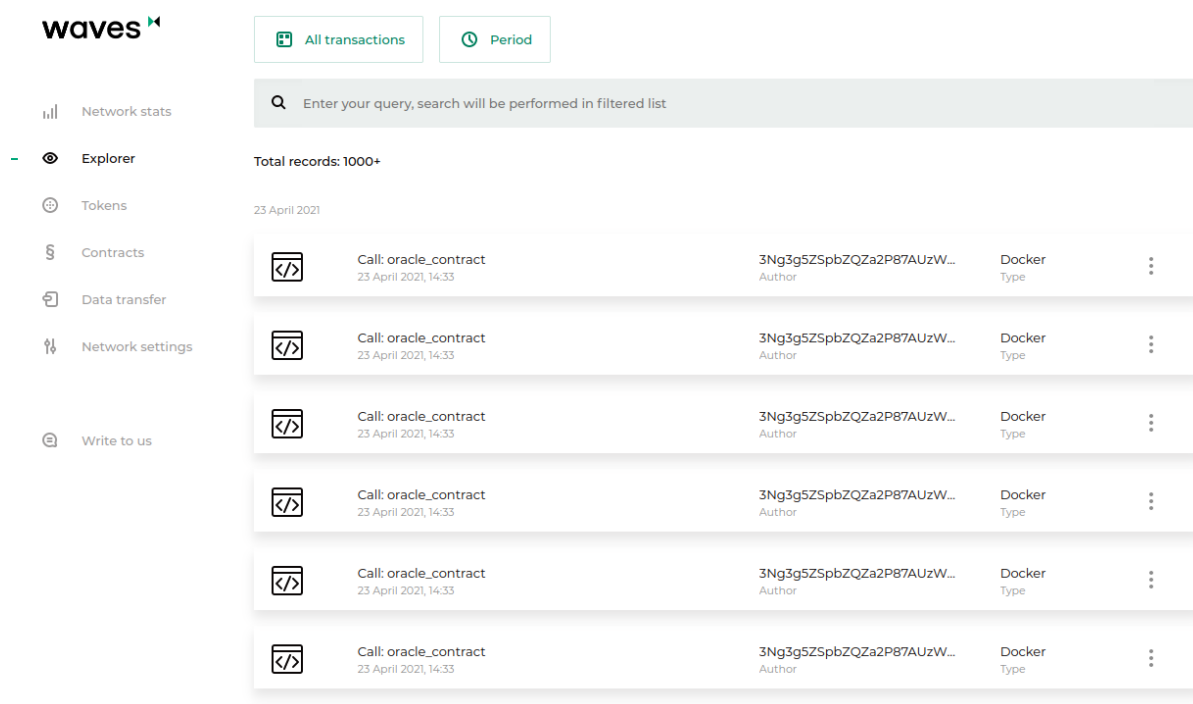
The relative chart shows dependence of WEST price from conventional assets in the following pairs:

- WEST - USDN;
- BTC - USD;
- BRENT - USD;
- Gold - USD;

The WEST price chart shows price of the WEST token in other cryptocurrencies:

- WEST - USDN;
- WEST - WAVES;
- WEST - BTC.

1.31.3 Explorer

The "Explorer" section contains information about transactions in the blockchain. Broadcasting timestamp is available as a search filter, as well as following categories:

- participants;
- data transactions;
- transaction identifiers;

- names of smart contracts;

- transaction signatures;

- number of a block containing transactions.

Additional filters are also available for showing of a definite transaction category:

- *Tokens* – token operations;

- *Contracts* – smart contract operations;

- *Data transactions*;

- *Permissions* – permission management;

- *Groups* – confidential data groups management;

- *Unconfirmed transactions* – UTX pool content.

The **Users** link situated in the end of filter list will direct you to the list of the network users with a filter according to their permissions.

1.31.4 Tokens
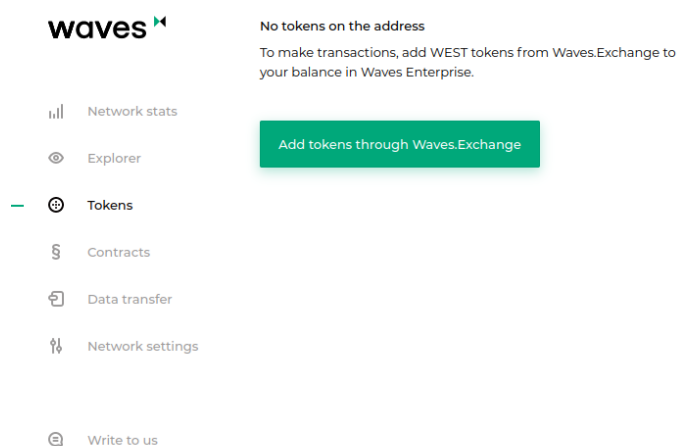
If you do not have tokens on your address, the "Tokens" section will show a button that will redirect you to the Waves Exchange.

In case you have tokens on your address, the tab will show your current balance, as well as buttons for transfer of tokens to other network participants, tokens leasing and issue. Issue of tokens requires the *issuer* permission of your address.

1.31.5 Contracts

The "Contracts" section contains information about smart contracts installed in the blockchain. It also allows to start a smart contract. To search for smart contracts, use filtering by transaction parameters which is available in the search bar:

- authors and senders of transactions;
- signatures;
- smart contract identifiers;
- smart contract names;
- Docker image name.

Additional filters are also available to display smart contract of the selected category:

- *My contracts* – the smart contracts developed and installed by you;
- *All contracts* – default value;
- *Disabled smart contracts* – smart contracts disabled by their developers with the use of the *106* transaction.

When you select a contract, its card opens.

The page of each smart contract contains the following tabs:

- **Information** – author address, image name, checksum, smart contract version and creation date;
- **Data** – the result of the last smart contract call;
- **Call** – on this tab, you can call the smart contract if you have sufficient balance on your address;
- **Update** – information about the last contract update;

- **Version history** – a table with Docker image names, creation timestamps and checksums for each smart contract version.

Contract call

Using the Client you can load parameters for the following transactions with csv or json:

- *CallContract Transaction*;

- *Data Transaction*,

To load parameters, on the **Call** tab, click the **Import from File (CSV, JSON)** link, and then upload the file. The json file must be an array of objects, each of which has the following keys:

- `value`;

- `key` – the name of the key, a string;

- `type` – one of the following values:

    - `integer`;

    - `string`;

    - `boolean`;

    - `binary (base64)`.

Learn more about smart contracts of the Waves Enterprise blockchain platform in the article *Smart contracts*.

1.31.6 Data transfer

The "Data transfer" section allows to sign and send data transactions into the blockchain. You can also create confidential data groups and send confidential data transactions into them in this section.

Learn more about confidential data groups in the article *Confidential data exchange*.

At the **Data record** tab allows you to create and send a data transaction. To do this, fill the "key-value" fields and choose a recipient address.
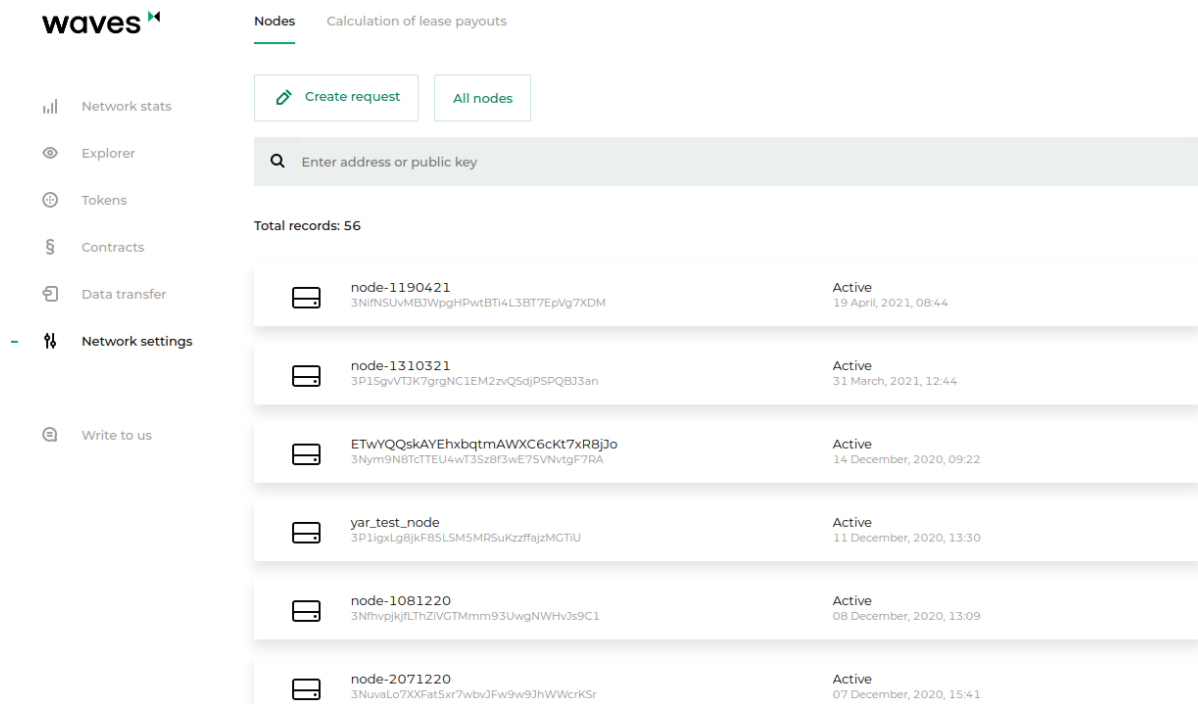
At the **Groups** tab, you can create and edit confidential data groups and send data transactions to them. This tab also shows confidential data groups you are a member of.

