Technical description of the Waves Enterprise platform

Release 1.6.0

https://wavesenterprise.com

Jan 26, 2024

CONTENTS

CHAPTER

ONE

DOCUMENTATION OF THE WAVES ENTERPRISE BLOCKCHAIN PLATFORM

Waves Enterprise blockchain platform is a complex distributed ledger system that allows to build public and private blockchain networks for performing of tasks in corporate and public sectors.

1.1 What is blockchain?

Blockchain is a continuous consequent chain of linked blocks that contain some information. This chain is filled with new blocks. Process of generating of new blocks is defined as *mining*. Each block contains a hash sum of previous block data. This makes impossible to change content of any block after its broadcasting in the network, because it requires modification of all chain blocks at all 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 system allows to provide security of transferred data and resolve the problem of trust between network participants.

1.2 What is the WE 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.

1.3 Solutions based on the WE blockchain platform

Waves Enterprise has developed following services based on the WE blockchain platform:

- *Waves Enterprise Mainnet* - the basic Waves Enterprise blockchain network.
- Voting - a blockchain service for organization of electronic voting. It allows to conduct fully fraud-protected transparent surveys, corporate polling and plebiscites.
- Data oracles - a service for secure redirecting of data required for business applications and smart contracts from external sources to the blockchain.

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

CHAPTER

TWO

SYSTEM REQUIREMENTS

Hardware and system requirements to a computer for deployment of a new Waves Enterprise node are stated below.

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

Hint: "Xmx" flag which defines the maximal size of the available JVM memory.

Environment requirements for the Waves Enterprise platform

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

CHAPTER

THREE

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 that allows you to test the basic features:

- signing and sending of transactions;
- obtaining of data from the blockchain;
- installation and call of smart contracts;
- transfer of confidential data between nodes;
- testing node monitoring with InfluxDB and Grafana.

Interaction with the platform can be performed both through the client application and through gRPC and REST API interfaces.

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 in it the **docker-compose.yml** file. 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://github.com/waves-enterprise/WE-releases/releases/download/v1.6.0/docker-
↪compose.yml
```

2. Open a 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:v1.6.
↪0
```

Wait for the message 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 file **./credentials.txt**:

```
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-bQ9gw6t0sA
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
```

Message when node and services start successfully:

```
Creating network "platf_we-network" with driver "bridge"
Creating node-2       ... done
Creating postgres     ... done
Creating node-0       ... done
Creating node-1       ... done
Creating auth-service ... done
Creating crawler      ... done
Creating data-service ... done
Creating frontend     ... done
Creating nginx-proxy  ... done
```

After successful launch of containers, the platform client will be available in your browser locally at **127.0.0.1** or **localhost**. The REST API of the node is located at **127.0.0.1/node-0** or **localhost/node-0**.

Note that the **80:80** port is provided for the local platform nginx server by default. If this port is occupied by another application in your system, change the ports parameter of the **nginx-proxy** section in the **docker-compose.yml** file, selecting the available port:

```
nginx-proxy:
  image: nginx:latest
  hostname: nginx-proxy
  container_name: nginx-proxy
```

(continues on next page)

(continued from previous page)

```
  ports:
    - "81:80"
```

After that, the client and the REST API will be available at **127.0.0.1:81** or **localhost:81**.

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

```
docker-compose down
```

3.2 Further actions

3.2.1 Node monitoring configuration in the Sandbox mode

A node, running in the Sandbox mode, has the ability to set up monitoring of its performance using **InfluxDB** and **Grafana**.

To install and configure monitoring, first stop the running containers using the `docker-compose down` command.

1. Install the Grafana Docker image:

```
docker run -d --name=grafana -p 3000:3000 grafana/grafana
```

2. Install the Docker image of InfluxDB:

```
docker run -d --name influxdb -p 8086:8086 -e INFLUXDB_DB=sandbox_influxdb -e␣
↪INFLUXDB_ADMIN_USER=sandbox_influxdb_admin -e INFLUXDB_ADMIN_
↪PASSWORD=sandbox_influxdb_pass quay.io/influxdb/influxdb:v2.0.3
```

Hint: Note that these steps are necessary to pre-install Grafana and InfluxDB. If you intend to use the Docker Compose, you can skip them: after modifying the *docker-compose.yml* file below and starting the containers, the images of both services will be downloaded and installed automatically, if not installed before.

---

3. Go to the `./configs/nodes/node-0` directory and open the node configuration file **node.conf** with administrator rights. At the end of the configuration file, add the following parameters of the monitoring services:

```
# Performance metrics
kamon {
  # Set to "yes", if you want to report metrics
  enable = yes

  # An interval within metrics are aggregated. After it, them will be sent to␣
↪the server
  metric.tick-interval = 1 second

  # Reporter settings
  influxdb {
    hostname = "localhost"
    port = 8086
```

(continues on next page)

(continued from previous page)

```
    database = "sandbox_influxdb"
    time-units = "ms"

    authentication {
      user = "sandbox_influxdb_admin"
      password = "sandbox_influxdb_pass"
    }

    environment.host = "node-0"
  }
}

# Non-aggregated data (information about blocks, transactions, ...)
metrics {
  enable = yes
  node-id = "node-0"

  influx-db {
    uri = "http://localhost:8086"
    db = "sandbox_influxdb"

    username = "sandbox_influxdb_admin"
    password = "sandbox_influxdb_pass"

    batch-actions = 100
    batch-flash-duration = 1s
  }
}
```

4. Open the **docker-compose.yml** file and add the monitoring services deployment parameters.

Full listing of the docker-compose.yml file

5. Launch the platform and enter the Grafana client application. To do this, start the platform containers using the `docker-compose up -d` command and wait for the deployment to complete.

Then open the Grafana client available on port **3000**:

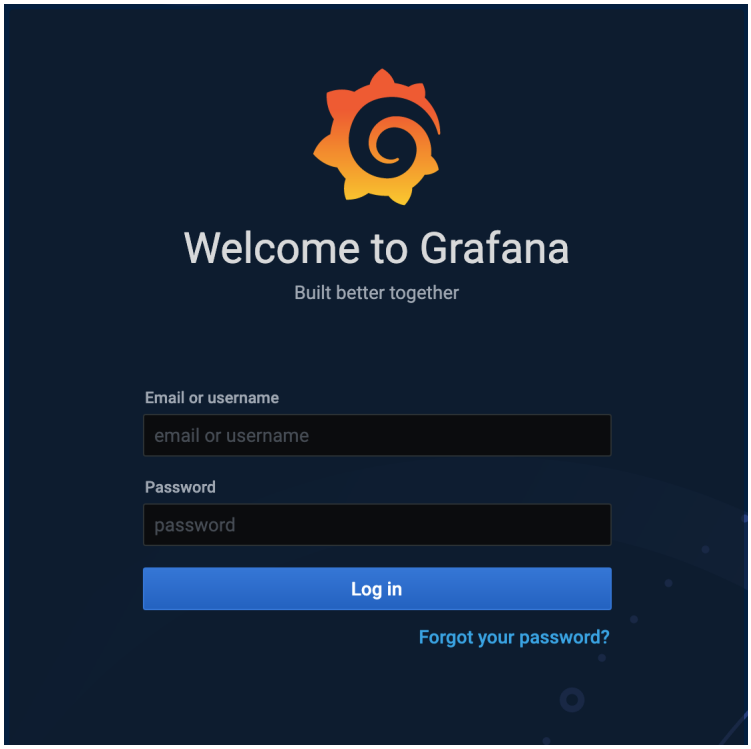
http://127.0.0.1:3000/login

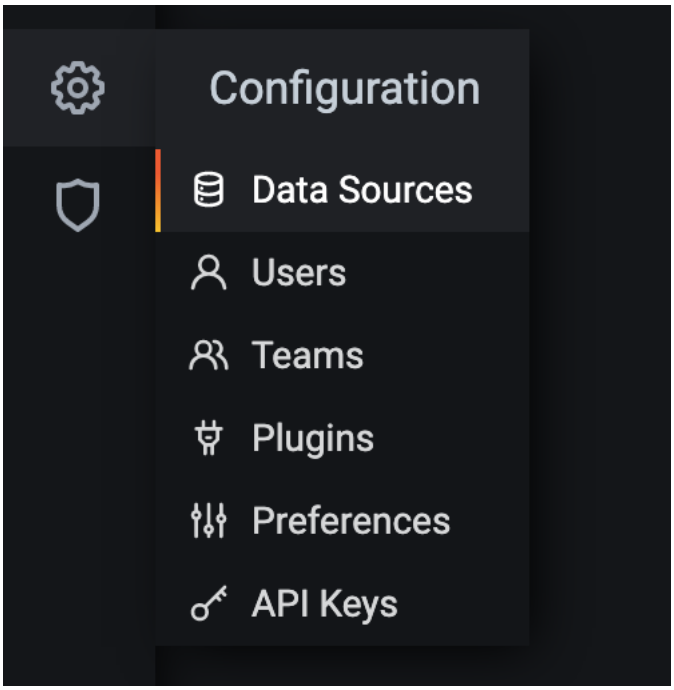
For client authentication, use the data set in the **environment** subcategory of **services: grafana:** section in the **docker-compose.yml** file:

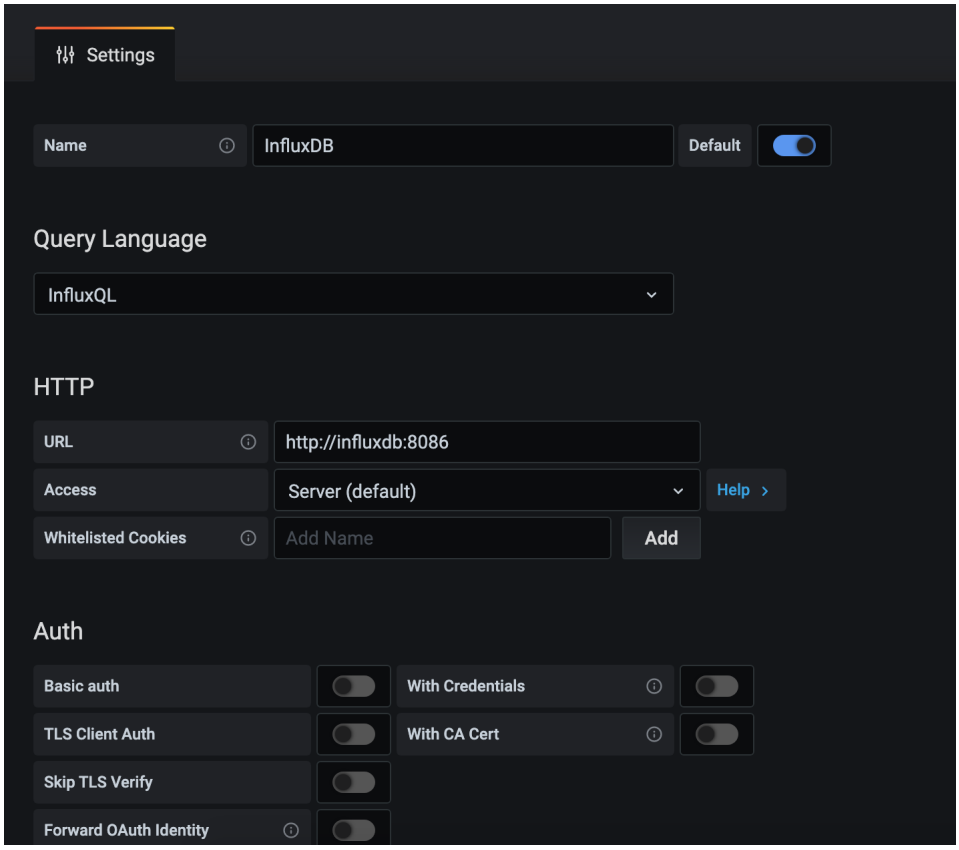
```
services:
  grafana:
    image: grafana/grafana:latest
    hostname: grafana
    container_name: grafana
    environment:
      GF_SECURITY_ADMIN_USER: 'admin'
      GF_SECURITY_ADMIN_PASSWORD: 'pass'
```

6. Connect the platform database to Grafana. To do this, click the **Configuration** tab and go to the **Data Sources** category:

Add a new data source by clicking **Add Data Source** and select **InfluxDB** from the suggested list.


Welcome to Grafana
Built better together
Email or username
email or username
Password
password
Log in
Forgot your password?


Configuration
Data Sources
Users
Teams
Plugins
Preferences
API Keys



Then configure the database used by the platform.

HTTP section, URL field: **http://influxdb:8086**

InfluxDB Details section: enter data set in the **environment** subcategory of the **services: influxdb:** section in the **docker-compose.yml** file:

```
influxdb:
    image: influxdb
    hostname: influxdb
    container_name: influxdb
    environment:
      - INFLUXDB_DB=influxdb // database field
      - INFLUXDB_ADMIN_USER=admin //  user field
      - INFLUXDB_ADMIN_PASSWORD=pass // password field
```

Click **Save & Test**. If the database is successfully connected, the message **Data source is working** will appear.

7. Set up data visualization in Grafana

To do this, click + (Add) on the left panel of the client and click the **Import** tab.

In the **Import via panel json** window, insert the json file containing the data visualization parameters.

Then click **Load**. In the **Options** window, select the desired panel name, or leave the default **sandbox** name.

Click the **Import** button.

You will then be able to view your demo platform metrics on the panel you have created. Go to the **Dashboard**/**Manage** tab and select the **sandbox** panel to open it.

See also

Deploying the platform in the trial mode (Sandbox)

Sandbox mode of the platform: fixing issues

3.2.2 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
```

(continues on next page)

(continued from previous page)

```
↪"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, enter to the localhost is performed 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`.

See also

Deploying the platform in the trial mode (Sandbox)

Node monitoring configuration in the Sandbox mode

See also

Transactions of the blockchain platform

Smart contracts

Confidential data exchange

gRPC tools

REST API methods

CHAPTER

FOUR

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 thr Mainnet, contact the technical support service for advice.

4.1 Generating balance

Please note that you must have at least **50,000 WEST** on the node address or in leasing. This amount constitutes the irreducible **generating balance** required to send transactions and mine. If the amount of tokens on the address becomes lower than the generating balance, the node loses the ability to be selected as a miner and to send transactions until the balance is replenished. Information about the generating balance in the Mainnet is updated once every 1,000 blocks. The node will be able to be selected as a miner and send transactions only after the generating balance is updated.

4.2 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.

4.3 Node deployment

Ознакомьтесь с *системными требованиями* к блокчейн-платформе.

After successful transfer of tokens, deploy the node:

1. Create a working directory and place in it the **docker-compose.yml** file. 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://github.com/waves-enterprise/WE-releases/releases/download/v1.6.0/docker-
↪compose.yml
```

2. Download the file `mainnet.conf` file from the official GitHub repository of Waves Enterprise, 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:v1.
↪6.0
```

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

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

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.

4.5 Further actions

4.5.1 Node update in the Mainnet

With each new release of the platform, we recommend that you update the nodes connected to Mainnet. All users whose nodes are running on Mainnet receive an email notifying them that their node version has been updated. 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 by 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 message `Migration finished sucessfully` and the node will continue to run.

Attention: If you are not using Docker Compose, contact the technical support team for instructions on how to update the node.

See also

Deploying a platform with connection to Mainnet

Mainnet: fixing issues

Waves Enterprise Mainnet fees

4.5.2 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-
↪0 ... error
tainer to be able to reuse that name.
```

(continues on next page)

(continued from previous page)

```
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

Waves Enterprise Mainnet fees

Generators

CHAPTER

FIVE

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. Our experts will help you configure the delivery of the platform 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.

5.1 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.

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

After that, you will be sent the YAML file for Kubernetes needed for the initial deployment of the platform and instructions on how to deploy and run the platform for your hardware configuration.

Before deployment, familiarize yourself with the blockchain platform *system requirements*.

5.2 Creation of a node account

After obtaining the license and initial deployment of the platform, you will need to create accounts for each node of your future network.

The generation of keys is performed with the AccountsGeneratorApp utility, which is included in the package *generators*. You can download this package from the official repository of Waves Enterprise on GitHub by selecting the platform version you use.

A node account includes an address and a key pair: public and private keys. The address and the public key will be shown on the command line during account creation using the **generators** 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 configuration file `accounts.conf` is used, which contains the parameters of account generation. This file is located in the directory of each node.

To create a node account, go to its directory and place in it the downloaded file **generators.jar**. Then run it by entering the file `accounts.conf` as an argument:

```
java -jar generators-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.

5.3 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 choosing authorization method by `api-key` hash. The principles of working with this configuration file will be discussed when configuring the authorization method of the 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 consensus, Docker smart contract execution and mining. All the parameters required for this are located in the **node.conf** file.

Installation and usage of the platform

5.3.1 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 article *Consensus algorithms*.

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

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

Select the preferred consensus type in the `type` field. Possible 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"

    ...
    }
```

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 are identical to those of the PoA consensus:

```
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 two 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` - опциональный параметр, определяющий, сколько времени после окончания раунда (параметр конфигурационного файла ноды: `round-duration`) майнер ожидает полный набор голосов от всех валидаторов.

При настройке CFT обратите внимание на следующие рекомендации:

- Параметр `sync-duration` должен быть отличен от нуля. Рекомендуется устанавливать значение **от 1 до 5 секунд**, в зависимости от размера и сложности транзакций.
- Примерный расчет значения параметра `finalization-timeout`: (`round-duration` + `sync-duration`) / 2. Не рекомендуется занижать это значение для ускорения финализации: если майнер наберет необходимое число голосов ранее окончания этого времени, он сразу выпустит финализирующий микроблок.
- Если в сети присутствует большое количество майнеров, ограничьте количество валидаторов раунда параметром `max-validators`. Механизм выбора валидаторов обеспечит равномерную ротацию всех валидаторов по раундам. Слишком большое количество валидаторов может отрицательно повлиять на производительность сети. Рекомендуемый диапазон значений: **от 5 до 10**.
- Если сеть работает под постоянной нагрузкой, установите параметр `full-vote-set-timeout`. До истечения этого периода времени майнер ждет полного набора голосов от валидаторов. Если валидатор сталкивается с какими-либо неполадками, сеть использует время `full-vote-set-timeout` для создания дополнительного временного промежутка, который позволяет отставшему валидатору завершить синхронизацию. Рекомендуемое значение: `sync-duration` * 2, не может привышать `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

5.3.2 General platform configuration: execution of smart contracts

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

```
docker-engine {
  enable = yes
  integration-tests-mode-enable = no
  # 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
  }
  contract-auth-expires-in = 1m
  grpc-server {
    # host = "192.168.97.3"
    port = 6865
    }
  }
  remove-container-on-fail = yes
}
```

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

- `integration-tests-mode-enable` - Docker contracts testing mode. When this option is enabled, smart contracts are executed locally in the container.
- `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 taken to execute the contract (in seconds).
- `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 of the cache with the execution status of transactions on Docker contracts;
- `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.
- `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`.

See also

Deployment of the platform in a private network

Development and usage of smart contracts

General platform configuration: consensus algorithm

General platform configuration: mining

Installation and usage of the platform

5.3.3 General platform configuration: mining

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

```
miner {
  enable = yes
  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
}
```

- `enable` - activation of the mining option. Enable - `yes`, disable - `no`.
- `quorum` - required number of node miners to create a block. The 0 value will allow to 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 the given period of time (in days).
- `micro-block-interval` - an interval between microblocks (in seconds).
- `min-micro-block-age` - a minimal age of the microblock (in seconds).
- `max-transactions-in-micro-block` - a maximum number of transactions in the microblock.
- `minimal-block-generation-offset` - a minimal time interval between blocks (in milliseconds).

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

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` - minimum age of 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.

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, TLS, and confidential data access groups. 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.

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

5.3.4 Precise platform configuration: gRPC and REST API authorization

Authorization is necessary to provide access to the gRPC and REST API tools of a node. For this purpose, the Waves Enterprise blockchain platform supports two types of authorization:

- `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 intruder can gain access to a node if the string `api-key` reaches him. If you want to improve security of your network, we recommend using JWT token authentication via an authorization service.

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

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 section `auth` 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"
}
```

- `api-key-hash` - hash from the REST API access key string.
- `privacy-api-key-hash` - hash from the key string to access **privacy** methods.

To fill these parameters you will need the `ApiKeyHash` utility from the package `generators-x.x.x.jar`, which you can download from the official repository of Waves Enterprise 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 {
  waves-crypto = yes
  api-key = "some string for api-key"
}
```

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

Enter the finished file `api-key-hash.conf` as an argument when you run the `ApiKeyHash` utility of the `generators` package:

```
java -jar generators-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` and `privacy-api-key-hash` parameters in the `auth` section of the node configuration file as indicated above.

type = "oauth2"

When selecting authorization by JWT-token, the `auth` section of the node configuration file looks like this:

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

The public key for oAuth is generated during the initial deployment of the node. It is located in the file `./auth-service-keys/jwtRS256.key.pub`. Copy the line between `-----BEGIN PUBLIC KEY-----` and `-----END PUBLIC KEY-----` and paste it as the `public-key` parameter of the `auth` section of the node configuration file.

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

See also

Deployment of the platform in a private network

Precise platform configuration: node gRPC and REST API configuration

Precise platform configuration: node gRPC and REST API configuration

Precise platform configuration: TLS

5.3.5 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
  }
```

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` - port for listening REST API requests.
- `tls` - enable/disable TLS for REST API requests. Enable - `yes`, disable - `no`.
- `cors` - support of cross-domain requests to REST API. Enable - `yes`, disable - `no`.

- `transactions-by-address-limit` - maximum number of transactions returned by `GET /transactions/address/{address}/limit/{limit}` method.
- `distribution-address-limit` - maximum number of addresses specified in the limit field and returned by `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 gRPC interface on the node.
- `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/disable TLS for gRPC requests. This option requires *setting of the node TLS*.

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

5.3.6 Precise platform configuration: TLS

In order to work with the node TLS, apart its configuration in the node config file, a user should get a keystore file itself with the use of the **keytool** utility:

```
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 config 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 keystore file with the filename **we.jks** will be obtained. In order to connect with 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 the node is located in one 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, a `we.cert` certificate file should be imported into the keystore:

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

Then also specify the relative path to `we.jks` in the `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 status. Possible options: `DISABLED` (disabled, in this case other options should be excluded or commented) 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

Precise platform configuration: gRPC and REST API authorization

Precise platform configuration: node gRPC and REST API configuration

5.3.7 Precise platform configuration: node gRPC and REST API configuration

If you use **privacy** API methods to manage confidential data, configure the access to confidential data for which the `privacy` section of the node configuration file is intended (example using the PostgreSQL database and enabling periodic deletion of files that are not in the blockchain):

```
privacy {

storage {
  vendor = postgres
  schema = "public"
  migration-dir = "db/migration"
```

(continues on next page)

(continued from previous page)

```
  profile = "slick.jdbc.PostgresProfile$"
  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
}

cleaner {
  enabled: yes
  interval: 10m
  confirmation-blocks: 100
  pending-time: 72h
}
}
```

Before changing it, 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.

If using PostgreSQL DBMS, you will need to install the JDBC interface. When using Amazon S3, the information must be stored on the Minio server.

After installing the appropriate DBMS for your project, proceed to configuring the block of the `privacy` section. Specify the DBMS you use in the `vendor` parameter:

- `postgres` - for PostgreSQL;
- `s3` - for Amazon S3.

If you do not use **privacy** API methods, specify `none` and comment out or delete the rest of the parameters.

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$"
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
```

(continues on next page)

(continued from previous page)

```
    numThreads = 20
    }
}
```

- `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` - name of the profile for JDBC access.
- `url` - address of the PostgreSQL database.
- `driver` - name of the JDBC (Java SataBase Connectivity) driver that allows Java applications to communicate with the database.
- `user` - user name to access the database.
- `password` - password to access the database.
- `connectionPool` - name of the connection pool, `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` - number of simultaneous connections to the database.

During the installation of the database running PostgreSQL, create an account to access the database. Then enter the login and password you specified in the `user` and `password` fields. When installing JDBC, set the profile name, which you then specify in the `profile` field.

In the `url` field, specify the address of the database you are using in the following format:

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

- `POSTGRES_ADDRESS` - PostgreSQL host address.
- `POSTGRES_PORT` - PostgreSQL host port number.
- `POSTGRES_DB` - name of the PostgreSQL database.

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 command `sslmode=require` to require a password when authorizing.

vendor = s3

When using 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
}
```

- `url` - address of the Minio server to store data. By default, Minio uses the port 9000.
- `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.
- `secret-access-key` - data access key in the S3 repository.
- `path-style-access-enabled = true` - unchangeable parameter to specify the path to S3 table.
- `connection-timeout` - period of inactivity before the connection is broken (in seconds).
- `connection-acquisition-timeout` - period of inactivity during connection establishment (in seconds).
- `max-concurrency` - number of concurrent accesses to the storage.
- `read-timeout` - period of inactivity when reading data (in seconds).

During installation of the Minio server, you will be prompted for a login and password to access the data. Enter your username in the `access-key-id` field and your password in the `secret-access-key` field.

cleaner section

The `cleaner` section is designed to configure the periodic deletion of confidential data that is stored in the database, but for one reason or another did not get into the blockchain (for example, in case of transaction rollback). This section includes the following parameters:

- `enabled` - enable/disable periodic deletion of files that did not hit the blockchain.
- `interval` - interval for cleaning the files.
- `confirmation-blocks` - the period of time in blocks during which the hash data transaction exists in the blockchain, and after which it is deleted.
- `pending-time` - the maximum period of time for which a file with data is saved without hitting the blockchain.

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

5.3.8 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
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 aJWT-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: node gRPC and REST API configuration

Precise platform configuration: TLS

5.3.9 Precise platform configuration: snapshot

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

```
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

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

5.4 Genesis block signing and starting the network

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 **generators** package. It uses the node configuration file `node.conf` that you set up as an argument:

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

As a result of the utility's work, the fields `genesis-public-key-base-58` and `signature` located in the `genesis` block of the `blockchain` section of the node configuration file will be filled 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"
```

After signing the genesis block, the platform is fully configured and ready to run the network. You can launch it according to the instructions received from Waves Enterprise specialists.

5.5 Attachment of the client application to the private network

Once the network is up and running, attach a 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 value `http://<computer network address>/<node address>`;
- Type of authorization on the node - select the type of authorization you configured earlier: by JWT-token or by `api-key`;
- Blockchain address - specify the address of your node;

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

Client description is provided in the article *Client*.

See also

Examples of node configuration files

Generators

Installation and usage of the platform

CHAPTER

SIX

EXAMPLES OF NODE CONFIGURATION FILES

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 the 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 storage of confidential data;
- the function of periodic deletion of sensitive data that is not on the blockchain is disabled.

