PascalCoin
Exchange Integration Guide

Compatible with **version 4** of PascalCoin

Introduction

This document will explain how to integrate PascalCoin V4 into your exchange with the help of the inbuilt JSON RPC API.

In PascalCoin there are two types of digital assets that can be exchanged:

- **PASC**
  This is the cryptocurrency and is the most common type of asset. It has 4 decimals.

- **PASA**
  These are PascalCoin accounts which are necessary to send and receive PASC.

Most exchanges will only trade PASC and not PASA. PASC can be exchanged in the same way as other cryptocurrencies are.

However, trading PASA is different since they are non-fungible assets. This means that some PASA are more valuable than others, due to their desirable numbers and/or names. For PASA exchanges, an auction-style market is recommended, such as PascWallet[1].

In order to support a large number of users, exchanges do not require a PASA for each user. They are able to integrate PascalCoin into their exchange using a single PASA owned by the exchange which acts as a custodian account for all their users.

User deposits can be identified via payload inside the deposit transactions.

This documentation focuses on integrating PASC.

Make sure to explore https://www.pascalcoin.org/development/rpc for examples on how to use the JSON-RPC API of PascalCoin.

[1] http://www.pascwallet.com

Install PascalCoin

Installing PascalCoin is simple and can be installed on Linux or Windows. Minimum requirements are 2GB RAM, 2 CPU's and at least 20GB of storage. You can find the downloads on our github.com releases page[2].

The server should be isolated from the public network! The following ports are used by your node:

Port 4004
This is the default PascalCoin network protocol port. It is used to keep your installation in sync with the network, so it must be accessible publicly.

Port 4003
This is the default JSON RPC port. This port should only be accessible by the IP’s that are allowed to communicate with PascalCoin to either query the blockchain or to create transactions.

The PascalCoin software releases contains 2 executables:

1. **GUI wallet**
   The GUI wallet is the wallet that is also used by the end-users of PascalCoin. It’s a full featured wallet, explorer, node and API server at the same time
2. **The Daemon**
   The daemon is a service that provides the same functionalities as the GUI, just without GUI.

It is advised to use the daemon for your server(s), but during development it might help to use the GUI wallet to have a visual feedback on everything that happens in your exchange wallet as well as the PascalCoin Blockchain.

To start the daemon use the following command:

```
./pascalcoin_daemon[.exe] -r
```

By default, the JSON-RPC API is only accessible from the same server. The GUI as well as the daemon can be configured to allow access from more IP’s - but **not** or all IP’s.

In the GUI wallet, select “Project” → “Options” from the main menu and change the “Allowed IP’s” for the JSON-RPC API to a list of IPs divided by a semicolon.

In the daemon, open the pascalcoin_daemoni.ini file in the same folder as the executable and edit the RPC_WHITELIST value. The same rules apply as for the GUI wallet.

[2] https://github.com/PascalCoin/PascalCoin/releases

Prepare your wallet

After installing and starting the software, the wallet will sync with the network. While this happens, you must set a wallet password to make sure your wallet is safe.

By default the password is an empty string, so the wallet is not protected. Everyone with access to it (e.g. JSON RPC) can execute private key related methods and possibly drain your wallet.
If you use the GUI, go to Project Private keys and set the wallet password. If you use the daemon, use the JSON-RPC method setwalletpassword[3] to change it.

A PascalCoin installation manages a single wallet. Each wallet contains one or more public/private key pairs as well as one or more public keys. Each public key holds one or more PASA.

Once you set the password, operations related to private keys (e.g. making a transaction) require you to unlock the wallet, execute the private key related JSON-RPC method and lock the wallet again using the lock[4] and unlock[5] JSON RPC API methods.

[3] https://www.pascalcoin.org/development/rpc#setwalletpassword
[4] https://www.pascalcoin.org/development/rpc#lock
[5] https://www.pascalcoin.org/development/rpc#unlock

Create your Exchange's key

In order to send/receive PASC with your installation, you'll need at least 1 key-pair. This key-pair will be associated to your exchange account (PASA).

On the first run, PascalCoin will examine your wallet for keys and if none are found, it will add one for you automatically.

To add new key pairs to your wallet, you can use the JSON RPC addnewkey[6] method.

