ElectrumSV

The ElectrumSV developers

Aug 07, 2023

GETTING STARTED

ElectrumSV is a wallet application for Bitcoin SV, a peer to peer form of electronic cash. As a wallet application it allows you to track, receive and spend bitcoin whenever you need to. But that's just the basics, as it manages and secures your keys it also helps you to do many other things.

Important: ElectrumSV can only be downloaded from electrumsv.io.

CHAPTER

ONE

GETTING STARTED

Before you can send and receive payments, you need to first create a wallet, and then create at least one account within it.

How do you know you have the official software and not malware?
Every person who had their coins stolen and was interested in investigating, identified that they had not downloaded from our official web site, and had instead obtained malware from some other fake site. Many swore they downloaded from the official site until their verified their download and found it to be malware. Read more about *verifying your download*.

How do you create a wallet?
Your wallet is a standalone container for all your bitcoin-related data. You should be able to create as many accounts as you need within it, each account containing separated funds much like a bank account. Read more about *creating a wallet*.

How do you create an account?
Each account in your wallet is much like a bank account, with the funds in each separated from the others. Read more about *creating a new account*.

How do you receive a payment from someone else?
Each account has the ability to provide countless unique and private receiving addresses and by giving a different one of these out to each person who will send you coins, allows you to receive funds from them. Read more about *receiving a payment*.

How do you make a payment to someone else?
By obtaining an address from another person, if you have coins in one of your accounts, you should be able to send some or all of those coins to that address. Read more about *making a payment*.

How do you scan the blockchain?
Whether you have restored the wallet, or you are an advanced user and you have given out a payment destination ElectrumSV does not know about, it is necessary to be able to find payments related to your wallet that exist on the blockchain. Read more about *making a payment*.

1.1 Verifying your download

Probably a dozen people have reported having their coins stolen, and any who were willing to investigate found they had downloaded a malware version of ElectrumSV and not an official download. More than one asserted that they had downloaded from our web site, but what they meant was that they had downloaded from a fake web site that had stolen our design.

Downloading an executable from a web site and running it is risky, and what you are putting your trust in, is that because you download from our official web site you are getting an executable you can safely run. Despite this, well meaning people have downloaded from fake versions of our web site, and paid the price for it.

It is in your best interests to verify your download is the official one. The goal of this page is to try and show you how to do that.

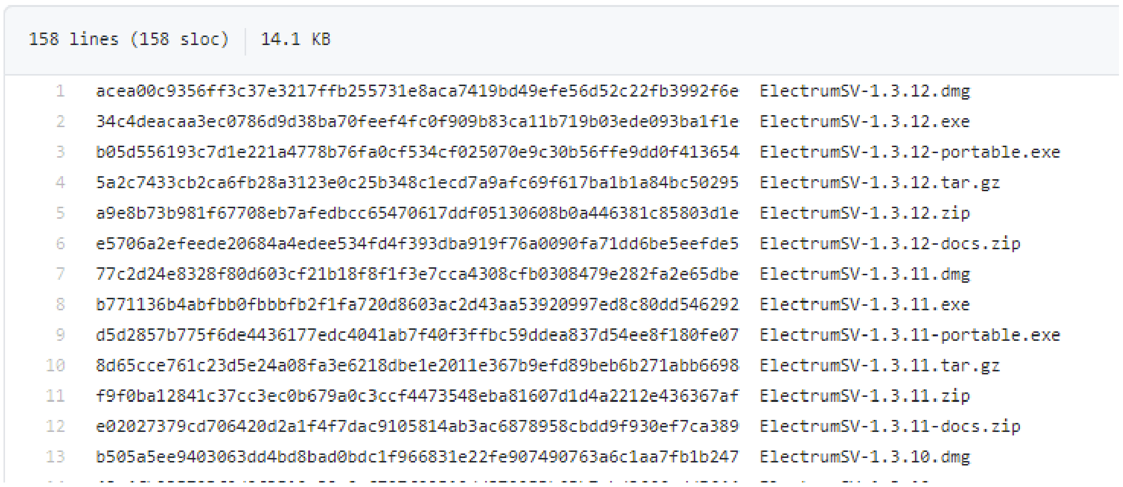
1.1.1 What are you verifying?

You will be checking the checksum (also known as a hash) of the file you downloaded. This is a standard algorithm that you can get lots of different software for, which will give you a series of letters and numbers that represent the uniqueness of your file. The algorithm we use for ElectrumSV is called SHA256 and we provide an official checksum for each file we make available. You will be comparing that official checksum to the one generated for your file. If it is the same, you should have the official version of that file. If it is different, you have downloaded malware instead.

The official checksums

We do not provide the checksums on the official web site where you find our download links, because this allows any attacker who manages to compromise the web site, to also replace the official checksums. Additionally, if there is a fake web site that offers both download links and checksums you should know something is fishy.

The official checksums are available from Github, where our open source code is located. You do not need to compare against the illustrative screenshot below, just click on the Github link and view them there.



158 lines (158 sloc) | 14.1 KB

```
acea00c9356ff3c37e3217ffb255731e8aca7419bd49efe56d52c22fb3992f6e  ElectrumSV-1.3.12.dmg
34c4deacaa3ec0786d9d38ba70feef4fc0f909b83ca11b719b03ede093ba1f1e  ElectrumSV-1.3.12.exe
b05d556193c7d1e221a4778b76fa0cf534cf025070e9c30b56ffe9dd0f413654  ElectrumSV-1.3.12-portable.exe
5a2c7433cb2ca6fb28a3123e0c25b348c1ecd7a9afc69f617ba1b1a84bc50295  ElectrumSV-1.3.12.tar.gz
a9e8b73b981f67708eb7afedbcc65470617ddf05130608b0a446381c85803d1e  ElectrumSV-1.3.12.zip
e5706a2efeede20684a4edee534fd4f393dba919f76a0090fa71dd6be5eefde5  ElectrumSV-1.3.12-docs.zip
77c2d24e8328f80d603cf21b18f8f1f3e7cca4308cfb0308479e282fa2e65dbe  ElectrumSV-1.3.11.dmg
b771136b4abfbb0fbbbfb2f1fa720d8603ac2d43aa53920997ed8c80dd546292  ElectrumSV-1.3.11.exe
d5d2857b775f6de4436177edc4041ab7f40f3ffbc59ddea837d54ee8f180fe07  ElectrumSV-1.3.11-portable.exe
8d65cce761c23d5e24a08fa3e6218dbe1e2011e367b9efd89beb6b271abb6698  ElectrumSV-1.3.11.tar.gz
f9f0ba12841c37cc3ec0b679a0c3ccf4473548eba81607d1d4a2212e436367af  ElectrumSV-1.3.11.zip
e02027379cd706420d2a1f4f7dac9105814ab3ac6878958cbdd9f930ef7ca389  ElectrumSV-1.3.11-docs.zip
b505a5ee9403063dd4bd8bad0bdc1f966831e22fe907490763a6c1aa7fb1b247  ElectrumSV-1.3.10.dmg
```

Fig. 1: The list SHA256 hashes for the official downloads.

1.1.2 Verifying your download

There is no easy way to check a download. Some level of technical competence is useful, although if you do not consider yourself technically competent and can follow instructions you should still be able to do it. Others have managed to do it, and as we get these instructions into a more approachable state over time, you should be able to as well.

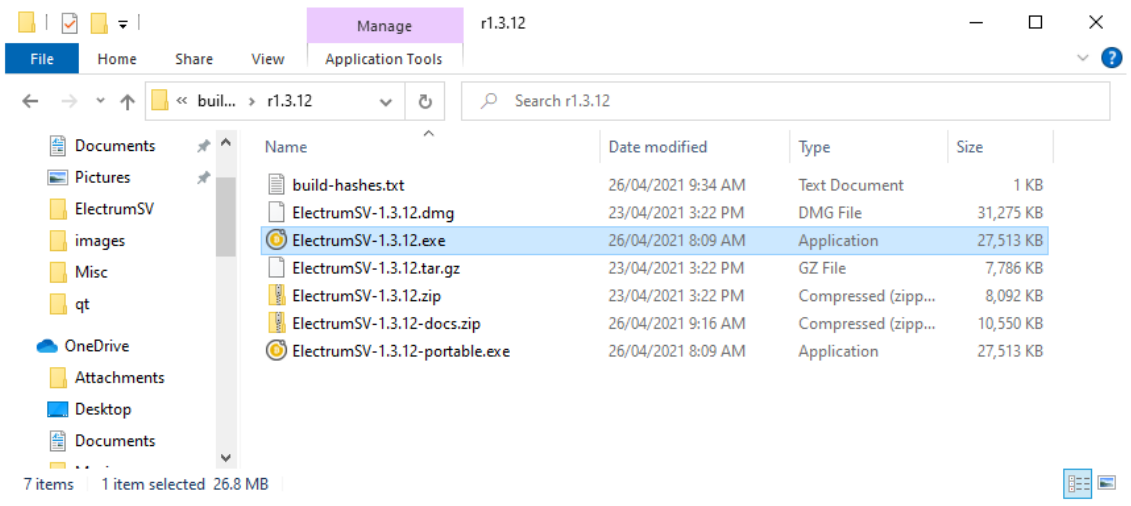
Find your operating system below, and check out the options listed for it. Some of them may be better than others, but some assurance that your download is legitimate is better than nothing.

Windows

Several methods of verifying your download on Windows are provided below. Any one should be good enough, but if you are a user who primarily uses a web browser you may need to learn to use the explorer or console.

Using the digital signatures

Thanks to the kindness of the Bitcoin Association, we now have the ability to sign our Windows executables from version 1.3.12 and above. In theory the presence of our signature on the executable you downloaded should be just as reliable as checking the checksum. You can check if the executable you downloaded has our signature, and if it is present you can assume that the file should be legitimate. Your first step is to find the executable you downloaded with the Windows explorer. You can open the Windows explorer with the `windows` and `e` key, then locate the directory your executable is located in.



Name	Date modified	Type	Size
build-hashes.txt	26/04/2021 9:34 AM	Text Document	1 KB
ElectrumSV-1.3.12.dmg	23/04/2021 3:22 PM	DMG File	31,275 KB
ElectrumSV-1.3.12.exe	26/04/2021 8:09 AM	Application	27,513 KB
ElectrumSV-1.3.12.tar.gz	23/04/2021 3:22 PM	GZ File	7,786 KB
ElectrumSV-1.3.12.zip	23/04/2021 3:22 PM	Compressed (zipp...	8,092 KB
ElectrumSV-1.3.12-docs.zip	26/04/2021 9:16 AM	Compressed (zipp...	10,550 KB
ElectrumSV-1.3.12-portable.exe	26/04/2021 8:09 AM	Application	27,513 KB

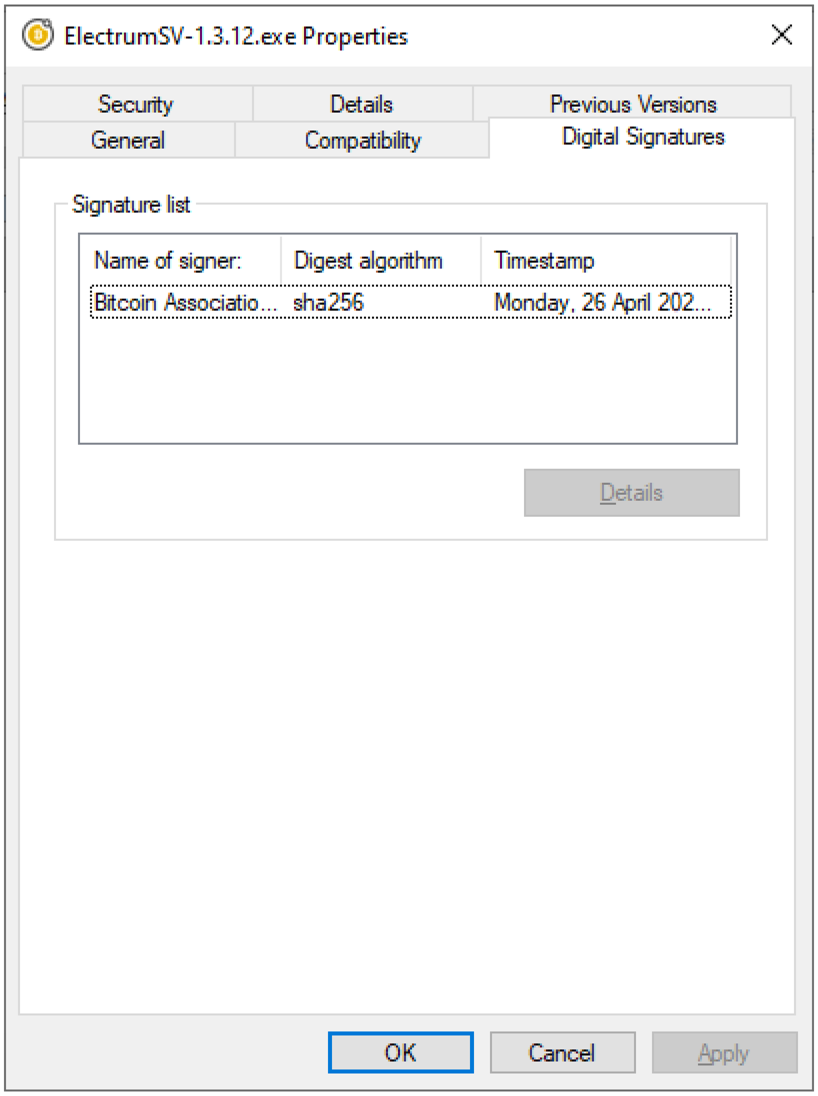
Fig. 2: Windows explorer.