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
waves-crypto = yes

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

# NTP settings
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
```

(continues on next page)

(continued from previous page)

```
# 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)
```

(continues on next page)

(continued from previous page)

```
          # - 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}
```

(continues on next page)

(continued from previous page)

```
  # 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

  # 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
}

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

(continues on next page)

(continued from previous page)

```
    # 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 {
  storage {
    vendor = postgres

    # for postgres vendor:
                    schema = "public"
                    migration-dir = "db/migration"
                    profile = "slick.jdbc.PostgresProfile$"
                    jdbc-config {
                      url = "jdbc:postgresql://postgres:5432/node-1"
                      driver = "org.postgresql.Driver"
                user = postgres
                      password = wenterprise
```

(continues on next page)

(continued from previous page)

```
                    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
  }

  cleaner {
    enabled: no

    # The amount of time between cleanups
    # interval: 10m

    # How many blocks the data hash transaction exists on the blockchain, after which it␣
↪will be removed from cleaner monitoring
    # confirmation-blocks: 100

    # The maximum amount of time that a file can be stored without getting into the␣
↪blockchain
    # pending-time: 72h
  }
}


# 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"
   #}
```

(continues on next page)

(continued from previous page)

```
  # 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
  }
}
}
```

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 {
  waves-crypto = yes
  chain-id = V
  amount = 1
  wallet = ${user.home}"/node/keystore.dat"
  wallet-password = "/FILL/"
  reload-node-wallet {
    enabled = false
    url = "http://localhost:6862/utils/reload-wallet"
  }
}
```

6.3 api-key-hash.conf

In this example, Waves Crypto encryption is enabled.

api-key-hash.conf:

```
apikeyhash-generator {
  waves-crypto = yes
  api-key = "some string for api-key"
}
```

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

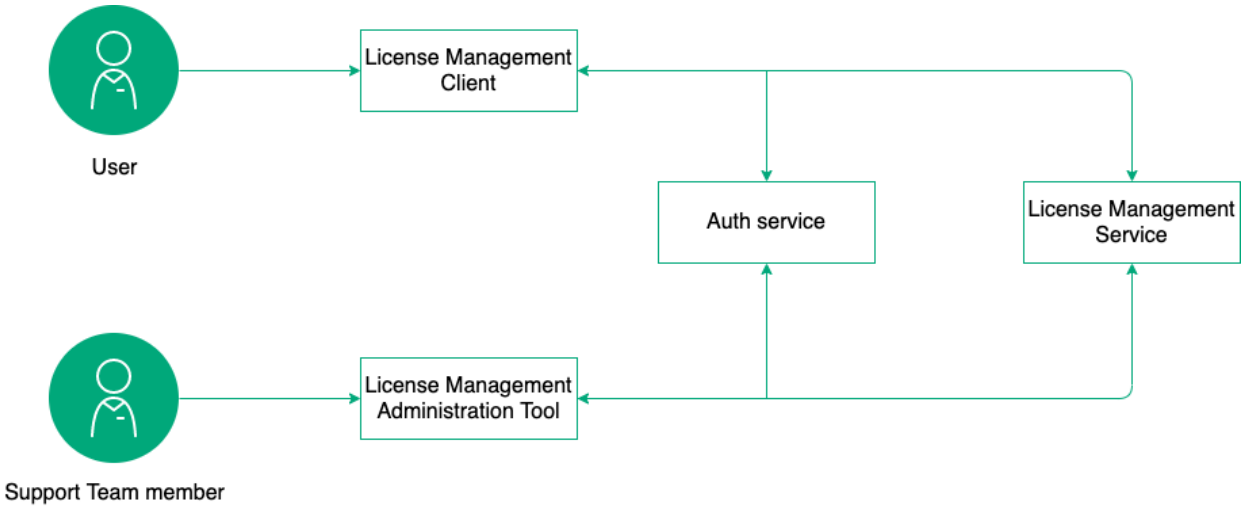
Generators

CHAPTER

SEVEN

LICENSES OF THE WAVES ENTERPRISE BLOCKCHAIN PLATFORM

The Waves Enterprise blockchain platform is a commercial product oriented for use in the corporate and government sectors and distributed through user licenses. The scheme for obtaining a license to use the platform 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

7.1 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 :ref:` Deploying the platform in the trial mode (Sandbox) <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 platforms in the implementation of projects not aimed at generating profit. It 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 Mainnet, there are *fees* for the transactions performed.

Each type of license applies to one node.

7.2 Duration of licenses

Licenses also differ in terms of duration, which is negotiated when the contract is concluded:

- 3 months is the standard validity period of a trial license.
- Lease for the duration of the use of the platform is determined by the time frame of the project implemented using the blockchain platform.
- 1 year.
- 2 years.
- Unlimited license.

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

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.

See also

Waves Enterprise Mainnet fees

CHAPTER

EIGHT

WAVES ENTERPRISE MAINNET FEES

Transaction number	Transaction 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	fee amount depends on number of addresses in a transaction
12	*Data Transaction*	0.05 WEST per kilobyte	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	fee amount depends on data size
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	

See also

licenses

Deploying a platform with connection to Mainnet

CHAPTER

NINE

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 and describes the data types used.

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

9.1 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**.

9.2 What the gRPC interface is for

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

9.2.1 gRPC: monitoring of blockchain events

The gRPC interface has 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_subscribe_on_request.proto` - a file that contains fields with request parameters;
- `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 a query `SubscribeOn(startFrom, transactionTypeFilter)` that initializes a subscription to the selected event group.

Query parameters:

startFrom - the moment when the event tracking starts:

- `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.

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

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 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` - fee amount for transactions within the block;
   - `block_size` - size of a block 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 transaction IDs 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` - fee amount for transactions within the block;
- `block_size` - size of a block 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` - authorization data were not 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

9.2.2 gRPC: obtaining node configuration parameters

To get the node configuration parameters, the method and request `NodeConfig` is used. This method is packed in the **util_node_info_service.proto** protobuf file.

The `NodeConfig` query does not require any additional 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` - a maximum number of transactions in a microblock;
- `min_micro_block_age` - minimum time of microblock existence (in seconds);
- `micro-block-interval` - interval between microblocks (in seconds);
- `pos_round info`: when using the PoS consensus algorithm, the `average_block_delay` parameter is displayed (the average block creation delay time, in seconds);
- `poa_round_info`: when using the PoA consensus algorithm, the parameters `round_duration` (block mining round length, in seconds) and `sync_duration` (block mining synchronization period, in seconds) are displayed.

See also

gRPC tools

9.2.3 gRPC: information about transaction according to their IDs

To get information about a transaction by its ID, two requests packaged in the **contract_transaction_service.proto** file can be sent to the gRPC interface of the node:

- `TransactionExists` - request for the existence of a transaction with the specified ID;
- `TransactionInfo` - request information about the transaction with the specified ID.

Both requests require entering the **tx_id** parameter - the ID of the transaction for which information is requested.

The response to the `TransactionExists` query:

- `exists` - Boolean data type: `true` - exists, `false` - does not exist.

The response to the ``TransactionInfo" query contains the following information about the transaction:

- `height` - the height of the blockchain at which the transaction has been sent;
- `transaction` - name of the transaction.

See also

gRPC tools

9.2.4 gRPC: obtaining information about smart contract state

To get node configuration parameters, there is a query `ContractExecutionStatuses`. The fields of this query are contained in the protobuf file **util_contract_status_service.proto**.

The `ContractExecutionStatuses` query requires the **tx_id** parameter - the transaction ID of the smart contract call whose status information is to be retrieved.

Information about smart contract state

The response to the `ContractExecutionStatuses` query outputs the following smart contract data:

- `sender` - participant who sent the smart contract to the blockchain;
- `tx_id` - smart contract call transaction ID;
- `Status` - information about execution of the smart contract: `0` - successfully executed (SUCCESS); `1` - executed with an error (ERROR); `2` - not executed (FAILURE);
- `code` - code of an error if occured during the execution of the smart contract;
- `message` - error message;
- `timestamp` - time of the smart contract call;
- `signature` - smart contract signature.

See also

gRPC tools

9.2.5 gRPC: obtaining information about UTX pool size

The query about the UTX pool size `UtxInfo` is sent as a subscription: after sending it, the response from the node comes once every 1 second. This request requires no additional parameters and is located 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.

See also

gRPC tools

9.2.6 gRPC: generation and checking of data digital signatures (PKI)

For networks using GOST cryptography, the gRPC interface has the ability to form a disconnected digital signature for transmitted data, as well as to verify it. For this purpose, two methods packed in the protobuf file **contract_pki_service.proto** are provided:

- **Sign** - generating a disconnected DS for the data transmitted in the request.
- **Verify** - verifying a disconnected DS for the data transmitted in the request.

Generating a disconnected digital signature

The **Sign** method requires following parameters:

- `input_data` - data for which an DS is required (as an array of bytes in **base64** encoding);
- `keystore_alias` - name of the storage for the DS private key;
- `password` - a password for the private key storage;
- `sig_type` - DS format. Supported formats: `1` - CAdES-BES; `2` - CAdES-X Long Type 1; `3` - CAdES-T.

The method response contains the `signature` field with generated digital signature in **base64** format.

Verifying a disconnected digital signature

The **Verify** method requires following parameters:

- `input_data` - data signed by an DS (as an array of bytes in **base64** encoding);
- `signature` - digital signature in the form of an array of bytes in **base64** encoding;
- `sig_type` - DS format. Supported formats: `1` - CAdES-BES; `2` - CAdES-X Long Type 1; `3` - CAdES-T.
- `extended_key_usage_list` - list of object identifiers (OIDs) of cryptographic algorithms that are used in DS generation (*optional field*).

Method's response contains a `status` field with boolean data type: `true` - signature is valid, `false` - signature is compromised.

Verifying an advanced qualified digital signature

The **verify** method has the ability to verify an advanced qualified digital signature. To verify the AQDS correctly, install the root AQDS certificate of the certification authority (CA) on your node, which will be used to validate the signature.

The root certificate is installed in the **cacerts** certificate storage of the Java virtual machine (JVM) you are using using the **keytool** utility:

```
sudo keytool -import -alias certificate_alias -keystore path_to_your_JVM/lib/security/
↪cacerts -file path_to_the_certificate/cert.cer
```

After the `-alias` flag, specify your preferred certificate name in the repository.

The cacerts certificate storage is located in the `/lib/security/` subdirectory of your Java virtual machine. To find out the path to the virtual machine on Linux, use the following command:

```
readlink -f /usr/bin/java | sed "s:bin/java::"
```

Then add `/lib/security/cacerts` to the resulting path and paste the resulting absolute path to **cacerts** after the `-keystore` flag.

After the `-file` flag, specify the absolute or relative path to the received EDS certificate of the Certification Authority.

The default password for **cacerts** is `changeit`. If necessary, you can change it using the **keytool** utility:

```
sudo keytool -keystore cacerts -storepasswd
```

See also

gRPC tools

9.2.7 gRPC: encryption and decryption methods

The gRPC interface of the node provides the ability to encrypt arbitrary data using the encryption algorithms of the Waves Enterprise blockchain platform, as well as to decrypt them. For this purpose, a set of requests packed in the **contract_crypto_service.proto** file is provided:

- `EncryptSeparate` - encryption of data with unique CEK keys separately for each recipient, each CEK is encrypted (*wrapped*) with a separate KEK key;
- `EncryptCommon` - data encryption with a single CEK key for all recipients, each CEK key is encrypted (*wrapped*) with a separate KEK key for each recipient;
- `Decrypt` - decrypt data.

Hint: Decryption of data is possible if the recipient's key is in the keystore of the node.

Encryption queries and responses

The `EncryptSeparate` and `EncryptCommon` queries require the following data:

- `sender` - an address of data sender;
- `password` - password to the encrypted data;
- `encryption_data` - data to be encrypted (as an array of bytes in **base64** encoding);
- `recipients_public_keys` - public keys of the recipients participating in the network;
- `crypto_algo` - cryptographic algorithm in use. Available values: `1` - GOST 28147-89; `2` - GOST 34.12-2015; `3` - AES.

The response to the `EncryptSeparate` request includes the following data for each recipient:

- `encrypted_data` - encrypted data;
- `public_key` - recipient public key;

- `wrapped_key` - result of key encryption for a recipient.

In response to the `EncryptCommon` query the following data is received:

- `encrypted_data` - encrypted data;
- `recipient_to_wrapped_structure` - a structure in the "key : value" format containing the public keys of the recipients with the corresponding key encryption results for each of them.

Decription query and response

When `Decrypt` is requested, the following data is entered:

- `recipient` - recipient's public key from the node keystore;
- `password` - password to the encrypted data;
- `encrypted_data` - encrypted data;
- `wrapped_key` - result of key encryption for a recipient;
- `sender_public_key` - the public key of the data sender;
- `crypto_algo` - cryptographic algorithm in use. Available values: `1` - GOST 28147-89; `2` - GOST 34.12-2015; `3` - AES.

In response to the `Decrypt` query, the `decrypted_data` field is received, containing the decrypted data in the form of an array of bytes in **base64** encoding.

See also

gRPC tools

9.2.8 gRPC: sending transactions into the blockchain

The gRPC interface supports the ability to send transactions to the blockchain by sending a `Broadcast` request packaged in the **transaction_public_service.proto** file.

The **Broadcast** method requires following parameters:

- `version` - transaction version;
- `transaction` - 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 article *Transactions of the blockchain platform*.

See also

gRPC tools

Description of transactions

Waves Enterprise Mainnet fees

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 above.

gRPC methods of the node, as opposed to REST API methods, do not require authorization. Also all the methods packaged in **protobuf** files that are placed in the **contract** directory are available for both node and smart contracts. When used in smart contracts, these methods require authorization.

See also

gRPC services used by smart contracts

CHAPTER

TEN

REST API METHODS

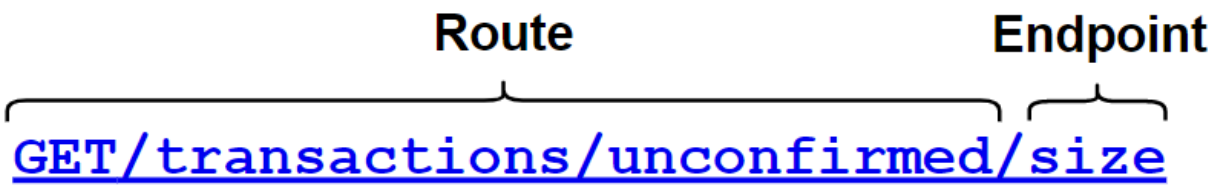
Attention: The Waves Enterprise blockchain platform is phasing out the REST API methods. No REST API methods will be developed in new versions of the platform.

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.

10.1 REST API usage

All REST API method calls are GET, POST or DELETE https queries to the URL `https://yournetwork.com/node-N/api-docs/swagger.json`, containing appropriate parameter sets. The desired groups of queries 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 method reference.

Example of a UTX pool size query:



Для использования практически всех методов REST API требуется авторизация по `api-key` или JWT-токену.

When authorizing by `api-key`, specify the value of the selected passphrase, and when authorizing by JWT-token - the value of **access**-token.

At the same time, for methods related to access to the node, only authorization by `api-key` is provided:

- access to the keystore of a node (e.g. the sign method);
- working with private data access groups;
- access to the node configuration.

If authorization by JWT-token is used, access to these methods will be denied.

10.2 What the platform REST API is for

You can use the REST API to perform the following tasks:

10.2.1 REST API: transactions

Methods of the `transactions` group are provided to work with transactions.

Signing and sending transactions

The node REST API uses a JSON representation of a transaction to send requests.

The basic principles of work with transactions are given in the *Transactions of the blockchain platform*. A description of the fields for each transaction is given in the *Description of transactions* section.

The **POST /transactions/sign** method is used to sign transactions. This method signs the transaction with the sender's private key stored in the keystore of the node. To sign requests with the key from the keystore of the node, be sure to specify the key pair password in the `password` field.

Example signature request *transaction 3*:

POST /transactions/sign:

```
{
   "type": 3,
   "version": 2,
   "name": "Test Asset 1",
   "quantity": 100000000000,
   "description": "Some description",
   "sender": "3FSCKyfFo3566zwiJjSFLBwKvd826KXUaqR",
   "decimals": 8,
   "reissuable": true,
   "password": "1234",
   "fee": 100000000
}
```

The **POST /transactions/sign** method in the response returns the fields needed to publish the transaction.

Example response with *transaction 3*:

POST /transactions/sign:

```
{
    "type": 3,
    "id": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "sender": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
    "senderPublicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
    "fee": 100000000,
    "timestamp": 1549378509516,
```

(continues on next page)

(continued from previous page)


```
    "proofs": [
↪"NqZGcbcQ82FZrPh6aCEjuo9nNnkPTvyhrNq329YWydaYcZTywXUwDxFAknTMEGuFrEndCjXBtrueLWaqbJhpeiG
↪" ],
    "version": 2,
    "assetId": "DnK5Xfi2wXUJx9BjK9X6ZpFdTLdq2GtWH9pWrcxcmrhB",
    "name": "Test Asset 1",
    "quantity": 10000,
    "reissuable": true,
    "decimals": 8,
    "description": "Some description",
    "chainId": 84,
    "script": "base64:AQa3b8tH",
    "height": 60719
}
```


The **POST /transactions/broadcast** method is designed to broadcast a transaction. The response fields of the **sign** method are input to this method. A transaction can also be sent to the blockchain using other tools given in the article *Transactions of the blockchain platform*.

In addition to separate methods for signing and sending transactions, there is a combined method **POST /transactions/signAndBroadcast**. This method signs and sends the transaction to the blockchain without intermediate transfer of information between the methods.

Example request and response of the method with *transaction 112*:

POST /transactions/signAndBroadcast:

Query:


```
{
"sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
"policyName": "Policy for sponsored v1",
"password":"sfgKYBFCF@#$fsdf()*%",
"recipients": [
  "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
  "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
  "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
  "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
],
"fee": 100000000,
"description": "Privacy for sponsored",
"owners": [
  "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
  "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
],
"type": 112
}
```


Response:

```
{
  "senderPublicKey": "3X6Qb6p96dY4drVt3x4XyHKCRvree4QDqNZyDWHzjJ79",
  "policyName": "Policy for sponsored v1",
  "fee": 100000000,
  "description": "Privacy for sponsored",
  "owners": [
    "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
    "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T"
  ],
  "type": 112,
  "version": 2,
  "sender": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "feeAssetId": "G16FvJk9vabwxjQswh9CQAhbZzn3QrwqWjwnZB3qNVox",
  "proofs": [

↪"3vDVjp6UJeN9ahtNcQWt5WDVqC9KqdEsrr9HTToHfoXFd1HtVwnUPPtJKM8tAsCtby81XYQReLj33hLEZ8qbGA3V
↪"
  ],
  "recipients": [
    "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
    "3NotQaBygbSvYZW4ftJ2ZwLXex4rTHY1Qzn",
    "3Nm84ERiJqKfuqSYxzMAhaJXdj2ugA7Ve7T",
    "3NtNJV44wyxRXv2jyW3yXLxjJxvY1vR88TF",
    "3NxAooHUoLsAQvxBSqjE91WK3LwWGjiiCxx"
  ],
  "id": "EyymzQcM2LrsgGDFFxeGn8DhahJbFYmorcBrEh8phv5S",
  "timestamp": 1585307711344
}
```


Information about transactions

The `transactions` group also includes the following methods of obtaining information about transactions in the blockchain:

GET /transactions/info/–id″

Obtaining information about a transaction by its `{id}` identifier. The transaction identifier is specified in the **POST /transactions/sign** or **POST /transactions/signAndBroadcast** methods response.

The method returns transaction data similar to the **POST /transactions/broadcast** and **POST /transactions/signAndBroadcast** methods responses.

Response example:

POST /transactions/signAndBroadcast:


```
{
  "type": 4,
  "id": "52GG9U2e6foYRKp5vAzsTQ86aDAABfRJ7synz7ohBp19",
  "sender": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "senderPublicKey": "CRxqEuxhdZBEHX42MU4FfyJxuHmbDBTaHMhM3Uki7pLw",
  "recipient": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "assetId": "E9yZC4cVhCDfbjFJCc9CqkAtkoFy5KaCe64iaxHM2adG",
  "amount": 100000,
  "fee": 100000,
  "timestamp": 1549365736923,
  "attachment": "string",
  "signature":
↪"GkkccUA79dBcwWgKjqB7vYHcnsj7caYETfncJhRkkaetbQon7DxbpMmvK9LYqUkirJp17geBJCRTNkHEoAjtsUm
↪",
  "height": 7782
}
```


GET /transactions/address/–address"/limit/–limit"

The method returns the data of the last `{limit}` transactions of the address `{address}`.

For each transaction, data similar to the **POST /transactions/broadcast** and **POST /transactions/signAndBroadcast** methods responses are returned.

Response example with one transaction:

GET /transactions/address/–address"/limit/–limit":


```
[
[
  {
    "type": 2,
    "id":
↪"4XE4M9eSoVWVdHwDYXqZsXhEc4q8PH9mDMUBegCSBBVHJyP2Yb1ZoGi59c1Qzq2TowLmymLNkFQjWp95CdddnyBW
↪",
    "fee": 100000,
    "timestamp": 1549365736923,
    "signature":
↪"4XE4M9eSoVWVdHwDYXqZsXhEc4q8PH9mDMUBegCSBBVHJyP2Yb1ZoGi59c1Qzq2TowLmymLNkFQjWp95CdddnyBW
↪",
    "sender": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
    "senderPublicKey": "CRxqEuxhdZBEHX42MU4FfyJxuHmbDBTaHMhM3Uki7pLw",
    "recipient": "3N9iRMou3pgmyPbFZn5QZQvBTQBkL2fR6R1",
    "amount": 1000000000
  }
]
]
```

GET /transactions/unconfirmed

The method returns the data of all transactions from the UTX pool of the node.

For each transaction, data similar to the **POST /transactions/broadcast** and **POST /transactions/signAndBroadcast** methods responses are returned.

Response example with one transaction:

GET /transactions/unconfirmed:


```
[
  {
    "type": 4,
    "id": "52GG9U2e6foYRKp5vAzsTQ86aDAABfRJ7synz7ohBp19",
    "sender": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
    "senderPublicKey": "CRxqEuxhdZBEHX42MU4FfyJxuHmbDBTaHMhM3Uki7pLw",
    "recipient": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
    "assetId": "E9yZC4cVhCDfbjFJCc9CqkAtkoFy5KaCe64iaxHM2adG",
    "amount": 100000,
    "fee": 100000,
    "timestamp": 1549365736923,
    "attachment": "string",
    "signature":
↪"GkkccUA79dBcwWgKjqB7vYHcnsj7caYETfncJhRkkaetbQon7DxbpMmvK9LYqUkirJp17geBJCRTNkHEoAjtsUm
↪"
  }
]
```


GET /transactions/unconfirmed/size

The method returns the number of transactions in the UTX pool as a number.

Response example:

GET /unconfirmed/info/–id″:


```
{
  "size": 4
}
```

GET /unconfirmed/info/–id″

The method returns the data of the transaction that is in the UTX pool by its `{id}`.

The method response contains transaction data similar to the **POST /transactions/broadcast** and **POST /transactions/signAndBroadcast** methods' responses.

Response example:

GET /unconfirmed/info/–id″:


```
{
  "type": 4,
  "id": "52GG9U2e6foYRKp5vAzsTQ86aDAABfRJ7synz7ohBp19",
  "sender": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "senderPublicKey": "CRxqEuxhdZBEHX42MU4FfyJxuHmbDBTaHMhM3Uki7pLw",
  "recipient": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "assetId": "E9yZC4cVhCDfbjFJCc9CqkAtkoFy5KaCe64iaxHM2adG",
  "amount": 100000,
  "fee": 100000,
  "timestamp": 1549365736923,
  "attachment": "string",
  "signature":
↪"GknccUA79dBcwWgKjqB7vYHcnsj7caYETfncJhRkkaetbQon7DxbpMmvK9LYqUkirJp17geBJCRTNkHEoAjtsUm
↪",
  "height": 7782
}
```


POST /transactions/calculateFee

The method returns the amount of commission for the sent transaction.

The request specifies parameters similar to **POST /transactions/broadcast** request. The method's response returns the identifier of the asset where the commission is charged (`null` for WAVES).

Response example:

GET /unconfirmed/info/–id″:


```
{
  "feeAssetId": null,
  "feeAmount": 10000
}
```

See also

REST API methods

Transactions of the blockchain platform

Description of transactions

10.2.2 REST API: generation and checking of data digital signatures (PKI)

A group of `pki` methods is provided for generating and verifying digital signatures.

The principle of `POST /pki/sign` and `POST /pki/verify` methods is similar to the *gRPC methods contract_pki_service.proto*.

All methods of the group are available only for networks with GOST cryptography.

GET /pki/keystoreAliases

The method returns a list with the names of all available signature private key storages.

Response example:

GET /pki/keystoreAliases:

```
{
   [
    "3Mq9crNkTFf8oRPyisgtf4TjBvZxo4BL2ax",
    "e19a135e-11f7-4f0c-9109-a3d1c09812e3"
   ]
 }
```

POST /pki/sign

The method generates a disconnected DS for the data transmitted in the request. The request consists of the following fields:

- `inputData` - data for which an DS is required (as an array of bytes in **base64** encoding);
- `keystoreAlias` - name of the storage for the DS private key;
- `password` - a password for the private key storage;
- `sigType` - DS format. Supported formats: `1` - CAdES-BES; `2` - CAdES-X Long Type 1; `3` - CAdES-T.

  Query example:

POST /pki/sign:

```
{
  "inputData" : "SGVsbG8gd29ybGQh",
  "keystoreAlias" : "key1",
  "password" : "password",
  "sigType" : 1,
}
```

The method returns the `signature` field containing the generated disconnected DS.

Response example:

POST /pki/sign:

```
{
  "signature" :
↪"c2RmZ3NkZmZoZ2ZkZ2hmZGpkZ2ZoamhnZmtqaGdmamtkZmdoZmdkc2doZmQjsndjfvnksdnjfn="
}
```

POST /pki/verify

The method is designed to verify the disconnected DS for the data transmitted in the request. The request consists of the following fields:

- `inputData` - data signed by an DS (as an array of bytes in **base64** encoding);
- `signature` - digital signature in the form of an array of bytes in **base64** encoding;
- `sigType` - DS format. Supported formats: `1` - CAdES-BES; `2` - CAdES-X Long Type 1; `3` - CAdES-T.
- `extended_key_usage_list` - list of object identifiers (OIDs) of cryptographic algorithms that are used in DS generation (*optional field*).

Query example:

POST /pki/verify:

```
{
 "inputData" : "SGVsbG8gd29ybGQh",
 "signature" : "c2RmZ3NkZmZoZ2ZkZ2hmZGpkZ2ZoamhnZmtqaGdmamtkZmdoZmdkc2doZmQ=",
 "sigType" : "CAdES_X_Long_Type_1",
 "extendedKeyUsageList": [
 "1.2.643.7.1.1.1.1",
 "1.2.643.2.2.35.2"
 ]
 }
```

Method's response contains a `sigStatus` field with boolean data type: `true` - signature is valid, `false` - signature is compromised.

Response example:

POST /pki/verify:

```
{
  "sigStatus" : "true"
}
```

Verifying an advanced qualified digital signature

The `POST /pki/verify` method has the ability to verify an advanced qualified digital signature. To verify the AQDS correctly, install the root AQDS certificate of the certification authority (CA) on your node, which will be used to validate the signature.

The root certificate is installed in the **cacerts** certificate storage of the Java virtual machine (JVM) you are using using the **keytool** utility:

```
sudo keytool -import -alias certificate_alias -keystore path_to_your_JVM/lib/security/
↪cacerts -file path_to_the_certificate/cert.cer
```

After the `-alias` flag, specify your preferred certificate name in the repository.

The cacerts certificate storage is located in the `/lib/security/` subdirectory of your Java virtual machine. To find out the path to the virtual machine on Linux, use the following command:

```
readlink -f /usr/bin/java | sed "s:bin/java::"
```

Then add `/lib/security/cacerts` to the resulting path and paste the resulting absolute path to **cacerts** after the ` *-keystore*` flag.

After the `-file` flag, specify the absolute or relative path to the received EDS certificate of the Certification Authority.

The default password for **cacerts** is `changeit`. If necessary, you can change it using the **keytool** utility:

```
sudo keytool -keystore cacerts -storepasswd
```

See also

REST API methods

Cryptography

10.2.3 REST API: encryption and decryption methods

REST API methods of the `crypto` group are provided to implement encryption methods.

The working principle of this group of methods is similar to the set *gRPC-methods contract_ crypto_ service.proto*.

POST /crypto/encryptSeparate

Encryption of data transmitted in the request, is performed with unique keys CEK separately for each recipient, each CEK is encrypted (*wrapped*) with a separate key KEK.

The following data are submitted in the query:

- `sender` - an address of data sender;
- `password` - password to the encrypted data;
- `encryptionText` - data to be encrypted (as a string);
- `recipients_public_keys` - public keys of the recipients participating in the network;
- `crypto_algo` - encryption algorithm in use. Available values: `gost-28147`; `gost-3412-2015-k`; `aes`.

If your network uses GOST encryption, only the algorithms `gost-28147` and `gost-3412-2015-k` are available to you. If GOST encryption is disabled, only the `aes` encryption algorithm is available.

Query example:

POST /crypto/encryptSeparate:

```
{
  "sender": "3MsHHc8LvyjPCKeSst9vsYcsHeQVzH6YJkL",
  "password": "",
  "encryptionText": "some string to encrypt",
  "recipientsPublicKeys": [
    "3MuNFC1Z8Tuy73pMzVUT6yowk4anWA8MNNE"
  ],
  "cryptoAlgo": "aes"
}
```

The response includes the following data for each recipient:

- `encrypted_data` - encrypted data;
- `public_key` - recipient public key;
- `wrapped_key` - result of key encryption for a recipient.

Response example:

POST /crypto/encryptSeparate:

```
{
 "encryptedText": "IZ5Kk5YNspMWl/jmlTizVxD6Nik=",
 "publicKey":
↪"5R65oLxp3iwPekwirA4VwwUXaySz6W6YKXBKBRL352pwwcpsFcjRHJ1VVHLp63LkrkxsNod64V1pffeiZz5i2qXc
↪",
 "wrappedKey":
↪"uWVoxJAzruwTDDSbphDS31TjSQX6CSWXivp3x34uE3XtnMqqK9swoaZ3LyAgFDR7o6CfkgzFkWmTen4qAZewPfBbwR
↪"
},
```

POST /crypto/encryptCommon

Encryption of data transmitted in the request with a single CEK key for all recipients, each CEK key is encrypted (*wrapped*) with a separate KEK key for each recipient.

The **POST /crypto/encryptCommon** request contains data similar to the **POST /crypto/encryptSeparate** request.

The response includes the following data for each recipient:

- `encrypted_data` - encrypted data;
- `recipient_to_wrapped_structure` - a structure in the "key : value" format containing the public keys of the recipients with the corresponding key encryption results for each of them.

Response example:

POST /crypto/encryptCommon:

```
{
  "encryptedText": "NpCCig2i3jzo0xBnfqjfedbti8Y=",
  "recipientToWrappedStructure": {

↪"5R65oLxp3iwPekwirA4VwwUXaySz6W6YKXBKBRL352pwwcpsFcjRHJ1VVHLp63LkrkxsNod64V1pffeiZz5i2qXc
↪":

↪"M8pAe8HnKiWLE1HsC1ML5t8b7giWxiHfvagh7Y3F7rZL8q1tqMCJMYJo4qz4b3xjcuuUiV57tY3k7oSig53Aw1Dkkw
↪",

↪"9LopMj2GqWxBYgnZ2gxaNxwXqxXHuWd6ZAdVqkprR1fFMNvDUHYUCwFxsB79B9sefgxNdqwNtqzuDS8Zmn48w3S
↪":

↪"Doqn6gPvBBeSu2vdwgFYMbDHM4knEGMbqPn8Np76mNRRoZXLDioofyVbSSaTTEr4cvXwzEwVMugiy2wuzFWk3zCiT3
↪"
   }
  }
```

POST /crypto/decrypt

Decryption of data encrypted with the cryptographic algorithm used by the network. Decryption is possible if the recipient's key is in the keystore of the node.

The following data are submitted in the query:

- `recipient` - recipient's public key from the node keystore;
- `password` - password to the encrypted data;
- `encryptedText` - encrypted string;
- `wrapped_key` - result of key encryption for a recipient;
- `senderPublicKey` – a public key of data sender;
- `crypto_algo` - encryption algorithm in use. Available values: `gost-28147`; `gost-3412-2015-k`; `aes`.

If your network uses GOST encryption, only the algorithms `gost-28147` and `gost-3412-2015-k` are available to you. If GOST encryption is disabled, only the `aes` encryption algorithm is available.

Query example:

POST /crypto/decrypt:

```
{
  "recipient": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "password": "12345qwert",
  "encryptedText":
↪"t859AE7idnjPpn3lUiorfzSGwcGPMVdOhQe1HAhoI0M0X0QPBc8TUhn+8pKRCL8evH2Ra9Vc",
  "wrappedKey": "2nfob2yW76xj2rQBWZkzFD2UjYymWqQUCpFqbSWQiSYnuaw6DZoAde8KsTCMxPFVHA",
  "senderPublicKey": "CgqRPcPnexY533gCh2SSvBXh5bca1qMs7KFGntawHGww",
  "cryptoAlgo": "aes"
}
```

The `decryptedText` field, which contains the decrypted string, arrives in response to the request.

Response example:

POST /crypto/decrypt:

```
{
"decryptedText": "some string for encryption",
}
```

See also

REST API methods

Cryptography

10.2.4 REST API: confidential data exchange and obtaining of information about confidential data groups

Learn more about confidential data exchange and access groups in the article *Confidential data exchange*.

A set of methods from the `Privacy` group is provided to implement these functions using the REST API:

POST /privacy/sendData

The method is designed to send sensitive data to the blockchain, available only to members of the access group defined for this data. The method request contains the following information:

- `sender` - blockchain address from which the data should be sent (corresponds to the value of the "privacy.owner-address" parameter in the configuration file of the node);
- `password` - password to access the private key in the node keystore;
- `policyId` - identifier of a group that will have access to the data to be forwarded;
- `info` - information about data being sent;
- `data` - string containing data in **base64** format;

- `hash` - sha256-hash data in **base58** format.

Examples of a query and a response:

POST /privacy/sendData:

Query:

```
{
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"password": "apgJP9atQccdBPA",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"info": {
  "filename":"Service contract #100/5.doc",
  "size": 2048,
  "timestamp": 1000000000,
  "author": "AIvanov@org.com",
  "comment": "some comments"
 },
 "data":
↪"TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSBOaGlzIHNpbmd1bGFyIHBhc3Np
↪",
 "hash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta"
 }
```

Response:

```
{
"senderPublicKey": "Gt3o1ghh2M2TS65UrHZCTJ82LLcMcBrxuaJyrgsLk5VY",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"dataHash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta",
"proofs": [

↪"2jM4tw4uDmspuXUBt6492T7opuZskYhFGW9gkbq532BvLYRF6RJn3hVGNLuMLK8JSM61GkVgYvYJg9UscAayEYfc
↪"
],
"fee": 110000000,
"id": "H3bdFTatppjnMmUe38YWh35Lmf4XDYrgsDK1P3KgQ5aa",
"type": 114,
"timestamp": 1571043910570
}
```

POST /privacy/sendDataV2

The **POST /privacy/sendDataV2** method is similar to the **POST /privacy/sendData** method, but allows you to attach a file in the Swagger window without having to convert it to **base64** format. The `Data` field is missing in this version of the method.

Examples of a query and a response:

POST /privacy/sendDataV2:

Query:


```
{
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"password": "apgJP9atQccdBPA",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"info": {
  "filename":"Service contract #100/5.doc",
  "size": 2048,
  "timestamp": 1000000000,
  "author": "AIvanov@org.com",
  "comment": "some comments"
 },
 "hash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta"
 }
```


Response:


```
{
"senderPublicKey": "Gt3o1ghh2M2TS65UrHZCTJ82LLcMcBrxuaJyrgsLk5VY",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"dataHash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta",
"proofs": [

↪"2jM4tw4uDmspuXUBt6492T7opuZskYhFGW9gkbq532BvLYRF6RJn3hVGNLuMLK8JSM61GkVgYvYJg9UscAayEYfc
↪"
],
"fee": 110000000,
"id": "H3bdFTatppjnMmUe38YWh35Lmf4XDYrgsDK1P3KgQ5aa",
"type": 114,
"timestamp": 1571043910570
}
```


GET /privacy/–policy-id″/recipients

The method is designed to get the addresses of all members recorded in group `{policy-id}`.

The response of the method returns an array of strings with the addresses of the members of the access group.

Response example:

GET /privacy/–policy-id″/recipients:

```
[
  "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "3Mx2afTZ2KbRrLNbytyzTtXukZvqEB8SkW7"
]
```

GET /privacy/–policy-id″/owners

The method is designed to get the addresses of the owners of access group `{policy-id}`.

The response of the method returns an array of strings with the addresses of the owners of the access group.

Response example:

GET /privacy/–policy-id″/owners:

```
[
  "3GCFaCWtvLDnC9yX29YftMbn75gwfdwGsBn",
  "3GGxcmNyq8ZAHzK7or14Ma84khwW8peBohJ",
  "3GRLFi4rz3SniCuC7rbd9UuD2KUZyNh84pn",
  "3GKpShRQRTddF1yYhQ58ZnKMTnp2xdEzKqW"
]
```

GET /privacy/–policy-id″/hashes

The method is designed to get an array of identification hashes of data that are bound to the `{policy-id}` access group.

The response of the method returns an array of strings with the identity hashes of the access group data.

Response example:

GET /privacy/–policy-id″/hashes:

```
[
  "FdfdNBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "eedfdNBVqYXrapgJP9atQccdBPAgJPwHDKkh6A"
]
```

GET /privacy/–policyId″/getData/–policyItemHash″

The method is designed to retrieve a packet of confidential data of access group `{policyId}` by the identification hash `{policyItemHash}`.

The response of the method returns the hash sum of the confidential data.

Response example:

GET /privacy/–policyId"/getData/–policyItemHash":

```
c29tZV9iYXNlNjRfZW5jb2RlZF9zdHJpbmc=
```

GET /privacy/–policyId"/getInfo/–policyItemHash"

The method is designed to retrieve a packet of confidential data of access group `{policyId}` by the identification hash `{policyItemHash}`.

The method response returns the following data:

- `sender` - an address of confidential data sender;
- `policy_id` - a confidential data group identifier;
- `type` - type of confidential data (`file`);
- `info` - file data array: `filename` - name of a file, `size` - file size, `timestamp` - a *Unix Timestamp* of file uploading (in milliseconds), `author` - file author, `comment` - optional comment to the file, `hash` - confidential data identifying hash.

Response example:

GET /privacy/–policyId"/getInfo/–policyItemHash":

```
{
  "sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
  "policy": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
  "type": "file",
  "info": {
    "filename": "Contract №100/5.doc",
    "size": 2048,
    "timestamp": 1000000000,
    "author": "AIvanov@org.com",
    "comment": "Comment"
  },
"hash": "e67ad392ab4d933f39d5723aeed96c18c491140e119d590103e7fd6de15623f1"
}
```

POST /privacy/forceSync

The method is designed to force a packet of confidential data. It is used if a transaction with confidential data for an access group is present in the blockchain, but for some reason this data was not written to the node's confidential data repository. In this case, the method allows to forcibly download the missing data.

The response includes the following data:

- `sender` - address of the node participating in the access group that sends the request;
- `policy` - a confidential data group identifier;
- `source` - address of the node from which the missing data should be downloaded. In case the node is unknown, set the parameter to `null` or leave the field empty: in this case the file will be downloaded from the storage of the first node in the access group list.

Method response contains a `result` field with data retrieval result and `message` field with possible error text. In case of successful reception, `success` is returned, confidential data is written to node storage.

If an error occurs, `error` is returned, the `message` field contains a description of the error.

Examples of a query and a response:

POST /privacy/forceSync:

Query:

```
{
  "sender": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "policy": "my_policy"
  "source": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
}
```

Response:

```
{
  "result": "error"
  "message": "Address '3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8' not in␣
↪policy 'my_policy'"
}
```

POST /privacy/getInfos

The method is designed to obtain an array of confidential data metadata by access group identifier and identification hash.

The response includes the following data:

- `policiesDataHashes` - an array of data with two elements for each individual access group: `policyId` - access group identifier; `datahashes` - an array of sensitive data hashes to get metadata for each of them.

The method response returns an array of data for each individual hash of sensitive data, similar to the response of the `GET /privacy/{policyId}/getInfo/{policyItemHash}` method.

Examples of a query and a response:

POST /privacy/getInfos:

Query:

```
{ "policiesDataHashes":
  [
   {
    "policyId": "somepolicyId_1",
    "datahashes": [ "datahash_1","datahash_2" ]
   },
   {
     "policyId": "somepolicyId_2",
     "datahashes": [ "datahash_3","datahash_4" ]
```

(continues on next page)

(continued from previous page)

```
  }
 ]
}
```

Response:

```
{
   "policiesDataInfo":[
      {
         "policyId":"somepolicyId_1",
         "datasInfo":[
            {
               "hash":
↪"e67ad392ab4d933f39d5723aeed96c18c491140e119d590103e7fd6de15623f1
↪",
               "sender":"3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
               "type":"file",
               "info":{
                  "filename":"Contract №100/5.doc",
                  "size":2048,
                  "timestamp":1000000000,
                  "author":"AIvanov@org.com",
                  "comment":"Comment"
               }
            },
            {
               "hash":
↪"e67ad392ab4d933f39d5723aeed96c18c491140e119d590103e7fd6de15623f1
↪",
               "sender":"3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
               "type":"file",
               "info":{
                  "filename":"Contract №101/5.doc",
                  "size":"2048",
                  "timestamp":1000000000,
                  "author":"AIvanov@org.com",
                  "comment":"Comment"
               }
            }
            }
         ]
      ]
   }
```

See also

REST API methods

Confidential data exchange

10.2.5 REST API: validation of addresses and aliases of network participants

The following methods of the `addresses` group are provided for validating addresses and aliases on the network:

GET /addresses/validate/–addressOrAlias″

Validation of a given recipient or its alias `{addressOrAlias}` on the blockchain network of a working node.

Response example:

GET /addresses/validate/–addressOrAlias″:

```
{
  addressOrAlias: "3HSVTtjim3FmV21HWQ1LurMhFzjut7Aa1Ac",
  valid: true
}
```

POST /addresses/validateMany

Validation of multiple addresses or aliases passed to the `addressesOrAliases` field as an array. The information in the response for each address is identical to the GET /addresses/validate/{addressOrAlias} method response.

Examples of query and esponse for one address, one existing and one non-existing alias:

POST /addresses/validateMany:

Query:

```
{
  addressesOrAliases: [
    "3HSVTtjim3FmV21HWQ1LurMhFzjut7Aa1Ac",
    "alias:T:asdfghjk",
    "alias:T:1nvAliDAl1ass99911%^&$$$ "
  ]
}
```

Response:

```
{
  validations: [
    {
      addressOrAlias: "3HSVTtjim3FmV21HWQ1LurMhFzjut7Aa1Ac",
      valid: true
    },
    {
      addressOrAlias: "alias:T:asdfghjk",
      valid: true
    },
    {
      addressOrAlias: "alias:T:1nvAliDAl1ass99911%^&$$$ ",
      valid: false,
      reason: "GenericError(Alias should contain only following characters: -.
↪01234567890_abcdefghijklmnopqrstuvwxyz)"
    }
  ]
}
```

See also

REST API methods

10.2.6 REST API: signing and validating messages in the blockchain

The following methods of the `addresses` group are provided for signing and validating messages:

POST /addresses/sign/–address″

The method signs the string passed in the `message` field with the addressee `{address}` private key and then serializes it in **base58** format. The response of the method returns the serialized string, the addressee's public key and signature.

Examples of a query and a response:

POST /addresses/sign/–address″:

Query:

```
{
 "message": "mytext"
}
```

Response:

```
{
  "message": "wWshKhJj",
  "publicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
  "signature":
```

(continues on next page)

(continued from previous page)

```
↪"62PFG855ThsEHUZ4N8VE8kMyHCK9GWnvtTZ3hq6JHYv12BhP1eRjegA6nSa3DAoTTMammhamadvizDUYZAZtKY9S
↪"
}
```

POST /addresses/verify/–address″

The method checks the signature of the message made by the `{address}`.

Examples of a query and a response:

POST /addresses/verify/–address″:

Query:

```
{
  "message": "wWshKhJj",
  "publickey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
  "signature":
↪"5kwwE9sDZzssoNaoBSJnb8RLqfYGt1NDGbTWWXUeX8b9amRRJN3hr5fhs9vHBq6VES5ng4hqbCUoDEsoQNauRRts
↪"
}
```

Response:

```
{
  "valid": true
}
```

POST /addresses/signText/–address″

The method signs the string passed in the `message` field with the private key of an `{address}`. Unlike the `POST /`addresses/sign/{address}` method, the string is passed in the original format.

Examples of a query and a response:

POST /addresses/signText/–address″:

Query:

```
{
  "message": "mytext"
}
```

Response:

```
{
  "message": "mytext",
  "publicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
  "signature":
↪"5kVZfWfFmoYn38cJfNhkdct5WCyksMgQ7kjwHK7Zjnrzs9QYRWo6HuJoGc8WRMozdYcAVJvojJnPpArqPvu2uc3u
```

(continues on next page)

(continued from previous page)

```
↪"
}
```

POST /addresses/verifyText/–address″

Checks the signature of the message made by the `{address}` via the POST method `/addresses/signText/{address}`.

Examples of a query and a response:

POST /addresses/verifyText/–address″:

Query:

```
{
  "message": "mytext",
  "publicKey": "C1ADP1tNGuSLTiQrfNRPhgXx59nCrwrZFRV4AHpfKBpZ",
  "signature":
↪"5kVZfWfFmoYn38cJfNhkdct5WCyksMgQ7kjwHK7Zjnrzs9QYRWo6HuJoGc8WRMozdYcAVJvojJnPpArqPvu2uc3u
↪"
}
```

Response:

```
{
  "valid": true
}
```

See also

REST API methods

10.2.7 REST API: information about configuration and state of the node, stopping the node

There are two groups of methods to get information about the node configuration:

- `node` - obtaining basic configuration parameters of a node, information about node state, stopping a node, changing a logging level;
- `anchoring` - the `GET /anchoring/config` query, which returns the `anchoring` section of the node configuration file.

To get the basic configuration parameters of a node, there are both methods that require authorization and open methods.

node group:

GET /node/config

The method returns the basic configuration parameters of a node.

Response example:

GET /node/config:


```
{
    {
      "version": "1.3.0-RC7",
      "gostCrypto": false,
      "chainId": "V",
      "consensus": "POA",
      "minimumFee": {
          "3": 0,
          "4": 0,
          "5": 0,
          "6": 0,
          "7": 0,
          "8": 0,
          "9": 0,
          "10": 0,
          "11": 0,
          "12": 0,
          "13": 0,
          "14": 0,
          "15": 0,
          "102": 0,
          "103": 0,
          "104": 0,
          "106": 0,
          "107": 0,
          "111": 0,
          "112": 0,
          "113": 0,
          "114": 0
      },
      "additionalFee": {
          "11": 0,
          "12": 0
      },
      "maxTransactionsInMicroBlock": 500,
      "minMicroBlockAge": 0,
      "microBlockInterval": 1000,
      "blockTiming": {
          "roundDuration": 7000,
          "syncDuration": 700
```


(continues on next page)

(continued from previous page)

```
    }
}
```

GET /node/owner

The method returns the address and the public key of the node owner.

Response example:

GET /node/config:

```
{
  "address": "3JFR1pmL6biTzr9oa63gJcjZ8ih429KD3aF",
  "publicKey": "EPxkVA9iQejsjQikovyxkkY8iHnbXsR3wjgkgE7ZW1Tt"
}
```

GET /node/status

The method returns information about the current state of the node.

Response example:

GET /node/status:

```
{
    "blockchainHeight": 47041,
    "stateHeight": 47041,
    "updatedTimestamp": 1544709501138,
    "updatedDate": "2018-12-13T13:58:21.138Z"
}
```

Also, if there are errors using GOST cryptography on a node, the method will return an error description:

GET /node/status:

```
{
    "error": 199,
    "message": "Environment check failed: Supported JCSP version is 5.0.40424, actual is␣
↪2.0.40424"
}
```

GET /node/version

The method returns a node version.

Response example:

GET /node/version:

```
{
  "version": "Waves Enterprise v0.9.0"
}
```

GET /node/logging

The method displays a list of loggers specified when configuring the node, and the logging level for each of them.

Node logging levels:

- `ERROR` - error logging;
- `WARN` - warning logging;
- `INFO` - node events logging;
- `DEBUG` - extended information about 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 of data from all logging levels.

Response example:

GET /node/logging:

```
ROOT-DEBUG
akka-DEBUG
akka.actor-DEBUG
akka.actor.ActorSystemImpl-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.core-DEBUG
com.github.dockerjava.core.command-DEBUG
com.github.dockerjava.core.command.AbstrDockerCmd-DEBUG
com.github.dockerjava.core.exec-DEBUG
```

POST /node/logging

The method is designed to change the logging level for selected loggers.

Query example:

POST /node/logging:

```
{
    "logger": "com.wavesplatform.Application",
    "level": "ALL"
}
```

POST /node/stop

The method stops the node, it has no response.

The GET /anchoring/config method:

The method outputs the `anchoring` section of the node configuration file.

Response example:

GET /anchoring/config:

```
{
 "enabled": true,
 "currentChainOwnerAddress": "3FWwx4o1177A4oeHAEW5EQ6Bkn4Lv48quYz",
 "mainnetNodeAddress": "https://clinton-pool.wavesenterpriseservices.com:443",
 "mainnetSchemeByte": "L",
 "mainnetRecipientAddress": "3JzVWCSV6v4ucSxtGSjZsvdiCT1FAzwpqrP",
 "mainnetFee": 8000000,
 "currentChainFee": 666666,
 "heightRange": 5,
 "heightAbove": 3,
 "threshold": 10
}
```

See also

REST API methods

Examples of node configuration files

10.2.8 REST API: information about network participants

There are three groups of methods for obtaining information about network participants:

- `addresses` - the methods designed to get information about the addresses of network members;
- `alias` - getting the participant's address by the alias set for him or the alias by the member's address;
- `leasing` - the `GET /leasing/active/{address}` query that outputs a list of leasing transactions in which the address was involved as sender or receiver.

addresses group:

GET /addresses

Obtaining all participant addresses, whose key pairs are stored in the keystore of the node.

Response example:

GET /addresses:

```
[
  "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "3Mx2afTZ2KbRrLNbytyzTtXukZvqEB8SkW7"
]
```

GET /addresses/seq/–from″/–to″

Obtaining addresses of the participants, which are stored in the keystore of a node in a given range: from the address `{from}` to the address `{to}`.

The format of the method response is identical to that of `GET /addresses`.

GET /addresses/balance/–address″

Getting the balance for the address `{address}`.

Response example:

GET /addresses/balance/–address″:


```
{
  "address": "3N3keodUiS8WLEw9W4BKDNxgNdUpwSnpb3K",
  "confirmations": 0,
  "balance": 100945889661986
}
```


POST /addresses/balance/details

Get detailed balance information for the list of addresses, which is specified as an array in the `addresses` field when prompted.

Parameters returned in the method response:

- `regular` - amount of tokens owned directly by the participant (**R**);
- `available` - participant's total balance, excluding funds leased by the participant (**A** = **R** - **L**);
- `effective` - participant's total balance, including funds leased to the participant and minus funds that the participant himself has leased (**E** = **R** + **F** - **L**);
- `generating` - participant's generating balance, including leased funds, for the last 1,000 blocks.

Variables in parentheses: **L** - funds leased by the participant to other participants, **F** - funds leased by the participant.

Response example for one address:

POST /addresses/balance/details:


```
[
  {
    "address": "3M4Bxh2VfzKFXqiQB8bDgRfVnPWrZUQ2MEF",
    "regular": 59899999999400000,
    "generating": 59899999999400000,
    "available": 59899999999400000,
    "effective": 59899999999400000
  }
]
```


GET /addresses/balance/details/–address″

Get detailed balance information for an individual address. The information in the response is identical to the `POST /addresses/balance/details` method.

Response example:

GET /addresses/balance/details/–address″:


```
[
  {
    "address": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
    "regular": 0,
    "generating": 0,
    "available": 0,
    "effective": 0
  }
]
```


GET /addresses/effectiveBalance/–address″

Obtaining the total balance of the address, including leased funds.

Response example:

GET /addresses/effectiveBalance/–address″:


```
{
   "address": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
   "confirmations": 0,
   "balance": 1240001592820000
}
```


GET /addresses/effectiveBalance/–address″/–confirmations″

Retrieves the balance for `{address}` after the number of confirmations >= `{confirmations}`. The participant's total balance is returned, including funds leased to the participant.

Response example for confirmations >= 1:

GET /addresses/effectiveBalance/–address″/–confirmations″:


```
{
  "address": "3N65yEf31ojBZUvpu4LCo7n8D73juFtheUJ",
  "confirmations": 1,
  "balance": 0
}
```

GET /addresses/generatingBalance/–address″/at/–height″

Obtaining the generating address balance at the specified block height `{height}`.

Response example:

GET /addresses/generatingBalance/–address″/at/–height″:

```
{
  "address": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "generatingBalance": 1011543800600
}
```

GET /addresses/scriptInfo/–address″

Obtaining data about the script installed on the address.

Parameters returned in the method response:

- `address` - address in **base58** format;
- `script` - script body in **base64** format;
- `scriptText` - the source code of the script;
- `complexity` - complexity of the script;
- `extraFee` - fee for outgoing transactions set by the script.

Script complexity - a number from 1 to 100 representing the amount of computing resources required to execute the script.

Response example:

GET /addresses/scriptInfo/–address″:

```
{
  "address": "3N3keodUiS8WLEw9W4BKDNxgNdUpwSnpb3K",
  "script":
↪"3rbFDtbPwAvSp2vBvqGfGR9nRS1nBVnfuSCN3HxSZ7fVRpt3tuFG5JSmyTmvHPxYf34SocMRkRKFgzTtXXnnv7upRHXJzZrLSQo8
↪",
  "scriptText": "ScriptV1(BLOCK(LET(x,CONST_LONG(1)),FUNCTION_CALL(FunctionHeader(==,
↪List(LONG, LONG)),List(FUNCTION_CALL(FunctionHeader(+,List(LONG, LONG)),List(REF(x,
↪LONG), CONST_LONG(1)),LONG), CONST_LONG(2)),BOOLEAN),BOOLEAN))",
  "complexity": 11,
  "extraFee": 10001
}
```

GET /addresses/publicKey/–publicKey″

The method returns the participant's address based on its public key.

Response example:

GET /addresses/publicKey/–publicKey″:

```
{
  "address": "3N4WaaaNAVLMQgVKTRSePgwBuAKvZTjAQbq"
}
```

GET /addresses/data/–address″

The method returns data written to the specified address using *transaction 12*.

Response example:

GET /addresses/data/–address″:

```
[
  {
    "key": "4yR7b6Gv2rzLrhYBHpgVCmLH42raPGTF4Ggi1N36aWnY",
    "type": "integer",
    "value": 1500000
  }
]
```

GET /addresses/data/–address″/–key″

The method returns data recorded at the specified address with key `{key}`. This key is specified in the *transaction 12* in the `data.key` field.

Response example:

GET /addresses/data/–address″/–key″:

```
{
  "key": "4yR7b6Gv2rzLrhYBHpgVCmLH42raPGTF4Ggi1N36aWnY",
  "type": "integer",
  "value": 1500000
}
```

alias group:

GET /alias/by-alias/–alias″

Obtaining a participant's address by his `{alias}`.

Response example:

GET /alias/by-alias/–alias″:

```
{
     "address": "address:3Mx2afTZ2KbRrLNbytyzTtXukZvqEB8SkW7"
}
```

GET /alias/by-address/–address″

Obtaining a participant's alias from his `{address}`.

Response example:

GET /alias/by-alias/–alias″:

```
[
    "alias:participant1",
]
```

The GET /leasing/active/–address″ method:

The method returns a list of lease creation transactions in which the address participated as sender or recipient.

Response example with one transaction:

GET /alias/by-alias/–alias″:

```
[
  {
    "type": 8,
    "id": "2jWhz6uGYsgvfoMzNR5EEGdi9eafyCA2zLFfkM4NP6T7",
    "sender": "3PP6vdkEWoif7AZDtSeSDtZcwiqSfhmwttE",
    "senderPublicKey": "DW9NKLYeyoEWDqJKhWv87EdFfTqpFtJBWoCqfCVwRhsY",
    "fee": 100000,
    "timestamp": 1544390280347,
    "signature":
↪"25kpwh7nYjRUtfbAbWYRyMDPCUCoyMoUuWTJ6vZQrXsZYXbdiWHa9iGscTTGnPFyegP82sNSfM2bXNX3K7p6D3HD
↪",
    "version": 1,
    "amount": 31377465877,
    "recipient": "3P3RD3yJW2gQ9dSVwVVDVCQiFWqaLtZcyzH",
```

(continues on next page)

(continued from previous page)

```
    "height": 1298747
  }
]
```

See also

REST API methods

10.2.9 REST API: information about a consensus algorithm in use

The methods of the `consensus` group are provided to obtain information about the consensus algorithm used.

GET /consensus/algo

The method returns the name of the consensus algorithm used.

Response example:

GET /consensus/algo:

```
{
    "consensusAlgo": "Leased Proof-of-Stake (LPoS)"
}
```

GET /consensus/settings

The method returns the parameters of the consensus algorithm used, specified in the node configuration file.

Response example:

GET /consensus/settings:

```
{
    "consensusAlgo": "Proof-of-Authority (PoA)",
    "roundDuration": "25 seconds",
    "syncDuration": "5 seconds",
    "banDurationBlocks": 50,
    "warningsForBan": 3
}
```

GET /consensus/minersAtHeight/–height″

The method returns the queue of miners at `{height}`. This method is available when using the *PoA consensus algorithm*.

Response example:

GET /consensus/minersAtHeight/–height″:

```
{
"miners": [
    "3Mx5sDq4NXef1BRzJRAofa3orYFxLanxmd7",
    "3N2EsS6hJPYgRn7WFJHLJNnrsm92sUKcXkd",
    "3N2cQFfUDzG2iujBrFTnD2TAsCNohDxYu8w",
    "3N6pfQJyqjLCmMbU7G5sNABLmSF5aFT4KTF",
    "3NBbipRYQmZFudFCoVJXg9JMkkyZ4DEdZNS"
],
"height": 1
}
```

GET /consensus/miners/–timestamp″

The method returns the queue of miners for `{timestamp}` (specified in **Unix Timestamp** format, in milliseconds). The method is available when using the *PoA consensus algorithm*.

Response example:

GET /consensus/miners/–timestamp″:

```
{
    "miners": [
        "3Mx5sDq4NXef1BRzJRAofa3orYFxLanxmd7",
        "3N2EsS6hJPYgRn7WFJHLJNnrsm92sUKcXkd",
        "3N2cQFfUDzG2iujBrFTnD2TAsCNohDxYu8w",
        "3N6pfQJyqjLCmMbU7G5sNABLmSF5aFT4KTF",
        "3NBbipRYQmZFudFCoVJXg9JMkkyZ4DEdZNS"
    ],
    "timestamp": 1547804621000
}
```

GET /consensus/bannedMiners/–height″

The method returns the list of banned miners at `{height}`. This method is available when using the *PoA consensus algorithm*.

Response example:

GET /consensus/bannedMiners/–height″:

```
{
  "miners": [
      "3N6pfQJyqjLCmMbU7G5sNABLmSF5aFT4KTF",
      "3NBbipRYQmZFudFCoVJXg9JMkkyZ4DEdZNS"
  ],
  "height": 440
}
```

GET /consensus/basetarget/–signature″

The method returns the basetarget value of block creation by its `{signature}`. The method is available when using the *PoS consensus algorithm*.

GET /consensus/basetarget

The method returns the basetarget value of current block creation. The method is available when using the *PoS consensus algorithm*.

GET /consensus/generatingbalance/–address″

The method returns the generating balance available for the `{address}` node, including funds leased to the participant. The method is available when using the ref:*PoS consensus algorithm <pos-consensus>*.

GET /consensus/generationsignature/–signature″

The method returns the generating signature value of block creation by its `{signature}`. The method is available when using the *PoS consensus algorithm*.

GET /consensus/generationsignature

The method returns the generating signature value of current block creation. The method is available when using the *PoS consensus algorithm*.

See also

REST API methods

Consensus algorithms

10.2.10 REST API: information about smart contracts

A set of methods from the `contracts` group is provided to obtain information about smart contracts loaded on the network.

GET /contracts

The method returns information on all smart contracts uploaded to the network. For each smart contract the following parameters are returned in the response:

- `contract_id` - smart contract identifier;
- `image` - name of the Docker image of the smart contract, or its absolute path in the repository;
- `imageHash` - hash sum of the smart contract;
- `version` - version of the smart contract;
- `active` - status of the smart contract at the time of sending the query: `true` - running, `false` - not running.

Response example for one smart contract:

GET /contracts:

```
[
  {
    "contractId": "dmLT1ippM7tmfSC8u9P4wU6sBgHXGYy6JYxCq1CCh8i",
    "image": "registry.wvservices.com/wv-sc/may14_1:latest",
    "imageHash": "ff9b8af966b4c84e66d3847a514e65f55b2c1f63afcd8b708b9948a814cb8957",
    "version": 1,
    "active": false
   }
 ]
```

POST /contracts

The method returns a set of "key:value" fields written to the stack of one or more smart contracts. The IDs of the requested smart contracts are specified in the `contracts` field of the request.

Response example for one smart contract:

POST /contracts:

```
{
  "8vBJhy4eS8oEwCHC3yS3M6nZd5CLBa6XNt4Nk3yEEExG": [
   {
   "type": "string",
   "value": "Only description",
   "key": "Description"
   },
```

(continues on next page)

(continued from previous page)

```
   {
   "type": "integer",
   "value": -9223372036854776000,
   "key": "key_may"
   }
  ]
 }
```

GET /contracts/status/–id″

The method returns the status of executed *transaction 103* to create a smart contract by transaction identifier `{id}`. However, if the node is restarted after sending the transaction to the blockchain, the method will not return the correct status of that transaction.

Parameters returned in the method response:

- `sender` - an address of transaction sender;
- `senderPublicKey` – a public key of transaction sender;
- `txId` - transaction identifier;
- `status` - transaction status: successfully hit the block, confirmed, rejected;
- `code` - error code (if any);
- `message` - message about the status of the transaction;
- `timestamp` - the *Unix Timestamp* (in milliseconds);
- `signature` - transaction signature.

Response example:

GET /contracts/status/–id″:

```
{
  "sender": "3GLWx8yUFcNSL3DER8kZyE4TpyAyNiEYsKG",
  "senderPublicKey": "4WnvQPit2Di1iYXDgDcXnJZ5yroKW54vauNoxdNeMi2g",
  "txId": "4q5Q8vLeGBpcdQofZikyrrjHUS4pB1AB4qNEn2yHRKWU",
  "status": "Success",
  "code": null,
  "message": "Smart contract transaction successfully mined",
  "timestamp": 1558961372834,
  "signature":
↪"4gXy7qtzkaHHH6NkksnZ5pnv8juF65MvjQ9JgVztpgNwLNwuyyr27Db3gCh5YyADqZeBH72EyAkBouUoKvwJ3RQJ
↪"
 }
```

GET /contracts/–contractId″

The method returns the result of smart contract execution by its `{contractId}` identifier.

Response example:

GET /contracts/–contractId″:

```
[
  {
    "key": "avg",
    "type": "string",
    "value": "3897.80146957"
  },
  {
    "key": "buy_price",
    "type": "string",
    "value": "3842"
  }
]
```

POST /contracts/–contractId″

The method returns key values from the `{contractId}` smart contract state. The query specifies the following data:

- `contract_id` - smart contract identifier;
- `limit` - a 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 sorting of keys.

Response example:

POST /contracts/–contractId″:

```
[
  {
    "type": "string",
    "key": "avg",
    "value": "3897.80146957"
  },
  {
    "type": "string",
    "key": "buy_price",
    "value": "3842"
  }
]
```

GET /contracts/executed-tx-for/–id″

The method returns the result of smart contract execution by identifier of a *transaction 105*.

The method's response returns transaction data 105, as well as the results of the execution in the `results` field.

Response example if a smart contract has not been executed:

GET /contracts/executed-tx-for/–id″:


```
{
  "type": 105,
  "id": "2UAHvs4KsfBbRVPm2dCigWtqUHuaNQou83CXy6DGDiRa",
  "sender": "3PKyW5FSn4fmdrLcUnDMRHVyoDBxybRgP58",
  "senderPublicKey": "2YvzcVLrqLCqouVrFZynjfotEuPNV9GrdauNpgdWXLsq",
  "fee": 500000,
  "timestamp": 1549365523980,
  "proofs": [
↪"4BoG6wQnYyZWyUKzAwh5n1184tsEWUqUTWmXMExvvCU95xgk4UFB8iCnHJ4GhvJm86REB69hKM7s2WLAwTSXquAs
↪"
  ],
  "version": 1,
  "tx": {
      "type": 103,
      "id": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky",
      "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
      "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
      "fee": 500000,
      "timestamp": 1550591678479,
      "proofs": [
↪"yecRFZm9iBLyDy93bDVaNo1PR5Qkkic7196GAgUt9TNH1cnQphq4yGQQ8Fxj4BYA4TaqYVw5qxtWzGMPQyVeKYv
↪" ],
      "version": 1,
      "image": "stateful-increment-contract:latest",
      "imageHash": "7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65",
      "contractName": "stateful-increment-contract",
      "params": [],
      "height": 1619
  },
  "results": []
}
```

GET /contracts/–contractId"/–key"

Returns the {key} value of the executed smart contract by its identifier.

Response example:

GET /contracts/–contractId"/–key":