1.31.7 Network settings

The "Network settings" sections allows to list information about nodes registered in the network, as well as to calculate leasing.

The **Node** tab shows information about the blockchain network:

- Public key;

- Address;

- Status;

- Address of a last transaction sender that have changed the node state;

- Last node state change timestamp;

- Presence of **miner** or **banned** permissions;

- Node membership in confidential data groups with information about these groups.

Search and filtering of nodes according to the following parameters are available:

- Name;
- Address;
- Public key;
- Activity in the network.
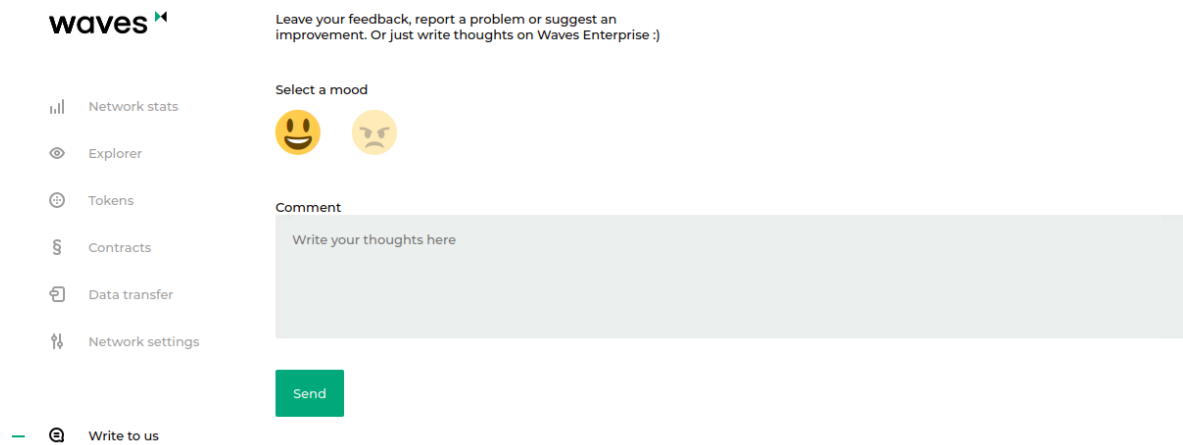
You can also send a request for connection of a new node to the network by pressing the *Create request*.

The **Calculation of lease payouts** tab contains the form for leasing calculation.

The calculation has the following algorithm:

1. A generating balance is requested from the leasing pool node for the beginning of a calculated period;

2. Leasing sum is calculated taking into account miner revenues (each miner should receive 40% for his own block and 60% for a previous block);

3. The sum is divided for each pool participant proportionately with a total sum of assets in leasing and the node generating balance at a defined height;

4. The calculated leasing sum is multiplied by a revenue percentage;

5. The node generating balance is re-calculated for a new height taking into account new and cancelled leasings.

1.31.8 Write to us

In the "Write to us" section, you can write any comment or message for the Waves Enterprise technical support service.

See also

Attachment of a client to the private blockchain

1.32 Generators

Generators is a set of utilities included in the supply package of the Waves Enterprise blockchain platform. Generators are supplied as a package file **generator-x.x.x.jar**, where `x.x.x` is the blockchain platform release version.

Generators for each version of the blockchain platform are available in the Waves Enterprise official GitHub repository.

In order to work with the generators, you have to install the Java Runtime Environment for your operating system. All components of the generator set are operated in the terminal or command line.

The generator set includes following utilities:

- **AccountsGeneratorApp** – node account generator;
- **GenesisBlockGenerator** – genesis block signer;
- **ApiKeyHash** – a generator for hash coding of an API key string required for node API authorization;

1.32.1 AccountsGeneratorApp

The **AccountsGeneratorApp** is used for configuration of a node account in a private network – a set of data about a blockchain network participant. To generate an account, you have to set up the **accounts.conf** file in the node directory.

accounts.conf configuration file example:

```
accounts-generator {
  crypto {
    type = WAVES
  }
  chain-id = T
  amount = 5
  wallet = ${user.home}"/node/wallet/wallet1.dat"
  wallet-password = "some string as password"
  reload-node-wallet {
    enabled = false
    url = "http://localhost:6869/utils/reload-wallet"
  }
}
```

Running of the **AccountsGeneratorApp**:

```
java -jar generator-x.x.x.jar AccountsGeneratorApp YourNode/accounts.conf
```

The generator creates a node public key (account) and stores it in the `keystore.dat` file in the directory of your node. If necessary, you can set a keypair password.

Hint: If you have set the password for your keypair, you have to state it in the `password` field while creating queries and transactions.

Learn more about node account generation in the *Creation of a node account* section.

1.32.2 GenesisBlockGenerator

The **GenesisBlockGenerator** is used for signing of a private network genesis block – the first block of a new network which contains transactions that define initial balances and permissions. To sign a genesis block, the generator uses the `blockchain.genesis` block of the **node.conf** node configuration file.

Running of the **GenesisBlockGenerator**:

```
java -jar generator-x.x.x.jar GenesisBlockGenerator YourNode/node.conf
```

The generator fills the fields `genesis-public-key-base-58` (a public key of a genesis block) and `signature` (genesis block signature) of a node configuration file.

Learn more about genesis block signing in the section *Genesis block signing*.

1.32.3 ApiKeyHash

The **ApiKeyHash** utility is used for authorization of the node API methods (gRPC and REST API interfaces for data exchange). For generation of a JWT token (in case of oAuth authorization) or a token based on an `api-key` string, the generator uses the **api-key-hash.conf** configuration file in the node directory.

Running of the **ApiKeyHash**:

```
java -jar generator-x.x.x.jar ApiKeyHash YourNode/api-key-hash.conf
```

The utility generates a JWT token or a hash of an entered `api-key` string, which are stated in the `auth` section of the node configuration file.

api-key-hash.conf example:

```
apikeyhash-generator {
  crypto {
    type = WAVES
  }
  api-key = "some string for api-key"
}
```

Learn more about gRPC and REST API authorization in the section *Precise platform configuration: gRPC and REST API authorization*.

See also

Architecture

1.33 Authorization and data services

The Waves Enterprise blockchain platform includes two external services:

- **Authorization service**, which provides authorization of all network components;
- **Data service**, which gathers blockchain data into a database and provides API for access to the gathered data.

1.33.1 Authorization service

This service provides authorization of all blockchain network components on the basis of the **oAuth 2.0** protocol. oAuth 2.0 is the open authorization framework which allows to grant a third party restricted access to a user's protected resources without disclosing login and password.

The general chart of the oAuth 2.0 authorization:

The object of the oAuth authorization is the JSON Web Token (JWT). Tokens are used for authorization of every query from a client to a server and have a limited lifetime. A client receives two tokens – **access** and **refresh**. An access token is used for authorization of queries for access to protected resources and storage of additional information about a user. A refresh token is used for receiving of a new access token and updating of a refresh token.

The authorization scheme of the Waves Enterprise blockchain platform:

The general authorization procedure is carried out as follows:

Sub	Auth key
Client 1	Ewt43twet
Client 2	36y21erqq
Client 3	6yt78ngy6
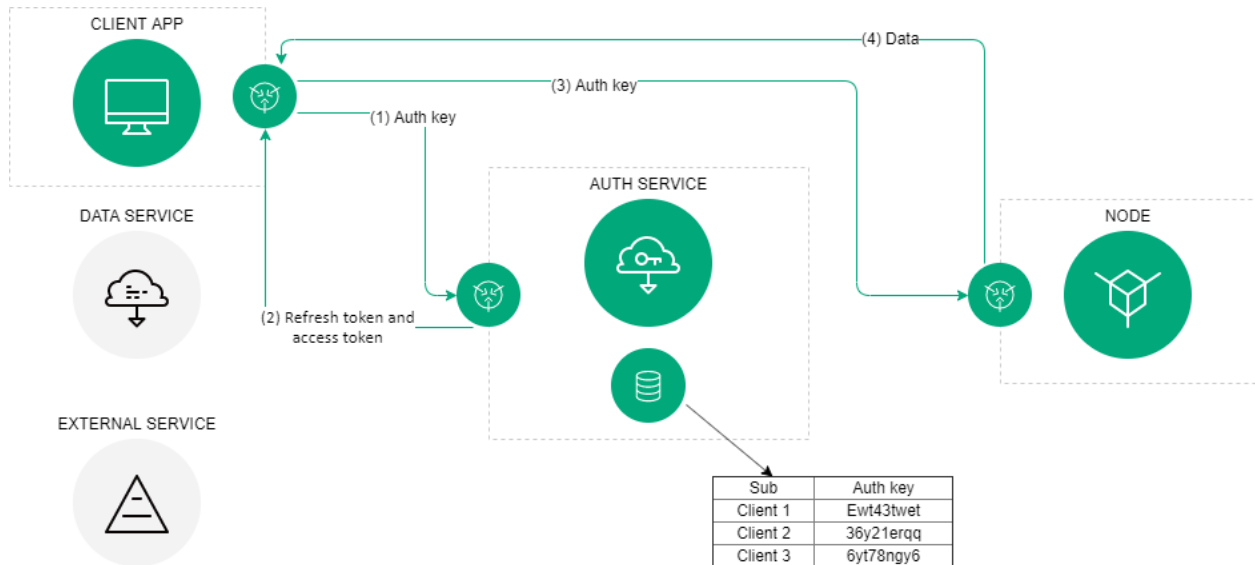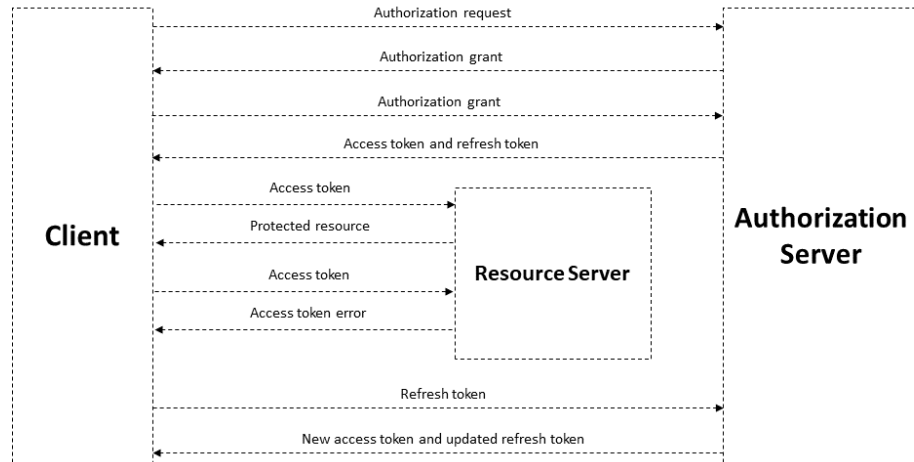
1. A client (blockchain network component: a corporate client, data exchange service or a third-party application) provides its authentication data once to the authorization service;