Right click on the file, and select `Properties`. This should open the properties window for the file, where you should select the `Digital Signatures` tab to see the signature.

From there click on `Details` and then `View Certificate`. You should see a certificate with the following information for the given version.

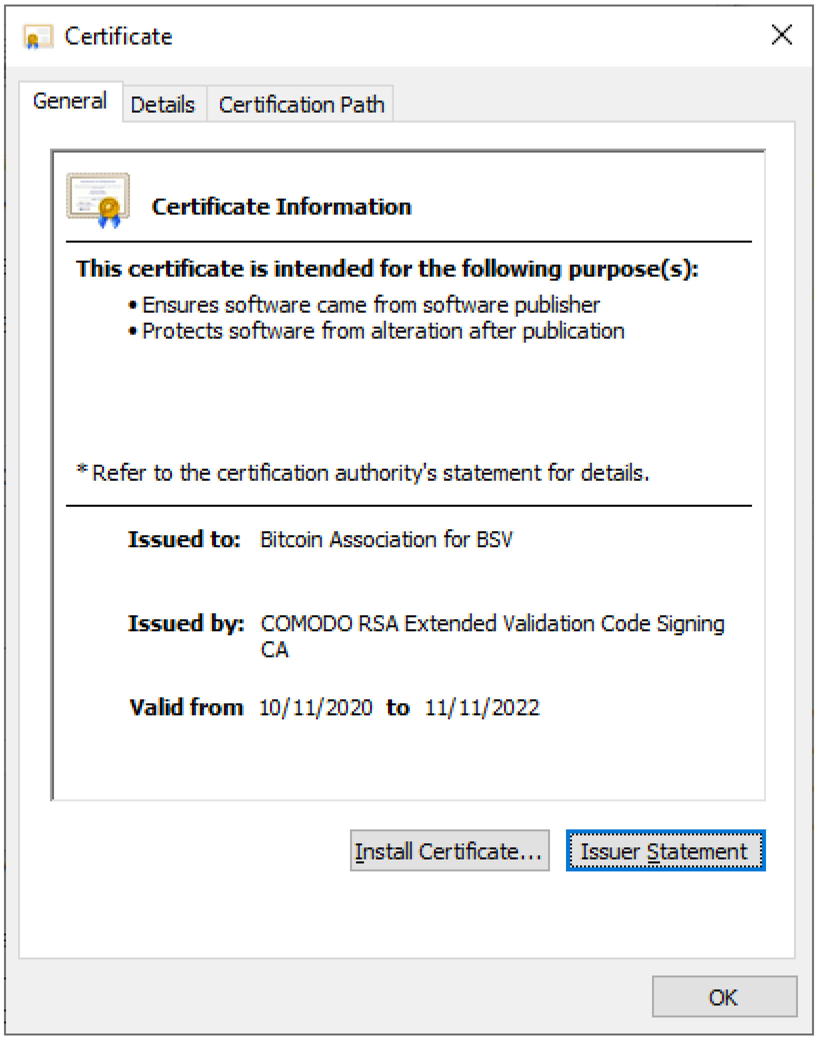
1.3.12 and above

The certificate should be issued to `Bitcoin Association for BSV`, be issued by `COMODO RSA Extended Validation Code Signing CA` and as of the time of writing be valid from `10/11/2020` to `11/11/2022`.



Name of signer:	Digest algorithm	Timestamp
Bitcoin Associatio...	sha256	Monday, 26 April 202...

Fig. 3: The digital signature.



Certificate

Certificate Information

This certificate is intended for the following purpose(s):

- Ensures software came from software publisher
- Protects software from alteration after publication

* Refer to the certification authority's statement for details.

Issued to: Bitcoin Association for BSV

Issued by: COMODO RSA Extended Validation Code Signing CA

Valid from 10/11/2020 to 11/11/2022

Fig. 4: The certificate the file was signed with.

Using certutil

`certutil` is already present in your Windows installation already. However, it requires opening a command prompt to run it, which might be something beyond some users. Press the *Windows* key and the `s` key at the same time, this will open the Windows searchy thing and there you can type `cmd` and then press the `enter` key to open a command prompt.

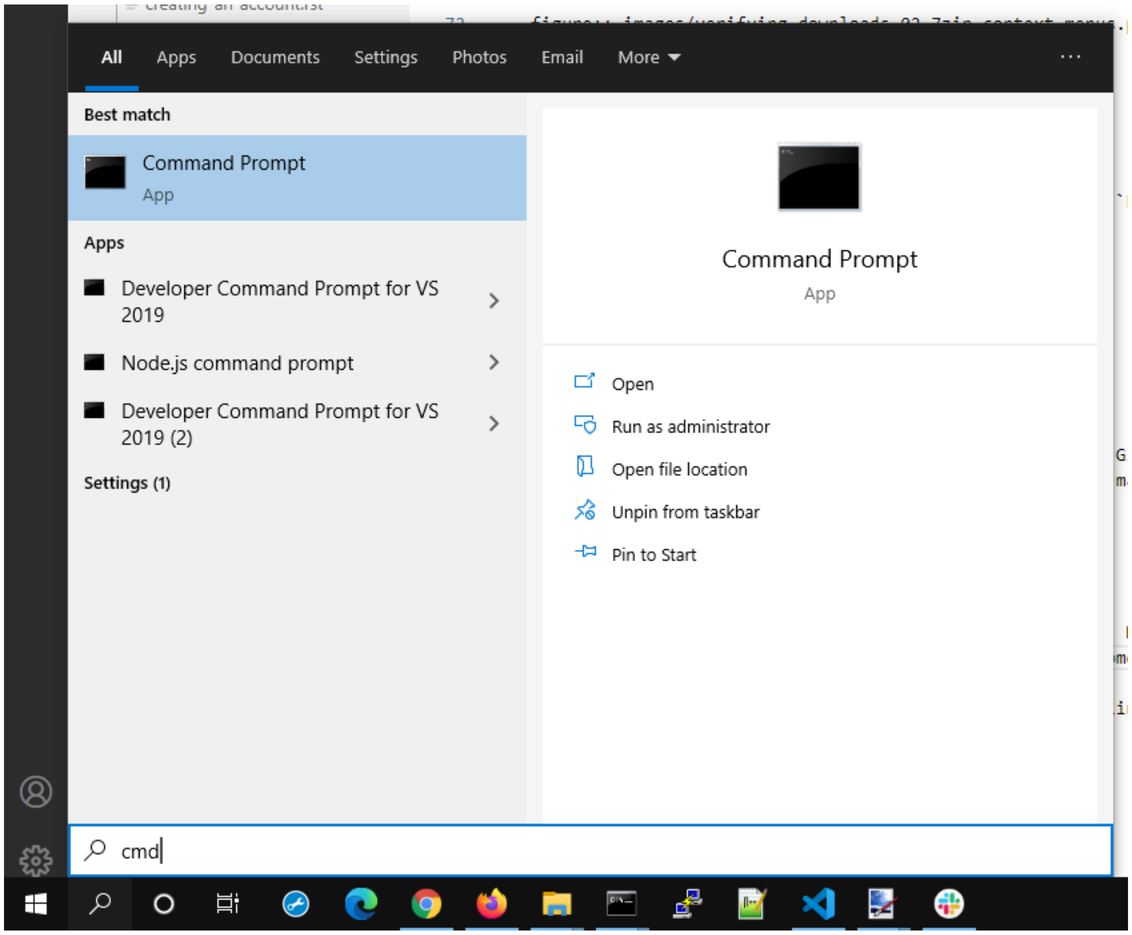
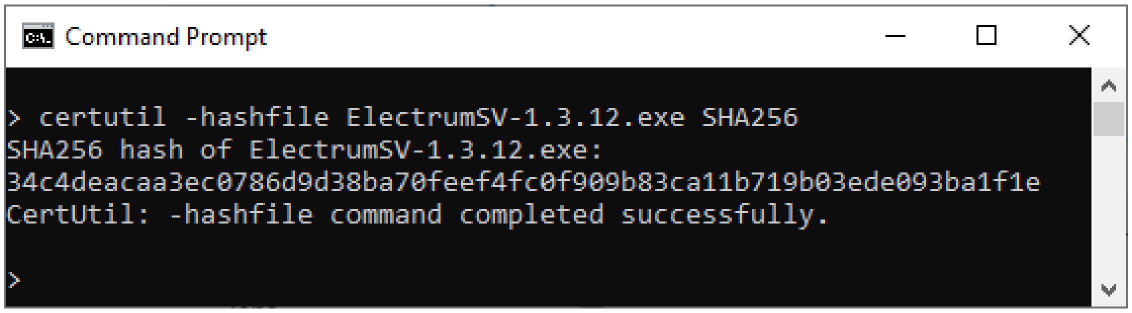


Fig. 5: Opening a command prompt.

Then you need to change the directory until you are in the same directory as the file you wish to get a checksum for. The `cd` command is used for this. Then you can use the certutil command to generate a SHA 256 checksum for that file. The syntax is `certutil --hashfile <filename> SHA256`, but remember you need to replace `<filename>` with the actual file name. You can see an illustration of this in the image below.

If you find the `ElectrumSV-1.3.12.exe` entry in the linked Github list, you can see it matches the certutil checksum result. The case of the letters does not matter, both lower case and upper case are equivalent. If you get a different result, and the command complains that it cannot find the file, then the file is not in the current directory. You need to use the `cd` command to change the current directory as mentioned above.



```
> certutil -hashfile ElectrumSV-1.3.12.exe SHA256
SHA256 hash of ElectrumSV-1.3.12.exe:
34c4deacaa3ec0786d9d38ba70feef4fc0f909b83ca11b719b03ede093ba1f1e
CertUtil: -hashfile command completed successfully.

>
```

Fig. 6: The certutil checksum result.

Using 7-Zip

This requires that you download the 7-Zip installer. Any of the non-standalone executables from the 7-Zip web site, should be fine. Download one and install it. Once it is installed, you should have a handy context menu available that can give you the SHA 256 checksum for your file. Simply select your file, open the context menu and generate the checksum. Do not reflect on the fact that no-one in their life ever wanted to "Share with Skype" and that they put it up the top before all the useful stuff.

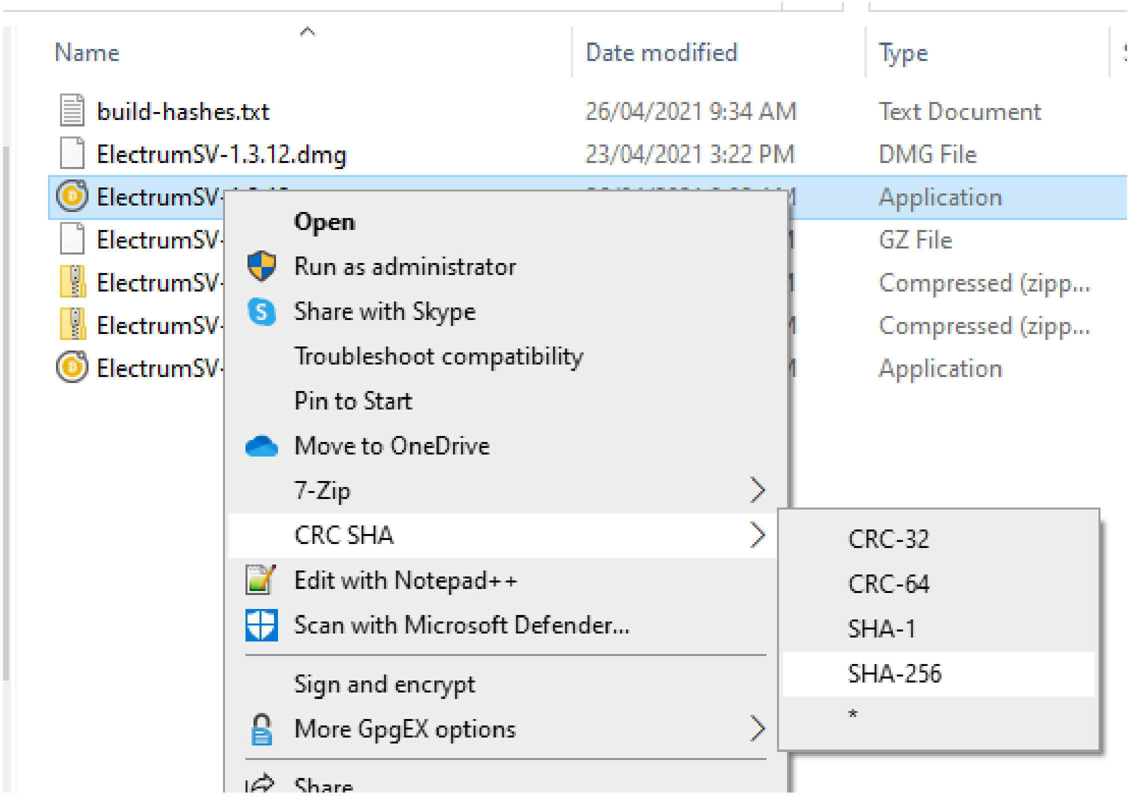
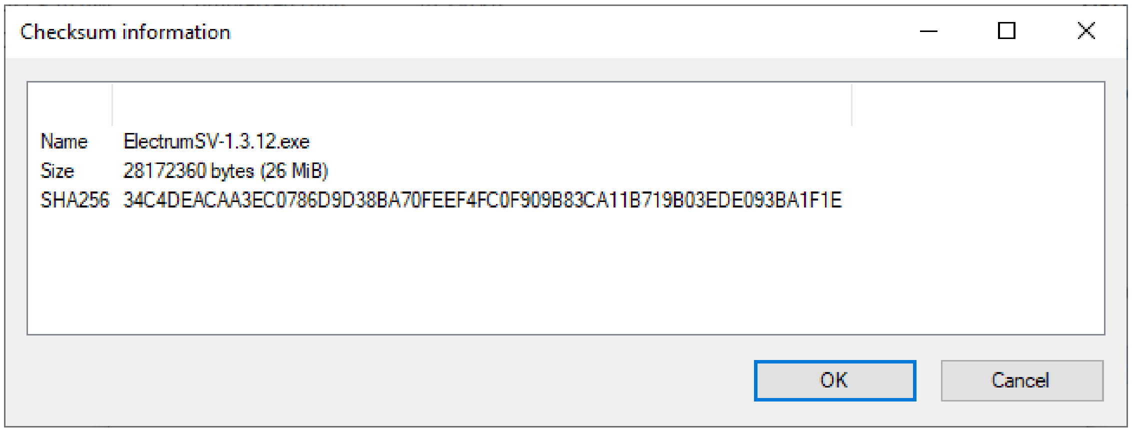


Fig. 7: The 7-Zip context menu.