```
{
  "key": "updated",
  "type": "integer",
  "value": 1545835909
}
```

See also

REST API methods

Smart contracts

Development and usage of smart contracts

10.2.11 REST API: information about network blocks

A group of `blocks` methods is provided to get information about the different blocks on the network.

GET /blocks/height

The method returns the number of the current block in the blockchain (block height).

Response example:

GET /blocks/height:

```
{
  "height": 7788
}
```

GET /blocks/height/–signature"

The method returns the block height by its `{signature}`.

The method response contains the `height` field, like the `GET /blocks/height` method.

GET /blocks/first

The method returns information about the genesis block of the network.

The response contains the following parameters:

- `reference` - hash sum of the genesis block;
- `blocksize` - size of the genesis block;
- `signature` - signature of the genesis block;
- `fee` - fees for transactions included in the genesis block;
- `generator` - address of creator of the genesis block;
- `transactionCount` - number of transactions *1* and *101* included in genesis block;
- `transactions` - array with the bodies of transactions 1 and 101 included in the genesis block;
- `version` - version of the genesis block;
- `timestamp` - **Unix Timestamp** of the genesis block (in milliseconds);
- `height` - height of the genesis block (**1**).

Response example:

GET /blocks/first:


```
{
    "reference":
↪"67rpwLCuS5DGA8KGZXKsVQ7dnPb9goRLoKfgGbLfQg9WoLUgNY77E2jT11fem3coV9nAkguBACzrU1iyZM4B8roQ
↪",
    "blocksize": 1435,
    "signature":
↪"4HENriUyMthzMSqWa5sYPFMATbzpQugTBMk6mXUh5HmnvHfUhmQk6EqmdhGvNFcUvTDrsyiVqkxtm8iiV2xNTSNK
↪",
    "fee": 0,
    "generator": "3MvQKx98a713B28rdUAtbWJ8DFJEXhnTjKs",
    "transactionCount": 26,
    "transactions": [
      {
        "type": 1,
        "id":
↪"2AdCY254MFSrgxpr6otBisV5Zz7neH8YoM6VGW5egoVJnwD8cJpYZVR42aVKTZnwGT9ee7LCpAGMNSUV86FEAGXu
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"2AdCY254MFSrgxpr6otBisV5Zz7neH8YoM6VGW5egoVJnwD8cJpYZVR42aVKTZnwGT9ee7LCpAGMNSUV86FEAGXu
↪",
        "recipient": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "amount": 1250000000000000
      },
      {
        "type": 1,
        "id":
```


(continues on next page)

(continued from previous page)


```
↪"5VC2LoFTbrfLkd48bjQkp8CmTyqXJSkJh723qxo9v5pz38tBUjRW9tHLuvwajSvkzQNFxrCc6Yjkgx5R2YR3x5VC
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"5VC2LoFTbrfLkd48bjQkp8CmTyqXJSkJh723qxo9v5pz38tBUjRW9tHLuvwajSvkzQNFxrCc6Yjkgx5R2YR3x5VC
↪",
        "recipient": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
        "amount": 300000000000000
      },
      {
        "type": 1,
        "id":
↪"4cmwEkSnBLc3TBTPUiT7HwmdER25X7GzCj2mgiEJ8K149vnNa1orBZUNstwNXtXFyKcQbkRPym39d9wJXTE4wgbU
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"4cmwEkSnBLc3TBTPUiT7HwmdER25X7GzCj2mgiEJ8K149vnNa1orBZUNstwNXtXFyKcQbkRPym39d9wJXTE4wgbU
↪",
        "recipient": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "amount": 100000000000000
      },
      {
        "type": 1,
        "id":
↪"5Etq3o1eWoN3bqR9cYV6149qxAE3ru4CoSCf1Mm5sSJEedcbmLhsbfg8rh4S6ESrAPq7ZEbghEgHjyb3xzUbDDRh
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"5Etq3o1eWoN3bqR9cYV6149qxAE3ru4CoSCf1Mm5sSJEedcbmLhsbfg8rh4S6ESrAPq7ZEbghEgHjyb3xzUbDDRh
↪",
        "recipient": "3N3jgxvmSsBBV4oz9BcKhT8War1em2sKoJn",
        "amount": 1000000000000000
      },
      {
        "type": 110,
        "id":
↪"3HewQJtzuaumzX4TvmN7fxVCgnsWTTaLeQjYBVDDuYoEW2ijWd7JME8h1gtsqepv5SDhHPvoMesVNm96br8WRgF8
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"3HewQJtzuaumzX4TvmN7fxVCgnsWTTaLeQjYBVDDuYoEW2ijWd7JME8h1gtsqepv5SDhHPvoMesVNm96br8WRgF8
↪",
        "targetPublicKey":
↪"56rV5kcR9SBsxQ9LtNrmp6V72S4BDkZUJaA6ujZswDneDmCTmeSG6UE2FQP1rPXdfpWQNunRw4aijGXxoK3o4puj
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP"
      },
      {
```


(continues on next page)

(continued from previous page)


```
        "type": 101,
        "id":
↪"5r4uLWn3rwmqbBygNj29iR4YsiV82dYWFeCbepAHhKGXqnn27vE6i811U9H2UZgX8zNQYZciyw3PR6nAdwjSPSp5
↪",
        "fee": 0,
        "timestamp": 1606211535609,
        "signature":
↪"5r4uLWn3rwmqbBygNj29iR4YsiV82dYWFeCbepAHhKGXqnn27vE6i811U9H2UZgX8zNQYZciyw3PR6nAdwjSPSp5
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "role": "permissioner"
      },
      {
        "type": 101,
        "id":
↪"4pBwjviNLtSPEBY5YB7ZdUXVSFnEk4rgscW8r9QQKxdxQZzjwjdq1ZnruMxQo7tomQVJf1Ni6SyVxSHrQZhBJaFN
↪",
        "fee": 0,
        "timestamp": 1606211535608,
        "signature":
↪"4pBwjviNLtSPEBY5YB7ZdUXVSFnEk4rgscW8r9QQKxdxQZzjwjdq1ZnruMxQo7tomQVJf1Ni6SyVxSHrQZhBJaFN
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "role": "miner"
      },
      {
        "type": 101,
        "id":
↪"5kwQwLH8oTy1ztF6xxsBxE3MDGio1NJm8F7Mtpynf3QTw9CWCsp5Fio5SxLmPxnB1bUVQHMCHbQCD4wXJLJgjSrp
↪",
        "fee": 0,
        "timestamp": 1606211535607,
        "signature":
↪"5kwQwLH8oTy1ztF6xxsBxE3MDGio1NJm8F7Mtpynf3QTw9CWCsp5Fio5SxLmPxnB1bUVQHMCHbQCD4wXJLJgjSrp
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "role": "connection_manager"
      },
      {
        "type": 101,
        "id":
↪"62xS2qkR7chFMSdryTjwB15BKd4CH5Hwn9PbzasZo1Qx6Bwg82nixMPKRQobDy3JW7cLmzMHi97hJk1JSDqhwUgM
↪",
        "fee": 0,
        "timestamp": 1606211535606,
        "signature":
↪"62xS2qkR7chFMSdryTjwB15BKd4CH5Hwn9PbzasZo1Qx6Bwg82nixMPKRQobDy3JW7cLmzMHi97hJk1JSDqhwUgM
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "role": "contract_developer"
      },
      {
```


(continues on next page)

(continued from previous page)


```
        "type": 101,
        "id":
↪"2sNwzGbwDL2Es53P8XY5wA9T9wwu3eXJbJUrtXJ9wg49urPjuBejWbidat2z3yZ8JrTpkWWFEsrerCtnC38XuRTJ
↪",
        "fee": 0,
        "timestamp": 1606211535605,
        "signature":
↪"2sNwzGbwDL2Es53P8XY5wA9T9wwu3eXJbJUrtXJ9wg49urPjuBejWbidat2z3yZ8JrTpkWWFEsrerCtnC38XuRTJ
↪",
        "target": "3MufokZsFzaf7heTV1yreUtm1uoJXPoFzdP",
        "role": "issuer"
      },
      {
        "type": 110,
        "id":
↪"4hLep3GngPEBH2xEmuUZ323muT8BstFdT552e42z6ZXCKGnF1PABGGjEiCkHfr6hMuyvRJ7axD9qoGeWQCU5yaCk
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"4hLep3GngPEBH2xEmuUZ323muT8BstFdT552e42z6ZXCKGnF1PABGGjEiCkHfr6hMuyvRJ7axD9qoGeWQCU5yaCk
↪",
        "targetPublicKey":
↪"5nGi8XoiGjjyjbPmjLNy1k2bus4yXLaeuA3Hb7BikwD9tboFwFXJYUmto5Joox76c3pp2Mr1LjgodUJuxryCJofQ
↪",
        "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c"
      },
      {
        "type": 101,
        "id":
↪"nj9Xfqm3pPLmuLsWfDZx4htKaNKAyvhen7tF95T9YwdmK1pqkiCjtaV9AxCwzEceViyo5rHPapigxPyCZdBWvRn
↪",
        "fee": 0,
        "timestamp": 1606211535604,
        "signature":
↪"nj9Xfqm3pPLmuLsWfDZx4htKaNKAyvhen7tF95T9YwdmK1pqkiCjtaV9AxCwzEceViyo5rHPapigxPyCZdBWvRn
↪",
        "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
        "role": "permissioner"
      },
      {
        "type": 101,
        "id":
↪"24AmxdGyH3afYRxPXn5zqvU1Fro1MwVQPDqwkdjCKLddSEiKVhyeMHTAVrRpHu83ZDPMyQkf3ty161PrujmGYtef
↪",
        "fee": 0,
        "timestamp": 1606211535603,
        "signature":
↪"24AmxdGyH3afYRxPXn5zqvU1Fro1MwVQPDqwkdjCKLddSEiKVhyeMHTAVrRpHu83ZDPMyQkf3ty161PrujmGYtef
↪",
        "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
        "role": "miner"
```


(continues on next page)

(continued from previous page)

```
    },
    {
      "type": 101,
      "id":
↪"4xsEQoh6Z4wDW6jT9UP3SqA1Yv5trbaGfF4uHajWxayBU8hrw2ZAYmtAWwDFytTdc6yqDepj6GwzxZuFYTq6638v
↪",
      "fee": 0,
      "timestamp": 1606211535602,
      "signature":
↪"4xsEQoh6Z4wDW6jT9UP3SqA1Yv5trbaGfF4uHajWxayBU8hrw2ZAYmtAWwDFytTdc6yqDepj6GwzxZuFYTq6638v
↪",
      "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
      "role": "connection_manager"
    },
    {
      "type": 101,
      "id":
↪"FSNaHMC11W3VskpGYfgxt3fqAMvt6gUmgy61CX8mm93QykuRp2E9Z8BtQc8w22Awc6W8CpXGJn6VcpkcBdAx4Tj
↪",
      "fee": 0,
      "timestamp": 1606211535601,
      "signature":
↪"FSNaHMC11W3VskpGYfgxt3fqAMvt6gUmgy61CX8mm93QykuRp2E9Z8BtQc8w22Awc6W8CpXGJn6VcpkcBdAx4Tj
↪",
      "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
      "role": "contract_developer"
    },
    {
      "type": 101,
      "id":
↪"4rfDMTGjbHENy3uiACLmfAHFJWyouhridZHGpynfV8S6aX3XmZHjUSfCvadn3KSzb8eHRq1kmzEaLMxvbqWkUKBY
↪",
      "fee": 0,
      "timestamp": 1606211535600,
      "signature":
↪"4rfDMTGjbHENy3uiACLmfAHFJWyouhridZHGpynfV8S6aX3XmZHjUSfCvadn3KSzb8eHRq1kmzEaLMxvbqWkUKBY
↪",
      "target": "3Mv79dyPX2cvLtRXn1MDDWiCZMBrkw9d97c",
      "role": "issuer"
    },
    {
      "type": 110,
      "id":
↪"4q5iXHv8jZ1qw5FptfBCz1cic14u1M4zCzE1i5qqEA4z6TQmeVFaqhZRpepFpdyGiSyKH4s6XqKPTgxuEJ8Sp4QQ
↪",
      "fee": 0,
      "timestamp": 1606211535610,
      "signature":
↪"4q5iXHv8jZ1qw5FptfBCz1cic14u1M4zCzE1i5qqEA4z6TQmeVFaqhZRpepFpdyGiSyKH4s6XqKPTgxuEJ8Sp4QQ
↪",
      "targetPublicKey":
↪"25GXtqKBAHTCrHuDoXvwQGXnHKBdeVcjdLvSmQ7SVFq4FDoMWzV78oRkgoS32AFDQ23DvfGFX6QpRkQRShQ4zMJy
```

(continues on next page)

(continued from previous page)


```
↪",
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ"
      },
      {
        "type": 101,
        "id":
↪"2gjzK3qSp89ywXCjEpvCHKSeyqoBYR2XCKegZ1ngGrQF8cDGXjA19HN8eYTgw8DRoXy62MM138EXXiZyV7oCaZrt
↪",
        "fee": 0,
        "timestamp": 1606211535599,
        "signature":
↪"2gjzK3qSp89ywXCjEpvCHKSeyqoBYR2XCKegZ1ngGrQF8cDGXjA19HN8eYTgw8DRoXy62MM138EXXiZyV7oCaZrt
↪",
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "role": "permissioner"
      },
      {
        "type": 101,
        "id":
↪"3zq1bCbeiNt4Z35rVtKwPo2MnW8peEcx2fQtgMseiJSb3TN7TKfU9auLEWKAgRXoNjpbpi9XA4aJw8Ly4gcpEaTv
↪",
        "fee": 0,
        "timestamp": 1606211535598,
        "signature":
↪"3zq1bCbeiNt4Z35rVtKwPo2MnW8peEcx2fQtgMseiJSb3TN7TKfU9auLEWKAgRXoNjpbpi9XA4aJw8Ly4gcpEaTv
↪",
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "role": "miner"
      },
      {
        "type": 101,
        "id":
↪"Aikgzt9ChSDfK4foF9oQJ8qRjV5cRyqF9okU9gr9JdpXh2LpyVB7GW4XSjmyc4MK9btPh3xd2whFDoCr8J5F4Hs
↪",
        "fee": 0,
        "timestamp": 1606211535597,
        "signature":
↪"Aikgzt9ChSDfK4foF9oQJ8qRjV5cRyqF9okU9gr9JdpXh2LpyVB7GW4XSjmyc4MK9btPh3xd2whFDoCr8J5F4Hs
↪",
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "role": "connection_manager"
      },
      {
        "type": 101,
        "id":
↪"48EGdWC133vQeydqMSXjmXJKB6L2brnu8Sh5W8r4anKCaUQZp5iKGrpVUAwsiUHfHrMXGA52roeoqo7abUHQbbVw
↪",
        "fee": 0,
        "timestamp": 1606211535596,
        "signature":
↪"48EGdWC133vQeydqMSXjmXJKB6L2brnu8Sh5W8r4anKCaUQZp5iKGrpVUAwsiUHfHrMXGA52roeoqo7abUHQbbVw
↪",
```


(continues on next page)

(continued from previous page)


```
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "role": "contract_developer"
      },
      {
        "type": 101,
        "id":
↪"FwNbJyr2Est9DFi5uch1ZfkQjDg13asqSsAdm37381aMWMrdaxcjqXMpKus1rxDcxZd5YnD4MNkz1ZpPgZ8nupn
↪",
        "fee": 0,
        "timestamp": 1606211535595,
        "signature":
↪"FwNbJyr2Est9DFi5uch1ZfkQjDg13asqSsAdm37381aMWMrdaxcjqXMpKus1rxDcxZd5YnD4MNkz1ZpPgZ8nupn
↪",
        "target": "3N9nNFySk1zVSVf9DUWR9DiBA1jEmmDDpaJ",
        "role": "issuer"
      },
      {
        "type": 110,
        "id":
↪"ps5vGHxv4DfTFnTXsqeS22hXQQm8uBf1mwnc7gtDvGxGGfEhDq8DvnCjtKukYmuEW6adz5NQGLbaqbMJK7ChYdA
↪",
        "fee": 0,
        "timestamp": 1606211535610,
        "signature":
↪"ps5vGHxv4DfTFnTXsqeS22hXQQm8uBf1mwnc7gtDvGxGGfEhDq8DvnCjtKukYmuEW6adz5NQGLbaqbMJK7ChYdA
↪",
        "targetPublicKey":
↪"5fbBNmkW9LJBUFNJW6vsjnmBzGf2AMwdqgHNvne2iYPMNW2wtDJGmF4PGnqyzTYJyYN3kWNWd4cFf9xBZ8Qi9Hki
↪",
        "target": "3N3jgxvmSsBBV4oz9BcKhT8War1em2sKoJn"
      },
      {
        "type": 101,
        "id":
↪"5BG3AhFnGbDcSDJ88KmXViU2tCxs4VNhXGjgocn2ZCcvCjTbxGjso4DKPkcajUNJBhPZHqgMmEKugVxqBMjNf2YY
↪",
        "fee": 0,
        "timestamp": 1606211535594,
        "signature":
↪"5BG3AhFnGbDcSDJ88KmXViU2tCxs4VNhXGjgocn2ZCcvCjTbxGjso4DKPkcajUNJBhPZHqgMmEKugVxqBMjNf2YY
↪",
        "target": "3N3jgxvmSsBBV4oz9BcKhT8War1em2sKoJn",
        "role": "permissioner"
      },
      {
        "type": 101,
        "id":
↪"HYoFXRgsyHGTa9JTnCDpJtBu6hr61LTYTA2zGPkUAVaTn6mhHfSKoVJbn91DN2gtqZxNreQnrV4GGnMR4cFikAE
↪",
        "fee": 0,
        "timestamp": 1606211535593,
        "signature":
```


(continues on next page)

(continued from previous page)


```
↪"HYoFXRgsyHGTa9JTnCDpJtBu6hr61LTYTA2zGPkUAVaTn6mhHfSKoVJbn91DN2gtqZxNreQnrV4GGnMR4cFikAE
↪",
        "target": "3N3jgxvmSsBBV4oz9BcKhT8War1em2sKoJn",
        "role": "contract_developer"
      },
      {
        "type": 101,
        "id":
↪"4snBMYD3dDw9pivJM2YFSJBPPtK4K43YGL8Qjw4APadgZCtqsR4yoo3CZC4bgf5ZffwVWQQzVmfSjxpzsiwCjNju
↪",
        "fee": 0,
        "timestamp": 1606211535592,
        "signature":
↪"4snBMYD3dDw9pivJM2YFSJBPPtK4K43YGL8Qjw4APadgZCtqsR4yoo3CZC4bgf5ZffwVWQQzVmfSjxpzsiwCjNju
↪",
        "target": "3N3jgxvmSsBBV4oz9BcKhT8War1em2sKoJn",
        "role": "issuer"
      }
    ],
    "version": 1,
    "poa-consensus": {
      "overall-skipped-rounds": 0
    },
    "timestamp": 1606211535610,
    "height": 1
  }
```


GET /blocks/last

The method returns the contents of the current block of the blockchain.

The current block is in the process of creation, until it is accepted by the miner nodes, the number of transactions in it may vary.

Parameters returned in the method response:

- `reference` - hash sum of the block;
- `blocksize` - size of the block;
- `features` - *features* running at the time of block creation;
- `signature` - block signature;
- `fee` - fees for transactions included in the block;
- ``generator" - address of creator of the block;
- `transactionCount` - number of transactions included in genesis block;
- `transactions` - array with bodies of transactions included in the block;;
- `version` - version of the block;
- `poa-consensus.overall-skipped-rounds` - number of missed mining rounds, when using the *PoA* consensus algorithm;
- `timestamp` - **Unix Timestamp** of the block (in milliseconds);

- `height` - height of the block.

Response example for an empty current block:

GET /blocks/last:


```
{
  "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
}
```


GET /blocks/at/–height″

The method returns the contents of the block at `height`.

Parameters returned in the method response:

- `reference` - hash sum of the block;
- `blocksize` - size of the block;
- `features` - *features* running at the time of block creation;
- `signature` - block signature;
- `fee` - fees for transactions included in the block;
- ``generator" - address of creator of the block;
- `transactionCount` - the number of transactions included in the block;
- `transactions` - array with bodies of transactions included in the block;;
- `version` - version of the block;
- `poa-consensus.overall-skipped-rounds` - number of missed mining rounds, when using the *PoA* consensus algorithm;
- `timestamp` - **Unix Timestamp** of the block (in milliseconds);
- `height` - height of the block.

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
}
```


GET /blocks/seq/–from″/–to″

The method returns the contents of blocks from height `{from}` to height `{to}`.

Parameters identical to the `GET /blocks/at/{height}` method are returned for each block.

GET /blocks/seqext/–from″/–to″

The method returns the contents of blocks with extended transaction information from height `{from}` to height `{to}`.

Other parameters returned for each block are identical to the `GET /blocks/at/{height}` method.

GET /blocks/signature/–signature″

The method returns the block content by its `{signature}`.

Parameters returned for each block are identical to the `GET /blocks/at/{height}` method.

GET /blocks/address/–address″/–from″/–to″

The method returns the contents of all blocks generated by the `{address}` from height `{from}` to height `{to}`.

The method response returns parameters identical to the `GET /blocks/at/{height}` method for each block.

GET /blocks/child/–signature″

The method returns an inherited block from the block with `{signature}`.

Parameters returned for each block are identical to the `GET /blocks/at/{height}` method.

GET /blocks/headers/at/–height″

The method returns the header of the block at `height`.

Parameters returned in the method response:

- `reference` - hash sum of the block;
- `blocksize` - size of the block;
- `features` - *features* running at the time of block creation;
- `signature` - block signature;
- `fee` - fees for transactions included in the block;
- ``generator" - address of creator of the block;
- `pos-consensus.base-target` - the coefficient adjusting the block release time when using the *PoS* consensus algorithm;
- `pos-consensus.generation-signature` - the signature needed to validate the block miner;
- `poa-consensus.overall-skipped-rounds` - number of missed mining rounds, when using the *PoA* consensus algorithm;
- `version` - version of the block;
- `timestamp` - **Unix Timestamp** of the block (in milliseconds);
- `height` - height of the block.

Response example:

GET /blocks/at/–height″:


```
{
  "reference":
↪"5qWJh9aQ2hkwnBWygGYmrBhzMe5inRZ2r6WhEXz3VJsiMtASWkvbsVeZGychZKzcPDbWmpzdhQwNQJ19PfK2dst9
↪",
  "blocksize": 589,
  "features": [
    0
  ],
  "signature":
↪"4U4Hmg4mDYrvxaZ3JVzL1Z1piPDZ1PJ61vd1PeS7ESZFkHsUCUqeeAZoszTVr43Z4NV44dqbLv9WdrLytDL6gHuv
↪",
  "fee": 5000000,
  "generator": "3NkZd8Xd4KsuPiNVsuphRNCZE3SqJycqv8d",
  "pos-consensus": {
    "base-target": 249912231,
    "generation-signature": "LM83w6eWQHnLJF2D9RQNdNcHAdnZLCLWrn5bfcoqcZy"
  },
```


(continues on next page)

(continued from previous page)


```
  "poa-consensus": {
    "overall-skipped-rounds": 2
  },
  "transactionCount": 2,
  "version": 12,
  "timestamp": 1568287320962,
  "height": 48260
}
```


GET /blocks/headers/seq/–from″/–to″

The method returns the headers of blocks from height `{from}` to height `{to}`.

The method response returns parameters identical to the `GET /headers/at/{height}` method for each block.

GET /blocks/headers/last

The method returns the header of the current block.

The method response returns parameters identical to the `GET /headers/at/{height}` method for each block.

See also

REST API methods

10.2.12 REST API: information about permissions of participants

The methods of the `permissions` group are used to obtain information about the roles of participants on the network.

For more information on participant permissions, see the article *Permissions*.

GET /permissions/–address″

The method returns information about the active permissions of the `{address}`, as well as the request generation time in Unix Timestamp format (in milliseconds).

Response example:

GET /permissions/–address″:


```
{
  "roles": [
      {
      "role": "miner"
      },
      {
      "role": "permissioner"
      }
  ],
  "timestamp": 1544703449430
}
```


GET /permissions/–address″/at/–timestamp″

The method returns information about participant roles of an `{address}`, active for a `{timestamp}`. The time is specified in Unix Timestamp format (in milliseconds).

Response example:

GET /permissions/–address″/at/–timestamp″:


```
{
  "roles": [
      {
      "role": "miner"
      },
      {
      "role": "permissioner"
      }
  ],
  "timestamp": 1544703449430
}
```


POST /permissions/addresses

The method returns roles for multiple addresses that are active at the specified point in time.

The method query contains the following data:

- `addresses` - list of addresses as an array of strings;
- `timestamp` - Unix Timestamp (in milliseconds).

Example of a query with two addresses:

POST /permissions/addresses:

```
{
    "addresses": [
        "3N2cQFfUDzG2iujBrFTnD2TAsCNohDxYu8w", "3Mx5sDq4NXef1BRzJRAofa3orYFxLanxmd7"
    ],
    "timestamp": 1544703449430
}
```

The method response returns an array of data `addressToRoles`, which contains the roles for each address as well as the `timestamp`.

Response example for two addresses:

POST /permissions/addresses:

```
{
    "addressToRoles": [
        {
        "address": "3N2cQFfUDzG2iujBrFTnD2TAsCNohDxYu8w",
        "roles": [
            {
              "role": "miner"
            },
            {
              "role": "permissioner"
            }
        ]
        },
        {
        "address": "3Mx5sDq4NXef1BRzJRAofa3orYFxLanxmd7",
        "roles": [
            {
              "role": "miner"
            }
        ]
        }
    ],
    "timestamp": 1544703449430
}
```

See also

REST API methods

Permissions

Permission management

10.2.13 REST API: information about address assets and balances

The methods of the `assets` group are provided for obtaining information about assets and address balances.

GET /assets/balance/–address″

The method returns the balance of all assets of the address.

Parameters returned in the method response:

- `address` - participant address;
- `balances` - object with balances of the participant:
  - `assetId` - asset identifier;
  - `balance` - asset balance;
  - `quantity` - total number of issued tokens of the asset;
  - `reissuable` - the reissuability of an asset;
  - `issueTransaction` - body of *transaction 3* for asset creation;
  - `minSponsoredAssetFee` - minimum fee for sponsored transactions;
  - `sponsorBalance` - funds allocated to pay for transactions on sponsored assets.

Response example:

GET /assets/balance/–address″:

```
{
  "address": "3Mv61qe6egMSjRDZiiuvJDnf3Q1qW9tTZDB",
  "balances": [
    {
      "assetId": "Ax9T4grFxx5m3KPUEKjMdnQkCKtBktf694wU2wJYvQUD",
      "balance": 4879179221,
      "quantity": 48791792210,
      "reissuable": true,
      "minSponsoredAssetFee" : 100,
      "sponsorBalance" : 1233221,
      "issueTransaction" : {
         "type" : 3,
         ...
      }
    },
    {
      "assetId": "49KfHPJcKvSAvNKwM7CTofjKHzL87SaSx8eyADBjv5Wi",
      "balance": 10,
      "quantity": 10000000000,
      "reissuable": false,
      "issueTransaction" : {
         "type" : 3,
         ...
```

(continues on next page)

(continued from previous page)

```
      }
    }
  ]
}
```

POST /assets/balance

The method returns the balance of all assets for several addresses passed to the `addresses` field. Parameters passed in response for each address are identical to the `GET /assets/balance/{address}` method.

Response example for one address:

GET /assets/balance/–address″:

```
{
  "address": "3Mv61qe6egMSjRDZiiuvJDnf3Q1qW9tTZDB",
  "balances": [
    {
      "assetId": "Ax9T4grFxx5m3KPUEKjMdnQkCKtBktf694wU2wJYvQUD",
      "balance": 4879179221,
      "quantity": 48791792210,
      "reissuable": true,
      "minSponsoredAssetFee" : 100,
      "sponsorBalance" : 1233221,
      "issueTransaction" : {
         "type" : 3,
         ...
      }
    },
    {
      "assetId": "49KfHPJcKvSAvNKwM7CTofjKHzL87SaSx8eyADBjv5Wi",
      "balance": 10,
      "quantity": 10000000000,
      "reissuable": false,
      "issueTransaction" : {
         "type" : 3,
         ...
      }
    }
  ]
}
```

GET /assets/balance/–address"/–assetId"

The method returns the balance of the address in the specified `{assetId}`.

Response example:

GET /assets/balance/–address"/–assetId":


```
{
  "address": "3Mv61qe6egMSjRDZiiuvJDnf3Q1qW9tTZDB",
  "assetId": "Ax9T4grFxx5m3KPUEKjMdnQkCKtBktf694wU2wJYvQUD",
  "balance": 4879179221
}
```


GET /assets/details/–assetId"

The method returns description of the specified `{assetId}`.

Response example:

GET /assets/details/–assetId":


```
{
  "assetId" : "8tdULCMr598Kn2dUaKwHkvsNyFbDB1Uj5NxvVRTQRnMQ",
  "issueHeight" : 140194,
  "issueTimestamp" : 1504015013373,
  "issuer" : "3NCBMxgdghg4tUhEEffSXy11L6hUi6fcBpd",
  "name" : "name",
  "description" : "Sponsored asset",
  "decimals" : 1,
  "reissuable" : true,
  "quantity" : 1221905614,
  "script" : null,
  "scriptText" : null,
  "complexity" : 0,
  "extraFee": 0,
  "minSponsoredAssetFee" : 100000
}
```


GET /assets/–assetId"/distribution

The method returns the number of asset tokens on all addresses using the specified asset.

Response example:

GET /assets/details/–assetId”:

```
{
  "3P8GxcTEyZtG6LEfnn9knp9wu8uLKrAFHCb": 1,
  "3P2voHxcJg79csj4YspNq1akepX8TSmGhTE": 1200
}
```

See also

REST API methods

10.2.14 REST API: blockchain peers

A group of `peers` methods is provided to work with blockchain peer nodes:

POST /peers/connect

The method is designed to connect a new participant node to your node.

Query example:

POST /peers/connect:

```
{
    "host":"127.0.0.1",
    "port":"9084"
}
```

Response example:

POST /peers/connect:

```
{
  "hostname": "localhost",
  "status": "Trying to connect"
}
```

GET /peers/connected

The method returns a list of connected nodes.

Response example:

GET /peers/connected:


```
{
  "peers": [
    {
      "address": "52.51.92.182/52.51.92.182:6863",
      "declaredAddress": "N/A",
      "peerName": "zx 182",
      "peerNonce": 183759
    },
    {
      "address": "ec2-52-28-66-217.eu-central-1.compute.amazonaws.com/52.28.66.217:6863",
      "declaredAddress": "N/A",
      "peerName": "zx 217",
      "peerNonce": 1021800
    }
  ]
}
```


GET /peers/all

The method returns a list of all known nodes.

Response example:

GET /peers/all:


```
{
  "peers": [
    {
      "address": "/13.80.103.153:6864",
      "lastSeen": 1544704874714
    }
  ]
}
```


GET /peers/suspended

The method returns a list of suspended nodes.