2. In case of a successful primary authentication, the authorization service saves the client`s authentication data in the data storage, generates the signed access and refresh tokens and sends them to the client. The tokens include the token lifetime and basic client`s data: identifier and role. Clients' authentication data is stored in the authorization service configuration file. Each time before sending a query to a third-party service, a client checks an access token lifetime. In case of token lifetime expiry, a client refers to the authorization service to obtai a new access token. For these queries to the authorization service, the client uses the refresh token;

3. With an actual access token, a client sends a query for obtaining of a third-party service data;

4. A third-party service checks an access token lifetime, its integrity and compares an authorization service public key, received in advance, with a key, which is stored in the access token signature. If the check is successful, a third-party service provides required data to a client.

See the following article for a description of authorization methods:

Authorization service: authorization variants

The authorization service supports two authorization options for accessing node API methods:

- `api-key` string hash authorization;
- `JWT token` authorization.

You can set the authorization option for accessing API methods in the *auth section* of the node configuration file.

Depending on a used authorization method, different values are put in queries or in the fields of the Swagger framework for providing access to the API:

- `ApiKey or PrivacyApiKey (apiKey)` – the value of the `api-key` string hash;
- `OAuth2 Bearer (apiKey)` – the value of the **access** token.

Api-key string hash authorization

A hash of a string defined by a user can be calculated with the use of the **ApiKeyHash** utility from the *generator* package. You can also generate a key string hash by yourself with the use of the *POST /utils/hash/secure* method.

A sample request with authorization by api-key hash:

```
curl -X POST
--header 'Content-Type: application/json'
--header 'Accept: application/json'
--header 'X-API-Key: 1' -d '1' 'http://2.testnet-pos.com:6862/transactions/calculateFee'
```

Available authorizations ✕

OAuth2 Bearer (apiKey)

Name: Authorization
In: header
Value:

> Fbt5fKHesnQG2CXmsKf4TC

> **Authorize** **Close**

ApiKey or PrivacyApiKey (apiKey)

Name: X-API-Key
In: header
Value:

> **Authorize** **Close**

JWT token authorization

If the oAuth protocol authorization is used, a client receives a **refresh** and an **access** tokens for access to the API methods. These tokens can be obtained with the use of the *authorization service REST API methods*.

The method *POST /v1/user* is used for registration of users. The method query contains following parameters:

- `login` – user login (e-mail address);

- `password` – the account password;

- `locale` – language of further e-mail notifications (possible options: `en` and `ru`);

- `source` – user type:

    - `license` – the blockchain platform usage *license* owner;

    - `voting` – a user of the Waves Enterprise Voting service.

After registration, a user has an opportunity to request **access** and **refresh** tokens.

The following methods are used to obtain and refresh authorization tokens:

1. *POST /v1/auth/login* – obtaining authorization token using login and password. This method is used for authorization of users.

2. *POST /v1/auth/token* – obtaining **refresh** and **access** authorization tokens for services and applications. This method requires no parameters in its query and returns values of the tokens in response. The method can be used only by the authorization service administrator.

3. *POST /v1/auth/refresh* – updating the **refresh** token. The token value is passed in the method query.

Note: A user must have a specific *authorization role* encrypted in the user's JWT token in order to invoke some of the *REST API methods*.

See also

Authorization and data services

data-sv-conf

REST API: authorization service methods

REST API: methods of the data service

Precise platform configuration: gRPC and REST API authorization

OAuth2 authorization roles

1.33.2 Data service

The data service is used for gathering blockchain data into a database. This service has its own API for access to the gathered data.

In the Waves Enterprise Mainnet, the data service operates in the autonomous mode, access to its API is restricted. For deployment in a private network, the data service is configured by the Waves Enterprise specialists, depending on the peculiarities of a project. You can also change data service parameters by yourself with the use of environment variables that are described in the article Data service: manual configuration.

1.33.3 API methods of the integration services

Definite REST API methods are available for the integration services for data exchange:

REST API: authorization service methods

GET /status

The method is used for obtaining of the authorization service status.

Example of the service response:

GET /status:

```
{
  "status": "string",
  "version": "string",
  "commit": "string"
}
```

POST /v1/user

The method is used for registration of a new user via the authorization service.

The method query contains following data:

- `login` – user login (e-mail address);
- `password` – account password;
- `locale` – e-mail notifications language (possible variants: *en* and *ru*);
- `source` – user type:
    - `license` – owner of a blockchain platform *license* ;
    - `voting` – user of the Waves Enterprise Voting service.

If the registration has been carried out successful, the method returns the 201 code. In case of another response, a user has not been registered.

GET /v1/user/profile

The method is used for obtaining of user data.

Example of the service response:

GET /v1/user/profile:

```
{
  "id": "string",
  "name": "string",
  "locale": "en",
  "addresses": [
    "string"
  ],
  "roles": [
    "string"
  ]
}
```

POST /v1/user/address

The method is used for obtaining of a user address identifier. The method query contains following data:

- `address` − user blockchain address;
- `name` − user name.

Example of the service response:

POST /v1/user/address:

```
{
  "addressId": "string"
}
```

GET /v1/user/address/exists

The method is used for checking of a user e-mail address. The method query contains a user e-mail address.

Example of the service response:

GET /v1/user/address/exists:

```
{
   "exist": true
}
```

POST /v1/user/password/restore

The method is used for restoring of an account password.

The method query contains following data:

- email – user e-mail;
- source – user type:
    - license – owner of a blockchain platform *license* ;
    - voting – user of the Waves Enterprise Voting service.

Example of the service response:

POST /v1/user/password/restore:

```
{
    "email": "string"
}
```

POST /v1/user/password/reset

The method is used for user password reset.

The method query contains following data:

- token – user authorization token;
- password – current user password.

Example of the service response:

POST /v1/user/password/reset:

```
{
   "userId": "string"
}
```

GET /v1/user/confirm/–code″

The method is used for confirmation of a password restoring code for a user account. The method query contains a confirmation code value.

POST /v1/user/resendEmail

The method is used for resending of a password recovery code to a specified e-mail.

The method query contains following data:

- `email` – user e-mail;
- `source` – user type:
    - `license` – owner of a blockchain platform *license* ;
    - `voting` – user of the Waves Enterprise Voting service.

The method response returns a user e-mail, to which a restoring code was sent.

Example of the service response:

POST /v1/user/resendEmail:

```
{
    "email": "string"
}
```

POST /v1/auth/login

The method is used for obtaining of a new authorization token for a user.

The method query contains following data:

- `name` – user name;
- `password` – account password;
- `locale` – e-mail notifications language (possible variants: *en* and *ru*);
- `source` – user type:
    - `license` – owner of a blockchain platform *license* ;
    - `voting` – user of the Waves Enterprise Voting service.

Example of the service response:

POST /v1/auth/login:

```
{
   "access_token": "string",
   "refresh_token": "string",
   "token_type": "string"
}
```

POST /v1/auth/token

The method is used for obtaining of authorization tokens for external services and applications. This method does not require any query parameters.

Example of the service response:

POST /v1/auth/token:

```
{
   "access_token": "string",
   "refresh_token": "string",
   "token_type": "string"
}
```

POST /v1/auth/refresh

The method is used for obtaining of a new **refresh** token. The method query contains a current **refresh** token value.

Example of the service response:

POST /v1/auth/refresh:

```
{
   "access_token": "string",
   "refresh_token": "string",
   "token_type": "string"
}
```

GET /v1/auth/publicKey

The method is used for obtaining of an authorization service public key. This method does not require any parameters in its query.