In this case, we selected the `SHA-256` menu option for the `ElectrumSV-1.3.12.exe` file and the following image shows the resulting checksum.

If you find the `ElectrumSV-1.3.12.exe` entry in the linked Github list, you can see it matches the 7-zip checksum result. The case of the letters does not matter, both lower case and upper case are equivalent.



```
Name    ElectrumSV-1.3.12.exe
Size    28172360 bytes (26 MiB)
SHA256  34C4DEACAA3EC0786D9D38BA70FEEF4FC0F909B83CA11B719B03EDE093BA1F1E
```

Fig. 8: The 7-Zip checksum result.

MacOS

The following approaches require the user to deal with the terminal. If you are unable to work out how to do this, remember you can always file a support request on the official ElectrumSV issue tracker.

shasum

This approach requires no application installation, but it does involve you being willing to use the `terminal` application. If you do not know how to locate this, start by opening the `launchpad` application using it's rocket icon in the dock.

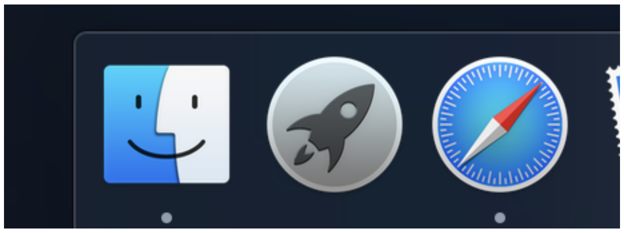


Fig. 9: Open the launchpad application search.

You should see the screen shown below. Enter `terminal` and it should show you one matching application which you should open.

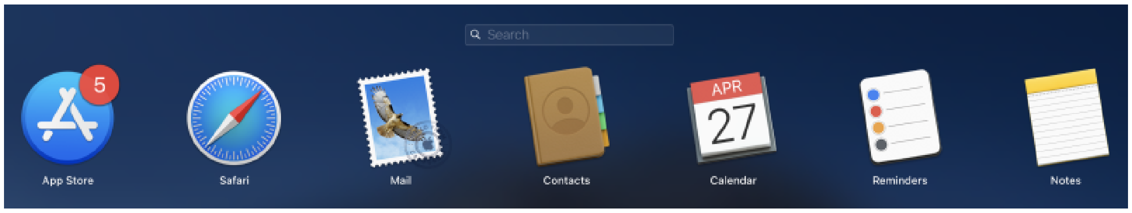
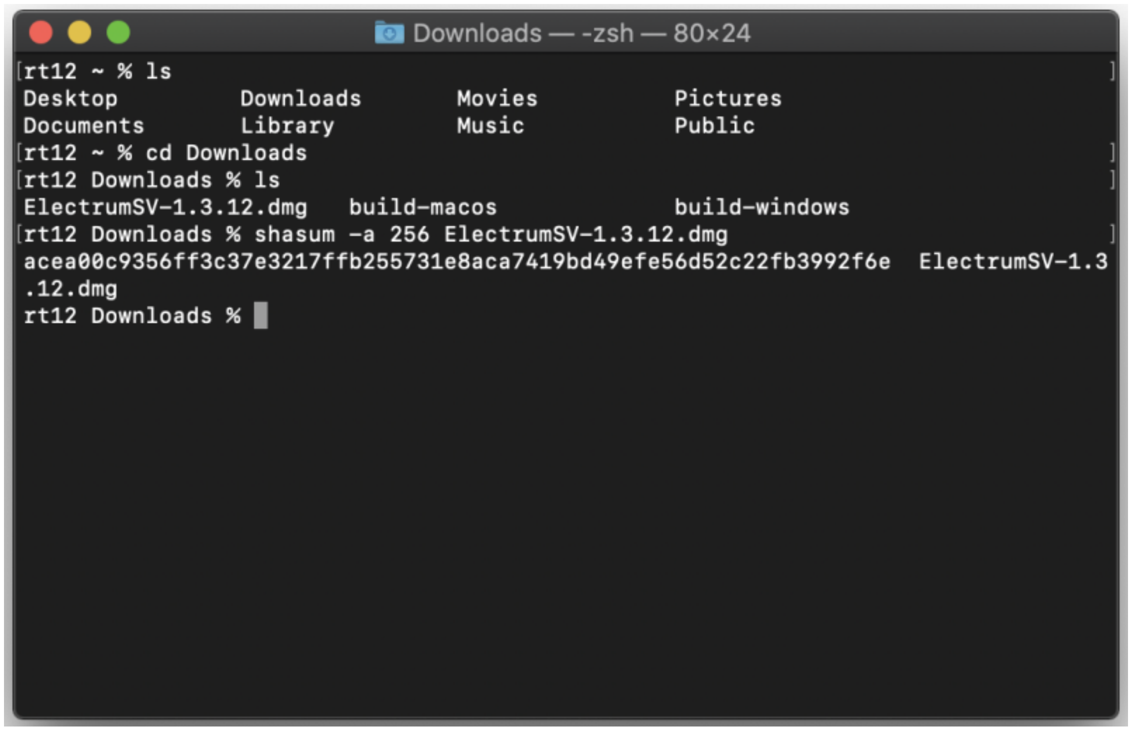


Fig. 10: Search for the 'terminal' application.

Work out what directory the terminal is looking at, and change it using the `cd` command. In the case shown below, the downloaded file was conveniently located in the `Downloads` folder and as this should also be the case for you the required commands should be the same. Type `cd Downloads` followed by `shasum -a 256 <filename>` where you replace <filename> with the actual file name of your download. Shown below, the file name was `ElectrumSV-1.3.12.dmg` and if you downloaded this file you also would use `shasum -a 256 ElectrumSV-1.3.12.dmg` as shown.



```
rt12 ~ % ls
Desktop          Downloads        Movies           Pictures
Documents        Library          Music            Public
rt12 ~ % cd Downloads
rt12 Downloads % ls
ElectrumSV-1.3.12.dmg   build-macos             build-windows
rt12 Downloads % shasum -a 256 ElectrumSV-1.3.12.dmg
acea00c9356ff3c37e3217ffb255731e8aca7419bd49efe56d52c22fb3992f6e  ElectrumSV-1.3
.12.dmg
rt12 Downloads %
```

Fig. 11: Run the 'shasum' application on your downloaded file.

If you find the `ElectrumSV-1.3.12.dmg` entry in the linked Github list, you can see it matches the `shasum` checksum result. The case of the letters does not matter, both lower case and upper case are equivalent. If you get a different result, and the command complains that it cannot find the file, then the file is not in the current directory. You need to use the `cd` command to change the current directory as mentioned above.

GNU Privacy Guard

By installing GNU Privacy Guard (GPG) you have a way to verify that the signatures provided by the developers for the files you download, prove those files came from those developers. This is quite involved to do, but it might be that you are more comfortable with this approach.

Start by downloading and installing GPG from the GPGTools web site. This gives you a way to check signatures for files. The next step is to obtain the keys for the ElectrumSV developers, and to register them with GPG. This is a little complicated so you need to follow these steps.

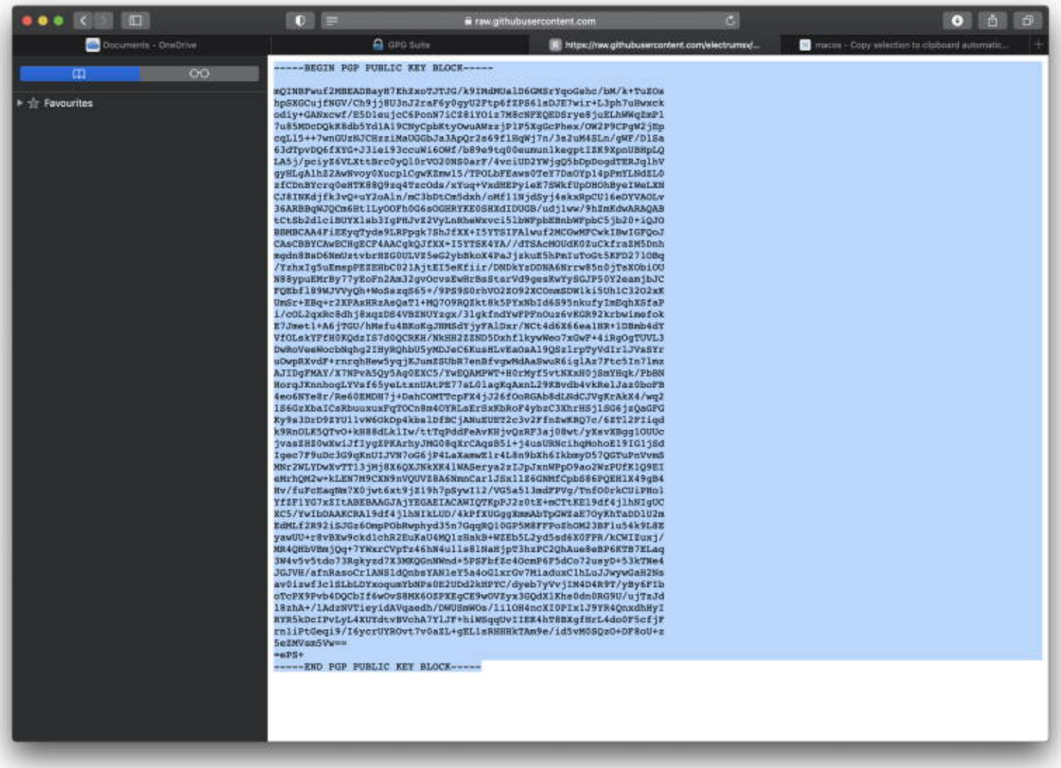
Open the `pubkeys` folder from the official ElectrumSV Github repository in Safari. You should see two files listed, `rt121212121.asc` and `kyuupichain.asc`. For each file perform the following key import actions.

Key import

Remember that this has to be done for all of the listed public keys in the ElectrumSV Github folder. Once you are viewing the raw page for a key, select (press `Command` with `a`) and copy (press `Command` and `c`) the key text.

As soon as you have copied the key text, the GPG application you installed will signal that it has detected a public key was copied. You will see it's icon in your dock jumping up and down. Click on it to import the key.