Calling this method will generate a new private/public key pair. The private key will be saved by the application internally. The Public Key is returned and accessible.

For operational and maintenance simplicity, it's a good idea to name your key in a relevant way in order to avoid later confusion. Key names are only stored in the wallet and are not public.

It is advised to start with only one key-pair for your integration. You can add other keys later and manage multiple PASAs to, e.g., avoid paying fees when withdrawing by using a load-balancing approach with multiple keys.

It is important that you backup your keys. As mentioned above, the keys are stored internally. They are stored in a file in the home directory of the user that runs the wallet.

Unix/Linux Systems
Your keys as well as the blockchain data are stored in the **`%APPDATA%/PascalCoin`** folder.

Windows Systems
Your keys as well as the blockchain data are stored in the **`$USER/PascalCoin`** folder.

The file WalletKeys.dat in the folder contains all your keys, so this is the file you'll need to backup. This file can also be copied to other installations of the software for re-use.

Please ensure that your wallet keys are backed up all the time.

To programmatically determine the available keys in PascalCoin, you can use the getwalletpubkeys[7] method.

[6] https://www.pascalcoin.org/development/rpc#addnewkey
[7] https://www.pascalcoin.org/development/rpc#getwalletpubkeys

Acquire your exchange account

In order to receive deposits and send withdrawals, your exchange needs an account (PASA). Getting an account is easy:

- **PascalCoin Team**
  Ask the PascalCoin team for a proper exchange account.
- **getpasa.com**
  Buy a random account for 3 PASC
- **pascwallet.com**
  Buy an account you like via this PASA exchange
- **Discord server**
  Join #free-pasa-bot to obtain a free random PASA
- **In-Wallet purchase**
  Use the wallets in-protocol PASA exchanging features. Many accounts are available for purchase directly in the network.

It is advised to use a recognizable and memorable PASA account number for your exchange.

PASA account numbers are 32 bit unsigned integers. They contain a checksum that makes it easier for users to verify if the account they entered is valid. The checksum is appended via “-” after the account number and is calculated as follows:

```
((AccountNumber * 101) MOD 89)+10
```

After you acquired an account, you are able to either receive or send PASC.

Understanding payloads

Accepting and assigning deposits is easier as it is in other currencies. Everything is managed via payloads, which, in essence, are nothing more than additional data that can be added to a transaction and can be read by certain persons. Payloads from the API are in HEX format (called HexaString in the docs) without any preceding 0X. You need to instruct your users to include a payload with their transaction to your exchange account to identify a deposit. This payload needs to be encoded to hex by you in case of a withdrawal or decoded by you in case of a deposit.

Payloads can be encrypted by the user. This functionality is explained in detail here[8], but to give you a brief introduction, we will explain the possibilities in short here. Please refer to the documentation for more info.

- **No encryption (none)**
  No encryption will take place and the payload is added to the operation "as is". Everyone can read the payload.

- **Destination public key (dest)**
  The encryption will be executed using the destination PASA public key. PascalCoin will take the public key that owns the receiving account (destination) and encrypts the payload. The payload can only be decrypted by the owner of the receiving public key (the one that also knows the private key). No-one else will be able to decrypt the value.

- **Senders public key (sender)**
  The encryption is done through the senders public key. PascalCoin will take the public key of the sender account and encrypts the payload.
  The payload can only be decrypted by the owner of the public key (the one that also knows the private key). No-one else will be able to decrypt the value.

- **AES256 with password (pwd)**
  The third and last encryption method is to use a password. PascalCoin will take the defined password to create an initialization vector (IV) and Key to encrypt the payload using AES256-CBC. The payload can be decrypted by everyone who knows the password.

This means that you cannot accept every transaction that arrives in your PASA. You can only read payloads the are either unencrypted or encrypted using your public key.

How you handle invalid/unrecognizable transactions is up to you. Sending back the transaction to the sender is a valid approach and is covered at the end of this document.

The withdraw payload should be free to set by your users on your exchange.

[8] https://www.pascalcoin.org/content/pascalcoin_payloads

Proposed workflow for accepting user deposits

Suppose the exchange account (PASA) is **12345-67**. All deposits and withdrawals will operate through this account on your exchange.