Example of the service response:

POST /v1/auth/refresh:

```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA7d9Oj/ZQTkkjf4UuMfUu
QIFDTYxYf6QBKMVJnq/wXyPYYkV8HVFYFizCaEciv3CXmBH77sXnuTlrEtvK7zHB
KvV87OHmZuazjIgZVSkOnOY7F8UUVNXnlzVD1dPsOGJ6orM41DnC1W65mCrP3bjn
fV4RbmykN/lk7McA6EsMcLEGbKkFhmeq2Nk4hn2CQvoTkupJUnOCP1dhO4bq1lQ7
Ffj9K/FJq73wSXDoH+qqdRG9sfrtgrhtJHerruhv3456e0zyAcDO8+sJUQFKY8OB
SZMEndVzFS2ub9Q8e7BfcNxTmQPM4PhHO5wuTqL32qt3uJBx2OI4lu3OND44ZrDJ
BbVog73oPjRYXj+kTbwUZI66SP4aLcQ8sypQyLwqKk5DtLRozSN00IrupJJ/pwZs
9zPEggL91TOrirbEhGlf5U8/6XN8GVXX4iMk2fD8FHLFJuXCD7Oj4JC2iWfFDC6a
uUkwUfqfjJB8BzIHkncoqOZbpidEE2lTWl+svuEu/wyP5rNlyMiE/e/fZQqM2+oO
cH5Qow6HH35BrloCSZciutUcd1U7YPqESJ5tryy1xn9bsMb+On1ocZTtvec/ow4M
RmnJwmOj1nd+cc19OKLG5/boeA+2zqWuOjCbWR9cOoCmgbhuqZCHaHTBEAKDWcsC
VRz5qD6FPpePpTQDb6ss3bkCAwEAAQ==
-----END PUBLIC KEY-----
```

See also

Authorization and data services

data-sv-conf

Authorization service: authorization variants

REST API: methods of the data service

REST API: methods of the data service

Following API methods are available for the data service:

Assets method group

The methods of the **Assets** group are used for obtaining of data about token sets (assets).

GET /assets

The method is used for obtaining the blockchain available assets list. The list consists of transactions for emission of the corresponding assets.

Response example:

GET /assets:

```
[
  {
    "index": 0,
    "id": "string",
    "name": "string",
    "description": "string",
    "reissuable": true,
    "quantity": 0,
    "decimals": 0
  }
]
```

POST /assets/count

The method returns a number of available assets in the blockchain.

Response example:

POST /assets/count:

```
{
  "count": 0
}
```

GET /assets/–id″

The method returns information about an available asset according to its `{id}`.

The response of the method contains following data:

- `index` – asset index number;
- `id` – asset identifier;
- `name` – asset name;
- `description` – asset description;
- `reissuable` – reissuability of an asset;
- `quantity` – the number of tokens in an asset;
- `decimals` – number of decimal places in a used token (WEST − **8**)

Response example:

GET /assets/–id″:

```
{
  "index": 14,
  "id": "12nx0qnhjd83",
  "name": "Demo asset",
  "description": "Demo asset",
  "reissuable": true,
  "quantity": 400,
  "decimals": 8
}
```

Blocks method group

GET /blocks/at/–height″

The method returns content of a block at a defined `height`.

The response of the method contains following parameters:

- `reference` – block hash sum;

- `blocksize` – size of a block;

- `features` – *features* activated at the moment of block generation;

- `signature` – block signature;

- `fee` – total fee for the transactions included in a block;

- `generator` – block creator address;

- `transactionCount` – number of transactions included in a block;

- `transactions` – array with bodies of transactions included in a block;

- `version` – block version;

- `poa-consensus.overall-skipped-rounds` – number of skipped mining rounds in case the *PoA* consensus algorithm is used;

- `timestamp` – block **Unix Timestamp** (in milliseconds);

- `height` – height of block generation.

Response example:

GET /blocks/at/–height″:

```
{
  "reference":
↪"hT5RcPT4jDVoNspfZkNhKqfGuMbrizjpG4vmPecVfWgWaGMoAn5hgPBJpC9696TL8wGDKJzkwewiqe8m26C4aPd
↪",
  "blocksize": 226,
  "features": [],
  "signature":
↪"5GAM7jfQScw4g3g7PCNNtz5xG3JzjJnW4Ap2soThirSx1AmUQHQMjz8VMtkFEzK7L447ouKHfj2gMvZyP5u94Rps
```

```
→",
  "fee": 0,
  "generator": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
  "transactionCount": 0,
  "transactions": [],
  "version": 3,
  "poa-consensus": {
    "overall-skipped-rounds": 1065423
  },
  "timestamp": 1615816767694,
  "height": 1826
}
```

Contracts method group

Methods of the **Contracts** group are used for obtaining of information about smart contracts of the blockchain.

GET /contracts

The method returns information about all smart contracts installed in the network. For each smart contract, following parameters are returned:

- `contractId` – smart contract identifier;

- `image` – name of a smart contract Docker image or its absolute path in its registry;

- `imageHash` – smart contract hash sum;

- `version` – smart contract version;

- `active` – smart contract status at the moment of the query: `true` – working, `false` – not working.

Example of an answer for one smart contract:

GET /contracts:

```
[
  {
    "contractId": "dmLT1ippM7tmfSC8u9P4wU6sBgHXGYy6JYxCq1CCh8i",
    "image": "registry.wvservices.com/wv-sc/may14_1:latest",
    "imageHash": "ff9b8af966b4c84e66d3847a514e65f55b2c1f63afcd8b708b9948a814cb8957",
    "version": 1,
    "active": false
  }
]
```

GET /contracts/count

The method returns a number of smart contracts on a blockchain that correspond with defined provisions and filters.

Response example:

GET /contracts/count:

```
{
  "count": 19
}
```

GET /contracts/info/–contractId″

The method returns information about a smart contract with a definite {contractId}.

Response example:

GET /contracts/id/–id″:

```
{
  "creator": "9yxe6Kw9eiCD2mTNvKdrcQ1EoQqzMy7p52USZftBtQhp",
  "contractId": "7zcrHAFZmcZ3EGs7JWL5jCrbizCppu2rcpDDuChNTF6K",
  "image": "registry.wvservices.com/waves-enterprise-public/east-contract:v1.2",
  "imageHash": "baef03e82e4ecc723b85876111cbe25ed390ad7c62169e8a3ba142b6a2ad3000",
  "version": 5,
  "active": true,
  "validationPolicy": {
      "type": "majority_with_one_of",
      "addresses": [
        "3NyJPnLBdEQiPdHoHHgQAYX6UVj6GKMxgMx",
        "3NmHrYoC8S2SUosy6UJp47bBwq2Cr2X6Yq1",
        "3NrKDuHjUG7vSCiMMD259msBKcPRm4MvaJu"
    ]
  },
  "apiVersion": "1.0"
}
```

GET /contracts/id/–id″/versions

The method returns version history of a smart contract with a definite {id}.

Example of a response for one version:

GET /contracts/id/–id″/versions:

```
[
  {
    "version": 0,
    "image": "string",
    "imageHash": "string",
    "timestamp": "string"
  }
]
```

GET /contacts/history/–id″/key/–key″

Возвращает историю изменений ключа {key} смарт-контракта по его {id}.

Example of a response for one key:

GET /contacts/history/–id″/key/–key″:

```
{
  "total": 777,
  "data": [
    {
      "key": "some_key",
      "type": "integer",
      "value": "777",
      "timestamp": 1559347200000,
      "height": 14024
    }
  ]
}
```

GET /contracts/senders-count

The method returns a number of unique participants that send transactions *104* for smart contract calls.

Response example:

GET /contracts/senders-count:

```
{
  "count": 777
}
```

GET /contracts/calls

The method returns a list of *104* transactions for smart contract calls with their parameters and results.

Example of a response for one transaction:

GET /contracts/calls:

```
[
  {
    "id": "string",
    "type": 0,
    "height": 0,
    "fee": 0,
    "sender": "string",
    "senderPublicKey": "string",
    "signature": "string",
    "timestamp": 0,
    "version": 0,
    "contract_id": "string",
    "contract_name": "string",
    "contract_version": "string",
    "image": "string",
    "fee_asset": "string",
    "finished": "string",
    "params": [
      {
        "tx_id": "string",
        "param_key": "string",
        "param_type": "string",
        "param_value_integer": 0,
        "param_value_boolean": true,
        "param_value_binary": "string",
        "param_value_string": "string",
        "position_in_tx": 0,
        "contract_id": "string",
        "sender": "string"
      }
    ],
    "results": [
      {
        "tx_id": "string",
        "result_key": "string",
        "result_type": "string",
        "result_value_integer": 0,
        "result_value_boolean": true,
        "result_value_binary": "string",
        "result_value_string": "string",
        "position_in_tx": 0,
        "contract_id": "string",
        "time_stamp": "string"
      }
```

(continues on next page)

```
    ]
  }
]
```

Privacy method group

Methods of the **Privacy** group are used for obtaining of information about confidential data groups.

GET /privacy/groups

The method returns a list of confidential data groups in the blockchain.

Example of a response for one group:

GET /privacy/groups:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

GET /privacy/groups/count

The method returns a number of confidential data groups in the blockchain.

Response example:

GET /privacy/groups/count:

```
{
  "count": 2
}
```

GET /privacy/groups/–address"

The method returns a list of confidential data groups that include a defined `{address}`.

Example of a response for one group:

GET /privacy/groups/–address″:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

GET /privacy/groups/by-recipient/–address″

The method returns a list of privacy data groups that include a defined {address} as a recipient of data.

Example of a response for one group:

GET /privacy/groups/by-recipient/–address″:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

GET /privacy/groups/–address″/count

The method returns a number of confidential data groups that include a defined {address}.