The GPG application will require you to approve the import, so go ahead and do that.


raw.githubusercontent.com
Documents – OneDrive
GPG Suite
https://raw.githubusercontent.com/electrumsv/...
macos - Copy selection to clipboard automatic...
Favourites

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFwuf2MBEADBayH7EhZxoTJTJG/k9IMdMUalD6GMSrYqoGebc/bM/k+TuZOa
hpSXGCujfNGV/Ch9jj8U3nJ2raF6y0gyU2Ftp6fZPS6lsDJE7wir+LJph7uHwxck
odiy+GANxcwf/E5DleujcC6PonN7iC28iYOiz7M8cNFEQEDSrye8juELhWWqZmPl
7u85MDcDQkK8db5Yd1Al9CNyCpbKtyOwuAMzzjPlP5XgGcPhex/OW2P9CPgW2jEp
cqLl5++7wnGUzMJCHzziMaUGGbJa3ApQr2s69flHqWj7n/3s2uM4SLn/gWF/DlSa
63dTpvDQ6fXYG+J3lei93ccuWi6OWf/b89e9tq00eumunlkegptIZK9XpnUBHpLQ
LA5j/pciyZ6VLXttBrcOyQl0rVO20NS0arF/4vciUD2YWjgQ5bDpDogdTERJqlhV
gyHLgAlhZ2AwNvoy0XucplCgwKZmwl5/TPOLbFEaws0TeY7Da0Ypl4pPmYLNdZL0
zfCDnBYcrq0eHTK88Q9zq4TzcOds/xYuq+VxdHEPyieE7SWkfUpOHOhByeIMeLXN
CJ8INKdjfk3vQ+uY2oAln/mC3bDtCm5dxh/oMf11NjdSyj4skxRpCU16eOYVAOLv
36ARBBqWJQCm6Ht1LyOOFh0G6sOGHRYKE0SHXdIDUGB/udjlww/9hZmKdwARAQAB
tCtSb2dlciBUYXlab3IgPHJvZ2VyLnRheWxvci5lbWFpbEBnbWFpbC5jb20+iQJO
BBMBCAA4FiEEyqTyds9LRPpgk7ShJfXX+I5YTSIFAlwuf2MCGwMFCwkIBwIGFQoJ
CAsCBBYCAwECHgECF4AACgkQJfXX+I5YTSK4YA//dTSAcMOUdK0ZuCkfraZM5Dnh
mgdn8BaD6NmUztvbrHZG0ULVZ5eG2ybBkoX4PaJjzkuE5hPmIuToGt5KFD27lOBq
/YzhxIg5uEmspPEZEHbC02lAjtEI5eKfiir/DNDkYzDDNA6Nrrw85n0jTsXObiOU
N88ypuEMrBy77yEoFn2Am32gvOcvsEwHrBsStarVd9gesKwYySGJP50Y2eamjbJC
FQEbfl89MJVVyQh+MoSazqS65+/9PS9S0rhVO2ZO92XCOmmSDWlki5Uh1C32O2xK
UmSr+EBq+r2XPAxHRzAsQaT1+MQ7O9RQZkt8k5PYxNbId6S95nkufyImEqhXSfaP
i/cOL2qxRc8dhj8xqzD84VBZNUYzgx/3lqkfndYwFPFnOuz6vKGR92krbwimefok
E7Jmetl+A6jTGU/hMsfu4BKoKgJHMSdYjyFAlDxr/NCt4d6X66ea1HR+1DBmb4dY
VfOLskYPfH0KQdzIS7d0QCRKH/NkSH2ZSND5DxhflkywWeo7xGwF+4iRgOgTUVL3
DwRoVeeWocbNqbg2IHyRQhbU5yMDJeC6KusHLvEaOaAl9QSzlrpTyVdIrlJVsSYr
uDwpRXvdF+rnrqhHew5yqjKJumZSUbR7enBfvgwMdAaSwuR6iglAz7Ftc5In7lmx
AJIDqFMAY/X7NPvA5Qy5Ag0EXC5/YwEQAMPWT+H0rMyf5vtNXxH0jSmYHqk/PbBN
HorqJKnnhogLYVsf65yeLtxnUAtPE77aL0lagKqAxnL29KBvdb4vkRelJaz0boFB
4eo6NYe8r/Re60EMDH7j+DahCOMTTcpFX4jJ26fOoRGAb8dLNdCJVgKrAkX4/wq2
l86GzXbaICsRbuuxuxFqTOCn8m4OYRLsErSxKbRoF4ybzC3XhrHSjl8G6jzQaGFG
Ky9a3DrD9ZYU1lvW6OkDp4kbalDfBCjANuEUET2c3v2FfnZwKRQ7c/6ZTl2FIiqd
k9RnOLK5QTvO+kH88dLklIw/ttTqPddFeAvKHjvQzRF3aj08wt/yXsvXBgg1OUUc
jvasZH20wXwiJfIygZPKArhyJMG08qXrCAqsB5i+j4usURNcihqMohoE19IGljSd
Igec7F9uDc3G9qKnUIJVN7oG6jP4LaXamwElr4L8n9bXh6IkbmyD57QGTuPnVvmS
MNr2WLYDwXvTT13jMj8X6QXJNkXK4lWASerya2zIJpJxnWPpD9ao2WzPUfK1Q9EI
eMrhQM2w+kLEN7M9CXN9nVQUVZ8A6NmnCar1JSxllZ6GNMfCpbS86PQEHlX49gB4
Hv/fuFcEaqNm7X0jwt6xt9jZ19h7pSywI12/VG5a51JmdFPVg/TnfO0rkCUiPHol
YfZF1YG7xZItABEBAAGJAjYEGAEIACAWIQTKpPJ2z0tE+mCTtKE19df4jlhNIgUC
XC5/YwIbDAAKCRAl9df4jlhNIkLUD/4kPfXUGgqXmmAbTpGWZaE7OyKhTaDDlU2m
EdMLf2R92iSJGz6OmpPObRwphydJ5n7GqqRQ10GP5M8FFPoZhOM2JBFlu54k9L8E
yawUU+r8vBXw9ckdlchR2EuKaU4MQlzBakB+WZEb5L2yd5sd6X0FPR/kCWIZuxj/
MR4QHbVBmjQq+7YWxrCVpTz46hN4ulls8lNaHjpT3hzPC2QhAue8eBP6KTB7ELaq
3W4v5v5tdo73Rgkyzd7X3MKQGnNWnd+5PSFbfZc4OcmP6F5dCo72usyD+53kTNe4
JGJVH/afnRasoCr1ANS1dQnbsYANleY5a4oGlxrGv7MiaduxC1hLuJJwywGaH2Ns
av0izwf3c1SLbLDYxoqumYbNPs0E2UDd2kMPYC/dyeb7yVvjIM4D4R9T/yBy6FIb
oTcPX9Pvb4DQCbIf6wOvS8MX6OZPXEgCE9wOVZyx3GQdXlKhs0dn0RG9U/ujTzJd
l8zhA+/lAdzNVTieyidAVqaedh/DWUSmWOs/1i1OH4ncXI0PIxlJ9YR4QnxdhHyI
NYR5kDcIPvLyL4XUYdtvBVchA7YlJF+hiWSqqUvIIEK4hT8BXgfHrL4do0F5cfjF
rnliPtGeqi9/I6ycrUYROvt7v0aZL+gELlsRHHHkTAm9e/id5vM0SQz0+DF8oU+z
5eZMVsm5Vw==
=ePS+
-----END PGP PUBLIC KEY BLOCK-----
```

Fig. 12: Select and copy the public key text.

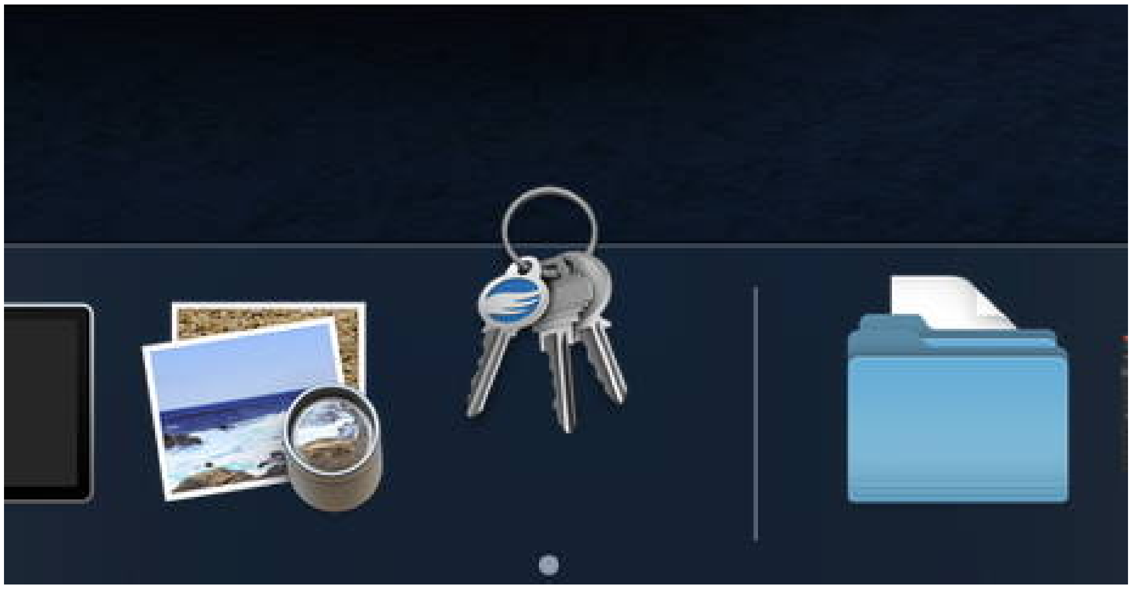
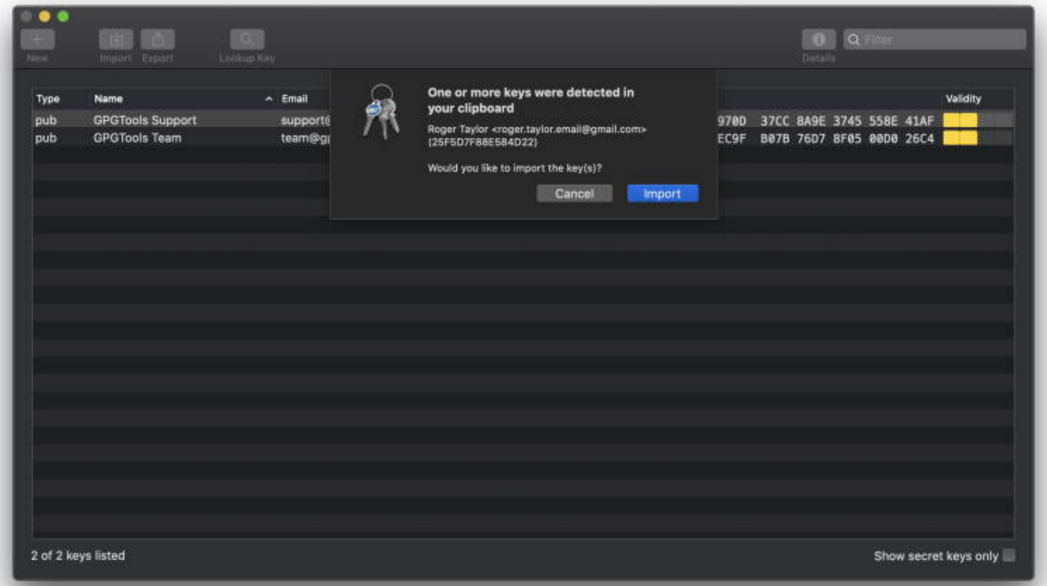


Fig. 13: Observe the GPG icon in the dock indicating that it can act on the copied key.



Type	Name	Email		Validity
pub	GPGTools Support	support@	970D 37CC 8A9E 3745 558E 41AF	
pub	GPGTools Team	team@gp	EC9F B07B 76D7 8F05 00D0 26C4	

One or more keys were detected in your clipboard

Roger Taylor <roger.taylor.email@gmail.com>
(25F5D7F88E584D22)

Would you like to import the key(s)?

2 of 2 keys listed

Fig. 14: Approve the public key import.

Once the public key is imported, you will see another sheet drop down to tell you if it was imported successfully or not. It will of course be successful.