This is a proposed workflow to show how it **can** work, but if your structure is completely different and not adaptable from your point of view feeld free to get in touch with the PascalCoin team.

Every action that alters the PascalCoin BlockChain is called an operation. There are various types available, but we will only cover 1 operation type which is an operation to transfer PASC from one account (PASA) to another account.

Users Table

Imagine your users table to look like this. Each user in your database contains a unique payload string to identify PascalCoin deposits. Payload strings are at max a string of 512 bytes.

ID	EMail	Name	pascal_payload	Other fields
523	bob@mail.com	Bob	XH42SD	...
10655	alice@mail.com	Alice	S82DG2	...

Deposits / Withdrawals Table(s)

Imagine you record each deposit and withdrawal of your users. This is how it might look like:

ID	type	User-ID	PASC OPHASH	status	Other fields
1	deposit	523	XXXXX..	success	...
2	withdraw	10655	YYYYY..	error	...
3	withdraw	10655	ZZZZZ..	pending	...

The operation hash of a PascalCoin operation is a string of 32 bytes[9].

[9] https://github.com/PascalCoin/PascalCoin/wiki/Ophash-(txID)

Accepting user deposits

Your node is operational. You have setup your keys and have a working PASA. Now you need to scan for deposits using the same workflow you would use for other cryptocurrencies.

Here is an example scenario:

- For Bob to deposit PASC into his exchange account he sends transactions to 12345-67 with payload XH42SD.

- For Alice to deposit PASC into her exchange account she sends transactions to 12345-67 with payload S82DG2.

The users payload encryption can either be none (no encryption) or dest (using your public key) for you to identify the deposit..

It is advised to follow this workflow:

1. **(optional) Get pending operations**
   If you want your users to get immediate feedback of a transaction to your account (deposit) or to their account (withdraw), use the method getpendings[10]. This will list all pending transactions that are not included in any block yet. Pending operations are visible more or less immediately and the amount of the transaction is already reflected in the receiving wallets balance (0-confirmation).

2. **Get operations of new block**
   Use getblockoperations[11] to fetch the operations of a block. Use getblockcount[12] to check the number of blocks (and check for new ones).

Both of these methods give you a list of operations. Loop all these operations (take care, the getblockoperations and getpendings results are paged). WIth this list, check if:

- **operation.optype = 1**
  This identifies a transaction.
- **operation.receivers[0].account = exchange account**
  This identifies the receiving account of the transaction (yours).
- **operation.receivers[0].payload = match in database**
  This is the payload of the transaction for you to use to identify your user.

[10] https://www.pascalcoin.org/development/rpc#getpendings
[11] https://www.pascalcoin.org/development/rpc#getblockoperations
[12] https://www.pascalcoin.org/development/rpc#getblockcount

Payloads are user submitted values, so make sure you don't use them in your (database) backend without proper escaping!

To decrypt a payload and associate the transaction with one of your users the following way is advised:

1. Take the payload as is and transform the Hex-String to a string (no encryption)
2. Check your user database if there is a payload associated with the payload of the transaction. If one is found, the transaction is valid.
3. If the above fails, try to decrypt the payload using the payloaddecrypt[13] RPC API and a possible set of passwords. PascalCoin will try to decrypt the payload with the available private keys in your wallet and the possible passwords (can be left empty). Repeat 1 and 2 with the decrypted payload.

If the above fails, the payload got encrypted and you'll will never know what the payload means. Possible reasons:

- The user used his own public key to encrypt the payload.
- The user used an unknown password to encrypt the payload.
- Or.. there is a misunderstanding and the sender simply used a wrong payload value.

In case the transaction can be identified successfully, record it in your deposit/withdrawal database.

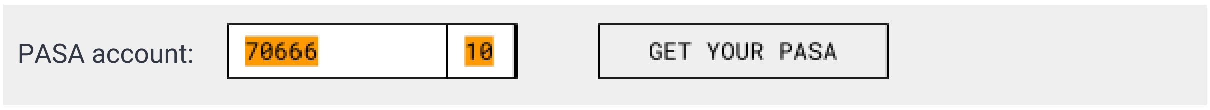
Each operation contains a field called maturation. It shows the age of an operation in terms of blocks. To verify a deposit to your account or to handle malicious transactions, it is advised to wait for at least 2 confirmations.