Response example:

GET /peers/suspended:

```
[
  {
    "hostname": "/13.80.103.153",
    "timestamp": 1544704754619
  }
]
```

POST /peers/identity

The method returns the public key of the node to which your node is connected for confidential data transfer.

The following parameters are passed in the request:

- `address` - blockchain address that corresponds to the `privacy.owner-address` parameter in the node configuration file;
- `signature` - digital signature from the value of the `address` field.

Query example:

POST /peers/identity:

```
{
  "address": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8",
  "signature":
↪"6RwMUQcwrxtKDgM4ANes9Amu5EJgyfF9Bo6nTpXyD89ZKMAcpCM97igbWf2MmLXLdqNxdsUc68fd5TyRBEB6nqf
↪"
}
```

The method response contains the `publicKey` parameter - the public key of the node associated with the `privacy.owner-address` parameter in its configuration file. If the *handshakes* check mode is disabled, the publicKey parameter is not shown.

Response example:

POST /peers/identity:

```
{
  "publicKey": "3NBVqYXrapgJP9atQccdBPAgJPwHDKkh6A8"
}
```

GET /peers/hostname/–address″

The method returns the host name and IP address of the node by its address in the Waves Enterprise network.

Response example:

GET /peers/hostname/–address″:

```
{
  "hostname": "node1.we.io",
  "ip": "10.0.0.1"
}
```

GET /peers/allowedNodes

Obtaining the current list of allowed network members at the time of the request.

GET /peers/allowedNodes:

```
{
  "allowedNodes": [
    {
      "address": "3JNLQYuHYSHZiHr5KjJ89wwFJpDMdrAEJpj",
      "publicKey": "Gt3o1ghh2M2TS65UrHZCTJ82LLcMcBrxuaJyrgsLk5VY"
    },
    {
      "address": "3JLp8wt7rEUdn4Cca5Hp9jZ7w8T5XDAKicd",
      "publicKey": "J3ffCciVu3sustgb5vxmEHczACMR89Vty5ZBLbPn9xyg"
    },
    {
      "address": "3JRY1cp7atRMBd8QQoswRpH7DLawM5Pnk3L",
      "publicKey": "5vn4UcB9En1XgY6w2N6e9W7bqFshG4SL2RLFqEWEbWxG"
    }
  ],
  "timestamp": 1558697649489
}
```

See also

REST API methods

10.2.15 REST API: hash calculation, working with scripts and sending auxiliary queries

For hashing, scripting and sending auxiliary requests to the node, there is a group of ``utils" methods:

Hashing: utils/hash

POST /utils/hash/fast

The method returns the hash sum of the string passed in the query.

Response example:

POST /utils/hash/fast:

```
{
  "message": "ridethewaves!",
  "hash": "DJ35ymschUFDmqCnDJewjcnVExVkWgX7mJDXhFy9X8oQ"
}
```

POST /utils/hash/secure

The method returns the double hash sum of the string passed in the query.

Response example:

POST /utils/hash/secure:

```
{
  "message": "ridethewaves!",
  "hash": "H6nsiifwYKYEx6YzYD7woP1XCn72RVvx6tC1zjjLXqsu"
}
```

Working with scripts: utils/script

This group of methods is designed to convert script code into **base64** format and decode them. Scripts are bound to accounts using the *13* transactions (binding a script to an address) and *15* (binding a script to an asset for an address).

POST /utils/script/compile

The method converts the script code to **base64** format.

Query example:

POST /utils/script/compile:

```
let x = 1
(x + 1) == 2
```

Parameters returned in the method response:

- `script` - script body in **base64** format;
- `complexity` - a number from 1 to 100 representing the amount of computing resources required to execute the script;
- `extraFee` - fee for outgoing transactions set by the script.

Response example:

POST /utils/script/compile:

```
{
  "script":
↪"3rbFDtbPwAvSp2vBvqGfGR9nRS1nBVnfuSCN3HxSZ7fVRpt3tuFG5JSmyTmvHPxYf34SocMRkRKFgzTtXXnnv7upRHXJzZrLSQo8
↪",
  "complexity": 11,
  "extraFee": 10001
}
```

POST /utils/script/estimate

The method is designed to decode and evaluate the complexity of a script passed in a request in **base64** format.

Parameters returned in the method response:

- `script` - script body in **base64** format;
- `scriptText` - the source code of the script;
- `complexity` - a number from 1 to 100 representing the amount of computing resources required to execute the script;
- `extraFee` - fee for outgoing transactions set by the script.

Response example:

POST /utils/script/compile:

```
{
  "script":
↪"3rbFDtbPwAvSp2vBvqGfGR9nRS1nBVnfuSCN3HxSZ7fVRpt3tuFG5JSmyTmvHPxYf34SocMRkRKFgzTtXXnnv7upRHXJzZrLSQo8
↪",
  "scriptText": "FUNCTION_CALL(FunctionHeader(==,List(LONG, LONG)),List(CONST_LONG(1),␣
↪CONST_LONG(2)),BOOLEAN)",
  "complexity": 11,
  "extraFee": 10001
}
```

Auxiliary queries

GET /utils/time

The method returns the current node time in two formats:

- `system` - system time of the node PC;
- `ntp` - network time.

Response example:

POST /utils/script/compile:

```
{
   "system": 1544715343390,
   "NTP": 1544715343390
}
```

POST /utils/reload-wallet

The method reloads a node's keystore. It applies if a new key pair was added to the keystore without restarting the node.

Response example:

POST /utils/reload-wallet:

```
{
   "message": "Wallet reloaded successfully"
}
```

See also

REST API methods

10.2.16 REST API: blockchain debug

Methods of the `debug` group are provided for debugging the blockchain network:

GET /debug/blocks/–howMany″

The method displays the size and full hash of the last blocks. The number of blocks is specified when prompted.

Response example:

GET /debug/blocks/–howMany″:


```
[
    {
        "226": "7CkZxrAjU8bnat8CjVAPagobNYazyv1HASubmp7YYqGe"
    },
    {
        "226": "GS3y9fUHAKCamq52TPsjizDVir8J7iGoe8P2XZLasxsC"
    },
    {
        "226": "B9LmhGGDdvcfUA9JEWvyVrT9sazZE6gibpAN13xUN7KV"
    },
    {
        "226": "Byb9MHtwYf3MFyi2tbhQ3GTdCct5phKq9REkbjQTzdne"
    },
    {
        "226": "HSxSHbiV4tZc8RaN6jxdhgtkAhjxuLn76uHxerMRUefA"
    }
]
```


GET /debug/info

The method displays general information about the blockchain needed for debugging and testing.

Response example:

GET /debug/info:


```
{
    "stateHeight": 74015,
    "extensionLoaderState": "State(Idle)",
    "historyReplierCacheSizes": {
        "blocks": 13,
        "microBlocks": 2
    },
    "microBlockSynchronizerCacheSizes": {
        "microBlockOwners": 0,
        "nextInventories": 0,
        "awaiting": 0,
        "successfullyReceived": 0
    },
    "scoreObserverStats": {
        "localScore": 42142328633037120000,
        "scoresCacheSize": 4
    },
    "minerState": "mining microblocks"
}
```


POST /debug/rollback

The method rolls the blockchain back to the specified height, removing all blocks after it. The following parameters are passed in the request:

- `rollbackTo` - the height to which the blockchain must be rolled back;
- `returnTransactionsToUtx` - return transactions that are contained in the rollback blocks to the UTX pool: `true` - return, `false` - delete.

Examples of a query and a response:

POST /debug/rollback:

Query:

```
{
    "rollbackTo": 100,
    "returnTransactionsToUtx": true
}
```

Response:


```
{
    "BlockId":
↪"4U4Hmg4mDYrvxaZ3JVzL1Z1piPDZ1PJ61vd1PeS7ESZFkHsUCUqeeAZoszTVr43Z4NV44dqbLv9WdrLytDL6gHuv
↪"
}
```

POST /debug/validate

The method validates transactions by their identifier and measures the time spent in milliseconds. The `id` of the transaction is passed in the query.

Response example:

POST /debug/validate:

```
{
    "valid": false,
    "validationTime": 14444
}
```

GET /debug/minerInfo

The method returns information about the miner.

Response example:

GET /debug/minerInfo:

```
[
    {
        "address": "3JFR1pmL6biTzr9oa63gJcjZ8ih429KD3aF",
        "miningBalance": 1248959867200000,
        "timestamp": 1585923248329
    }
]
```

GET /debug/historyInfo

The method displays the history of the last block.

Response example:

GET /debug/historyInfo:

```
{
"lastBlockIds": [
"37P4fvexYHPUzNPRRqYbRYxGz7x3r5jFznck7amaS6aWnHL5oQqrqCzsSh1HvYKnd2ZhU6n6sWYPb3hxsY8FBfmZ
↪",
"5RRu1qtesz4KvrVp4fxzQHebq2fRanNsg3HJKwD4uChqySm7vFHCdHKU6iZYXJDVmfSxiE9Maeb6sM2JireaWLbx
↪",
"3Lo27JfjekcZnJsYEe7st7evDZ6TgmCUBtiZrSxUCobKL48DZQ4dXMfp89WYjEykH15HEHSXzqMSTQigE8vEcN2r
↪",
"r4RuxEXAqgfDMKVXRWmZcGMaWKDsAvVxfXDtw8d6bamLR61J1gaoesargYSoZQqRbDrBcefLprk7D78fA728719
↪",
"3F4Up46crZbpKVWUeieL6GeSrVMYm7JJ7aX6aHD6B8wedFggSKv8d3H39Qy9MLEauFBU9m3qZV1U8emhmnqwmLbg
```

(continues on next page)

(continued from previous page)


```
↪",
"QSuBkEtVe9nik5T5S33ogeCbgKy7ihBkS2pwYayK23m4ANier83ThpajEzvpbyPy9pPWZc5St8mYUKxXDscKuRC
↪",
"4udpNnz3e1M1GbVZxtwfg8gpF6EbiKxRCRBwi6iRMyLsvh5J2Ec9Wqyu2sq2KYL75o12yiP8TszworeUfuxNmJ5g
↪",
"5BZYZ4RZAJjM5KKCaHpyUsXnb4uunnM5kcfTojc5QzQo3vyP2w3YD4qrALizkkQQR4ziS77BoAGb56QCecUtHFFN
↪",
"5JwfLaF1oGxRXVCdDbFuKpxrvxgLCGU3kCFwxUhLL8G3xV211MrKBuAuQ4MaC5uN574uV9U8M6HfHTMERnfr5jGJ
↪",
"4bysMhz14E1rC7dLYScfVVqPmHqzi8jdhcnkruJmCNL86TwV2cbF7G9YVchvTrv9qbQZ7JQownV59gRRcD26zm16
↪"
],
"microBlockIds": []
}
```


GET /debug/configInfo

The method completely outputs the used node configuration file.

Response example:

GET /debug/configInfo:


```
{
  "node": {
    "anchoring": {
      "enable": "no"
 },
    "blockchain": {
      "consensus": {
        "type": "pos"
 },
      "custom": {
        "address-scheme-character": "K",
        "functionality": {
          "blocks-for-feature-activation": 10,
          "feature-check-blocks-period": 30,
          "pre-activated-features": {
            ...
    "wallet": {
      "file": "wallet.dat",
      "password": ""
    },
    "waves-crypto": "yes"
    }
  }
```

DELETE /debug/rollback-to/–signature″

The method rolls the blockchain up to the block with the specified `{signature}`.

Response example:

DELETE /debug/rollback-to/–signature″:

```
{
    "BlockId":
↪"4U4Hmg4mDYrvxaZ3JVzL1Z1piPDZ1PJ61vd1PeS7ESZFkHsUCUqeeAZoszTVr43Z4NV44dqbLv9WdrLytDL6gHuv
↪"
}
```

GET /debug/portfolios/–address″

The method displays the current balance of the transactions in the UTX pool of the `{address}` node.

Response example:

GET /debug/portfolios/–address″:

```
{
    "balance": 104665861710336,
    "lease": {
        "in": 0,
        "out": 0
    },
    "assets": {}
}
```

POST /debug/print

The method outputs the current messages of the logger that has a DEBUG logging level.

The answer is output in the `"message": "string"` format

GET /debug/state

The method displays the current state of the node.

Response example:

GET /debug/state:


```
{
    "3JD3qDmgL1icDaxa3n24YSjxr9Jze5MBVVs": 4899000000,
    "3JPWx147Xf3f9fE89YtfvRhtKWBHy9rWnMK": 17528100000,
    "3JU5tCoswHH7FKPBUowySWBnQwpbZiYyNhB": 300021381800000,
    "3JCJChsQ2CGyHc9Ymu8cnsES6YzjjJELu3a": 75000362600000,
    "3JEW9XnPC8w3qQ4AJyVTDBmsVUp32QKoCGD": 5000000000,
    "3JSaKNX94deXJkywQwTFgbigTxJa36TDVg3": 6847000000,
    "3JFR1pmL6biTzr9oa63gJcjZ8ih429KD3aF": 1248938560600000,
    "3JV6V4JEVc3a9uSqRmdUMvMKMfZa16HbGmq": 4770000000,
    "3JZtYeGEZHjb2zQ6EcSEo524PdafPn6vWkc": 900000000,
    "3JMMFLX9d1rmXaBK9AF7Wuwzu4vRkkoVQBC": 4670000000,
    "3JJDpPDqSPokKp5jEmzwMzmaPUyopLZjW1C": 800000000,
    "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t": 994280900000
}
```


GET /debug/stateWE/–height″

The method displays the node's state at the specified `{height}`.

Response example:

GET /debug/stateWE/–height″:


```
{
    "3JPWx147Xf3f9fE89YtfvRhtKWBHy9rWnMK": 17528100000,
    "3JU5tCoswHH7FKPBUowySWBnQwpbZiYyNhB": 300020907600000,
    "3JCJChsQ2CGyHc9Ymu8cnsES6YzjjJELu3a": 75000350600000,
    "3JSaKNX94deXJkywQwTFgbigTxJa36TDVg3": 6847000000,
    "3JFR1pmL6biTzr9oa63gJcjZ8ih429KD3aF": 1248960085800000,
    "3JWDUsqyJEkVa1aivNPP8VCAa5zGuxiwD9t": 994280900000
}
```


See also

REST API methods

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 a icon at the beginning of the article.

If authorization is not required, you will see a icon.

See also

Precise platform configuration: node gRPC and REST API configuration

CHAPTER

ELEVEN

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 article *Smart contracts*.

11.1 Preparing to work

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 Waves Enterprise Mainnet, it is configured by default 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. You can also load a Docker contract image from a repository not specified in the node configuration file using transaction 103, which initiates the creation of a smart contract. For more information, see *Development and installation of a smart contract* and *description of the transaction 103*.

When working in the Waves Enterprise Mainnet, the Waves Enterprise open registry is pre-installed in the configuration file.

11.2 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 smart contract authorization parameters (when using REST API) or used **protobuf** files (when using gRPC).

Examples of Python smart contract code using gRPC and REST 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:

11.2.1 Example of a smart contract with gRPC

This section describes an example of creating a simple smart contract in Python. The smart contract uses a 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:

```
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):
        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)
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
    # 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 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 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 in 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:

```
#!/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:

```
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. Contact the Waves Enterprise Technical Support team to place your smart contract in the public repository if you are working in the Waves Enterprise Mainnet.

If you work on a private network, *build your smart contract yourself and place it in your own registry*.

How a smart contract with gRPC works

Once called, the 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

11.2.2 Example of a smart contract with the use of REST API

Program description and listing

This section describes an example of creating and running a simple smart contract. The contract increments the number passed to it each time it is called.

Program listing:

```
import json
import os
import requests
import sys


def find_param_value(params, name):
    for param in params:
        if param['key'] == name: return param['value']
    return None


def print_success(results):
    print(json.dumps(results, separators=(',', ':')))


def print_error(message):
    print(message)
    sys.exit(3)


def get_value(contract_id):
    node = os.environ['NODE_API']
    if not node:
        print_error("Node REST API address is not defined")
    token = os.environ["API_TOKEN"]
    if not token:
        print_error("Node API token is not defined")
    headers = {'X-Contract-Api-Token': token}
    url = '{0}/internal/contracts/{1}/sum'.format(node, contract_id)
    r = requests.get(url, verify=False, timeout=2, headers=headers)
    data = r.json()
    return data['value']
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    command = os.environ['COMMAND']
    if command == 'CALL':
        contract_id = json.loads(os.environ['TX'])['contractId']
        value = get_value(contract_id)
        print_success([{
            "key": "sum",
            "type": "integer",
            "value": value + 1}])
    elif command == 'CREATE':
        print_success([{
            "key": "sum",
            "type": "integer",
            "value": 0}])
    else:
        print_error("Unknown command {0}".format(command))
```

Step-by-step description of the smart contract operation:

- The program expects to get a data structure in json format with a ``params" field;
- reads the value of the `a` field;
- returns the result as the value of the field "{a}+1" in json format.

Example of input parameters:

```
"params":[
    {
        "key":"a",
        "type":"integer",
        "value":1
    }
]
```

Authorization of a smart contract with REST API

To work with the *node REST API*, the smart contract needs authorization. For the smart contract to work correctly with the API methods, follow these steps:

1. The following parameters must be defined in the environment variables of the smart contract:
   - `NODE_API` - URL to the node REST API of;
   - `API_TOKEN` - authorization token for the smart contract;
   - `COMMAND` - commands to create and call a smart contract;
   - `TX` - transaction required for operation of a smart contract (*103* - *107*).
2. The smart contract developer assigns the value of the `API_TOKEN` variable to the `X-Contract-Api-Token` query header. In the `API_TOKEN` variable the node writes the JWT authorization token when the contract is created and executed.
3. The contract code must pass the received token in the request header (`X-Contract-Api-Token`) every time the API of the node is accessed.

Development of a smart contract

1. Place the **contract.py** file with the code in the directory that will contain your smart contract files.

2. Create a **run.sh** shell script, which will run the smart contract code in the container:

```
#!/bin/sh

python contract.py
```

Place the **run.sh** file in the root directory of your smart contract.

3. Create a **Dockerfile** script file to build and control the startup of your smart contract. When developing in Python, your smartcontract image can be based on the official Alpine Linux-based Python image `python:alpine3.8`.

Dockerfile example:

```
FROM python:alpine3.8
ADD contract.py /
ADD run.sh /
RUN chmod +x run.sh
CMD exec /bin/sh -c "trap : TERM INT; (while true; do sleep 1000; done) & wait"
```

Place the **Dockerfile** in the root directory of your smart contract.

4. Contact the Waves Enterprise Technical Support team to place your smart contract in the public repository if you are working in the Waves Enterprise Mainnet.

If you work on a private network, *build your smart contract yourself and place it in your own registry*.

See also

Development and usage of smart contracts

REST API methods

11.3 Uploading of a smart contract into a registry

Contact the Waves Enterprise Technical Support team to place your smart contract in the public repository if you are working in the Waves Enterprise Mainnet.

When working on a private network, upload a Docker image of the smart contract to your own registry:

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 an 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 transaction 103 when creating the smart contract.

11.4 Installing of a smart contract into the blockchain

After uploading the smart contract to the repository, install it on the network using the *103* 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, \
```

(continues on next page)

(continued from previous page)

```
    "image": "my-contract:latest", \
    "imageHash":
↪"7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65", \
    "contractName": "my-contract", \
    "params": [], \
    "height": 1619 \
}
}' 'http://my-node:6862/transactions/broadcast'
```

11.5 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,
```

(continues on next page)

(continued from previous page)

```
"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'
```

See also

Smart contracts

General platform configuration: execution of smart contracts

CHAPTER

TWELVE

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;
- GOST encryption standards;
- 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 instructions for installing it, are available at the Waves Enterprise GitHub repository.

12.1 Contents

12.1.1 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 "Using the JS SDK in a node with oAuth".

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 section "Options for creating a seed phrase".

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 :ref:` Transaction description <tx-list>`.

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 section "Examples of creating and sending transactions".

Cryptographic node methods used by the JavaScript SDK

Three cryptographic methods are available for the JavaScript SDK:

- `crypto/encryptCommon` - encryption of data for all recipients with a single key CEK, which in turn is wrapped by unique keys KEK for each recipient;
- `crypto/encryptSeparate` - encrypt text separately for each recipient with a unique key;
- `crypto/decrypt` - decrypt data, provided that the key of the message recipient is in the keystore of the node.

The **signature-generator** component also supports both GOST and Waves cryptography algorithms.

See also

JavaScript SDK

Description of transactions

REST API: encryption and decryption methods

12.1.2 JS SDK installation and initialization

If you are going to 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

12.1.3 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": 40000000000,
  "fee": 100000
}
```

A transaction body can be left blank and fill in 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.

Transactions *3*, *13*, *14* and *112* use the `description` text field, and transactions *4* and *6* 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': 'vostok'} });
}

(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,
```

(continues on next page)

(continued from previous page)

```
    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 a 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

Waves Enterprise Mainnet fees

12.1.4 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';
```

(continues on next page)

(continued from previous page)

```
const fetch = (url, options = {}) => {
  const headers = options.headers || {}
  return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key': 'vostok'} });
}

(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': 'vostok'} });
}

(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) {
```

(continues on next page)

(continued from previous page)

```
        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': 'vostok'} });
}

(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(),
    }
```

(continues on next page)

(continued from previous page)

```
    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

        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': 'vostok'} });
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
↪config`)).json();
```

(continues on next page)

(continued from previous page)

```
    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: 'vostok-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';
```

(continues on next page)

(continued from previous page)

```
const fetch = (url, options = {}) => {
    const headers = options.headers || {}
    return nodeFetch(url, { ...options, headers: {...headers, 'x-api-key': 'vostok'} });
}

(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': 'vostok'} });
}

(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(),
```

(continues on next page)

(continued from previous page)

```
  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":""
  },
```

(continues on next page)

(continued from previous page)

```
    "fee": 5000000,
    "password": "sfgKYBFCF@#$fsdf()*%",
    "timestamp": Date.now(),
    "version": 3,
    "apiKey": 'vostok',
  }
  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': 'vostok'} });
}

(async () => {
    const { chainId, minimumFee, gostCrypto } = await (await fetch(`${nodeAddress}/node/
↪config`)).json();

    const wavesApiConfig = {
        ...MAINNET_CONFIG,
```

(continues on next page)

(continued from previous page)

```
        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);
```

(continues on next page)

(continued from previous page)

```
        console.log('Broadcast BURN result: ', burnResult)
    } catch (err) {
        console.log('Broadcast error:', err)
    }

})();
```

See also

JavaScript SDK

12.1.5 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

12.1.6 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

CHAPTER

THIRTEEN

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 divided into groups with access to confidential data.

13.1 Creation of a confidential data group

Anyone on the network can create a confidential data access group. Before you create an access group, decide on the list of members that will be part of it. Then sign and submit the transaction *112 CreatePolicy*:

1. In the `recipients` field, enter the comma-separated addresses of participants who will have access to confidential data.
2. In the `owners` field, add the comma-separated addresses of the group members who will be given administrator rights. The administrators of the access group, in addition to accessing confidential data, will be able to change the composition of the access group.

When you send a transaction, you will receive the ID of the created access group (`policyId`). You will need it when you change the composition of its 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 its composition, as will the participants added to the `owners` field.

13.2 Updating a confidential data group

Only members of a confidential data group added to the `owners` field when creating the group, as well as its creator himself (**group owners**) can change the composition of the access group.

To do this, sign and submit the transaction *113 UpdatePolicy*:

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.

13.3 Sending confidential data into the network

REST API methods **POST /privacy/sendData** and **POST /privacy/sendDataV2** are used to send confidential data to the network. These methods require authorization.

With the **POST /privacy/sendData** and **POST /privacy/sendDataV2** methods, you can send data up to **20 megabytes**.

When sending data, include the following information in your request:

- `sender` - blockchain address from which the data should be sent (corresponds to the value of the "privacy.owner-address" parameter in the configuration file of the node);
- `password` - password to access the private key in the node keystore;
- `policyId` - identifier of a group that will have access to the data to be forwarded;
- `info` - information about data being sent;
- `data` - string containing data in **base64** format;
- `hash` - sha256-hash data in **base58** format.

Examples of query and response of the **POST /privacy/sendData** method:

POST /privacy/sendData:

Query:

```
{
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"password": "apgJP9atQccdBPA",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"info": {
  "filename":"Service contract #100/5.doc",
  "size": 2048,
  "timestamp": 1000000000,
  "author": "AIvanov@org.com",
  "comment": "some comments"
 },
 "data":
↪"TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSBOaGlzIHNpbmd1bGFyIHBhc3Np
↪",
 "hash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta"
 }
```

Response:

```
{
"senderPublicKey": "Gt3o1ghh2M2TS65UrHZCTJ82LLcMcBrxuaJyrgsLk5VY",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"dataHash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta",
"proofs": [

↪"2jM4tw4uDmspuXUBt6492T7opuZskYhFGW9gkbq532BvLYRF6RJn3hVGNLuMLK8JSM61GkVgYvYJg9UscAayEYfc
↪"
```

(continues on next page)

(continued from previous page)


```
],
"fee": 110000000,
"id": "H3bdFTatppjnMmUe38YWh35Lmf4XDYrgsDK1P3KgQ5aa",
"type": 114,
"timestamp": 1571043910570
}
```


The **POST /privacy/sendDataV2** method allows you to attach a file in the Swagger window without having to convert it to the **base64** format. The `Data` field is missing in this version of the method.

Examples of query and response of the **POST /privacy/sendDataV2** method:

POST /privacy/sendDataV2:

Query:


```
{
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"password": "apgJP9atQccdBPA",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"info": {
  "filename":"Service contract #100/5.doc",
  "size": 2048,
  "timestamp": 1000000000,
  "author": "AIvanov@org.com",
  "comment": "some comments"
 },
 "hash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta"
 }
```


Response:


```
{
"senderPublicKey": "Gt3o1ghh2M2TS65UrHZCTJ82LLcMcBrxuaJyrgsLk5VY",
"policyId": "4gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaC",
"sender": "3HYW75PpAeVukmbYo9PQ3mzSHdKUgEytUUz",
"dataHash": "FRog42mnzTA292ukng6PHoEK9Mpx9GZNrEHecfvpwmta",
"proofs": [