You can confirm the key was imported successfully, by observing that it is now present in the list in the GPG application.

Go ahead and import any keys you haven't imported already, then you are all set to verify the signature of an ElectrumSV download when you need to.

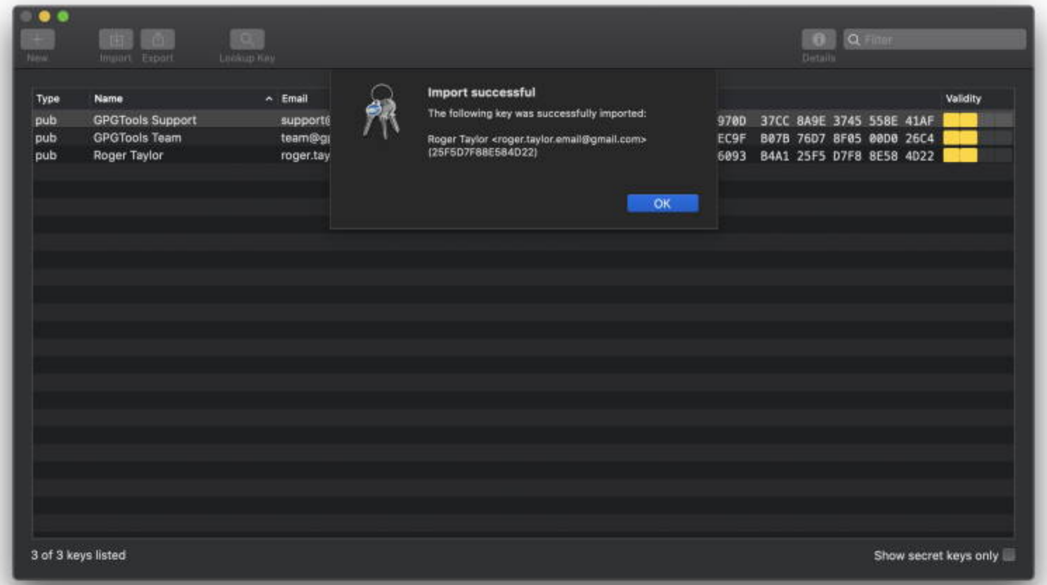
Verify a download

Let's say you have downloaded `ElectrumSV-1.3.12.dmg` from the official ElectrumSV downloads page. You now need to find and download the signature for that file, so that you can verify it was created by the ElectrumSV developers. The signatures are located on the official ElectrumSV web site, under it's download folder. The `.dmg` you downloaded was for version `1.3.12` so locate the folder by that name, and look inside it. You should see the signature file `ElectrumSV-1.3.12.dmg.sig`, which is what you need to download `ElectrumSV-1.3.12.dmg`.

Open the context menu for the `ElectrumSV-1.3.12.dmg` file (press `Control` when you click on the file). You will see a `Services` sub-menu, with an additional `OpenPGP: Verify Signature of File` beneath it. Click on this verify sub-menu.

The GPG application will verify the `.dmg` using the detected matching `.dmg.sig` file and let you know the result.

As you can see the signature was verified. If you want to go through the process of trusting the ElectrumSV keys, there is a link there you can use for next time.



Import successful

The following key was successfully imported:

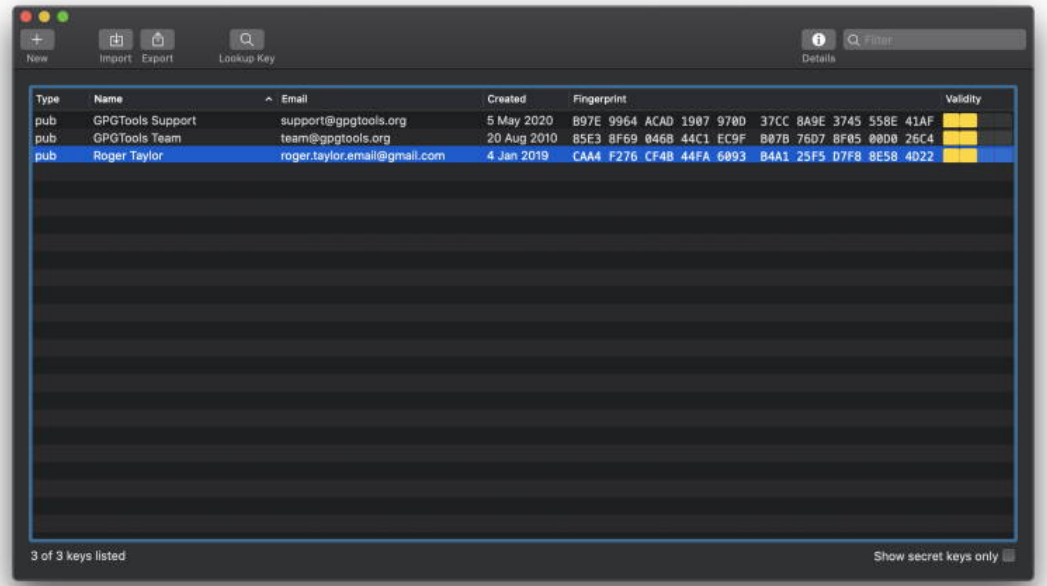
Roger Taylor <roger.taylor.email@gmail.com>
(25F5D7F88E584D22)

Type	Name	Email		Validity
pub	GPGTools Support	support@	970D 37CC 8A9E 3745 558E 41AF	🟨
pub	GPGTools Team	team@g	EC9F B07B 76D7 8F05 00D0 26C4	🟨
pub	Roger Taylor	roger.tay	6093 B4A1 25F5 D7F8 8E58 4D22	🟨

3 of 3 keys listed

Show secret keys only

Fig. 15: Observe the successful public key import.

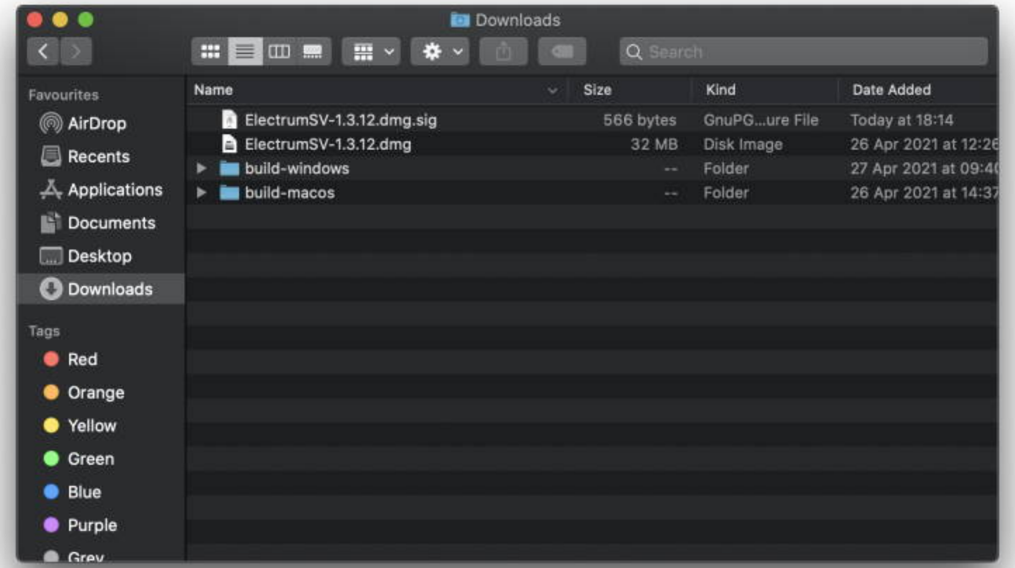


Type	Name	Email	Created	Fingerprint	Validity
pub	GPGTools Support	support@gpgtools.org	5 May 2020	B97E 9964 ACAD 1907 970D 37CC 8A9E 3745 558E 41AF	🟨
pub	GPGTools Team	team@gpgtools.org	20 Aug 2010	85E3 8F69 046B 44C1 EC9F B07B 76D7 8F05 00D0 26C4	🟨
pub	Roger Taylor	roger.taylor.email@gmail.com	4 Jan 2019	CAA4 F276 CF4B 44FA 6093 B4A1 25F5 D7F8 8E58 4D22	🟨

3 of 3 keys listed

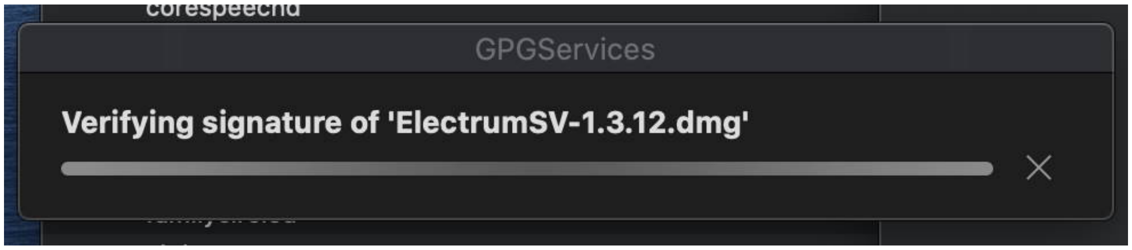
Show secret keys only

Fig. 16: Observe the imported public key is present in your GPG application.



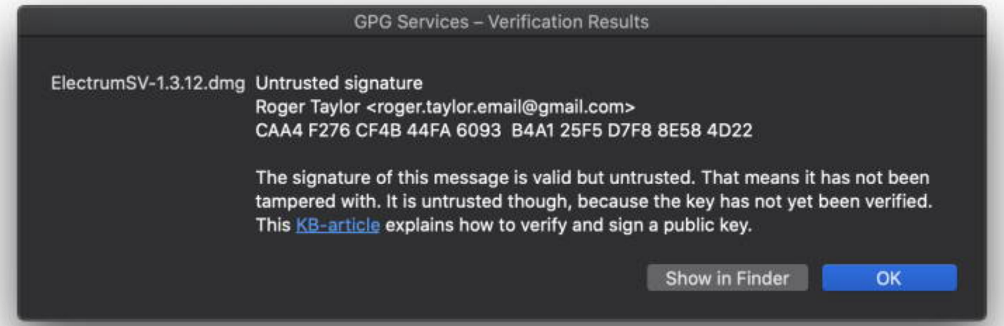
Name	Size	Kind	Date Added
ElectrumSV-1.3.12.dmg.sig	566 bytes	GnuPG...ure File	Today at 18:14
ElectrumSV-1.3.12.dmg	32 MB	Disk Image	26 Apr 2021 at 12:26
build-windows	--	Folder	27 Apr 2021 at 09:40
build-macos	--	Folder	26 Apr 2021 at 14:37

Fig. 17: Confirm you have downloaded both the `.dmg` and the matching `.dmg.sig` files.



Verifying signature of 'ElectrumSV-1.3.12.dmg'

Fig. 18: Open the context menu and select the OpenPGP verify entry.



ElectrumSV-1.3.12.dmg Untrusted signature
Roger Taylor <roger.taylor.email@gmail.com>
CAA4 F276 CF4B 44FA 6093 B4A1 25F5 D7F8 8E58 4D22

The signature of this message is valid but untrusted. That means it has not been tampered with. It is untrusted though, because the key has not yet been verified. This KB-article explains how to verify and sign a public key.

Fig. 19: Observe the verification result.

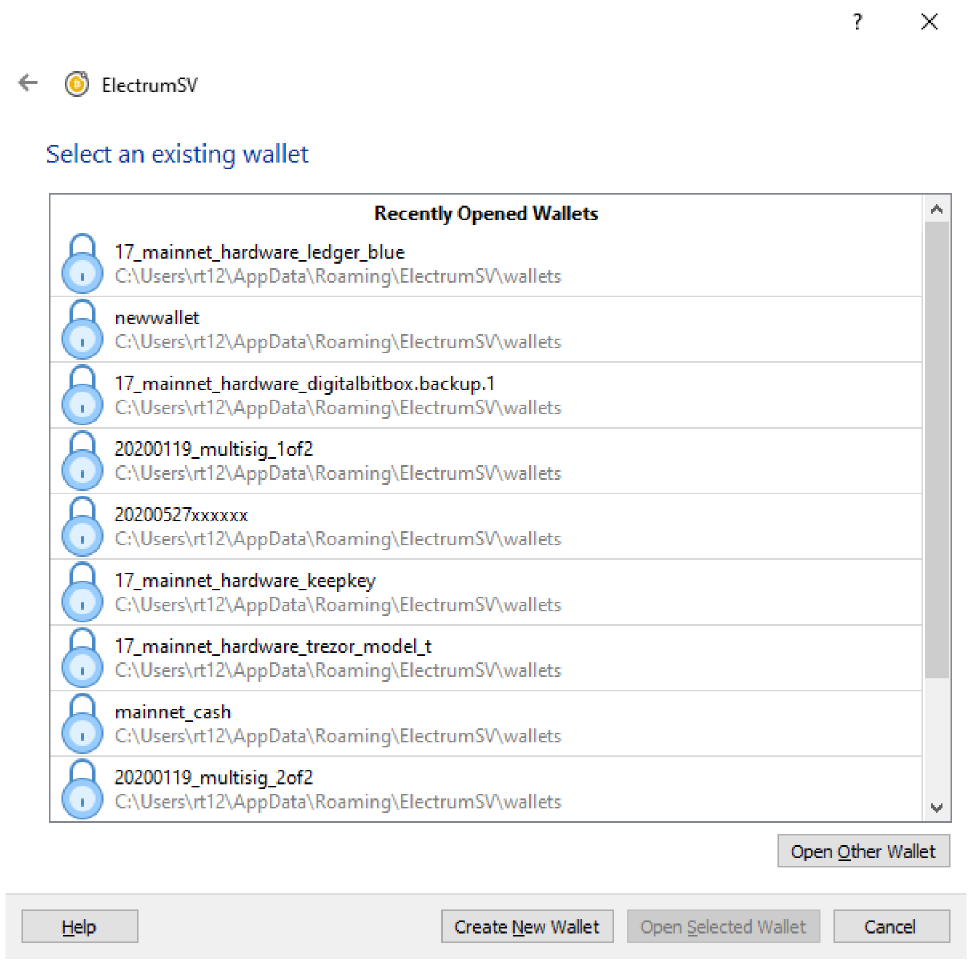
1.2 Creating a wallet

From ElectrumSV 1.3 and beyond, a wallet is now a container for your accounts. This guide shows you how to create an empty wallet with no accounts. After creating the wallet, you will of course want to add an account to it, in order to be able to start using it.

1.2.1 Choosing the location and file name

The first step is to choose where to store your wallet, and what it's file name should be. If you choose not to store your wallet in the default location that ElectrumSV uses, it is likely that you will quickly be able to find it again in the "Recently Opened Wallets" list when you open it again in the future.

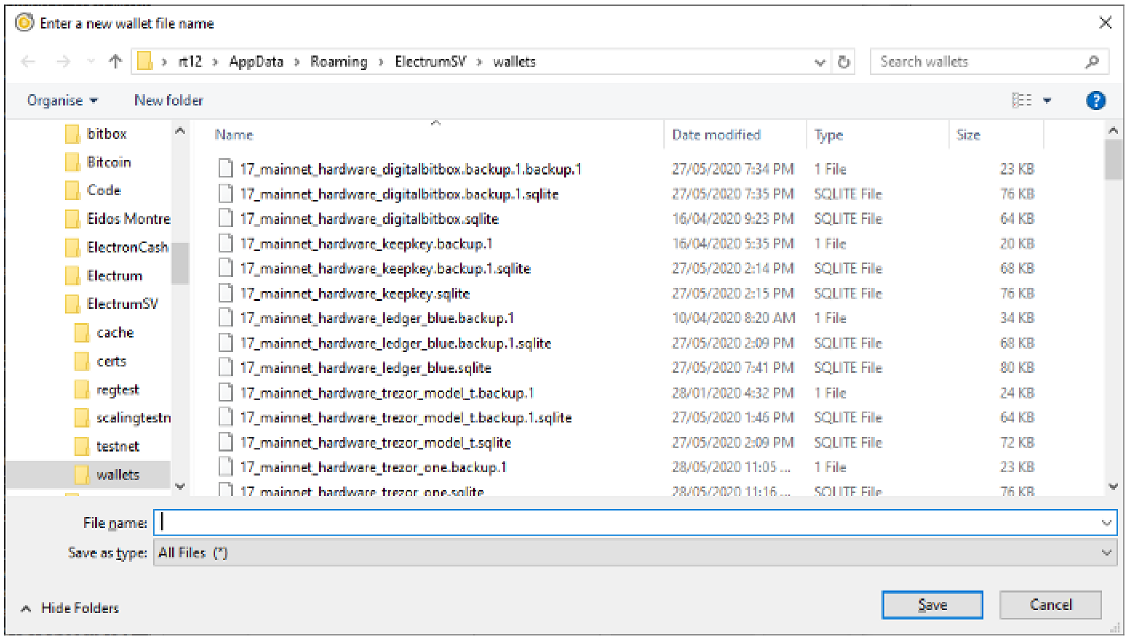
Start off at the wallet selection page.



17_mainnet_hardware_ledger_blue