Response example:

GET /privacy/groups/–address″/count:

```
{
  "count": 1
}
```

GET /privacy/groups/id/–id″

The method returns information about a privacy data group with a definite {id}.

Response example:

GET /privacy/groups/id/–id″:

```
{
  "id": "string",
  "name": 0,
  "description": "string",
  "createdAt": "string"
}
```

GET /privacy/groups/id/–id″/history

The method returns a history of changes of a confidential data access group with a definite {id}. The history is returned as a list of sent *112-114 transactions* with their descriptions.

Example of a response for one transaction:

GET /privacy/groups/id/–id″/history:

```
{
  "id": "string",
  "name": 0,
  "description": "string",
  "createdAt": "string"
}
```

GET /privacy/groups/id/–id″/history/count

The method returns a number of 112-114 transactions sent for changing of an access group with a definite {id}.

Response example:

GET /privacy/groups/id/–id″/history/count:

```
{
  "count": 0
}
```

GET /privacy/nodes

The method returns a list of available nodes in the blockchain.

Example of a response for one node:

GET /privacy/nodes:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

GET /privacy/nodes/count

The method returns a number of available nodes in the blockchain.

Response example:

GET /privacy/nodes/count:

```
{
  "count": 0
}
```

GET /privacy/nodes/publicKey/–targetPublicKey″

The method returns information about a node according to its `{targetPublicKey}`.

Response example:

GET /privacy/nodes/publicKey/–targetPublicKey″:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

GET /privacy/nodes/address/–address″

The method returns information about a node according to its `{address}`.

Response example:

GET /privacy/nodes/address/–address″:

```
[
  {
    "id": "string",
    "name": 0,
    "description": "string",
    "createdAt": "string"
  }
]
```

Transactions method group

Methods of the **Transactions** group are used for obtaining of information about transactions in the blockchain.

GET /transactions

The method returns a list of transactions corresponding with provisions of a search query and applied filters.

Important: The **GET /transactions** method returns not more than 500 transactions for one query.

Example of a response for one transaction:

GET /transactions:

```
[
 {
    "id": "string",
    "type": 0,
    "height": 0,
    "fee": 0,
    "sender": "string",
    "senderPublicKey": "string",
    "signature": "string",
    "timestamp": 0,
    "version": 0
 }
]
```

GET /transactions/count

The method returns a number of transactions corresponding with provisions of a search query and applied filters.

Response example:

GET /transactions/count:

```
{
"count": "116"
}
```

GET /transactions/–id″

The method returns a transaction according to its `{id}`.

Response example:

GET /transactions/–id″:

```
{
"id": "string",
"type": 0,
"height": 0,
"fee": 0,
"sender": "string",
"senderPublicKey": "string",
"signature": "string",
"timestamp": 0,
"version": 0
}
```

Users method group

Methods of the **Users** group are used for obtaining information about participants of the blockchain network.

GET /users

The method returns a list of participants corresponding with provisions of a search query and applied filters.

Example of a response for one participant:

GET /users:

```
[
  {
    "address": "string",
    "aliases": [
      "string"
    ],
    "registration_date": "string",
    "permissions": [
      "string"
    ]
  }
]
```

GET /users/count

The method returns a number of participants corresponding with filters applied in the query.

Example of a response for one participant:

GET /users/count:

```
{
  "count": 1198
}
```

GET /users/–userAddressOrAlias″

The method returns information about a participants according to his address or alias.

Response example:

GET /users/–userAddressOrAlias″:

```
{
  "address": "string",
  "aliases": [
    "string"
  ],
  "registration_date": "string",
  "permissions": [
    "string"
  ]
}
```

GET /users/contract-id/–contractId″

The method returns a list of participants that have ever called a smart contract with a definite `{contractId}`.

Example of a response for one participant:

GET /users/contract-id/–contractId″:

```
{
  "address": "string",
  "aliases": [
    "string"
  ],
  "registration_date": "string",
  "permissions": [
    "string"
  ]
}
```

POST /users/by-addresses

The method returns a list of participants for a definite set of addresses.

Example of a response for one participant:

POST /users/by-addresses:

```
{
  "address": "string",
  "aliases": [
    "string"
  ],
  "registration_date": "string",
  "permissions": [
    "string"
  ]
}
```

Methods for obtaining of information about data transactions (12)

This group of methods is called via the `/api/v1/txIds/` route.

GET /api/v1/txIds/–key″

The method returns a list of identifiers for data transactions that include the defined {key}.

Example of a response for one transaction:

GET /api/v1/txIds/–key″:

```
[
  {
    "id": "string"
  }
]
```

GET /api/v1/txIds/–key″/–value″

The method returns a list of identifiers for data transactions that include defined {key} and {value}.

Example of a response for one transaction:

GET /api/v1/txIds/–key″/–value″:

```
[
  {
      "id": "string"
  }
]
```

GET /api/v1/txData/–key″

The method returns bodies of data transactions that include a defined {key}.

Example of a response for one transaction:

GET /api/v1/txData/–key″:

```
[
  {
    "id": "string",
    "type": "string",
    "height": 0,
    "fee": 0,
    "sender": "string",
    "senderPublicKey": "string",
    "signature": "string",
    "timestamp": 0,
    "version": 0,
    "key": "string",
```

```
    "value": "string",
    "position_in_tx": 0
  }
]
```

GET /api/v1/txData/–key″/–value″

The method returns bodies of data transactions that include defined {key} and {value}.

Example of a response for one transaction:

GET /api/v1/txData/–key″/–value″:

```
[
  {
    "id": "string",
    "type": "string",
    "height": 0,
    "fee": 0,
    "sender": "string",
    "senderPublicKey": "string",
    "signature": "string",
    "timestamp": 0,
    "version": 0,
    "key": "string",
    "value": "string",
    "position_in_tx": 0
  }
]
```

Leasing method group

GET /leasing/calc

The method returns a total sum for leasing of tokens in a specified block interval.

Response example:

GET /leasing/calc:

```
{
  "payouts": [
    {
      "leaser": "3P1EiJnPxFxGyhN9sucXfB2rhQ1ws4cmuS5",
      "payout": 234689
    }
  ],
```

```
    "totalSum": 4400000,
    "totalBlocks": 1600
}
```

Stats method group

Methods of the **Stats** group are used for obtaining statistical data and blockchain monitoring.

GET /stats/transactions

The method returns information about transactions that have been send within a specified time period.

Response example:

GET /stats/transactions:

```
{
  "aggregation": "day",
  "data": [
    {
      "date": "2020-03-01T00:00:00.000Z",
      "transactions": [
        {
          "type": 104,
          "count": 100
        }
      ]
    }
  ]
}
```

GET /stats/contracts

The method returns information about transactions *104* within a specified time period.

Response example:

GET /stats/contracts:

```
{
  "aggregation": "day",
  "data": [
    {
      "date": "2020-03-01T00:00:00.000Z",
      "transactions": [
        {
          "type": 104,
```

```
            "count": 100
        }
    ]
    }
  ]
}
```

GET /stats/tokens

The method returns information about turnover of tokens in the blockchain within a specified time period.

Response example:

GET /stats/tokens:

```
{
  "aggregation": "day",
  "data": [
    {
      "date": "2020-03-01T00:00:00.000Z",
      "sum": "12000.001"
    }
  ]
}
```

GET /stats/addresses-active

The method returns addresses that have been active within a specified time period.

Response example:

GET /stats/addresses-active:

```
{
  "aggregation": "day",
  "data": [
    {
      "date": "2020-03-01T00:00:00.000Z",
      "senders": "12",
      "recipients": "12"
    }
  ]
}
```

GET /stats/addresses-top

The method returns addresses that have been the most active senders or recipients within a specified time period.

Response example:

GET /stats/addresses-top:

```
{
  "aggregation": "day",
  "data": [
    {
      "date": "2020-03-01T00:00:00.000Z",
      "senders": "12",
      "recipients": "12"
    }
  ]
}
```

GET /stats/nodes-top

The method returns addresses of nodes that have created the largest number of blocks within a specified time period.

Response example:

GET /stats/nodes-top:

```
{
  "limit": "10",
  "data": [
    {
      "generator": "3NdPsjaFC7NeioGVF6X4J5A8FVaxdtKvAba",
      "count": "120",
      "node_name": "Genesis Node #5"
    }
  ]
}
```

GET /stats/contract-calls

The method returns a list of smart contracts that have been mostly called within a specified time period.

Response example:

GET /stats/contract-calls:

```
{
  "limit": "5",
  "data": [
    {
      "contract_id": "Cm9MDf7vpETuzUCsr1n2MVHsEGk4rz3aJp1Ua2UbWBq1",
      "count": "120",
      "contract_name": "oracle_contract",
      "last_call": "60.321"
    }
  ]
}
```

GET /stats/contract-last-calls

The method returns a list of last smart contract calls according to their IDs and names.

Response example:

GET /stats/contract-last-calls:

```
{
  "limit": "5",
  "data": [
    {
      "contract_id": "Cm9MDf7vpETuzUCsr1n2MVHsEGk4rz3aJp1Ua2UbWBq1",
      "contract_name": "oracle_contract",
      "last_call": "60.321"
    }
  ]
}
```

GET /stats/contract-types

The method returns a list of blockchain smart contracts according to their images and hashes.