↪"2jM4tw4uDmspuXUBt6492T7opuZskYhFGW9gkbq532BvLYRF6RJn3hVGNLuMLK8JSM61GkVgYvYJg9UscAayEYfc
↪"
],
"fee": 110000000,
"id": "H3bdFTatppjnMmUe38YWh35Lmf4XDYrgsDK1P3KgQ5aa",
"type": 114,
"timestamp": 1571043910570
}
```


Sending request of these types will result in a *114 PolicyDataHash* transaction, which will send a hash of confidential data to the blockchain.

See also

Description of transactions

REST API: confidential data exchange and obtaining of information about confidential data groups

CHAPTER

FOURTEEN

PERMISSION MANAGEMENT

All permissions of the blockchain platform are described in the article *Permissions*. Permissions can be arbitrarily combined for any address, individual permissions can be revoked at any time.

The *102 Permission Transaction* is used for managing the permissions of participants. This transaction can be signed using the `sign` method of the node REST API and sent using the corresponding gRPC or REST API method. Only a member with the **permissioner** permission can send a transaction to the blockchain.

Regardless of the sending method used, the transaction includes the following fields:

- `type` - type of transaction for managing the authority of the participants (`type = 102`);
- `sender` - address of the participant with authority to send transaction 102 (**permissioner** permission);
- `password` - key pair password in the node keystore, optional field;
- `proofs` - transaction signature;
- `target` - address of the participant for whom you want to set or remove permissions;
- `role` - member's permission which you want to set or remove;
- `opType` - type of operation: `add` (add a permission) or `remove` (remove a permission);
- `dueTimestamp` - **Unix Timestamp** of the action (in milliseconds), optional field.

The received response of the `sign` method is sent to the `broadcast` method of node gRPC or REST API.

See also

Description of transactions

REST API: information about permissions of participants

CHAPTER

FIFTEEN

CONNECTION AND REMOVING OF NODES

When working in Waves Enterprise Mainnet, member nodes are connected to the network and removed from it :ref:` with the help of Waves Enterprise specialists <mainnet-general>`.

In a private network, the connection and removal of new members is performed after manual configuration and the start of the first node.

15.1 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 node state of the network participants. Furthermore, 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 article *Participant role assignment*.
6. Start the new node.

15.2 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

Permission management

Architecture

CHAPTER

SIXTEEN

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*.

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` 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

REST API: information about configuration and state of the node, stopping the node

CHAPTER

SEVENTEEN

ARCHITECTURE

17.1 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. In 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.

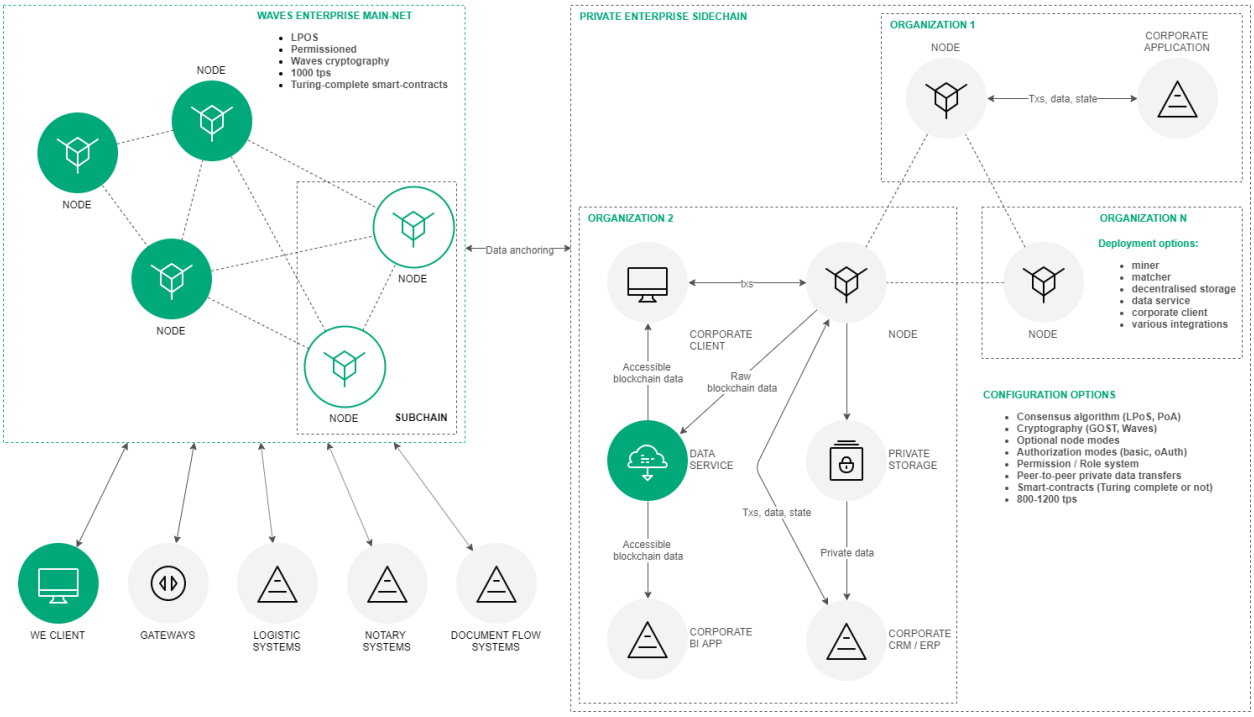
This two-part arrangement allows to optimize the network for high processing loads, increase information transmission rate, as well as to enhance concurrence and availability of data. Usage of the anchoring mechanism increases trust to data in sidechains, because they are validated in the master blockchain.

Platform architecture scheme:

17.2 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 keys are protected with the password.



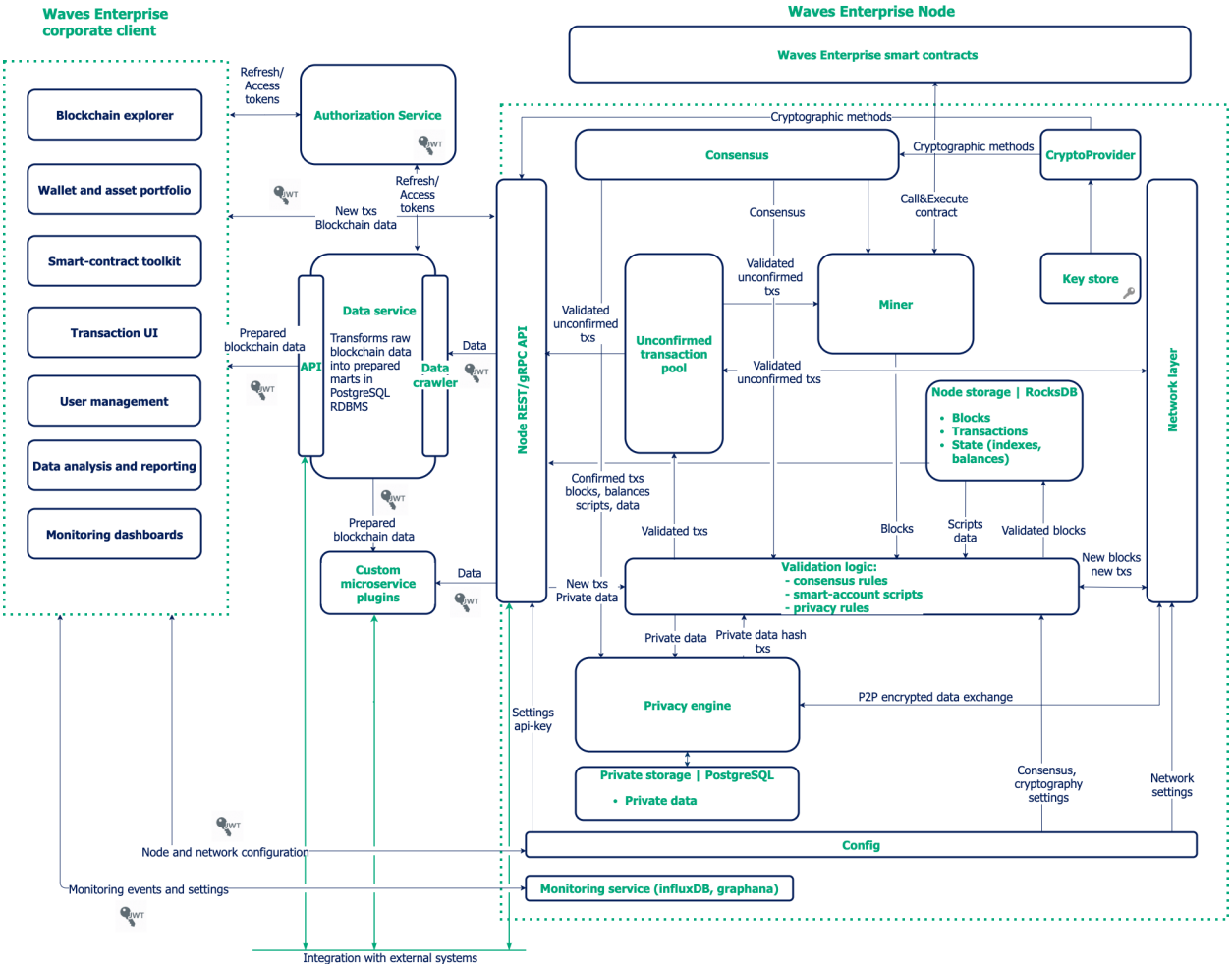
- **Network layer** – the logic layer that provides interaction of nodes at the applied level via the network protocol over the TCP.
- **Node storage** – the system component based on RocksDB that provides storage of 'key-value' pairs for the entire set of confirmed transactions and blocks, as well as 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 contains also 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
Blockchain explorer
Wallet and asset portfolio
Smart-contract toolkit
Transaction UI
User management
Data analysis and reporting
Monitoring dashboards
Refresh/ Access tokens
Authorization Service
JWT
Refresh/ Access tokens
New txs Blockchain data
Data service
Transforms raw blockchain data into prepared marts in PostgreSQL RDBMS
API
Data crawler
Prepared blockchain data
Data
Prepared blockchain data
Custom microservice plugins
Data
Waves Enterprise Node
Waves Enterprise smart contracts
Cryptographic methods
Consensus
Cryptographic methods
CryptoProvider
Consensus
Call&Execute contract
Validated unconfirmed txs
Miner
Key store
Validated unconfirmed txs
Unconfirmed transaction pool
Validated unconfirmed txs
Node REST/gRPC API
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-account 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
Config
Node and network configuration
Monitoring service (influxDB, graphana)
Monitoring events and settings
Integration with external systems

See also

Waves-NG blockchain protocol

Consensus algorithms

Cryptography

Examples of node configuration files

Authorization and data services

Generators

CHAPTER

EIGHTEEN

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.

18.1 Description of a mining round

Each mining round consists of the following stages:

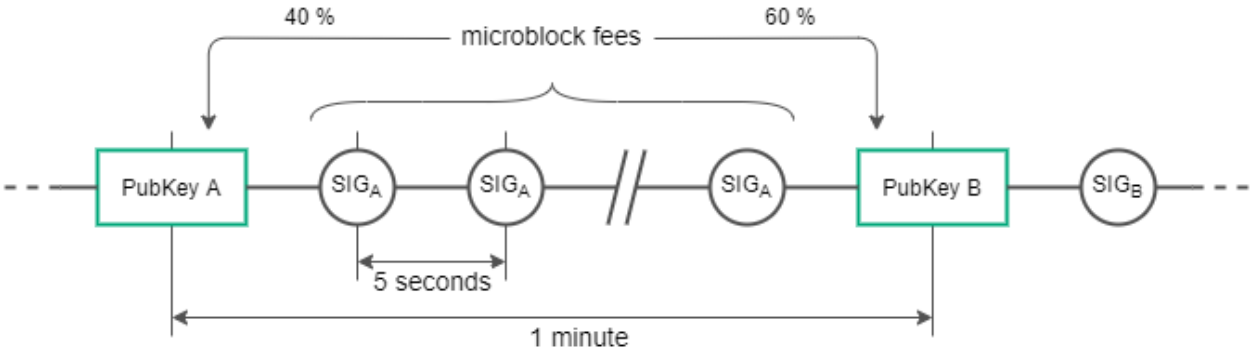
1. A used consensus algorithm defines a round miner and the time gor generation of a **key block** which does not contain transactions.
2. The round miner generates a key block which contains only service information:
   - public key of the miner for validation of microblock signatures;
   - a miner 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, that 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 genration 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.

18.2 Miner fee mechanism

The Waves-NG protocol supports financial motivation for miners. Each transaction in the Waves Enterprise blockchain requires a fee in WAVES 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%** if the total fee.

The fee charging transaction is carried out for each 100 blocks in order to provide an additional checking interval:



18.3 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

CHAPTER

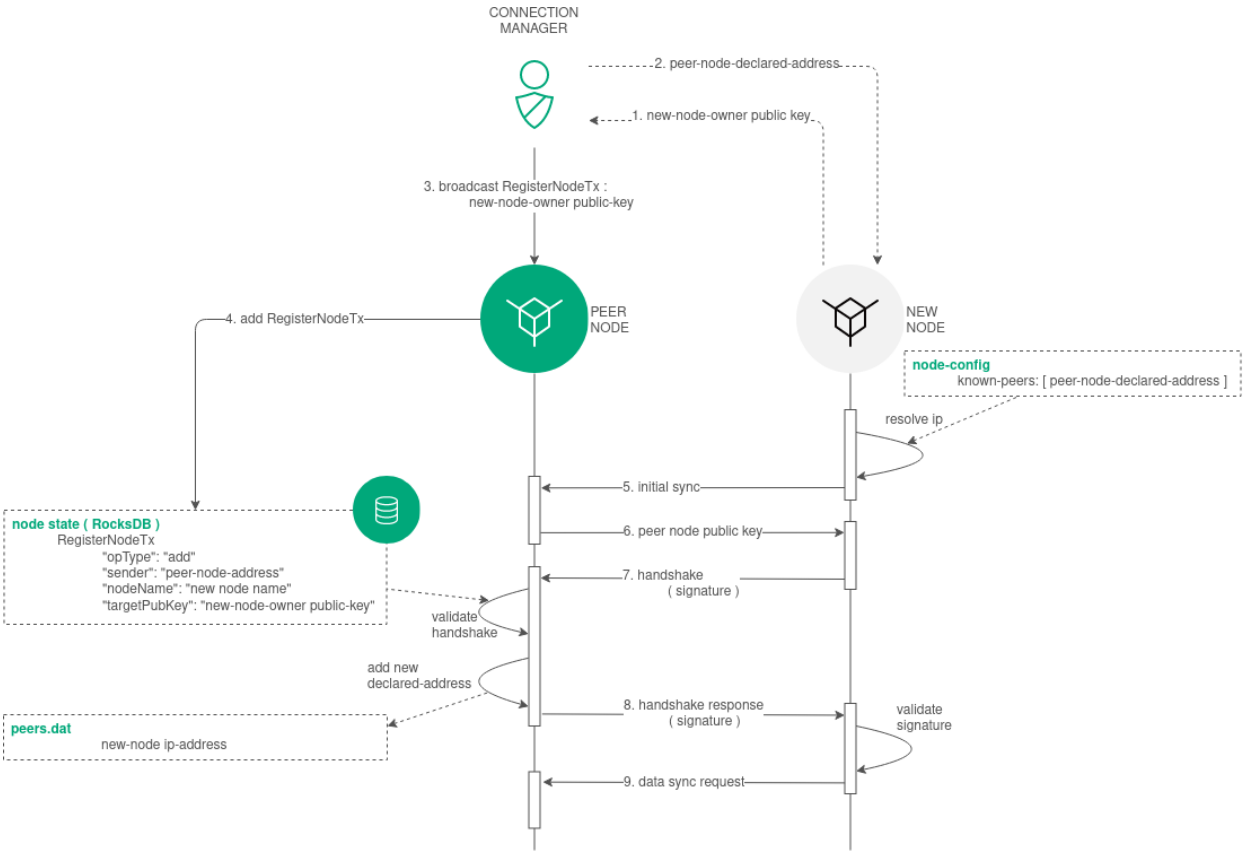
NINETEEN

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. Consequently, 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. In case of a successful check, in response to the **PeerIdentityRequest**, peers send their public keys to the new node. The new node saves these public keys in its temporary address storage for primary connection with peers. After saving of 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

Connection and removing of nodes

Permissions

CHAPTER

TWENTY

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.

20.1 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.

20.2 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 nodes with the `miner` role vote for their features during the `feature-check-blocks-period`.
2. After the `feature-check-blocks-period`, the system performs counting of votes - identifiers of each feature in 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.

20.3 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.

20.4 Preliminary activation of features

All features available for voting can be also forcibly activated while starting a new blockchain. This can be set in the `pre-activated-features` block of the `blockchain` section in the node configuration fail:

```
pre-activated-features = {
  ...
  101 = 0
  }
```

Blockchain height for activation of a certain feature is stated after an equal mark in front of every feature.

20.5 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

See also

rest-sf

CHAPTER

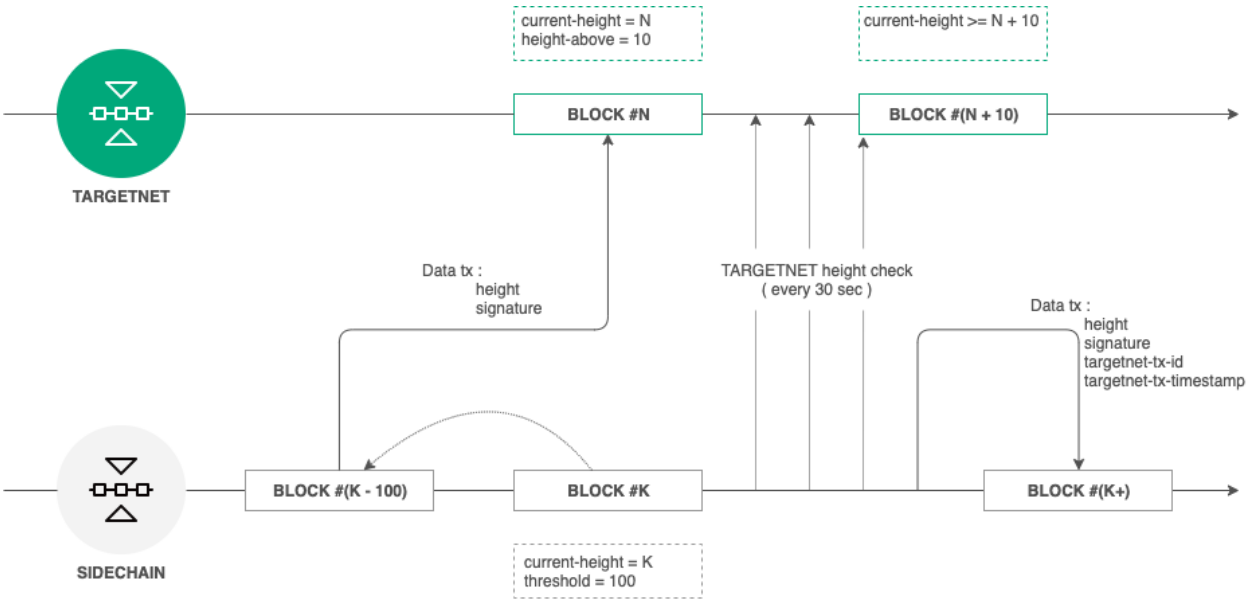
TWENTYONE

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 participantsm blocks and transactions, that carries 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 occurencies.

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.

21.1 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.

21.2 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 a Targetnet;
- `targetnet-tx-timestamp` - date and time of creation of a Targetnet anchoring tranaction.

21.3 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 1: Anchoring error types

Code	Error message	Possible reason
0	`Unknown error`	Unknown error has occured while broadcasting a transaction into a Targetnet
1	`Fail to create data transaction for Targetnet`	Creating of a transaction for broadcasting into a Targetnet has not been completed and returned an error
2	`Fail send transaction to Targetnet`	Broadcasting of a transaction into a Targetnet has not been completed and returned an error (that can occur |br| due to a JSON query error)
3	`Invalid http status of response from Targetnet` |br| `transaction broadcast`	Broadcasting of a transaction into the Targetnet returned a code |br| other than 200
4	`Fail to parse http body of response from Targetnet` |br| `transaction broadcast`	Broadcasting of a transaction into a Targetnet returned a |br| unrecognizable JSON query
5	`Targetnet return transaction with id='$TargetnetTxId'` \|br\| `but it differ from transaction that we sent` \|br\| `id='$sentTxId`	Broadcasting of a transaction into a Targetnet returned an identifier |br| that differs from a first transaction
6	`Targetnet didn't respond on transaction info request`	A Targetnet has not responded to a query on transaction information
7	`Fail to get current height in Targetnet`	The current height of a Targetnet has not been obtained
8	`Anchoring transaction in Targetnet disappeared after` |br| `height rise enough`	Anchoring transaction has has not been found in a Targetnet after increase of height |br| to the `height-above` value
9	`Fail to create sidechain anchoring transaction`	Failed to broadcast an anchoring transaction in a private blockchain
10	`Anchored transaction in sidechain was changed during` |br| `Targetnet height arise await, looks like` |br| `a rollback has happened`	While waiting for acknowledgement of a transaction in a Targetnet, a rollback occured |br| in a private blockchain, a transaction identifier has been changed

See also

Precise platform configuration: anchoring

CHAPTER

TWENTYTWO

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*.

22.1 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 for registration of incoming `SnapshotNotification` messages at a node, sending of `SnapshotRequest` messages and snapshot loading. If a node receives the `SnapshotNotification` message, the sender address is added to the array of addresses that have a 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.

22.2 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 cnsensus 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 receiving of the `SnapshotNotification`, the nodes initiate the `SnapshotRequest` messages for downloading of a ready snapshot. In case of expiration of snapshot receiving timeout or an error while downloding 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 integrity of smart contracts, members and parameters of confidential data access groups, permissions of participants. In case of successful verification of the snapshot, the node sends the messages about availability of the snapshot (`SnapshotNotification`) to its peers. After this, peers of the node can send a request on snapshot download to it.

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*.

22.3 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.

22.4 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

CHAPTER

TWENTYTHREE

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.

Smart contracts can be developed in any programming language and do not have any restrictions for their internal logic. In order to split startup and performance of a smartcontract and the blockchain platform itself, smart contracts start and work in Docker containers.

When a smart contract starts in a blockchain network, nobody can change, spoof it or restrict its operation without interference with the entire network. This aspect allows to provide security of business applications.

A smart contract has an access to the node state for data exchange via gRPC and REST API interfaces.

Each network participant can create and call smart contracts. 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 with the Waves Enterprise technical support service. After approval of your request, your smart contract will be uploaded into the registry, you will be able to call the smart contract with the use of the platform client or a REST API query to your node.

If you are going to use smart contracts in your own private blockchain network, you have to create your own registry for smart contract uploading and calling.

The general chart of smart contract operation is provided below:

23.1 Development and installation of smart contracts

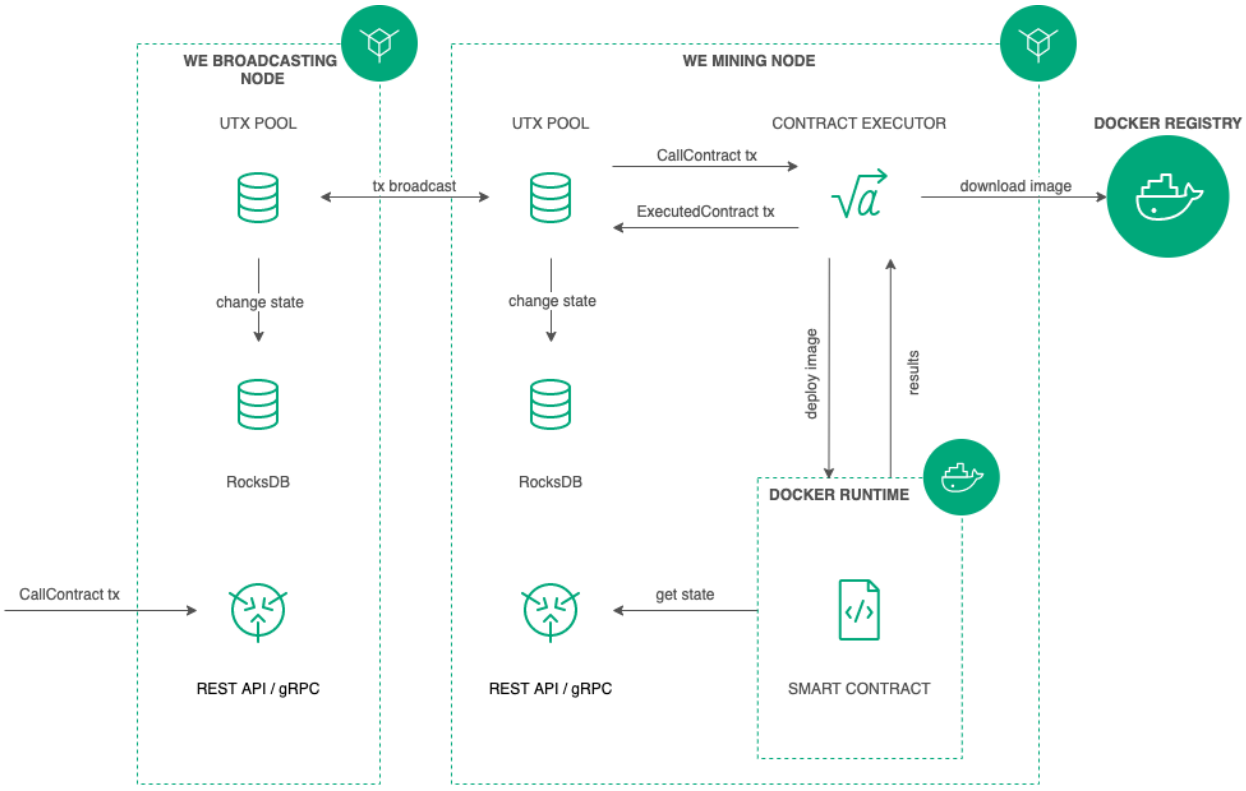
Practical instructions on development of smart contracts, as well as an example of a smart contract in Python are listed in the article *Development and usage of smart contracts*.

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 work in the Waves Enterprise Mainnet, it already has the pre-set parameters



for installing of smart contracts 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 *last 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 of a 103 transaction, the node downloads the image of a smart contract stated in the `image` field. After that, the downloaded image is checked and started in the Docker container.

23.2 Call of a smart contract and saving of results of its operation

A smart contract is called for operation by a network participant with the use of the *104 CallContract Transaction*. This transaction transfers the ID of the Docker container of the smart contract, as well as its input and output parameters in 'key-value' pairs. The container starts if it has not been started before.

Results of a smart contract operation are stored in its state with the use of the *105 ExecutedContract Transaction*.

23.3 Restriction of smart contract calls

In order to disable calls of a definite smart contract in the blockchain, send the *106 DisableContract Transaction* with the ID of the smart contract Docker container. This transaction can be sent only by the developer 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.

23.4 Updating of 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: A certain smart contract can be updated only by a participant who has sent a 103 transaction for this smart contract and has the **contract_developer** permission.

23.5 Parallel operation of smart contracts

The Waves Enterprise platform allows to run multiple smart contracts simultaneously. This option is supported only by smart contracts that use the gRPC interface for data exchange.

Concept of parallel operation of smart contracts:

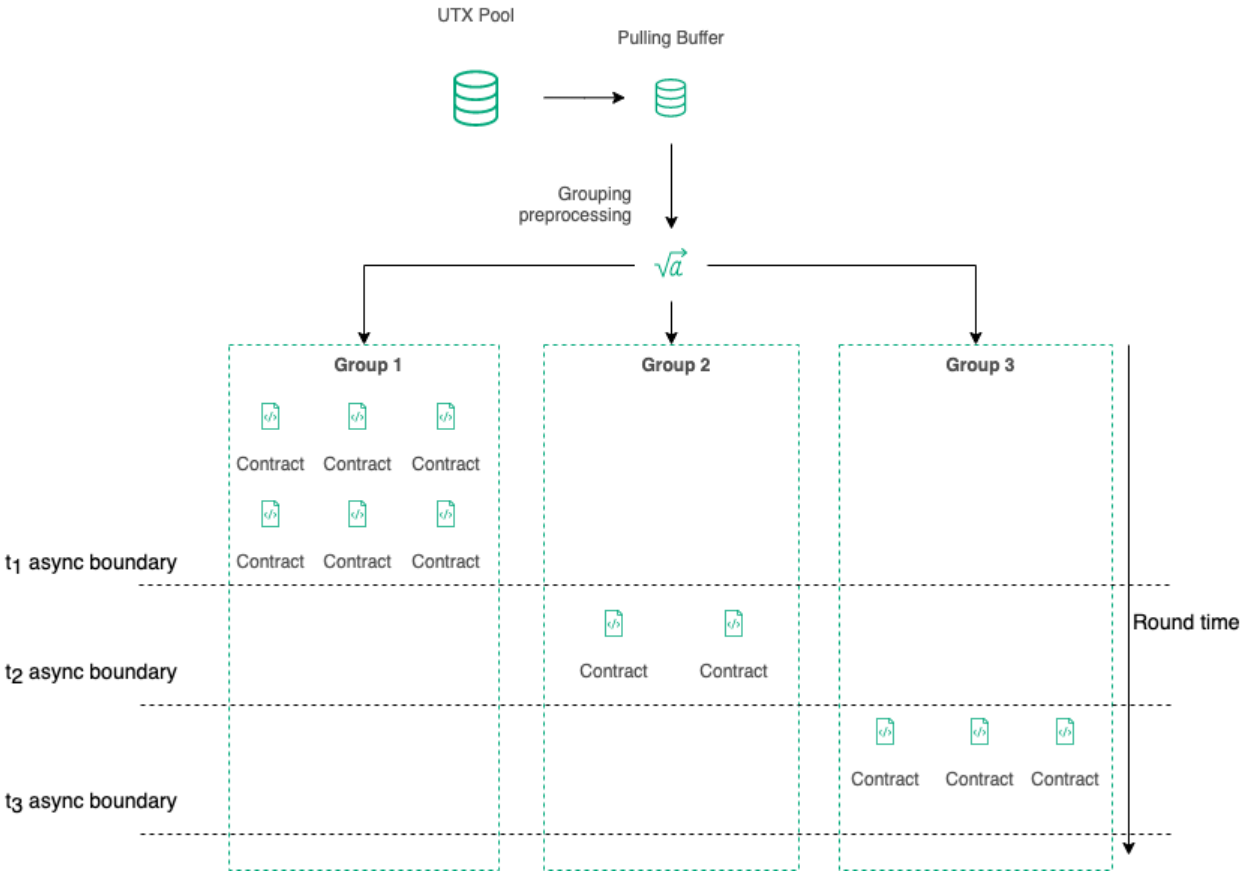
1. A smart contract developer enables the `async-factor` parameter in the code of his smart contract. This parameter defines the maximum allowed number of simultaneous transactions within one smart contract.
2. Upon start, the smart contract transfers the `async-factor` value to the node.
3. When operation of smart contracts begins, transactions for call of smart contracts are transferred from the UTX pool into the buffer for unprocessed smart contract transactions up to its filling.
4. Transaction in the buffer are divided into groups depending on smart contract IDs. Only one group of transactions can operate simultaneously, maximum number of transaction corresponds with the `async-factor` parameter.
5. When a next smart contract starts its operation, one cell in the transaction buffer becomes free. In the same time, a cell is blocked in case of transfer of transactions from the UTX pool. That means, operations of the buffer filling and processing of contract calls are performed simulteneously, that allows to avoid time gaps while transferring new transactions.

The value of the `async-factor` parameter can be set in advance in the interval from 1 to 999, as well as to be calculated dynamically. You can set a fixed value of this parameter as a constant, but it is recommended to set a calculable value. For instance, a contract can request a number of free processor cores and pass

this value as `async-factor`. This number will be used for parallel processing of transactions with a definite smart contract.

If the `async-factor` parameter is not defined, all transactions with a smart contract will be processed in a consecutive way.

Parallel operation of smart contracts is illustrated with the chart below:



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.

23.6 API methods available for smart contracts

In order to exchange data with smart contracts and nodes, the platform provides the gRPC and REST API methods. Usage of these methods allows to perform a wide range of operations with the blockchain.

Learn more:

23.6.1 gRPC services used by smart contracts

General instructions on usage of gRPC for development of smart contracts are provided in the article *Example of a smart contract with gRPC*.

Smart contracts that use the gRPC for data exchange with the node can use the services that are listed in the protobuf files with names starting with `contract`:

contract˙address˙service.proto

A set of methods for obtaining of participant addresses from the node keystore and data stored in addresses.

``**GetAddresses**`` - the method for obtaining of all addresses of participants, whose key pairs are stored in the node keystore. The method returns the``addresses`` string array.

``**GetAddressData**`` - the method for obtaining of all data stored at a definite address with the use of the transaction *12*. The method query contains following parameters:

- `address` - the address containing data to be obtained;
- `limit` - a 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 data stored at the address.

contract˙contract˙service.proto

A set of methods for work with smart contracts.

``**Connect**`` - the method for connection of the smart contract to the node. The method query should contain following parameters:

- `connection_id` - the connection identifier of the smart contract (see *Authorization of smart contracts with gRPC*);
- `async_factor` - maximum number of simultaneously processed transaction of the smart contract (see *Parallel operation of smart contracts*).

``**CommitExecutionSuccess**`` - the method for obtaining of a result of a successful of a smart contract with a definite ID and transferring of this result to the node.

``**CommitExecutionError**`` - the method for obtaining of an error occured as a result of operation of a smart contract with a definite ID and transferring of this error to the node.

``**GetContractKeys**`` - the method for obtaining of results of operation of a smart contract with a definite ID. The method query contains the following data:

- `contract_id` - smart contract identifier;
- `limit` - a 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 sorting of keys.

The method response returns the `Entries` array containing the results of a smart contract operation.

``**GetContractKey**`` - the method for obtaining of a definite result of a smart contract operation according to the key of this result. The method query contains the following data:

- `contract_id` - smart contract identifier;
- `key` - the required key.

The method response returns the `entry` data array which contains the result of smart contract operation according to the required key.

contract˙crypto˙service.proto

A set of encryption and decryption methods. See details in the article *gRPC: encryption and decryption methods*.

contract˙permission˙service.proto

A set of methods for obtaining of information about permissions of participants.

``**GetPermissions**`` - the method for obtaining of a list of all address permissions valid at the moment of the query transfer. The method query contains the following data:

- `address` - the required address;
- `timestamp` - the *Unix Timestamp* (in milliseconds) for the required moment of time.

The response of the method returns the `roles` array containing permissions for the required address and the entered `timestamp`.

``**GetPermissionsForAddresses**`` - the method for obtaining of all permissions valid at the moment of the query transfer for multiple addresses. The method query contains the following data:

- `addresses` - a string array containing required addresses;
- `timestamp` - the *Unix Timestamp* (in milliseconds) for the required moment of time.

The method response returns an `address_to_roles` array containing permissions for each required address, ad well as the entered `timestamp`.

contract˙pki˙service.proto

A set of methods for generation and checking of electronic signatures. See details in the article *gRPC: generation and checking of data electronic signatures (PKI)*.

contract˙privacy˙service.proto

A set of methods for obtaining of information about confidential data groups, as well as for work with confidential data.

Learn more about confidential data exchange and access groups in the article *Confidential data exchange*.

``GetPolicyRecipients`` - the method for obtaining of addresses in a confidential data group with a definite `policy_id`. The method response returns a `recipients` string array which contains addresses of confidential data group participants.

``GetPolicyOwners`` - the method for obtaining of owners of a confidential data group with a definite `policy_id`. The response of the method returns the `owners` string array, which contains addresses of confidential data group owners.

``GetPolicyItemData`` - the method for obtaining of a confidential data package according to its identification hash. This method is available if the address of the method caller belongs to the confidential data group.

The method query contains a confidential group identifier `policy_id` and an identifying hash of required confidential data `item_hash`. The response of the method returns the `data` string containing a hash of a required confidential data package.

``GetPolicyItemInfo`` - the method for obtaining of information about a confidential data package according to its identification hash. This method is available if the address of the method caller belongs to the confidential data group.

The method query contains a confidential group identifier `policy_id` and an identifying hash of required confidential data `item_hash`. The method response returns the following data:

- `sender` - an address of confidential data sender;
- `policy_id` - a confidential data group identifier;
- `type` - type of confidential data (`file`);
- `info` - an array containing a file data: * `filename` - name of the file; * `size` - size of the file; * `timestamp` - a *Unix Timestamp* (in milliseconds) for uploading of the file to the network; * `author` - author of the file; * `comment` - an optional comment to the file;
- `hash` - identifying hash of confidential data.

contract˙transaction˙service.proto

A set of methods for obtaining of information about transactions that have been sent to the blockchain. See details in the article *gRPC: information about transaction according to their IDs*.

contract˙util˙service.proto

This protobuf file contains the **``GetNodeTime``** method, which is used for obtaining of a node current time. The method returns the current node time in two formats:

- `system` - system time of the node PC;
- `ntp` - network time.

See also

Smart contracts

sc-rest

Development and usage of smart contracts

General platform configuration: execution of smart contracts

23.6.2 REST API methods available for smart contracts

General instructions on development of smart contracts with the use of REST API are provided in the article *Example of a smart contract with the use of REST API*.

Smart contracts that use the REST API interface for data exchange can use following methods:

``Addresses`` method set:

- *GET /addresses*
- *GET /addresses/publicKey/{publicKey}*
- *GET /addresses/balance/{address}*
- *GET /addresses/data/{address}*
- *GET /addresses/data/{address}/{key}*

``Crypto`` method set:

- *POST /crypto/encryptSeparate*
- *POST /crypto/encryptCommon*
- *POST /crypto/decrypt*

``Privacy`` method set:

- *GET /privacy/{policy-id}/getData/{policy-item-hash}*
- *GET /privacy/{policy-id}/getInfo/{policy-item-hash}*
- *GET /privacy/{policy-id}/hashes*
- *GET /privacy/{policy-id}/recipients*

``Transactions`` method set:

- *GET /transactions/info/{id}*
- *GET /transactions/address/{address}/limit/{limit}*

``Contracts`` method set:

- *GET /internal/contracts/{contractId}/{key}*
- *GET /internal/contracts/executed-tx-for/{id}*
- *GET /internal/contracts/{contractId}*
- GET /internal/contracts

To ensure a better performance, smart contracts can use the `Contracts` methods with a dedicated route `/internal/contracts/`. Endpoints of this route are identical with conventional methods of the `Contracts` group.

``PKI`` method group:

- *PKI /verify*

See also

Smart contracts

gRPC services used by smart contracts

Development and usage of smart contracts

General platform configuration: execution of smart contracts

See also

Development and usage of smart contracts

General platform configuration: execution of smart contracts

CHAPTER

TWENTYFOUR

TRANSACTIONS OF THE BLOCKCHAIN PLATFORM

Transaction is a separate operation in the blockchain changing the network state and performed .

24.1 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: работа с транзакциями*);
- with the use of the *JavaScript SDK*.

The transaction signature is inserted into the `proofs` while sending transactions into the blockchain. As a rule, this field contains one signature of a participant which has been sent a 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*)

24.2 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 a 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 transactionsof the Waves Enterprise blockchain platform:

24.2.1 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.

The format of responses returned by a node via the gRPC interface is defined in t he protobuf filed (see *gRPC tools*).

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 data structure:

Field	Data type	Description
type	Byte	Transaction number (**1**)
id	Byte	Transaction identifier
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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 issue of tokens.

Data structure of a query for transaction 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 a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number
id	Byte	Transaction identifier
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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 (WEST - **8**)
description	Array[byte]	An arbitrary description of a transaction
chainId	Byte	Identifying byte of the network (Mainnet - **87**, or **V**)
script	Array[Byte]	Script for validation of a transaction, an *optional field*
height	Int	Height of transaction execution

4. Transfer Transaction

A transaction of transfer of tokens from one address to another.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**4**)
version	Byte	Transaction version
sender	ByteStr	Address of a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	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 a transaction (**in base58 format**), *optional field*
sender	ByteStr	Address of a 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	Address of recipient of tokens
id	Byte	Transaction identifier
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**

5. Reissue Transaction

Transaction for token re-issue.

Data structure of a query for transaction 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 a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
quantity	Long	Amount of tokens to be re-issued
sender	ByteStr	Address of a 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 a transaction (in milliseconds), **optional field**
height	Int	Height of transaction execution

6. Burn Transaction

Transaction for burning of tokens: decreases amount of tokens at the sender address, and, with this, decreases a total amount of tokens in the blockchain. The burned tokens cannot be restored.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**6**)
version	Byte	Transaction version
sender	ByteStr	Address of a 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*
attachment	Byte	Comment to a transaction (**in base58 format**), *optional field*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
amount	Long	Number of tokens to be burnt
sender	ByteStr	Address of a 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 a transaction (in milliseconds), **optional field**
height	Int	Height of transaction execution

8. Lease Transaction

Leasing of tokens to another address. The tokens in leasing are taken into account in a 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.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**8**)
version	Byte	Transaction version
sender	ByteStr	Address of a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
amount	Long	Number of tokens for leasing
sender	ByteStr	Address of a 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	Height of transaction execution
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**

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 obtaining of transaction data.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**9**)
version	Byte	Transaction version
fee	Long	*WE Mainnet transaction fee*
sender	ByteStr	Address of a transaction sender
password	String	Keypair password in the node keystore, *optional field*
txId	Byte	ID of a leasing transaction

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
leaseId	Byte	ID of a leasing transaction
sender	ByteStr	Address of a 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 a transaction (in milliseconds), **optional field**
height	Int	Height of transaction execution

10. CreateAlias Transaction

Creation of an alias for a sender address. An alias can be used in transactions as a recipient identifier.

Data structure of a query for transaction signing:

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 a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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

11. MassTransfer Transaction

Transfer of tokens to several recipients (1 - 100 addresses). A transaction fee depends on a number of addresses.

Data structure of a query for transaction signing:

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
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 a transaction (**in base58 format**), *optional field*
sender	ByteStr	Address of a 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 fields `recipient`: ` and ``amount` separated with 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 }
]
```