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
newwallet
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
17_mainnet_hardware_digitalbitbox.backup.1
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
20200119_multisig_1of2
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
20200527xxxxxx
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
17_mainnet_hardware_keepkey
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
17_mainnet_hardware_trezor_model_t
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
mainnet_cash
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets
20200119_multisig_2of2
C:\Users\rt12\AppData\Roaming\ElectrumSV\wallets

Fig. 20: The wallet selection page.

You will be presented with a file dialog that lets you choose where your wallet will be stored, and what it will be named. It defaults to the standard ElectrumSV wallet location on your operating system. Enter a file name, and click "Save" (or press the enter key).



Name	Date modified	Type	Size
17_mainnet_hardware_digitalbitbox.backup.1.backup.1	27/05/2020 7:34 PM	1 File	23 KB
17_mainnet_hardware_digitalbitbox.backup.1.sqlite	27/05/2020 7:35 PM	SQLITE File	76 KB
17_mainnet_hardware_digitalbitbox.sqlite	16/04/2020 9:23 PM	SQLITE File	64 KB
17_mainnet_hardware_keepkey.backup.1	16/04/2020 5:35 PM	1 File	20 KB
17_mainnet_hardware_keepkey.backup.1.sqlite	27/05/2020 2:14 PM	SQLITE File	68 KB
17_mainnet_hardware_keepkey.sqlite	27/05/2020 2:15 PM	SQLITE File	76 KB
17_mainnet_hardware_ledger_blue.backup.1	10/04/2020 8:20 AM	1 File	34 KB
17_mainnet_hardware_ledger_blue.backup.1.sqlite	27/05/2020 2:09 PM	SQLITE File	68 KB
17_mainnet_hardware_ledger_blue.sqlite	27/05/2020 7:41 PM	SQLITE File	80 KB
17_mainnet_hardware_trezor_model_t.backup.1	28/01/2020 4:32 PM	1 File	24 KB
17_mainnet_hardware_trezor_model_t.backup.1.sqlite	27/05/2020 1:46 PM	SQLITE File	64 KB
17_mainnet_hardware_trezor_model_t.sqlite	27/05/2020 2:09 PM	SQLITE File	72 KB
17_mainnet_hardware_trezor_one.backup.1	28/05/2020 11:05 ...	1 File	23 KB
17_mainnet_hardware_trezor_one.sqlite	28/05/2020 11:16 ...	SQLITE File	76 KB

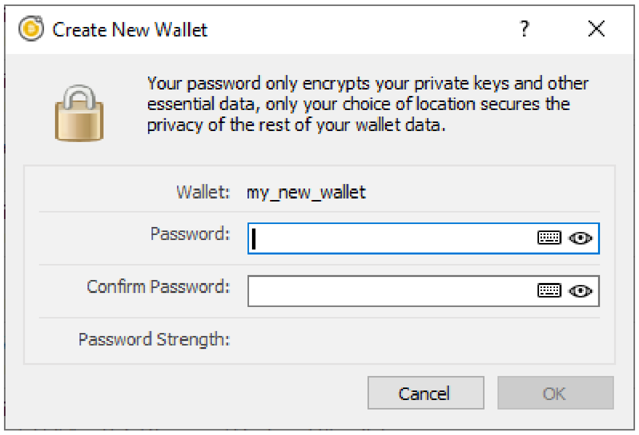
File name:

Save as type: All Files (*)

Fig. 21: The wallet file name dialog.

1.2.2 Add a mandatory password

The next step is setting a password for your new wallet. We require a password and there is no way to opt out, but you can always enter something like "password" or "123456" if you wish. For now this is also required for hardware and watch-only wallets, where there is no key or seed word data to encrypt.



Your password only encrypts your private keys and other essential data, only your choice of location secures the privacy of the rest of your wallet data.

Wallet: my_new_wallet

Password:

Confirm Password:

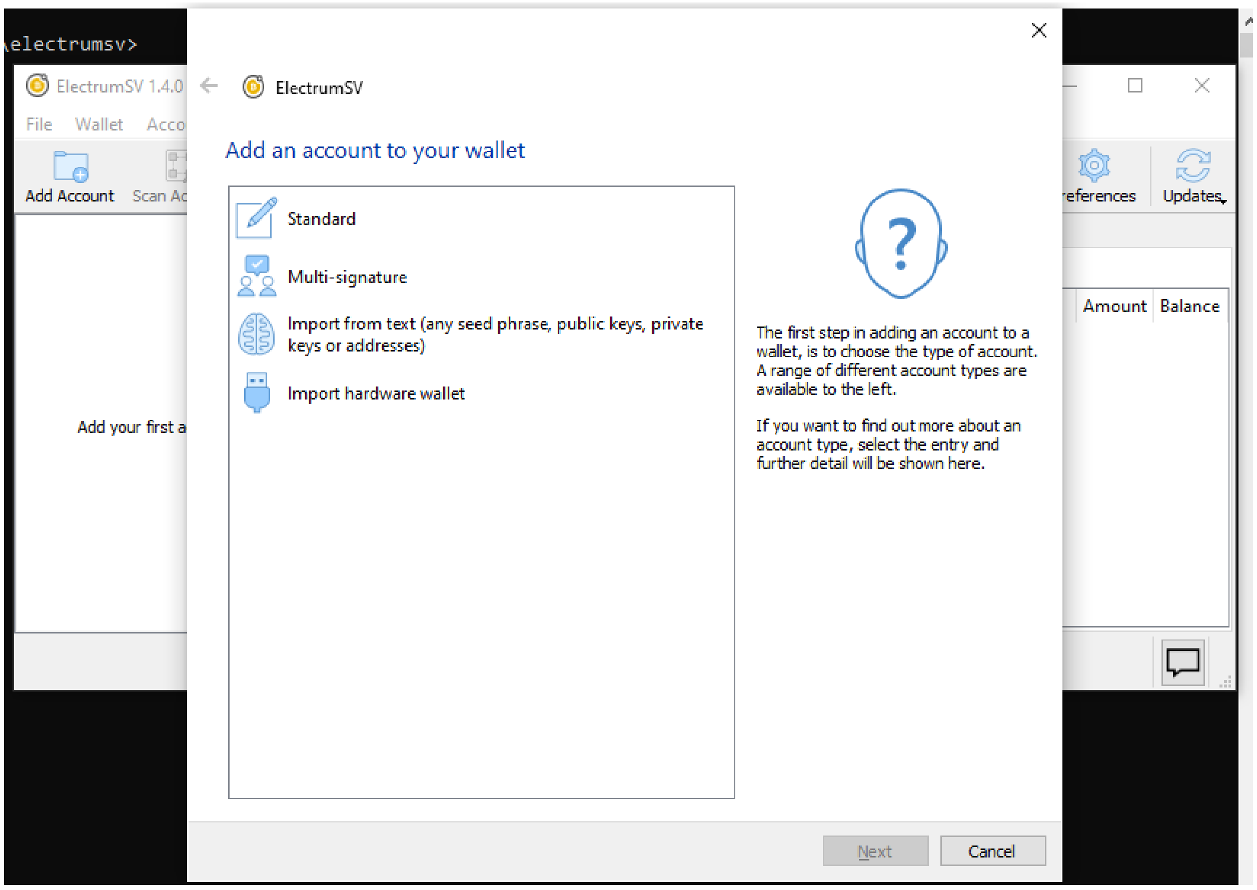
Password Strength:

Fig. 22: The wallet file name dialog.

Once you have entered a password, and confirmed it, the "OK" button will become enabled and you can click it (or just press the enter key) to open the new wallet. On doing so, the wallet window will open and you will be presented with the account creation dialog.

If you dismiss the dialog either intentionally, or accidentally, before creating an account you can bring it back up by clicking the "Add Account" button.

Congratulations, you have created a new empty wallet. It will not be usable until you have created an account, and various parts of the user interface will indicate this.



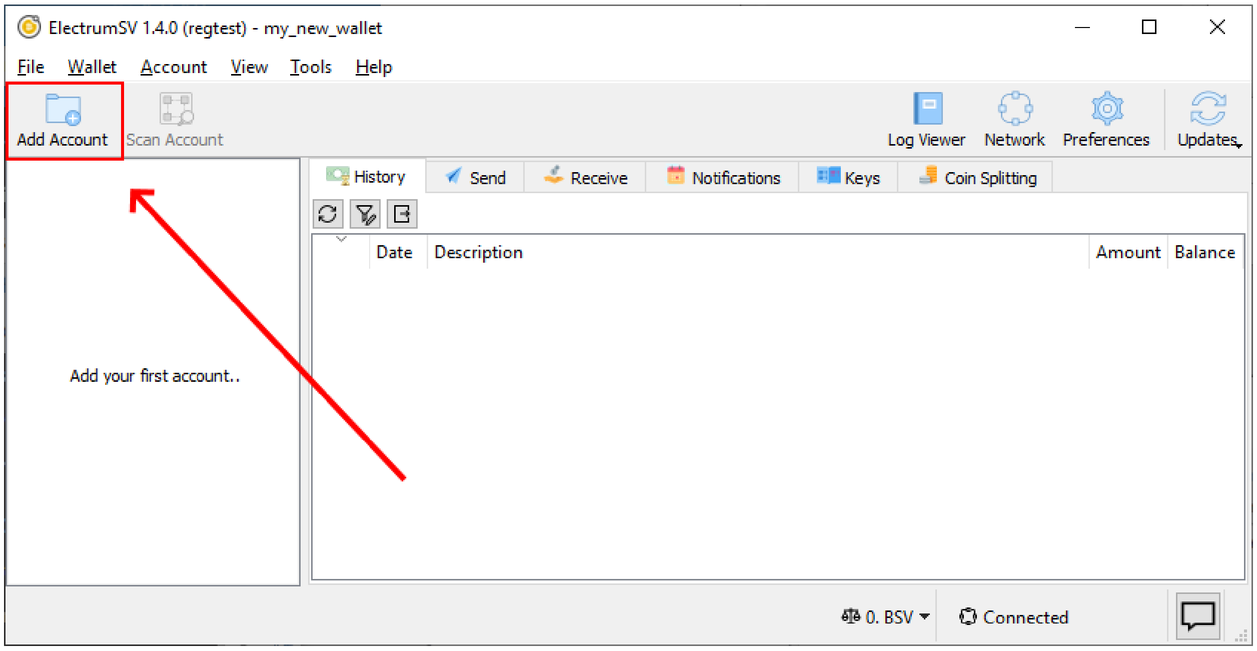
Add an account to your wallet

Import from text (any seed phrase, public keys, private keys or addresses)

The first step in adding an account to a wallet, is to choose the type of account. A range of different account types are available to the left.

If you want to find out more about an account type, select the entry and further detail will be shown here.

Fig. 23: The new wallet's wallet window with account creation dialog.



Add your first account..

Fig. 24: The receiving tab is disabled.

1.3 Creating an account

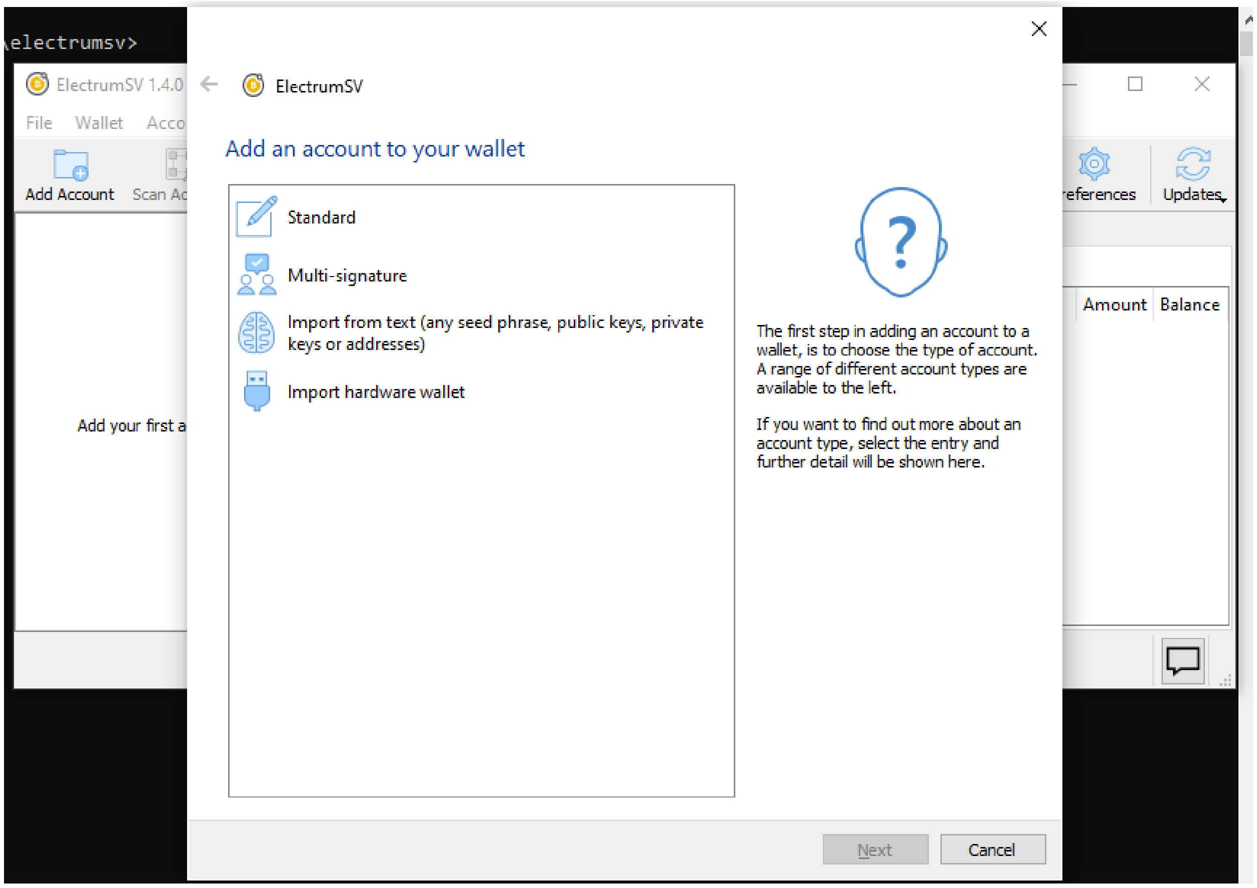
If you are reading this, you likely have a new wallet that has no accounts, and you want to add one to it. We support addition of a wide variety of account types:

- A new “Standard” account. This is the equivalent of creating a new ElectrumSV seed-word based wallet in 1.2.5 and earlier.
- A multi-signature account. Use this if you are creating a new multi-signature account, or restoring an existing one from master public keys, seed words and so on.
- Importing from text. Use this to import your seed words, whether Electrum seed words, BIP39 seed words from another wallet, private keys, public keys, master public keys, master private keys, and so on.
- Importing a hardware wallet. If you have an existing hardware wallet that has a seed set up on it, then you can use this to add an account that links to it and uses it to sign. If you have a hardware wallet that does not have a seed set up on it, you should also be able to use this to set it up unless the device is a Ledger. Do not buy a Ledger.

This guide solely covers creating a “Standard” account.

1.3.1 Adding a new account

On creating a new wallet, the first thing you will be presented with is the window for adding a new account.



The first step in adding an account to a wallet, is to choose the type of account. A range of different account types are available to the left.

If you want to find out more about an account type, select the entry and further detail will be shown here.

Fig. 25: The account creation window.

If you dismiss this window, accidentally or otherwise, you can re-open it by clicking on the "Add Account" button on the left hand side of the wallet window toolbar. Click it and it will open the account wizard which allows all supported types of accounts to be created.

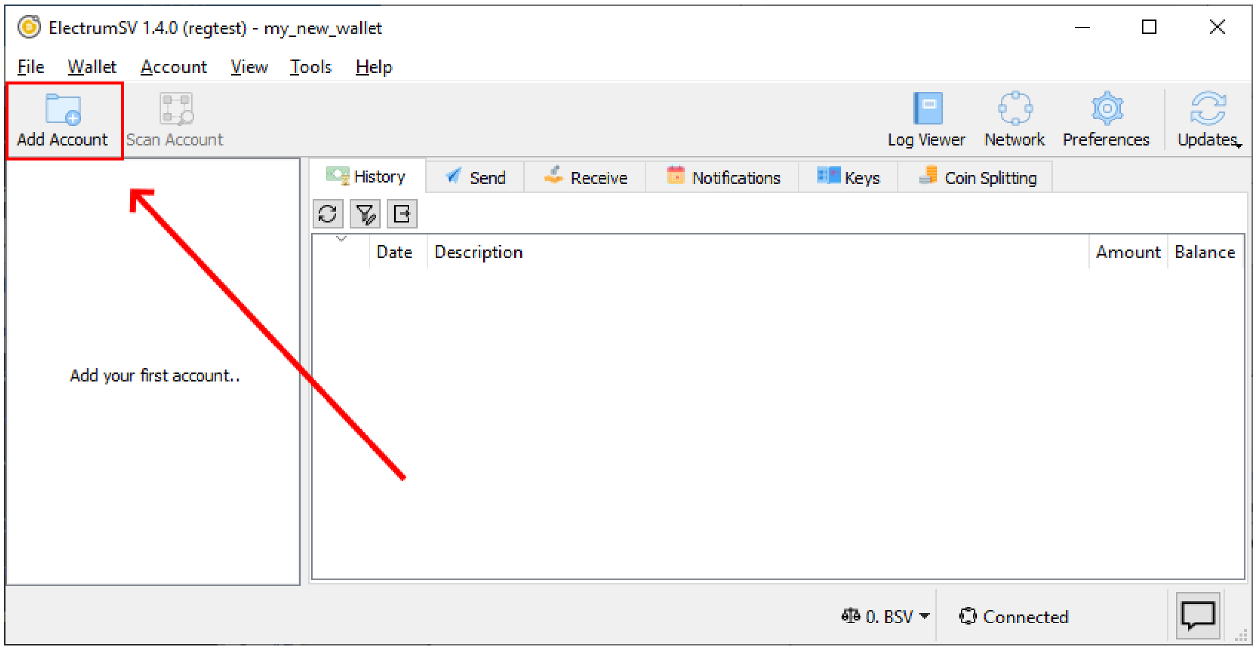
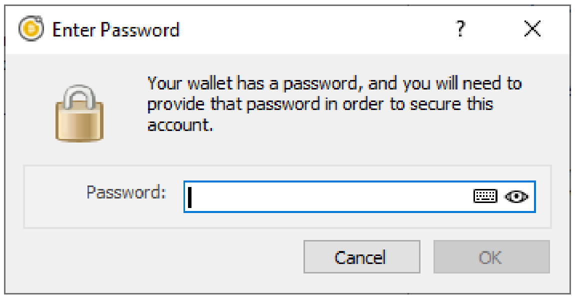


Fig. 26: The "Add Account" button highlighted.

1.3.2 Creating a new "Standard" Account

Double-click on the "Standard" entry to proceed. Or if you prefer to work for it, click the "Next" button or press the enter key. You will be asked for your password so that the generated seed words and private key data can be encrypted into your wallet. This also verifies you have the ability to really use this wallet, and should able to add an account.

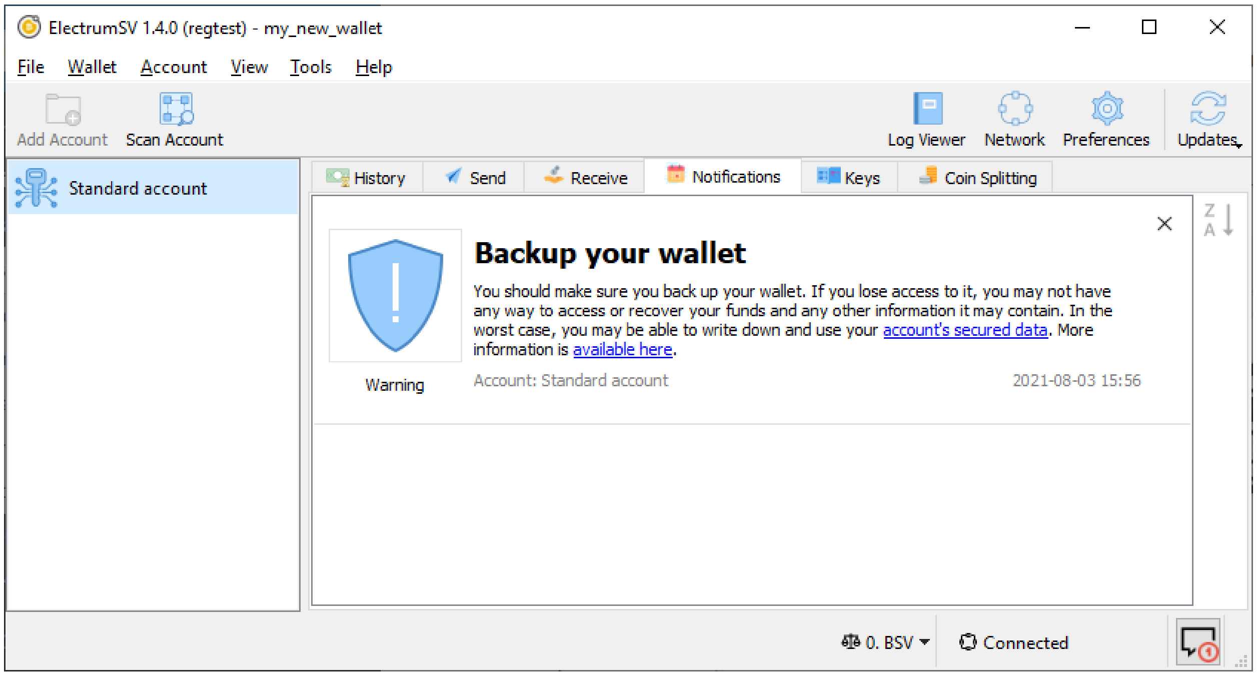


Your wallet has a password, and you will need to provide that password in order to secure this account.

Fig. 27: The password dialog.

You will immediately see that the account has been added to your wallet. You will note that at no point did you have to copy down your new seed words, or confirm them. You will be reminded to back them up by the wallet, and can do so at your leisure and own risk.

The "Notifications" tab will be shown every time you own your wallet as long as you have not dismissed the "Backup your wallet" notification. It is advised you go and back up your secured data immediately, as it instructs you to.



Backup your wallet

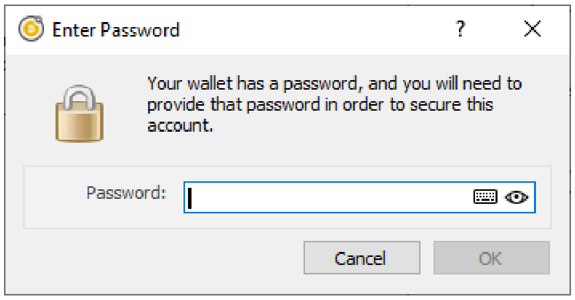
You should make sure you back up your wallet. If you lose access to it, you may not have any way to access or recover your funds and any other information it may contain. In the worst case, you may be able to write down and use your account's secured data. More information is available here.

Account: Standard account 2021-08-03 15:56

Fig. 28: The first thing you see on creating your new account.

1.3.3 Follow the link to your secured data

If you click on the “account’s secured data” link, it will take you directly to that secured data. But first it will need your password so it can decrypt that data for display.

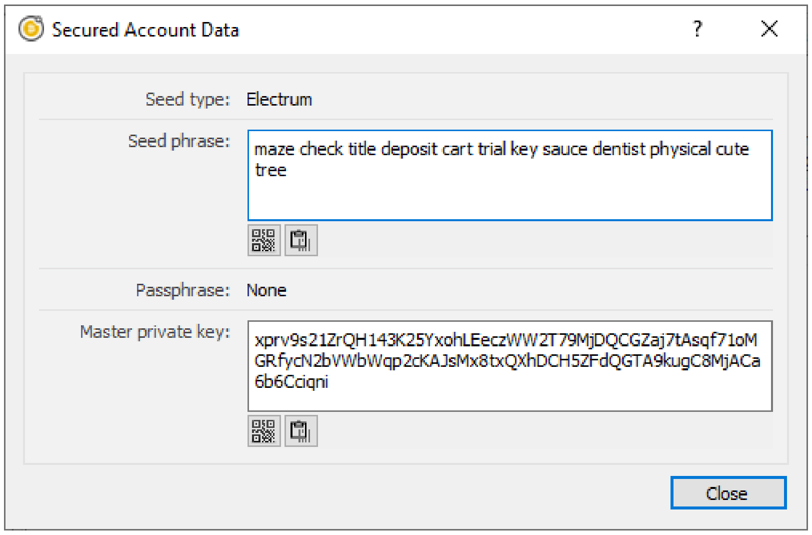


Your wallet has a password, and you will need to provide that password in order to secure this account.

Fig. 29: The password dialog.

Having entered the correct password you will see the secured data.

Congratulations, now write down the seed words somewhere safe. I recommend you look into SAFEWORDS to help you with this. You can dismiss the notification by clicking on the “X” in it’s top right corner.



Secured Account Data

Seed type: Electrum

Seed phrase: maze check title deposit cart trial key sauce dentist physical cute tree