[13] https://www.pascalcoin.org/development/rpc#addnewkey

Withdrawing

When a user wants to withdraw it’s PASC from your exchange, your withdrawal form should (at best) look like this:

PASA Account



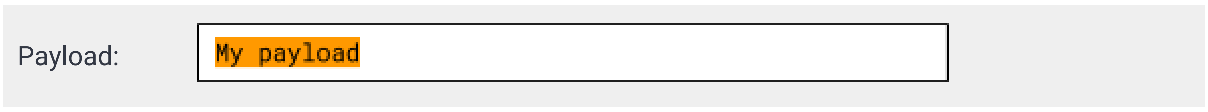
PASA account: `70666` `10` GET YOUR PASA

The account to withdraw to.

Checksums of accounts are an important feature to make sure the user did not input a wrong account number.

Additionally you could add a button for users to get their own PASA. You can either distribute your own PASA and subtract the price from the withdrawal or use an external service like getpasa.com.

Payload



Payload: `My payload`

The payload the user wants to specify. You can add another option to let the user define the encryption method of the payload, but if not it is advised to not use any encryption (none).

Use the sendto[14] JSON RPC method to create a new transaction.

When an operation is created, it will be in the pending state. So no block is associated with it and it will have no maturation.

The resulting operation hash starts with 8 zeros until the operation is included. After the operation got included, the first 8 bytes will represent the block number, so the ophash will change! The JSON-RPC findoperation[15] method is able to find the operation with either the pending ophash or the altered ophash after the operation is included. Just make sure your backend can handle the change of an operation hash.

[14] https://www.pascalcoin.org/development/rpc#sendto
[15] https://www.pascalcoin.org/development/rpc#findoperation

Checklist

This checklist can be used to make sure your implementation works.

- ❐ **Check that your node is secured**
  Port 4004 open, port 4003 only accessible for allowed IPs

- ❐ **Deposit with valid payload using no encryption.**
  This deposit should be successful.

- ❐ **Deposit with valid payload using destination encryption.**
  This deposit should be successful.

- ❐ **Deposit with valid payload using sender encryption.**
  This deposit should fail. Optional: send back the transaction with the origin ophash.

- ❐ **Deposit with valid payload using a password.**
  This deposit should fail. Optional: send back the transaction with the origin ophash.

- ❐ **Withdraw to an invalid account.**
  This should fail, use getaccount[16] to check if an account exists.

- ❐ **Withdraw to a valid account.**
  This should be successful.

- ❐ **Check if your wallet is password protected.**
  Calls to payloaddecrypt or sendto should return an error code.

[16] https://www.pascalcoin.org/development/rpc#getaccount

Appendix

A list of technical explanations.

HexaString Conversion

Pseudo Code from string to HexaString.

```
originalString
hexaString

foreach 8BitChar in originalString
    byte = to-decimal(8BitChar)
    hex = to-hex(8BitChar)
    hex = pad-left(hex, length=2, '0')
    append hex to hexaString
end foreach
```

Pseudo Code from HexaString to string.

```
hexaString
originalString

foreach 2chars in hexaString
    dec = to-decimal(2chars)
    8BitChar = to-ascii(dec)
    append 8BitChar to originalString
 end foreach
```

Examples:

Decrypted	Encrypted
Hello World	48656C6C6F20576F726C6421
email@domain.de	656d61696C40646F6D61696E2E6465
€	E282AC

Checksum calculation

The formula to calculate the checksum of an account is as follows:

```
((AccountNumber * 101) MOD 89)+10
```

Examples:

Account Number	Checksum
0	10
70666	10
257521	93
6780	24

Deposit Payload Decryption

Simple Pseudo-Code example on how to decode payloads.

```
Operation

Payload = Operation.receiver[n].payload
Decoded = HexaToString(payload)

If NOT DB.User.exists(Decoded)
  JSON-RPC.decodepayload(Payload)
  Decoded = HexaToString(payload)
  If NOT DB.User.exists(Decoded)
    NotFound()
  Else
    Deposit()
EndIf

Deposit()
```