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:

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
senderPublicKey	PublicKeyAccount	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 a 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*
id	Byte	Data transaction ID
type	Byte	Transaction number (**12**)
version	Byte	Transaction version
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**

Example of the `data` field:

```
"data": [
      {
       "key": "objectId",
       "type": "string",
       "value": "obj:123:1234"
      }, {...}
     ]
```

13. SetScript Transaction

A transaction to bind the script to an account or delete the script. An account with a script tied 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.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**13**)
version	Byte	Transaction version
sender	ByteStr	Address of a 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 is in **base64** format. If you leave this field empty (*null*), the script will be detached from the account

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**13**)
id	Byte	ID of a script setting transaction
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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 a transaction (**in base58 format**), *optional field*
height	Byte	Height of transaction execution

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.

Data structure of a query for transaction signing:

Field	Data type	Description
sender	ByteStr	Address of a 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

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**14**)
id	Byte	Sponsorship transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
assetId	Byte	Sponsorship asset (token) ID - *optional field*
timestamp	Long	The**Unix Timestamp** of a 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

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.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**15**)
version	Byte	Asset script transaction version
sender	ByteStr	Address of a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**15**)
id	Byte	Asset script transaction ID
sender	ByteStr	Address of a transaction sender
sender-PublicKey	PublicKey-Account	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
proofs	List(ByteStr)	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	Array[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

101. GenesisPermission Transaction

A transaction to assign the first network administrator who distributes permissions to other participants.

Data structure of a query for transaction 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 a 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`)

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**101**)
timestamp	Long	The**Unix Timestamp** of a 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. Only a participant with the **permissioner** permission can send 102 transactions to the blockchain.

Possible permissions for the `role` field:

- permissioner
- sender
- blacklister
- miner
- issuer
- contract_developer
- connection_manager
- banned

For a description of the permissions, see the article *Permissions*.

Data structure of a query for transaction signing:

Field	Data type	Description
type	Byte	Transaction number (**102**)
sender	ByteStr	Address of a transaction sender
password	String	Keypair password in the node keystore, *optional field*
senderPublicKey	PublicKeyAccount	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
dueTimestamp	Long	Role **Unix Timestamp** (in milliseconds) - *optional field*
version	Byte	Transaction version

Data structure of a query for transaction broadcasting:

Field	Data type	Description
senderPublicKey	PublicKeyAccount	Transaction sender public key
role	String	A permission to be assigned (for an administrator - `permissioner`)
sender	ByteStr	Address of a 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**)
dueTimestamp	Long	Role **Unix Timestamp** (in milliseconds) - *optional field*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
target	ByteStr	Address of a first administrator to be appointed

103. CreateContract Transaction

Creating a smart contract. The byte representation of this transaction after it is signed must not exceed **150 kilobytes**.

The `feeAssetId` field of this transaction is optional and is only used for gRPC-enabled smart contracts. The value of the `version` field for this type of smart contracts is **2**.

Transaction 103 can only be signed by a user with the role **contract_developer**.

Data structure for transaction signing request:

Field	Data type	Description
fee	Long	*WE Mainnet transaction fee*
image	Array[Bytes]	Smart contract Docker image name
imageHash	Array[Bytes]	Smart contract Docker image hash
contractName	Array[Bytes]	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	ByteStr	Address of a transaction sender
password	String	Keypair password in the node keystore, *optional field*
params	List[DataE	Input and output data of a smart contract. Entered using the fields `type value` and `key` separated with a comma - *optional field*
params.l	Byte	Parameter key
params.t	Byte	Parameter type. Possible values: `binary bool integer string`
params.v	Byte	Parameter value
type	Byte	Transaction number (**103**)
version	Byte	Transaction version

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**103**)
id	Byte	ID of a CreateContract transaction
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
proofs	List(ByteS	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
image	Array[Bytes]	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)
imageHash	Array[Bytes]	Smart contract Docker image hash
contractName	Array[Bytes]	Smart contract name
params	List[DataE	Input and output data of a smart contract. Entered using the fields `type value` and `key` separated with a comma - *optional field*
params.ke	Byte	Parameter key
params.ty	Byte	Parameter type. Possible values: `binary bool integer string`
params.va	Byte	Parameter value
height	Byte	Height of transaction execution

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 a request to broadcast a smart contract from a not installed repository:

```
{
  "type": 103,
  "id": "ULcq9R7PvUB2yPMrmBdxoTi3bcRmQPT3JDLLLZVj4Ky",
  "sender": "3N3YTj1tNwn8XUJ8ptGKbPuEFNa9GFnhqew",
  "senderPublicKey": "3kW7vy6nPC59BXM67n5N56rhhAv38Dws5skqDsjMVT2M",
  "fee": 500000,
  "timestamp": 1550591678479,
  "proofs": [
↪"yecRFZm9iBLyDy93bDVaNo1PR5Qkkic7196GAgUt9TNH1cnQphq4yGQQ8Fxj4BYA4TaqYVw5qxtWzGMPQyVeKYv
↪" ],
  "version": 1,
  "image": "customregistry.com:5000/stateful-increment-contract:latest",
  "imageHash": "7d3b915c82930dd79591aab040657338f64e5d8b842abe2d73d5c8f828584b65",
  "contractName": "stateful-increment-contract",
  "params": [],
  "height": 1619
  }
```

104. CallContract 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 second version of the transaction: if the `version` field of the smart contract creation transaction is set to `2`. The contract is updated using the transaction *107*. When a contract is created, transaction 104 is automatically created, which calls the contract to verify it.

If the contract is not executed or is executed with an error, then transactions 103 and 104 are deleted and do not enter the block.

Data structure for transaction signing request:

Field	Data type	Description
contractId	ByteStr	Smart contract ID
fee	Long	*WE Mainnet transaction fee*
sender	ByteStr	Address of a transaction sender
password	String	Keypair password in the node keystore, *optional field*
type	Byte	Transaction number (**104**)
params	List[DataEnt	Input and output data of a smart contract. Entered using the fields `type` `value` and `key` separated with a comma - *optional field*
params.k	Byte	Parameter key
params.ty	Byte	Parameter type. Possible values: `binary` `bool` `integer` `string`
params.va	Byte	Parameter value
version	Byte	Transaction version

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**104**)
id	Byte	Smart contract call transaction ID
sender	ByteStr	Address of a transaction sender
sender-PublicKey	PublicKey-Account	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
proofs	List(ByteStr	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
contractId	ByteStr	Smart contract ID
params	List[DataEnt	Input and output data of a smart contract. Entered using the fields `type` `value` and `key` separated with a comma - *optional field*
params.key	Byte	Parameter key
params.typ	Byte	Parameter type. Possible values: `binary` `bool` `integer` `string`
params.valu	Byte	Parameter value

105. ExecutedContract Transaction

Writing of the result of smart contract execution to its state. The byte representation of this transaction after signing must not exceed **150 kilobytes**.

Transaction 105 contains all fields (body) of transaction 103 or 104 of the smart contract whose execution result must be written to its state (the `tx` field). The result of the smart contract's execution is entered into its stack from the corresponding parameters of the `params` field of transaction 103 or 104.

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 broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**105**)
id	Byte	ExecutedContract transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
password	String	Keypair password in the node keystore, *optional field*
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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[_]]	A list of possible results of smart contract execution
height	Byte	Height of transaction execution

106. DisableContract Transaction

Disabling of a smart contract. 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 role **contract_developer**.

Data structure for transaction signing request:

Field	Data type	Description
sender	ByteStr	Address of a 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

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**106**)
id	Byte	DisableContract transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
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

107. UpdateContract Transaction

Updating of 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.

Data structure for transaction signing request:

Field	Data type	Description
image	Array[Bytes]	Smart contract Docker image name
sender	ByteStr	Address of a transaction sender
password	String	Keypair password in the node keystore, *optional field*
fee	Long	*WE Mainnet transaction fee*
contractId	ByteStr	Smart contract ID
imageHash	Array[Bytes]	Smart contract Docker image hash
type	Byte	Transaction number (**107**)
version	Byte	Transaction version

Data structure of a query for transaction broadcasting:

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 broadcasting:

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 a transaction (in milliseconds), **optional field**
signature	ByteStr	Transaction signature (**in base58 format**)
version	Byte	Transaction version
targetPubKey	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.

Data structure for transaction signing request:

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 a 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*

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**111**)
id	Byte	RegisterNode transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
version	Byte	Transaction version
targetPubKey	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*

112. CreatePolicy Transaction

Creation of a confidential data group consisting of addresses stated in a transaction.

Data structure for transaction signing request:

Field	Data type	Description
sender	ByteStr	Address of a transaction sender
policyName	String	Name of an access group to be created
password	String	Keypair password in the node keystore, *optional field*
recipients	Aray[Byt	Array of addresses of a group participants separated by commas
fee	Long	*WE Mainnet transaction fee*
description	Array[byte]	An arbitrary description of a transaction (**in base58 format**)
owners	Aray[Byt	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

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**112**)
id	Byte	CreatePolicy transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
policyName	String	Name of an access group to be created
recipients	Aray[Bytes]	Array of addresses of a group participants separated by commas
owners	Aray[Bytes]	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a transaction (in milliseconds), **optional field**
proofs	List(ByteStr)	Array of transaction proofs (**in base58 format**)
height	Byte	Height of transaction execution
description	Array[byte]	An arbitrary description of a transaction (**in base58 format**)
version	Byte	Transaction version

113. UpdatePolicy Transaction

Updating of a confidential data group

Data structure for transaction signing request:

Field	Data type	Description
policyId	String	Confidential data group identifier
password	String	Keypair password in the node keystore, *optional field*
sender	ByteStr	Address of a transaction sender
recipients	Aray[Byte	Array of addresses of a group participants separated by commas
fee	Long	*WE Mainnet transaction fee*
opType	String	Type of operation: `add` - add participants; `remove` - remove participants
owners	Aray[Byte	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
type	Byte	Transaction number (**113**)
version	Byte	Transaction version

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**113**)
id	Byte	UpdatePolicy transaction ID
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
policyId	String	Confidential data group identifier
recipients	Aray[Bytes]	Array of addresses for adding or removing of group participants separated by commas
owners	Aray[Bytes]	Array of addresses of group administrators separated by commas: administrators are entitled to change an access group
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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 permission
description	Array[byte]	An arbitrary description of a transaction (**in base58 format**)
version	Byte	Transaction version

114. PolicyDataHash Transaction

Sending of a 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 broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**114**)
id	Byte	Transaction identifier
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
policyId	String	Name of an access group to be created
dataHash	String	Confidential data hash to be sent
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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

120. AtomicTransaction

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.

Atomic transaction support 2 and more transactions of the following types:

- *4. Transfer Transaction*, ver. 3
- *102. Permission Transaction*, ver. 2
- *103. CreateContract Transaction*, ver. 3
- *104. CallContract Transaction*, ver. 4
- *105. ExecutedContract Transaction*, ver. 1 and 2
- *106. DisableContract Transaction*, ver. 3
- *107. UpdateContract Transaction*, ver. 3
- *112. CreatePolicy Transaction*, ver. 3
- *113. UpdatePolicy Transaction*, ver. 3
- *114. PolicyDataHash Transaction*, ver. 3

An atomic transaction itself does not require a fee: its total fee is summed up from fee of transactions included into it.

Learn more about atomic transactions: *Atomic transactions*

Data structure for transaction signing request:

Field	Data type	Description
type	Byte	Transaction number (**120**)
sender	ByteStr	Address of a 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

Data structure of a query for transaction broadcasting:

Field	Data type	Description
type	Byte	Transaction number (**114**)
id	Byte	Transaction identifier
sender	ByteStr	Address of a transaction sender
senderPublicKey	PublicKeyAccount	Transaction sender public key
fee	Long	*WE Mainnet transaction fee*
timestamp	Long	The**Unix Timestamp** of a 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

See also

Description of transactions

Waves Enterprise Mainnet fees

24.2.2 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*	2
4	*Transfer Transaction*	3
5	*Reissue Transaction*	2
6	*Burn Transaction*	2
8	*Lease Transaction*	2
9	*Lease Cancel Transaction*	2
10	*Create Alias Transaction*	3
11	*Mass Transfer Transaction*	2
12	*Data Transaction*	2
13	*Set Script Transaction*	1
14	*Sponsorship Transaction*	1
15	*Set Asset Script Transaction*	1
101	*Genesis Permission Transaction*	No version
102	*Permission Transaction*	2
103	*Create Contract Transaction*	3
104	*Call Contract Transaction*	4
105	*Executed Contract Transaction*	2
106	*Disable Contract Transaction*	3
107	*Update Contract Transaction*	3
110	*Genesis Resgister Node Transaction*	1
111	*Register Node Transaction*	1
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

Waves Enterprise Mainnet fees

CHAPTER

TWENTYFIVE

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 realized through the *120 AtomicTransaction*, which is a container consisting of two or more signed transactions.

Atomic transaction support 2 and more transactions of the following types:

- *4. Transfer Transaction*, ver. 3
- *102. Permission Transaction*, ver. 2
- *103. CreateContract Transaction*, ver. 3
- *104. CallContract Transaction*, ver. 4
- *105. ExecutedContract Transaction*, ver. 1 and 2
- *106. DisableContract Transaction*, ver. 3
- *107. UpdateContract Transaction*, ver. 3
- *112. CreatePolicy Transaction*, ver. 3
- *113. UpdatePolicy Transaction*, ver. 3
- *114. PolicyDataHash Transaction*, ver. 3

The key peculiarity of transactions that are supported by atomic transactions, is an *atomicBadge* field. This field contains a *trustedSender* value: a trusted address of a transaction sender for including into the *120* transaction container. If a sender address is not specified, an address of a sender of the 120 transaction becomes a sender of a included transaction.

25.1 Processing of atomic transactions

Atomic transactions have two signatures. First signature belongs to its sender and is used for broadcasting. Second signature is generated by a miner and is used for including of 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 transactions included into the atomic container.

Validation of included transactions is carried out as follows:

- There should be more than one included transactions.
- All transactions should have different identifiers.
- An atomic transaction should contain only supported transaction types.

Including of an atomic transaction to another atomic transaction is not allowed.

There should not be executed transactions inside an atomic transaction to be sent into the UTX pool, the `miner` field should be empty. This field is filled during transferring of the transaction into a block.

There should not be 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` fiels 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 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.

25.2 Generating of atomic transactions

An access to the node *REST API* is required for generating of an atomic transaction.

1. A user picks supported transactions that should be used as an atomic operation.
2. After that, a user fills fields of all transactions and signs them.
3. A user fills the `transactions` field of an atomic transaction with data of signed, but not broadcasted transactions.
4. After filling an atomic transaction with data of all included transaction, a user signs it and broadcasts into the blockchain.

Data structures for signing and broadcasting of an atomic transaction, are listed in the *list of transactions*.

Attention: If you create an atomic transaction including a *114* transaction, set its `broadcast` value as `false` while signing.

See also

Description of transactions

Waves Enterprise Mainnet fees

CHAPTER

TWENTYSIX

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 organisation of an overall workflow.

These problems are resolved by the **consensus** - an algorithm which coordinates work of the blockchain particiants 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. Anyway, voting guarantees achievement of a consensus that will be profitable for the entire network.

The Waves Enterprise blockchain platform supports three consensus algorithms:

26.1 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.

26.1.1 Proof of Stake

The mechanism for delegating of rights for block generation is based on the number of tokens in the user's account. The more tokens a user has, the more likely this user will be able to generate a block.

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 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 a next miner;
- T_{min} - a constant value defining a time interval between blocks (**5 seconds**);
- C_1 - a constant value correcting a form of interval allocation between blocks (**70**);

- C_2 - a constant value that is, by default, equal to the BaseTarget (**5E17**) and serving 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 in its turn is calculated as:

$$(S > R_{max} \rightarrow T_b = T_p + max(1, \frac{T_p}{100})) \wedge (S < R_{min} \wedge \wedge T_b > 1 \rightarrow T_b = T_p - max(1, \frac{T_p}{100}))$$

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 - average time of generation of at least three last blocks;
- T_p - a 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 system participants, which reduces the cost of deploying private networks. No additional emission is required, which in PoW systems is used for rewarding miners for finding a new block. In PoS systems, a miner receives a reward in the form of fees for transactions which appeared in its block.

26.1.2 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

26.2 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.

26.2.1 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, which 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 which 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. In 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 , which 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, which 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 moners 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, which 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.

26.2.2 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.

26.2.3 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 nd 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 in the same time the P_{ban} is also achieved, this miner will not be excluded from the queue.

26.2.4 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.

26.2.5 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

26.3 CFT consensus algorithm

When information is exchanged extensively in a corporate blockchain, consistency between 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 CFT (Crash Fault Tolerance) consensus algorithm is designed to prevent such situations.

26.3.1 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 up to restoring of network cohesion.

Консенсус CFT, также как и PoA, зависит от текущего времени, а время начала и окончания каждого раунда рассчитывается на основе временной метки genesis-блока. Основные параметры, на основе которых формируется алгоритм для определения майнера текущего блока, также идентичны параметрам алгоритма PoA (см. раздел Описание алгоритма). Для валидации блоков в блок `consensus` конфигурационного файла ноды были добавлены три новых параметра:

- **max-validators** – лимит валидаторов, участвующих в голосовании в конкретном раунде.
- **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** - опциональный параметр, определяющий, сколько времени после окончания раунда (параметр конфигурационного файла ноды: `round-duration`) майнер ожидает полный набор голосов от всех валидаторов.

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 finalization of the last block (parameter of the node configuration file: `finalization-timeout`).
- V_{max} - limit of validators taking part in voting (parameter of the node configuration file: `max-validators`).

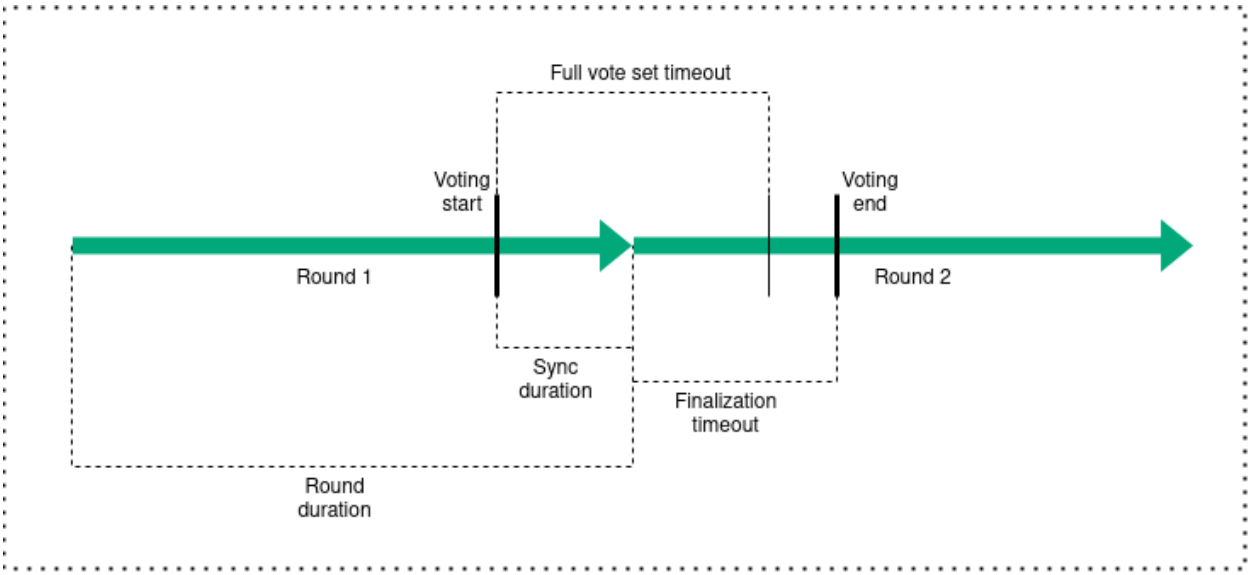
26.3.2 Voting

Общая схема раунда при использовании CFT выглядит следующим образом:

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 a 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.

26.3.3 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.

In order to exclude returning of your transactions into the UTX pool, it is highly recommended to work not with the current (liquid) blockm but with a finalized one that has been already validated by the network participants.

26.3.4 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 evenly distribute the load of the network.