Response example:

GET /stats/contract-types:

```
{
  "limit": "5",
  "data": [
    {
      "id": "Cm9MDf7vpETuzUCsr1n2MVHsEGk4rz3aJp1Ua2UbWBq1",
      "image": "registry.wvservices.com/waves-enterprise-public/oracle-contract:v0.1",
      "image_hash": "936f10207dee466d051fe09669d5688e817d7cdd81990a7e99f71c1f2546a660",
      "count": "60",
```

<div align="right">(continues on next page)</div>

```
        "sum": "6000"
    }
  ]
}
```

GET /stats/monitoring

The method returns general information about the network.

Response example:

GET /stats/monitoring:

```
{
  "tps": "5",
  "blockAvgSize": "341.391",
  "senders": "50",
  "nodes": "50",
  "blocks": "500000"
}
```

Anchoring method group

Methods of the **Anchoring** group are used for obtaining of information about anchoring rounds.

GET /anchoring/rounds

The method returns a list of transactions that have been sent in anchoring rounds in accordance with specified provisions and filters.

Response example:

GET /anchoring/rounds:

```
[
  {
    "height": 0,
    "sideChainTxIds": [
      "string"
    ],
    "mainNetTxIds": [
      "string"
    ],
    "status": "string",
    "errorCode": 0
  }
]
```

GET /anchoring/round/at/–height″

The method returns information about an anchoring round at a specified block {height}.

Response example:

GET /anchoring/round/at/–height″:

```
{
  "height": 0,
  "sideChainTxIds": [
    "string"
  ],
  "mainNetTxIds": [
    "string"
  ],
  "status": "string",
  "errorCode": 0
}
```

GET /anchoring/info

The method returns information about the blockchain anchoring.

Response example:

GET /anchoring/info:

```
{
  "height": 0,
  "sideChainTxIds": [
    "string"
  ],
  "mainNetTxIds": [
    "string"
  ],
  "status": "string",
  "errorCode": 0
}
```

Auxiliary methods of the data service

GET /info

The method returns information about a data service in use.

Response example:

GET /info:

```
{
  "version": "string",
  "buildId": "string",
  "gitCommit": "string"
}
```

GET /status

The method returns information about status of the data service.

Response example:

GET /status:

```
{
  "status": "string"
}
```

See also

Authorization and data services

data-sv-conf

REST API: authorization service methods

Authorization service: authorization variants

See also

Authorization service: authorization variants

data-sv-conf

REST API: authorization service methods

REST API: methods of the data service

Precise platform configuration: gRPC and REST API authorization

OAuth2 authorization roles

1.34 Differences between the opensource and the commercial versions of the Waves Enterprise blockchain platform

The Waves Enterprise blockchain platform exists in the commercial and the opensource versions.

The commercial version of the Waves Enterprise blockchain platform is intended for use in the corporate and government sectors and is distributed through *user licenses*. To purchase a commercial version of the Waves Enterprise platform, contact the Waves Enterprise sales team by email: sales@wavesenterprise.com.

The release of the Waves Enterprise opensource version distributed under the Apache 2.0 license is available on GitHub. The *30000 blocks limit on blockchain height* does not apply to the opensource version.

Table 11: Differences between the opensource and commercial versions of the Waves Enterprise blockchain platform

Functionality	Opensource	Corporate
Containerized Smart Contracts	✔	✔
Confidential data exchange	✔	✔
Consensus algorithms: LPoS, PoA, CFT	✔	✔
Anchoring	✔	✔
Cryptography:		
• Waves (Curve25519, Blake2b256 and Keccak256)	✔	✔
• GOST	✖	✔
Support of TLS	✖	✔
PKI support v1	✖	✔

Correspondingly, the *node configuration files* are different for the opensource and commercial versions of the Waves Enterprise blockchain platform. The following sections of the node configuration file are not available in the opensource version:

- *node.tls*

- node.network.tls

- *node.api.rest.tls*

- *node.api.grpc.tls*

- *node.docker-engine.docker-tls*

- node.license

- *cryptography setup*

See also

  Licenses of the Waves Enterprise blockchain platform

1.35 External components of the platform

1.35.1 External proprietary components of the platform

Table 12: List of proprietary components

Name	Version	License	License type	License link	Architecture component
CryptoPro CSP, including CryptoPro JCSP	5.0 R2	CRYPTO-PRO JSC license	Proprietary	https://www.cryptopro. ru/download?pid=1417	Node

1.35.2 External open-source components of the platform

Table 13: List of open-source components

Name	Version	License	License type	License link	Architecture component
postgres	13.x	PostgreSQL License	Freeware, opensource	https://github.com/postgres/ postgres/blob/master/ COPYRIGHT	Data crawler
nodejs	12.21	MIT License	Freeware, opensource	https://raw.githubusercontent.com/ nodejs/node/master/LICENSE	Data crawler, data service, client
npm	6.14.:	The Artistic License 2.0	Freeware, opensource	https://github.com/npm/cli/blob/ latest/LICENSE	Data crawler, data service, client
netty	4.1.x	Apache License 2.0	Freeware, opensource	https://github.com/netty/netty/ blob/4.1/LICENSE.txt	Node
rocksdb	6.13.:	Apache License 2.0	Freeware, opensource	https://github.com/facebook/ rocksdb/blob/master/LICENSE. Apache	Node
docker-java	3.2.x	Apache License 2.0	Freeware, opensource	https://github.com/docker-java/ docker-java/blob/master/LICENSE	Node
akka (http, grpc)	10.1.:	Apache License 2.0	Freeware, opensource	https://github.com/akka/akka/ blob/master/LICENSE	Node
swagger-ui	3.23.:	Apache License 2.0	Freeware, opensource	https://github.com/swagger-api/ swagger-ui/blob/master/LICENSE	Node
nginx	1.18.:	BSD License	Freeware, opensource	https://nginx.org/LICENSE	Client

See also

> *System requirements*
>
> *Installing CryptoPro CSP license*

1.36 Official resources and contacts

1.36.1 Blockchain platform official resources

- Official website of the Waves Enterprise blockchain platform
- Github page of the project
- Official website of the Waves blockchain platform

1.36.2 How to contact with us

- Waves Enterprise technical support service
- Feedback form of the blockchain platform client
- Official Telegram chat in English: Waves Enterprise Group
- Official Telegram chat in Russian: Waves Enterprise

1.37 Glossary

Authorization
Granting a participant the rights to perform certain operations on the blockchain (in particular, to use API methods)

Address
The identifier of a network member derived from its public key. Each address has its own balance and state

Account
A set of data about a network member used to identify him or her

Alias
The conditional name of a network member associated with its address. An alias is assigned to a member using the transaction 10 and can be specified in transactions instead of the address of a specific member

Anchoring
Algorithm for checking data in a private blockchain for invariance by validating it in a larger network

Asset
A digital asset in blockchain. An asset is a set of tokens

Atomic transaction
A container transaction consisting of several other transactions. If one of the transactions placed in the atomic is not executed, all other transactions are also not executed

Balance
Number of tokens owned by the address in the blockchain

Block
A set of transactions recorded in the blockchain, signed by the miner and containing a link to the signature of the previous block. Block size is limited to 1 Mb or 6000 transactions

Blockchain
A decentralized, distributed, and publicly accessible digital registry that records information in such a way that any individual record cannot be changed once it is made without changing all subsequent blocks

Validation
Confirmation of data invariability (integrity)

Generator
An auxiliary utility that allows you to create key pairs or key strings

Generating balance
Minimum balance, giving the address the right to mine

Access group

List of addresses with access to sensitive data on the blockchain

Data crawler

Service for extracting data from a node and loading it into a data preparation service

Smart contract execution

Execution of program code embedded in a smart contract in a blockchain

Key block

Initial block of a mining round, containing service information:

- public key of the miner for validation of microblock signatures;

- a miner fee for a previous block;

- the miner signature;

- a reference to a previous block

Fee

The amount of tokens an address pays for the transactions it sends to the blockchain

Consensus

Algorithm of coordination of information recorded in the blockchain between its participants

License

A document granting the right to use the Waves Enterprise blockchain platform

Leasing

Leasing of tokens on a participant's balance to other participants. Leasing is used to create a generating balance from the participant taking tokens on lease, as well as to increase the probability of the participant's selection by the miner of the next round when using the LPoS consensus algorithm

Miner

Node, having the right to create new blockchain blocks

Mining

The process of creating new blockchain blocks

Migration

The process of changing key blockchain parameters

Microblock

A set of transactions applied to a blockchain state. The number of transactions in a microblock is limited to 500 units. Microblocks form a network block. Microblocks are generated only under load: if there are no transactions, only blocks are released.

Node

A network participant's computer with the Waves Enterprise blockchain platform software installed and a network address assigned

Node update

Updating the Waves Enterprise blockchain platform software installed on a network member's computer

Image

A smart contract template that contains its code and is used to create a Docker container in which the smart contract is executed

Rollback

Sending an already created block for re-mining due to failures occurring on blockchain nodes

Peer

Node network address

Transaction signing

Adding to the body of the transaction the public key of its creator, used to confirm the integrity of the transaction in the blockchain

Private network, sidechain

A blockchain network separate from Waves Enterprise Mainnet and with its own registered participants

Private key

A string combination of characters for transactional signing and token access, to which only its owner has access. The private key is inextricably linked to the public key

Transaction broadcasting

Writing a transaction to a blockchain block during a mining round

Public network

A large blockchain network where each participant is known and registered in advance (e.g., Waves Enterprise Mainnet)

Public key

A string combination of characters inextricably linked to the private key. The public key is attached to transactions to confirm the correctness of the user's signature made on the private key

Unconfirmed transaction pool (UTX pool)

A component of the Waves Enterprise blockchain platform that stores unconfirmed transactions until they are verified and sent to the blockchain

Round

The process of mining a block by a blockchain network participant

Repository

Smart contract image repository deployed with Docker Registry software

Permission

Granting or denying of certain operations in the blockchain

Network message

Network event information sent by a node to other nodes in the blockchain

Smart contract

A separate application which saves its entry data in the blockchain, as well as the output results of its algorithm

Snapshot

A set of all the blockchain current data on accounts, smart contracts, sensitive data access groups, permissions and registered nodes. A snapshot contains no history of changing values, transactions or blocks.

Creation of a smart contract

Upload a new smart contract to the blockchain using transaction *103*

Soft fork

Mechanism for activating pre-built blockchain functionality

State

Blockchain transaction history stored in the database of each node

Address state

Data set of an individual address: balances, information about sent data transactions, results of execution of smart contracts called by the address

Smart contract state

Current data on the smart contract execution result, such as calculation result. This data is recorded

and updated using the *104* transaction. Smart contract parameters such as publication time, information about whether the smart contract has been disabled, and other information about the smart contract itself are not included in the smart contract state and are stored outside of the blockchain in the Docker registry.

Token

1. A blockchain unit used to motivate participants to mine on the network.

   When using the *platform on the Mainnet*, the WEST system token is used. In addition to the system token, you can create and use *other tokens*.

   Unlike blockchain platforms where you need to publish a *ERC-20* standard smart contact to create a new token, the Waves Enterprise network provides the native way to issue tokens via a *token issue transaction*.

2. The object used to authorize the blockchain participant

Transaction

A separate operation in the blockchain that changes the network state and is performed on behalf of a participant. By sending a transaction the participant sends a request to the network with a set of data necessary to change the state accordingly

AQDS

Advanced qualified digital signature based of Public Key Infrastructure (PKI). AQDS is issued by an accredited Certification Authority (CA). As a rule, the validity period of an AQDS is limited to one year

Participant

User of the Waves Enterprise blockchain platform software, sending transactions to the blockchain

Fork

The formation of a new blockchain branch

Keystore

A closed repository where key pairs of blockchain nodes are stored

Hash

A unique set of characters generated from raw data using a given algorithm. Hash allows to uniquely identify the raw data

Keystring hash

A set of characters generated from a key string specified by the participant and used to authorize him in the blockchain

Service Endpoint

HTTP or HTTPS address to which the HTTP method refers. The endpoint performs a specific task, accepts parameters and returns data.

API method

A separate procedure called by a member via the API of the blockchain platform (gRPC or REST API) and designed to perform a specific operation in the blockchain

CEK

Content Encryption Key – data encryption key. The key is used to encrypt text data

Crash Fault Tolerance (CFT)

A PoA-based consensus algorithm that prevents blockchain forks from occurring in the event of any malfunction by one or more participants

Genesis block

Initial block of the blockchain network, containing service transactions for the distribution of primary roles and balances of participants

KEK

Key Encryption Key used to encrypt the content encryption key (CEK)

Leased Proof of Stake (LPoS)

The PoS consensus algorithm that enables a participant to lease tokens to other participants

Liquid block

Block state during a mining round from the formation of its key block to the formation of the next key block

MVCC (Multiversion concurrency control)

A mechanism for managing concurrent access to the state of smart contracts through multiversionality. With this mechanism, the node supports the ability to execute multiple transactions of any smart contracts in parallel, while ensuring data consistency.

JWT (JSON Web Token)

JSON-formatted object used to authorize a blockchain participant using the oAuth protocol

PKI

Public Key Infrastructure in which each key is represented by two parts: public and private. For more information, see. Public key infrastructure

Proof of Authority (PoA)

Consensus algorithm, in which the ability to verify transactions and create new blocks is given to the more authoritative nodes

Proof of Stake (PoS)

A consensus algorithm in which the node that checks transactions and mines in the next round is chosen based on its current balance

Sandbox

Blockchain platform trial mode

Seed phrase

A set of 24 randomly defined words to restore access to the address balance

Targetnet

A blockchain network into which data from a private network is anchored

1.38 What is new at Waves Enterprise

1.38.1 1.15.0

The 1.15.0 is the last released version, and is marked as **latest** in this documentation.

The following articles have been modified:

- *Confidential smart contracts*
- *Confidential data exchange*
- *Smart contracts*
- *WASM Smart contracts*
- *Atomic transactions*

- *General platform configuration: execution of smart contracts*
- *gRPC services used by Docker smart contracts*
- *gRPC: obtaining information on the results of the execution of a smart contract call*
- *gRPC: transfer of confidential smart contract data*
- *REST API: working with confidential smart contracts*
- *REST API: information about smart contracts*
- *System errors*

The 1.15.0 version contains critical fixes, see release description for details.

1.38.2 1.14.0

The following articles have been added:
- *WASM Smart contracts*
- *Development and usage of WASM smart contracts*
- *Smart Account*
- *System errors*

The following articles have been modified:
- *Smart contracts*
- *Calling a Docker smart contract and saving the results of its operation*
- *List of available feature identifiers*
- *103. CreateContract Transaction*
- *104. CallContract Transaction*
- *107. UpdateContract Transaction*
- *105. ExecutedContract Transaction*
- *13. SetScript Transaction*
- *120. Atomic Transaction*
- *Atomic transactions*
- *OAuth2 authorization roles*
- *REST API: information about a consensus algorithm in use*
- *gRPC: transfer of confidential smart contract data*
- *GET /contracts/status/{id}*
- *GET /contracts/executed-tx-for/{id}*
- *REST API: validation of addresses and aliases of network participants*

The 1.14.0 version contains critical fixes, see release description for details.

1.38.3 1.13.0

The following articles have been added:

- *Confidential smart contracts*
- *gRPC: transfer of confidential smart contract data*
- *REST API: working with confidential smart contracts*
- *OAuth2 authorization roles*
- *Precise platform configuration: setting the fee for the transactions sent to the blockchain*
- *GET /permissions/contract-validators*
- *GET /permissions/contract-validators/{height}*

The following articles have been modified:

- *Role management*
- *gRPC: monitoring of blockchain events*
- *GET /addresses/scriptInfo/{address}*
- *Development and usage of smart contracts*
- *Smart contracts*
- *GET /contracts/{contractId}*
- *POST /contracts/{contractId}*
- *GET /contracts/executed-tx-for/{id}*
- *103. CreateContract Transaction*
- *104. CallContract Transaction*
- *105. ExecutedContract Transaction*
- *107. UpdateContract Transaction*
- *114. PolicyDataHash Transaction*
- *List of available feature identifiers*
- *Actual versions of transactions*
- *Precise platform configuration: confidential data groups configuration*
- *Versions of Docker smart contract API*
- *GET /consensus/minersAtHeight/{height}*
- *Deploying the platform in the trial mode (Sandbox)*
- *System requirements*
- *Precise platform configuration: gRPC and REST API authorization*

The 1.13.0 version contains critical fixes, see release description for details.

1.38.4 1.12.3

The following articles have been added:

- *GET /contracts/balance/details/{ContractID}*
- *Precise platform configuration: logging*

The following articles have been modified:

- *GET /leasing/active/{address}*
- *105. ExecutedContract Transaction*
- *List of available feature identifiers*
- *GET /contracts/status/{id}*
- *Information on the results of the execution of a smart contract call*
- *REST API: information about network blocks*
- *gRPC: monitoring of blockchain events*
- *REST API: information about configuration and state of the node, stopping the node*
- *gRPC: obtaining node information*
- *REST API: retrieving certificates*
- *gRPC: retrieving certificates*
- *GET /transactions/unconfirmed/size*
- *gRPC: obtaining information about UTX pool size*
- *Signing and sending transactions*
- *Sending transactions into the blockchain*
- *POST /transactions/signAndBroadcast*
- *REST API: confidential data exchange and obtaining of information about confidential data groups*
- *gRPC: handling confidential data*
- *addresses group*
- *gRPC: information about the network members' addresses*
- *What the platform REST API is for*
- *Authorization and data services*
- *Authorization service: authorization variants*
- *POST /pki/verify*
- *Development and usage of smart contracts*
- *Installing a smart contract into the blockchain*
- *Calling a Docker smart contract and saving the results of its operation*
- *POST /addresses/verifyText/{address}*
- *GET /addresses/scriptInfo/{address}*
- *Environment requirements for the Waves Enterprise blockchain platform*
- *POST /node/logging*

- *Node start with a snapshot*

The 1.12.3 version contains critical fixes, see release description for details.

1.38.5 1.12.2

The following articles have been added:

- *GET /privacy/%policyId%/transactions*
- *Launching the network*

The following articles have been modified:

- *Activation of blockchain features*
- *Atomic transactions*
- *120. Atomic Transaction*
- *3. Issue Transaction*
- *5. Reissue Transaction*
- *6. Burn Transaction*
- *8. Lease Transaction*
- *9. LeaseCancel Transaction*
- *10. CreateAlias Transaction*
- *11. MassTransfer Transaction*
- *12. Data Transaction*
- *14. Sponsorship Transaction*
- *102. Permission Transaction*
- *103. CreateContract Transaction*
- *104. CallContract Transaction*
- *106. DisableContract Transaction*
- *107. UpdateContract Transaction*
- *111. RegisterNode Transaction*
- *112. CreatePolicy Transaction*
- *113. UpdatePolicy Transaction*
- *114. PolicyDataHash Transaction*
- *Actual versions of transactions*
- *Versions of Docker smart contract API*
- *Precise platform configuration: confidential data groups configuration*
- *Confidential data exchange*
- *GET /contracts/status/{id}*
- *gRPC: obtaining information on the results of the execution of a smart contract call*
- *Permission management*

- *Role management*

The 1.12.2 version contains critical fixes, see release description for details.