26.3.5 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
}
```

Рекомендации по конфигурации CFT см. в разделе *General platform configuration: consensus algorithm*.

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

CHAPTER

TWENTYSEVEN

CRYPTOGRAPHY

The Waves Enterprise platform gives an opportunity to choose a cryptographic algorithm depending on peculiarities of a project.

27.1 Hash coding

Hash coding is performed consequently by the functions **Blake2b256** and **Keccak256** or the "Streebog" function in accordance with the GOST R 34.11-2012 Information Technology – Cryptographic Information Security – Hash Function . Size of an output data block is **256 bits**.

27.2 Electronic signature

Algorithms for key generation, producing and checking of electronic signatures are based on the **Curve25519** elliptic curve (ED25519 with X25519 keys) or correspond with the **GOST R 34.10-2012** Information Technology – Cryptographic Information Security – Signature and verification processes of electronic digital signature.

Learn more about generation and verification of electronic signatures with the use of the *gRPC* and *REST API* methods.

27.3 Data encryption

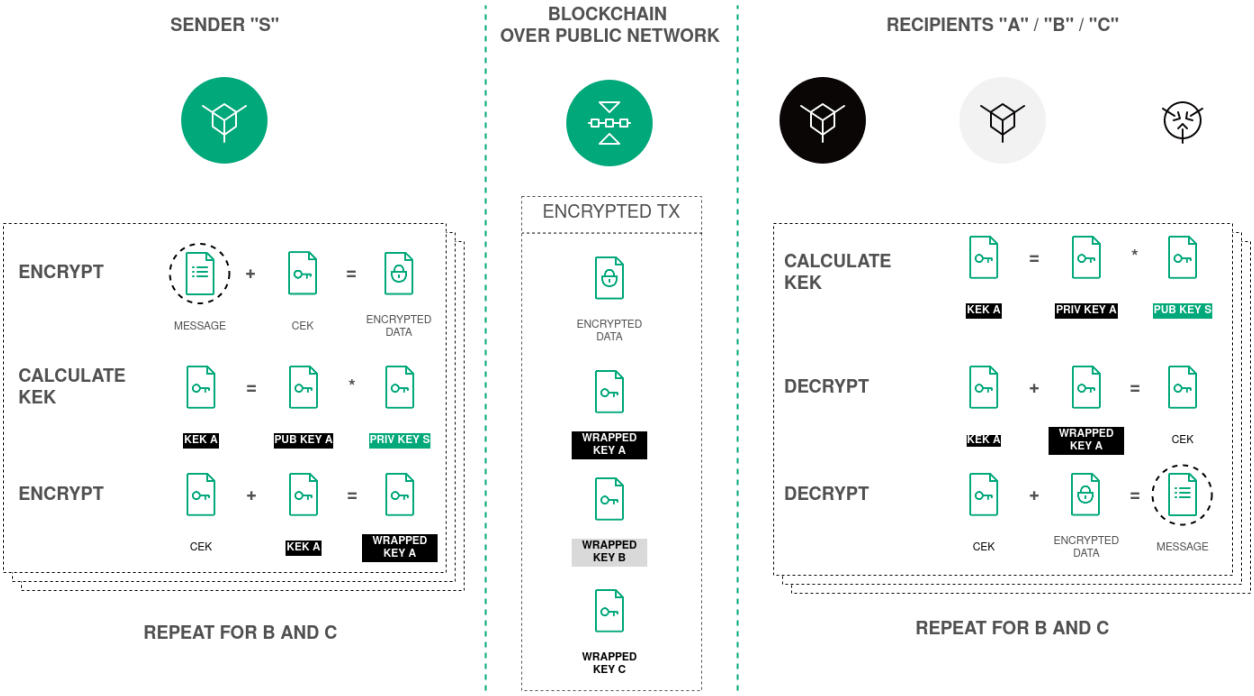
The platform supports data encryption with the use of session keys on the basis of the **Diffie–Hellman protocol**. This operation is used for encryption of any text information, for instance, data of smart contracts that should not be available for other blockchain participants. Encryption can be performed individually for every recipient with generation of a unique cipher text, as well as with generation of a unified cipher text for a group of recipients.

The algorithms used for symmetric encryption correspond with the **AES** standard or the **GOST R 34.12-2015** Information Technology – Cryptographic Information Security – Block ciphers.

Here you can see the scheme of the text data encryption procedure based on the Diffie-Hellman protocol:

Symmetric CEK and KEK keys are used for encryption and decryption.

CEK (Content Encryption Key) is used for text data encryption. **KEK (Key Encryption Key)** is used for encryption of a CEK.



A CEK is randomly generated by a blockchain node with the use of corresponding hash coding algorithms. A KEK is generated by a node on the basis of the Diffie-Hellman algorithm with the use of public and private ktys of a sender and recipients, this key is used for encryption of a CEK.

A symmetric CEK is unavailable for reading and is not demonstrated during the encryption process. It is transferred from a sender to a recipient in an encrypted format (**wrappedKey**) via insecure channels together with an encrypted message. An example of such insecure channel is a data transaction *112* for recording of data into a blockchain or smart contract state. A KEK will not be transferred from a sender to a recipient: it is restored by a recipient on the basis of his closed key and a known public key of a sender (**Diffie-Hellman key exchange** algorithm).

Learn more about *encryption* woth the use of gRPC methods.

In order to encrypt network channels, the **TLS v. 1.2 (AES-256 CBC SHA)** algorithm is used. If the GOST encryption is used, the protocol establishes a TLS-like connection with the use of the 'Kuznyechik' encryption algorithm.

See also

gRPC: encryption and decryption methods

REST API: encryption and decryption methods

CHAPTER

TWENTYEIGHT

PERMISSIONS

The Waves Enterprise blockchain platform realises a permissioned blockchain model: only authorized participants can have an 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*.

28.1 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*.

blacklister

A participant with the `blacklister` role is entitled to temporarily or constantly restrict activity of other participants by adding the `banned` role to their accounts. To do this, a `blaklister` 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.

connection-manager

A participant with the `connection-manager` role is entitled to connect and disconnect network nodes. As a rule, a network administrator is also appointed as a `connection-manager`.

Learn more about node connection and disconnection in the article *Connection and removing of nodes*.

banned

A participant with the `banned` role cannot perform any actions in the blockchain temporarily or constantly . An address with the `banned` role is added to the **blacklist** of nodes.

28.2 Permission management

A 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.

The process of participant permission adding and removing is described in the article *Permission management*.

Prior to sending of the 102 transaction, a node checks the following:

1. A sender of the 102 transaction is not included in the **blacklist**.
2. A sender address has the `permissioner` role.
3. The `permissioner` role of the address is active at the moment of transaction sending.
4. A role stated in the 102 transaction is not active in case of its adding to the address and, vice versa, is not active in case of its removing.

Adding and removing of permissions is performed upon broadcasting of corresponding transactions in the blockchain. Permissions can be arbitrary combined for any address, definite permissions can be removed at any moment.

See also

REST API: information about permissions of participants

Description of transactions

CHAPTER

TWENTYNINE

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, statistcal 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 groups;
- **Network settings** - information about network nodes, registration of new nodes and leasing calculation;
- **Write to us** - form of contact with the Waves Enterprise technical support service.

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 rignt 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.

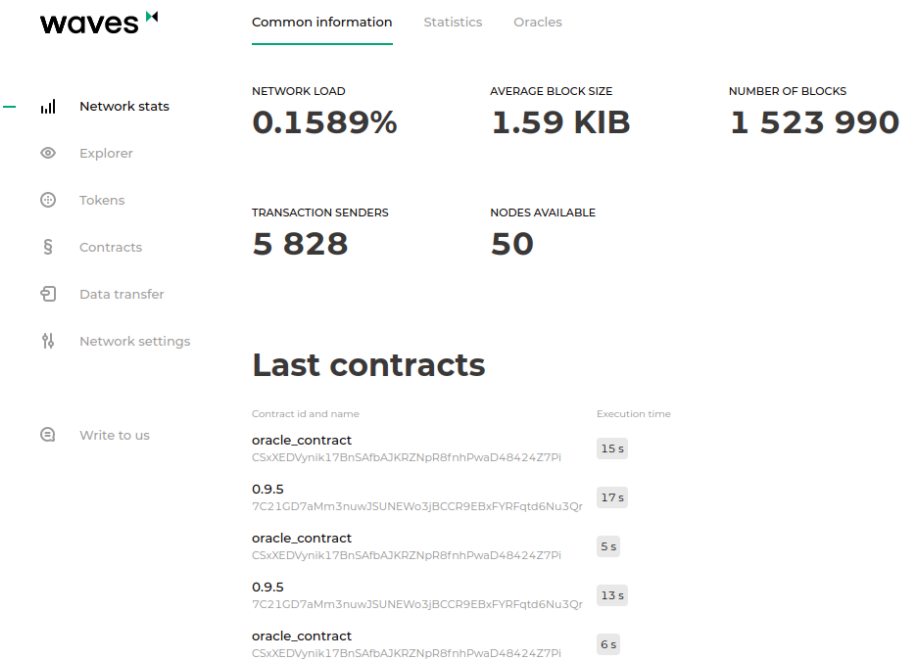
29.1 Network stats

The **General information** tab 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.

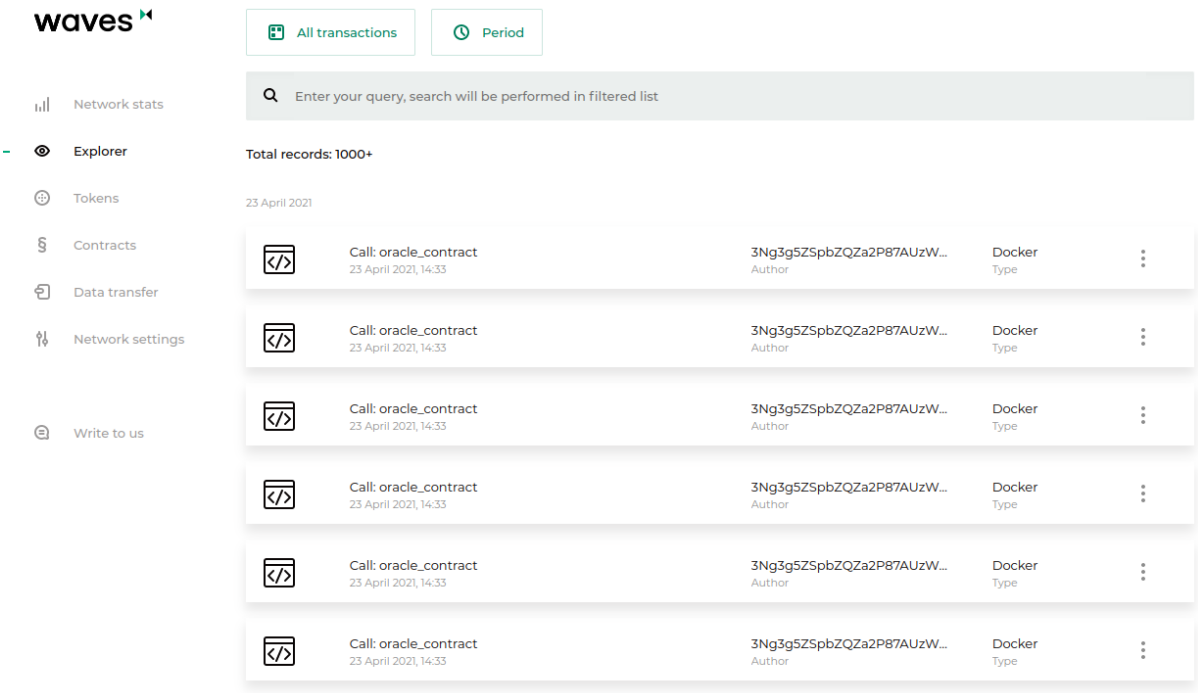
The relative chart shows dependence of WEST price from conventional assets in the following pairs:

- 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.

29.2 Explorer



The "Explorer" section contains information about transactions in the blockchain. Broadcasting timestamp is availabe as a a search filter, as well as following categories:

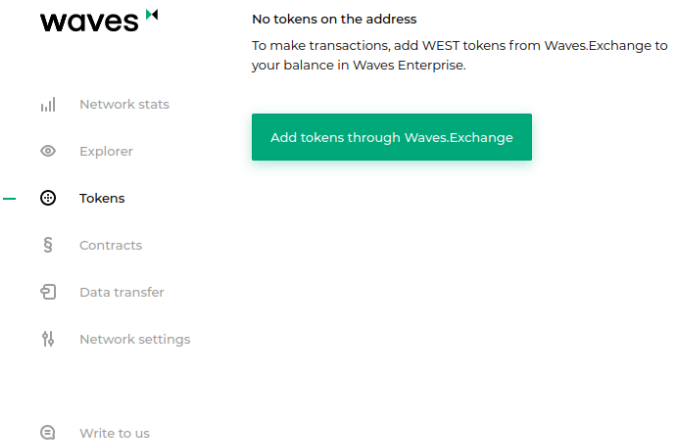
- participants;
- data transactions;
- transaction identifiers;
- names of smart contracts;
- transaction signatures;
- number of a block containing transactions.

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* - management of confidential data groups;
- *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.

29.3 Tokens



If you do not have tokens on your address, the tab 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.

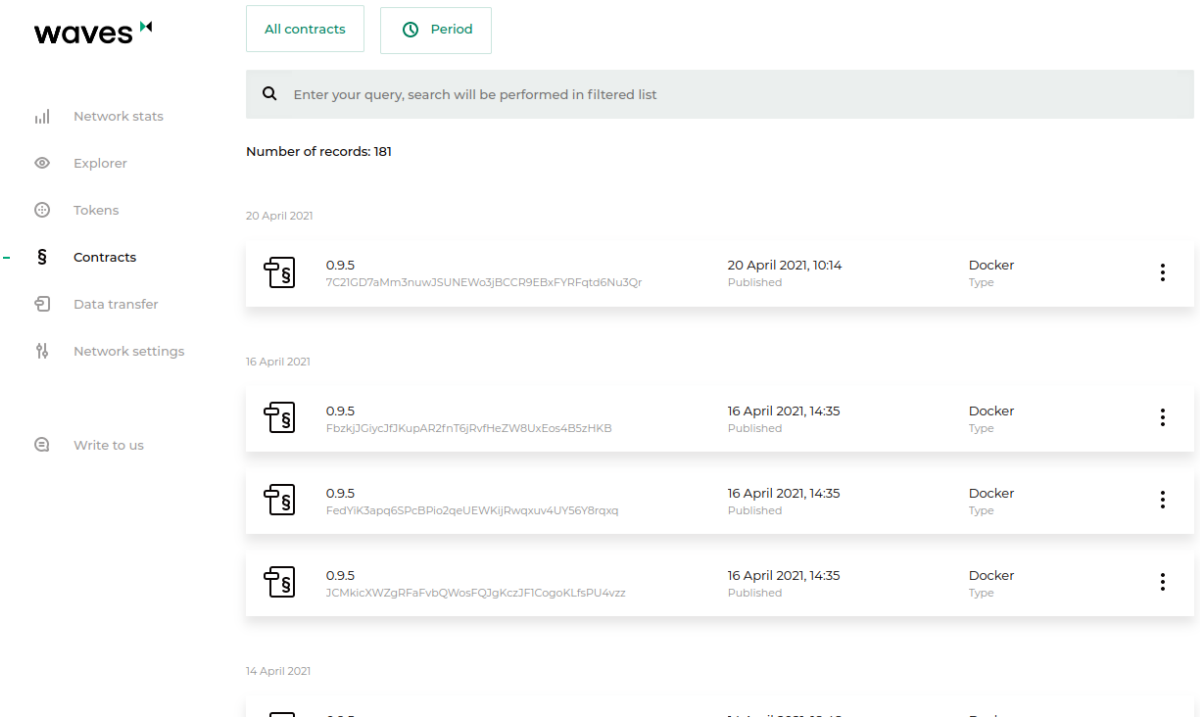
29.4 Contracts

The "Contracts" section contains information about smart contracts installed in the blockchain. It also allows to start definite smart contracts. The filtering according to following transaction parameters is available for searching of smart contracts:

- authors and senders of transactions;
- signatures;
- smartcontract identifiers;
- smart contract names;
- Docker image name.

Additional filters are also available for showing of definite smart contract categories:

- *My contracts* - 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.

The page of each smart contract contains four tabs:

- **Information** - author address, image name, checksum, smart contract version and creation date;
- **Data** - result of the last smart contract call;
- **Call** - at this tab, you can call the smart contract if you have sufficient balance on your address;
- **Version history** - a table with Docker image names, creation timestamps and checksums for each smart contract version.

Learn more about smart contracts of the Waves Enterprise blockchain platform in the article *Smart contracts*.

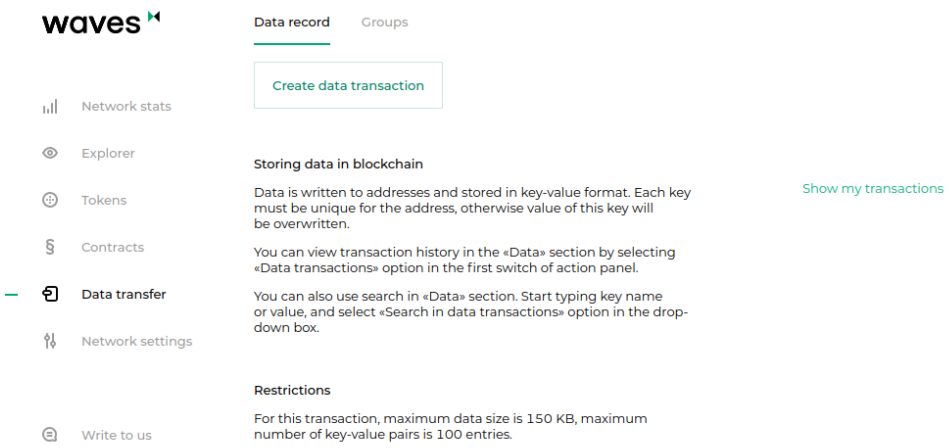
29.5 Data transfer

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.



29.6 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.

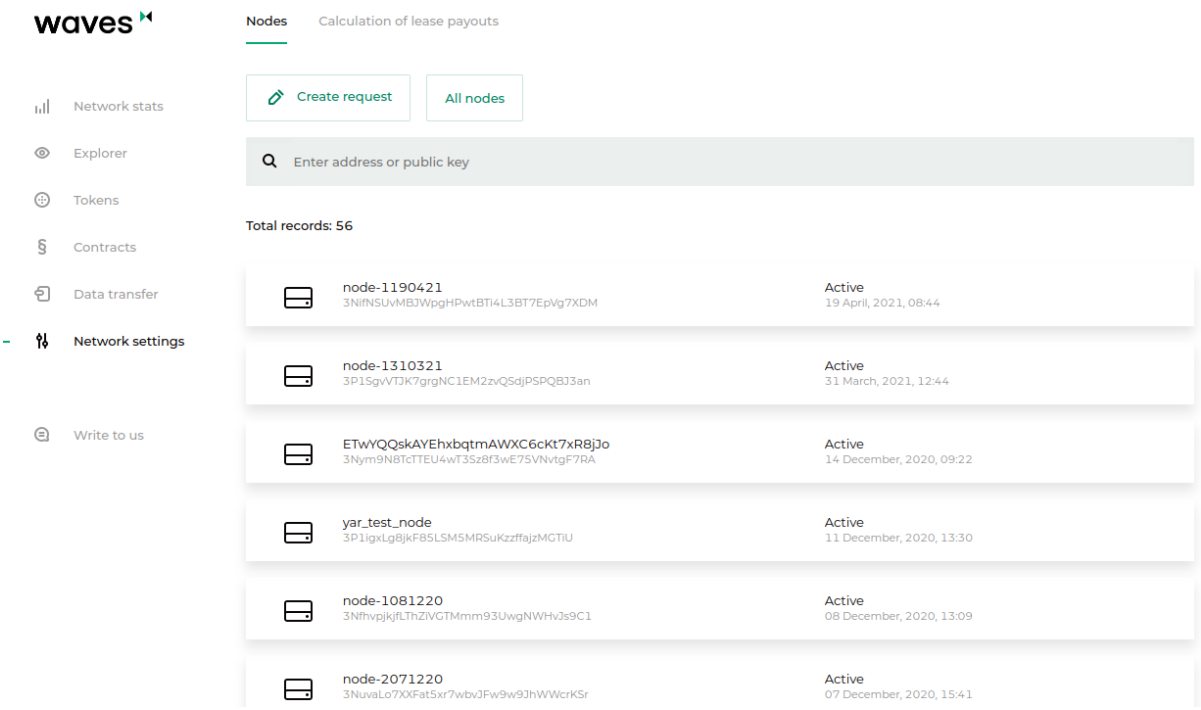
Search and filtering of nodes according to the following parameters are available:

- 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:



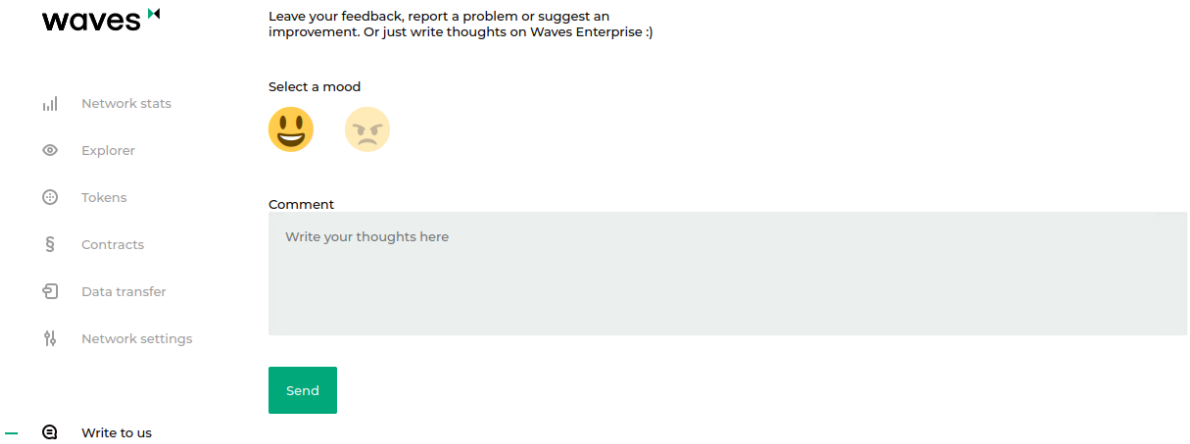
1. A generating balance is requested from the leasing pool node for the beginning of a calculated period;
2. Leasing sum is calculated taking into account miner revenues (each miner should receive 40% for his own block and 60% for a previous block);
3. The sum is divided for each pool participant proportionately 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.

29.7 Write to us

In this 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


waves
Leave your feedback, report a problem or suggest an improvement. Or just write thoughts on Waves Enterprise :)
Network stats
Explorer
Tokens
Contracts
Data transfer
Network settings
Write to us
Select a mood
Comment
Write your thoughts here
Send

CHAPTER

THIRTY

GENERATORS

Generators is a set of utilities included in the supply package of the Waves Enterprise blockchain platform. Generators are developed as a package file **generator-x.x.x.jar**, where `x.x.x` is a blockchain platform release version.

Generators for each version of the blockchain platform are available in the official GitHub repository of Waves Enterprise.

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;

30.1 AccountsGeneratorApp

The **AccountsGeneratorApp** is used for configuration of a node account in a private network - a set of data about a blockchain network participant. In order to generate an account, you have to set up the **accounts.conf** file in the node directory.

Running of the **AccountsGeneratorApp**:

```
java -jar generators-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 generating in the section *Creation of the node account*.

30.2 GenesisBlockGenerator

The **GenesisBlockGenerator** is used for signing of a private network genesis block - a 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 generators-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 *Signing of a genesis block and start of the platform*.

30.3 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 generators-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, that are stated in the `auth` section of the node configuration file.

Learn more about gRPC and REST API authorization in the section *Additional configuration of the platform: configuration of the gRPC and REST API authorization*.

See also

Architecture

CHAPTER

THIRTYONE

AUTHORIZATION AND DATA SERVICES

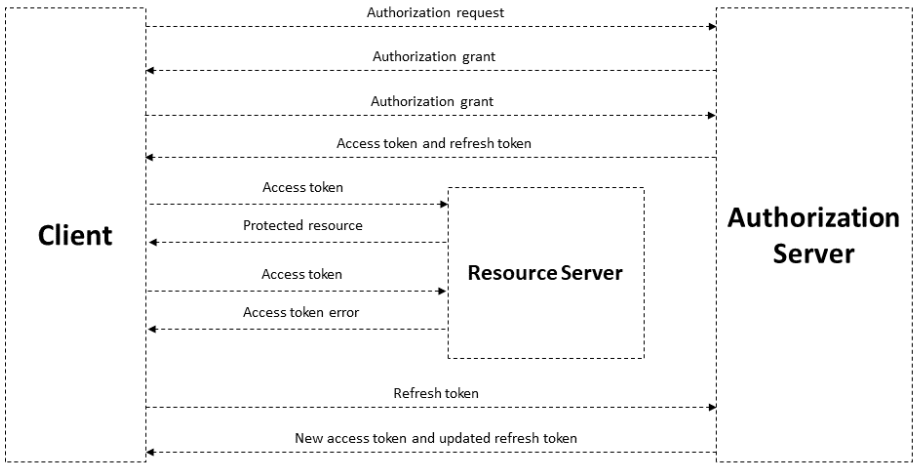
The Waves Enterprise blockchain platform includes two external services:

- **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.

31.1 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 restricted access to protected resources of a user to a third party without disclosing of logins and password.

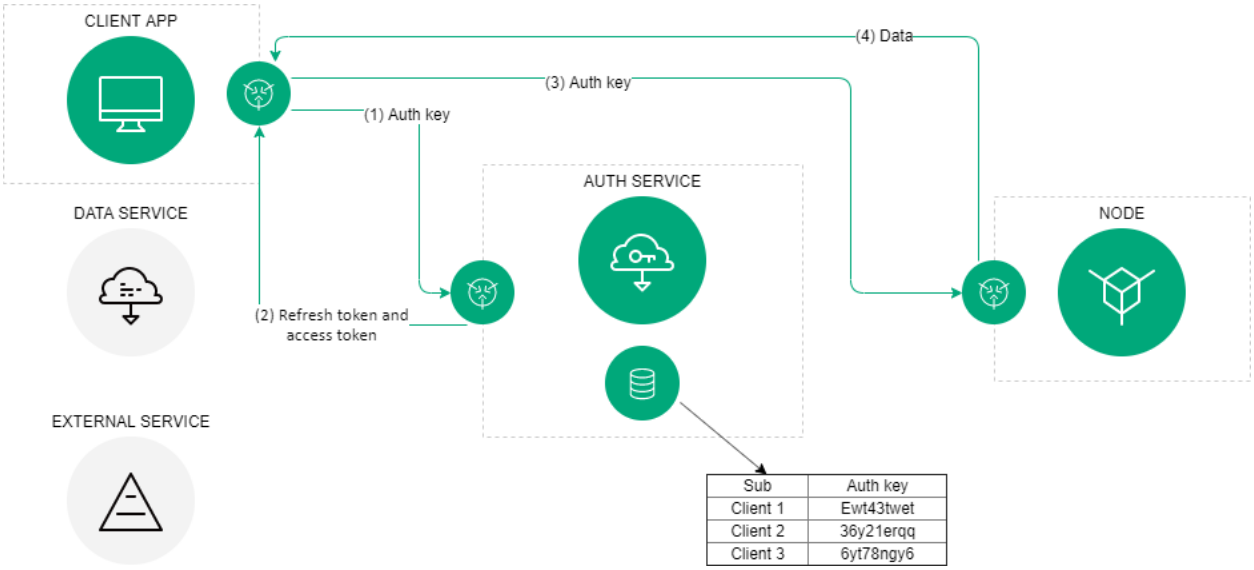
The general chart of the oAuth 2.0 authorization:



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:

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 authentication data of the client in the data storage, generates the signed access and refresh tokens and sends them to the client. The tokens contain their lifetime and basic data of the client: its identifier and role. Authentication data of clients are 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 expiry, a client refers to the authorization service for obtaining of a new access token. For this queries to the authorization service, a client uses a 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 an access token signature. In case of a successfull check, a third party service provides required data to a client.

Description of authorization methods is provided in the article Authorization service: authorization variants.

31.2 Data service

The data service is used for gathering of 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 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.

31.3 API methods of the integration services

Definite REST API methods are available for the integration services for data exchange:

31.3.1 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` - language of e-mail notifications (possible variants: *en* and *ru*);
- `source` - user type: `license` - owner of a blochchain 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: :animate: fade-in-slide-down

```
{
  "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: :animate: fade-in-slide-down

```
{
   "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 blochchain platform license ; * `voting` - user of the Waves Enterprise Voting service.

Example of the service response:

POST /v1/user/password/restore: :animate: fade-in-slide-down

```
{
   "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: :animate: fade-in-slide-down

```
{
  "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 restoring code to a specified e-mail.

The method query contains following data:

- `email` - user e-mail;
- `source` - user type: * `license` - owner of a blochchain 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` - language of e-mail notifications (possible variants: *en* and *ru*);
- `source` - user type: * `license` - owner of a blochchain 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
KvV870HmZuazjIgZVSkOnOY7F8UUVNXnlzVD1dPsOGJ6orM41DnC1W65mCrP3bjn
fV4RbmykN/lk7McA6EsMcLEGbKkFhmeq2Nk4hn2CQvoTkupJUnOCP1dhO4bq1lQ7
Ffj9K/FJq73wSXDoH+qqdRG9sfrtgrhtJHerruhv3456e0zyAcDO8+sJUQFKY80B
SZMEndVzFS2ub9Q8e7BfcNxTmQPM4PhHO5wuTqL32qt3uJBx2OI4lu3OND44ZrDJ
BbVog73oPjRYXj+kTbwUZI66SP4aLcQ8sypQyLwqKk5DtLRozSNOOIrupJJ/pwZs
9zPEggL91T0rirbEhGlf5U8/6XN8GVXX4iMk2fD8FHLFJuXCD7Oj4JC2iWfFDC6a
uUkwUfqfjJB8BzIHkncoqOZbpidEE2lTWl+svuEu/wyP5rNlyMiE/e/fZQqM2+o0
cH5Qow6HH35BrloCSZciutUcd1U7YPqESJ5tryy1xn9bsMb+On1ocZTtvec/ow4M
RmnJwmOj1nd+cc19OKLG5/boeA+2zqWu0jCbWR9c0oCmgbhuqZCHaHTBEAKDWcsC
VRz5qD6FPpePpTQDb6ss3bkCAwEAAQ==
-----END PUBLIC KEY-----
```

See also

Authorization and data services

data-sv-conf

auth-sv-var

REST API: methods of the data service

31.3.2 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 of a list of assets available in the blockchain. The list consists of transactions for emission of 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 blockchcain.

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` - number of tokens in an asset;
- `decimals` - number of digits after decimal dot 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 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 mining rounds to be skipped in case of use of the *PoA* consensus algorithm;
- `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` - status of a smart contract at the moment of a 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/id/–id″

The method returns information about a smart contract with a definite `{id}`.

Response example:

GET /contracts/id/–id″:


```
{
  "id": "string",
  "type": 0,
  "height": 0,
  "fee": 0,
  "sender": "string",
  "senderPublicKey": "string",
  "signature": "string",
  "timestamp": 0,
  "version": 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"

Returns a history of changes of a `{key}` key for a smart contract with a definite `{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,
```


(continues on next page)

(continued from previous page)

```
        "result_value_boolean": true,
        "result_value_binary": "string",
        "result_value_string": "string",
        "position_in_tx": 0,
        "contract_id": "string",
        "time_stamp": "string"
      }
    ]
  }
]
```

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 of 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"
```


(continues on next page)

(continued from previous page)

```
  ]
}
```

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 of 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"
    }
```


(continues on next page)

(continued from previous page)

```
  ]
}
```

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",
      "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

auth-sv-var

CHAPTER

THIRTYTWO

OFFICIAL RESOURCES AND CONTACTS

32.1 Blockchain platform official resources

- Official website of the Waves Enterprise blockchain platform
- Github page of the project
- Official website of the Waves blockchain platform

32.2 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

CHAPTER

THIRTYTHREE

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

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 stack. Microblocks form a network block, the number of transactions in a microblock is limited to 500 units

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
Stored blockchain data set:

- states of network addresses;
- states of smart contracts in the network;
- data of miners of the previous rounds;
- information of confidential data access groups.

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 smart contract performance data recorded and updated with the transaction *104*

Token

1. A blockchain unit used to motivate participants to mine on the network
2. The object used to authorize the blockchain participant

Transaction
A separate operation in the blockchain changing the network state and performed

Participant
User of the Waves Enterprise blockchain platform software, sending transactions to the blockchain

Forl
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

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

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

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

JWT-токен (JSON Web Token)
JSON-formatted object used to authorize a blockchain participant using the oAuth protocol

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

CHAPTER

THIRTYFOUR

WHAT IS NEW AT WAVES ENTERPRISE

34.1 1.6.0

The 1.6.0 is the last released version, and and marked as **latest** in this documentation.

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*

34.2 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.

34.3 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

Following articles have been modified:

- Cryptography
- Managing permissions
- Transactions
- Preparation of configuration files
- Changes in the node configuration file

- Description of the node configuration file parameters and sections
- Consensus configuration
- Node API tools
- JavaScript SDK
- *Glossary*
- Content of the Docker configuration article has been transferred to the new article Preparing to work
- The article Docker smart contracts with the use of the node REST API has been removed from the index

34.4 1.4.0

Following articles have been added:

- Atomic transactions
- Working in the web client
- JavaScript SDK

Following articles have been modified:

- *Architecture*
- Transactions
- Authorization type configuration for the REST API and gRPC access
- Node API tools
- Node update

34.5 1.3.1

Following articles have been added:

- Parallel contract execution

Following articles have been modified:

- Creating a smart contract
- Docker configuration

34.6 1.3.0

Following articles have been modified:

- Client
- 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 configuration
- Addresses REST API methods
- Node REST API methods
- Contracts REST API methods
- Privacy REST API methods
- *System requirements*

34.7 1.2.3

Following articles have been modified:

- Docker smart contracts
- Description of the node configuration file parameters and sections
- Privacy data access groups configuration

34.8 1.2.2

Following articles have been added:

- Debug REST API methods
- Full REST API description on the API docs

Following articles have been modified:

- Installing and running the Waves Enterprise platform

34.9 1.2.0

Following articles have been added:

- A new section Integration services, which includes Authorization service and Data service
- 'Obtaining a license' section was added
- A new REST API Licenses method was added
- New article: Smart contract run with gRPC
- New article: gRPC services available to smart contract

Following articles have been modified:

- Installing and running the Waves Enterprise platform
- Updated: Cryptography. Part of information was moved into Data encryption operations
- Changes in the node configuration file
- Transactions

34.10 1.1.2

Following articles have been modified:

- Demo version
- Changes in the node configuration file
- 'Node installation' section was converted into 'Installing and running the Waves Enterprise platform'
- Connecting participants to the network
- Anchoring configuration
- Authorization type configuration for the REST API access
- Connection of the node to the "Waves Enterprise Partnernet"
- Connection of the node to the "Waves Enterprise Mainnet"
- *System requirements*

34.11 1.1.0

Following articles have been added:

- API methods available to smart contract
- Demo version
- Changes in the node configuration file

Following articles have been modified:

- Docker smart contracts
- Example of starting a contract
- Node installation
- Additional services deploy

34.12 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

INDEX

T

U

V