1.38.6 1.12.1

The following articles have been added:
- *Tokens of the Waves Enterprise blockchain platform*
- *Handling tokens from a Docker smart contract*

The following articles have been modified:
- *gRPC services used by Docker smart contracts*
- *contract_contract_service.proto*
- *Permissions*
- *gRPC tools*
- *Smart contracts*
- *List of available feature identifiers*
- *Glossary*
- *POST /utils/hash/secure*

The 1.12.1 version contains critical fixes, see release description for details.

1.38.7 1.12.0

The following articles have been added:
- *General platform configuration: cryptography*
- *REST API: retrieving certificates*
- *gRPC: retrieving certificates*

The following articles have been modified:
- *Anchoring*
- *node.conf*
- *Examples of node configuration files*
- *Licenses of the Waves Enterprise blockchain platform*
- *Obtaining a private network license and associated files*
- *Deployment of the platform in a private network*
- *Platform configuration for operation in a private network*
- *gRPC: obtaining node configuration parameters*
- *REST API: information about configuration and state of the node, stopping the node*
- *REST API: information about smart contracts*
- *Generators*
- *POST /utils/hash/fast*

- *POST /privacy/sendData*
- *POST /privacy/sendDataV2*
- *POST /privacy/sendLargeData*
- *Sending confidential data to the blockchain*
- *Sending confidential data to the blockchain*
- *Activation of blockchain features*
- *gRPC: obtaining node configuration parameters*
- *Use Ledger Nano Devices with Waves Enterprise Client*
- *REST API: signing and validating messages in the blockchain*
- *REST API: encryption and decryption methods*
- *REST API: generation and checking of data digital signatures*
- *Signing and sending transactions*
- *Sending transactions into the blockchain*
- *REST API: confidential data exchange and obtaining of information about confidential data groups*
- *gRPC: handling confidential data*
- *gRPC: obtaining information on the results of the execution of a smart contract call*
- *103. CreateContract Transaction*
- *External components of the platform*
- *Cryptography*
- *General platform configuration: consensus algorithm*
- *General platform configuration: mining*
- *Genesis block signing*
- *REST API: information about address assets and balances*
- *Development and usage of smart contracts*
- sc-example-rest
- *103. CreateContract Transaction*
- *104. CallContract Transaction*
- *107. UpdateContract Transaction*
- *Activation of blockchain features*

The 1.12.0 version contains critical fixes, see release description for details.

1.38.8 1.11.0

Following articles have been added:

- *Differences between the opensource and the commercial versions of the Waves Enterprise blockchain platform*
- *WE Contract SDK (Java/Kotlin Contract SDK) Client*

The following articles have been modified:

- *Deployment of the platform in a private network*
- *Deploying a platform with connection to Mainnet*
- *Precise platform configuration: node gRPC and REST API configuration*
- *Precise platform configuration: TLS*
- *General platform configuration: execution of smart contracts*
- *Licenses of the Waves Enterprise blockchain platform*

The 1.11.0 version contains critical fixes, see release description for details.

1.38.9 1.8.4

Following articles have been added:

- *Use Ledger Nano Devices with Waves Enterprise Client*
- *Constructing smart contracts with JS Contract SDK*
- *Constructing smart contracts with Java/Kotlin Contract SDK*

The following articles have been modified:

- *103. CreateContract Transaction*
- *Contracts method group*
- *General platform configuration: execution of smart contracts*
- *REST API usage*
- *Mainnet fees*

The 1.8.4 version contains critical fixes, see release description for details.

1.38.10 1.8.2

The 1.8.2 version contains critical fixes, see release description for details.

1.38.11 1.8.0

The following articles have been modified:

- *Precise platform configuration: confidential data groups configuration*
- *REST API: encryption and decryption methods*
- *Glossary*
- *System requirements*
- *Precise platform configuration: TLS*
- *Example of how to prepare artefacts for TLS*
- *Precise platform configuration: node gRPC and REST API configuration*
- *General platform configuration: execution of smart contracts*
- *node.conf*
- *General platform configuration: mining*
- *Environment requirements for the Waves Enterprise blockchain platform*
- *gRPC tools*
- *gRPC: monitoring of blockchain events*
- *gRPC: obtaining node information*
- *contract_transaction_service.proto*
- *gRPC: obtaining information on the results of the execution of a smart contract call*
- *gRPC: obtaining information about UTX pool size*
- *contract_pki_service.proto*
- gRPC: encryption and decryption methods
- *gRPC: handling transactions*
- *gRPC: handling confidential data*
- *REST API: confidential data exchange and obtaining of information about confidential data groups*
- *gRPC: retrieving auxiliary information*
- *gRPC: information about the network members' addresses*
- *Confidential data exchange*
- *REST API: information about configuration and state of the node, stopping the node*
- *Smart contracts*
- *Activation of blockchain features*
- *Client*
- *Data immutability in a blockchain*

The 1.8.0 version contains critical fixes, see release description for details.

1.38.12 1.7.3

The 1.7.3 version contains critical fixes, see details in the release description.

1.38.13 1.7.2

The following articles have been modified:

- generating_balance
- *Creation of a node account*
- *Genesis block signing*
- *Smart contract validators fee mechanism*
- *Glossary*

1.38.14 1.7.0

Following article has been added:

Precise platform configuration: node in the watcher mode

1.38.15 1.6.2

The following articles have been modified:

- *Description of transactions*
- *gRPC services used by Docker smart contracts*
- *Smart contracts*
- *Permissions*
- *Snapshooting*
- *Activation of blockchain features*
- *System requirements*

1.38.16 1.6.0

The structure and content of the documentation have been fully changed, the landing page with the search line and quick access to the basic sections have been added.

The following articles describing the snapshot mechanism developed in the 1.6.0 version have been added:

- *Snapshooting*
- *Node start with a snapshot*
- *Precise platform configuration: snapshot*

1.38.17 1.5.2

The article *CFT consensus algorithm* has been changed.

The 1.5.2 version contains critical fixes, see details in the release description.

1.38.18 1.5.0

Following articles have been added:

- *CFT consensus algorithm*
- Preparing to work
- gRPC methods of the node
- *Monitoring of events in the blockchain with the use of the gRPC*

The following articles have been modified:

- *Cryptography*
- *Managing permissions*
- *Transactions*
- Preparing configuration files
- Changes to the node config file
- Description of the node configuration file parameters and sections
- Consensus setup
- API instruments of the node
- JavaScript SDK
- *Glossary*
- Content of the Docker Config section has been transferred into the new Preparing to work section
- The section Docker smart contracts with the use of the node REST API has been deleted from the index

1.38.19 1.4.0

Following articles have been added:

- *Atomic transactions*
- *Working in the web client*
- *JavaScript SDK*

The following articles have been modified:

- *Architecture*
- *Transactions*
- Authorization type configuration for the REST API and gRPC access
- REST API instruments of the node
- Updating a Mainnet node

1.38.20 1.3.1

Following articles have been added:

- Parallel contract execution

The following articles have been modified:

- Smart contract creation
- Docker setup

1.38.21 1.3.0

The following articles have been modified:

- gRPC methods of the node
- The "Role model" and "Access managing" sections have been converted to a section Permissions managing
- Description of the node configuration file parameters and sections
- Privacy data access groups configuration
- Docker setup
- REST API Addresses methods
- REST API Node methods
- REST API Contracts methods
- REST API Privacy methods
- System requirements

1.38.22 1.2.3

The following articles have been modified:

- Docker smart contract
- Description of the node configuration file parameters and sections
- Privacy access groups configuration

1.38.23 1.2.2

Following articles have been added:

- REST API Debug methods
- The complete REST API description on the API Documentation page

The following articles have been modified:

- Installing and running the platform

1.38.24 1.2.0

Following articles have been added:

- A new Integration services section, which includes Authorization service и Data preparation service.
- Obtaining a license section was added.
- A new REST API Licenses method was added.
- A new *Smart contract run with gRPC* section was added
- A new gRPC services available to smart contract section was added.

The following articles have been modified:

- Installing and running the platform
- Updated: *Cryptography*. Part of information was moved into Data encryption operations
- Changes in the node configuration file
- Transactions

1.38.25 1.1.2

The following articles have been modified:

- Demo version
- Changes in the node configuration file
- 'Node installation' section was converted into 'Installing and running the platform'
- Connecting participants to the network
- Anchoring Configuration
- Authorization type configuration for the REST API access
- Connection of the node to the "Partnernet"
- Connection of the node to the "Mainnet"
- *System requirements*

1.38.26 1.1.0

Following articles have been added:

- API methods available to smart contract
- Sandbox
- Changes in the node configuration file

The following articles have been modified:

- Docker Smart Contracts
- Example of starting a contract
- Node installation
- Additional services deploy

1.38.27 1.0.0

Following articles have been added:

- Authorization service

Following articles have been changed:

- Node configuration

- Connection to Mainnet and Partnernet

- *REST API*

- Node installation

Changes in the `node.conf` *configuration file*

- The NTP server article has been added

- The `auth` section for authorization type configuration has been added in the REST API article