Passphrase: None

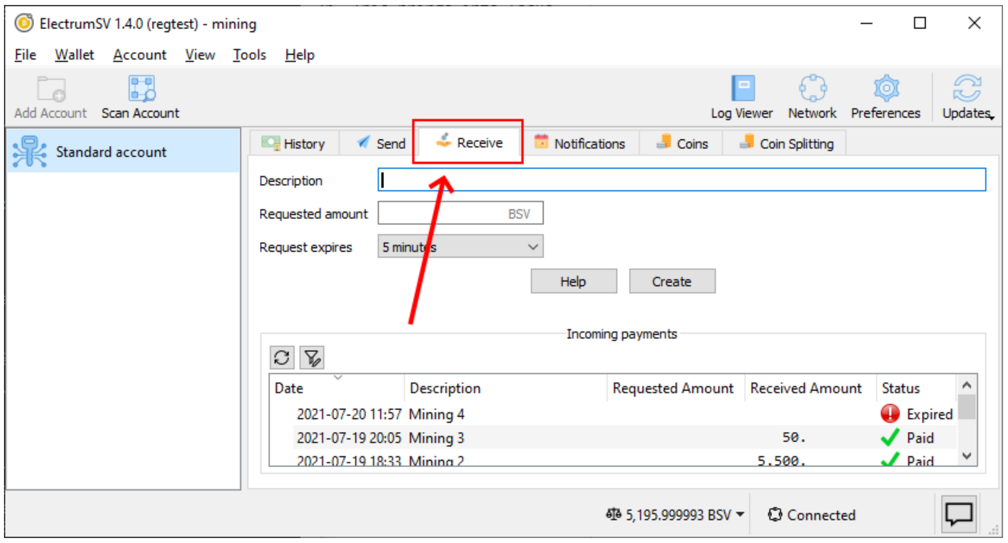
Master private key: xprv9s21ZrQH143K25YxohLEeczWW2T79MjDQCGZaj7tAsqf71oMGRfycN2bVWbWqp2cKAJsMx8txQXhDCH5ZFdQGTA9kugC8MjACa6b6Cciqni

Fig. 30: The secured data dialog.

1.4 Receiving a payment

An incoming payment is where you declare that you expect an incoming payment, and an address or payment destination is allocated for you to give out. The wallet then monitors the blockchain for usage of this payment destination for a payment, and closes out the incoming payment when either an expiry time has passed or payment is detected, retrieved and processed.

When you want to create an incoming payment, the first thing you do is go to the "Receive" tab. Here you can see a simple form you fill out to create a new incoming payment, and a list showing existing incoming payments.



ElectrumSV 1.4.0 (regtest) - mining

Description:
Requested amount: BSV
Request expires: 5 minutes

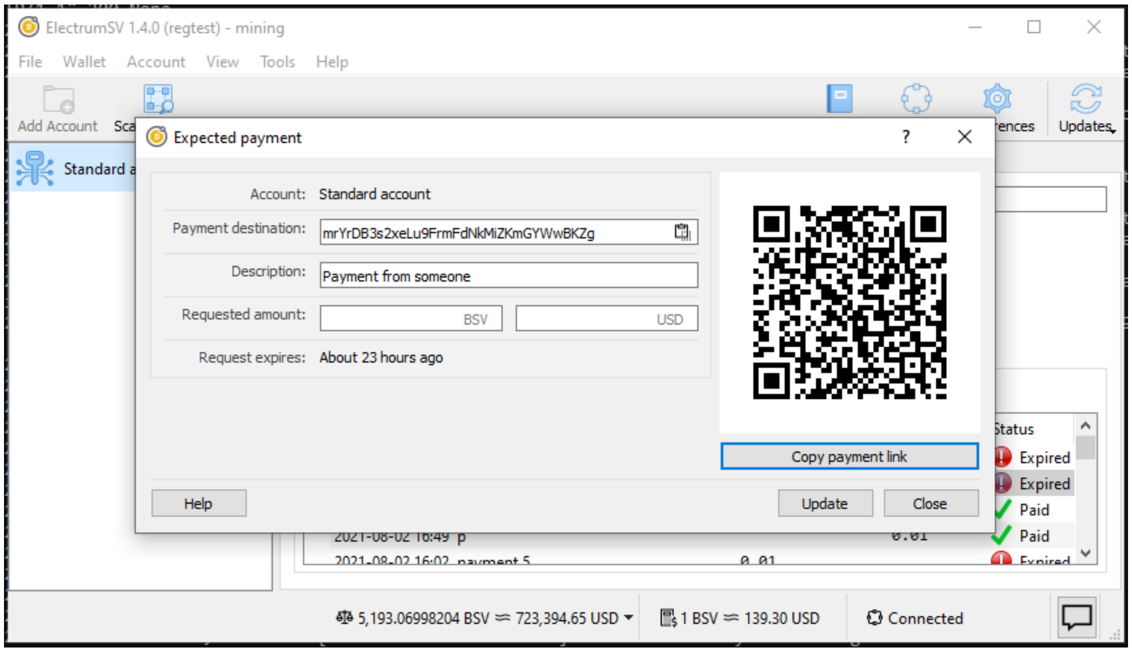
Incoming payments

Date	Description	Requested Amount	Received Amount	Status
2021-07-20 11:57	Mining 4			Expired
2021-07-19 20:05	Mining 3		50.	✓ Paid
2021-07-19 18:33	Mining 2		5,500.	✓ Paid

5,195.999993 BSV   Connected

Fig. 31: The "Receive" tab where incoming payments are created and viewed.

The minimum information you need to provide when creating an incoming payment is the description. If you provide no amount, it will be closed and considered paid if it receives any payment before the expiration period ends. You can also opt not to provide an expiration period, and have it sit there indefinitely. If it expires before the payer pays, then you can of course scan the blockchain to detect it later.



Account: Standard account
Payment destination: mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg
Description: Payment from someone
Requested amount: BSV USD
Request expires: About 23 hours ago

Fig. 32: The "Expected payment" dialog shown when you create an incoming payment.

This list is updated in real time. As the expiration period for an incoming payment ends, an expected payment is marked as expired and we stop watching the blockchain for activity related to it. As we detect activity related to an expected payment, we process those transactions and if there is no requested amount or the incoming value in those transactions is greater than or equal to the requested amount we mark the expected payment as paid and stop watching the blockchain for activity related to it.

1.4.1 Giving out the payment destination

There are several ways you can give the payment destination for your newly created expected payment, to whomever you expect to pay it.

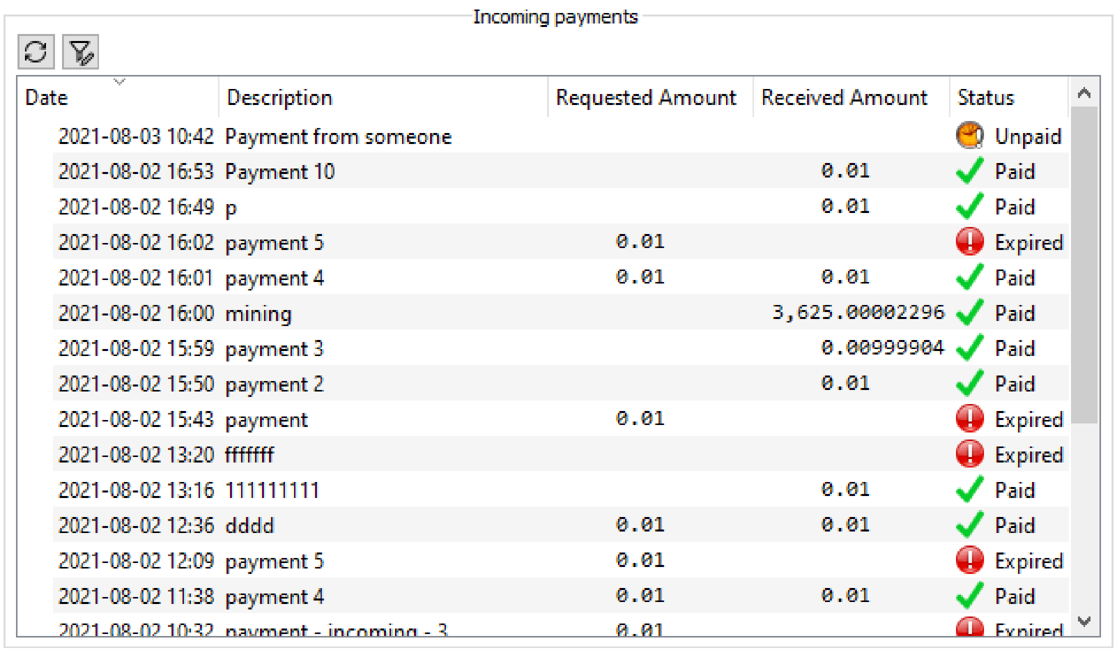
Copying the payment link

The advantage of copying the payment link and giving it to your payer, is that they should just be able to paste it into their wallet and they should see all the main details get incorporated into their wallet's user interface. The payment destination itself, the amount and the description of what the payment is for.

The payment link from the screenshot above is as follows. It will of course differ for your payment. The specification that defines these payment URLs is BIP21.

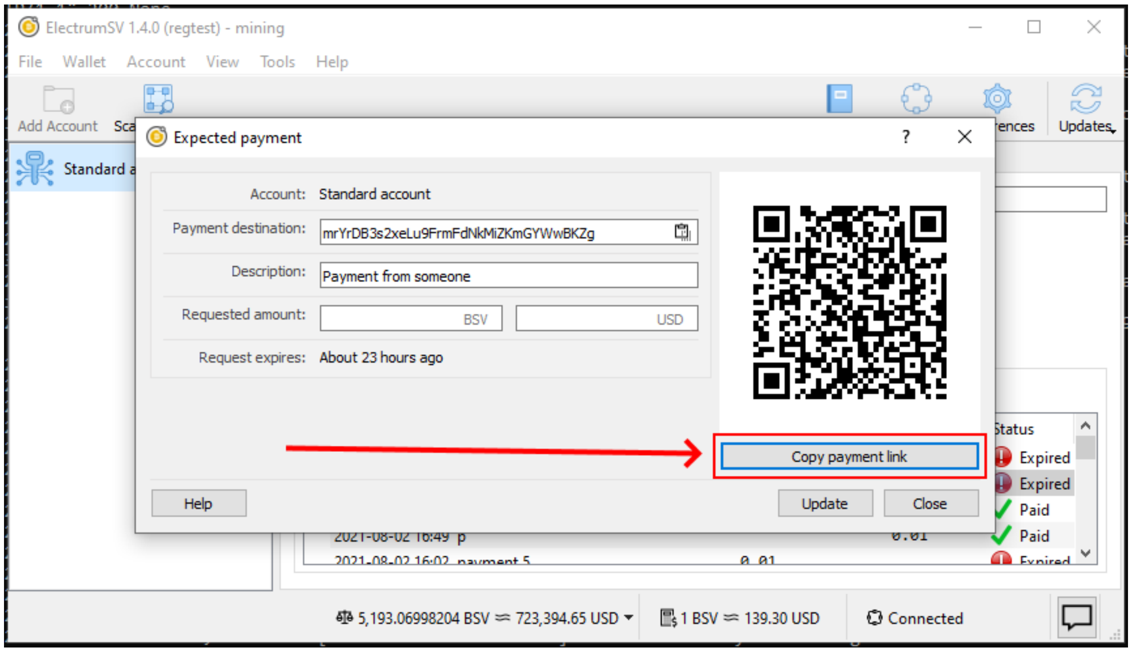
```
bitcoin:mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg?sv&message=Payment%20from%20someone
```

If you paste this payment link into the "Pay to" field in the "Send" tab, you can see how ElectrumSV and ideally other wallets will populate for form for the payer. Note that the address in the payment link is a testnet address, and is not usable on the real Bitcoin SV blockchain - the mainnet.



Date	Description	Requested Amount	Received Amount	Status
2021-08-03 10:42	Payment from someone			Unpaid
2021-08-02 16:53	Payment 10		0.01	✓ Paid
2021-08-02 16:49	p		0.01	✓ Paid
2021-08-02 16:02	payment 5	0.01		Expired
2021-08-02 16:01	payment 4	0.01	0.01	✓ Paid
2021-08-02 16:00	mining		3,625.00002296	✓ Paid
2021-08-02 15:59	payment 3		0.00999904	✓ Paid
2021-08-02 15:50	payment 2		0.01	✓ Paid
2021-08-02 15:43	payment	0.01		Expired
2021-08-02 13:20	fffffff			Expired
2021-08-02 13:16	111111111		0.01	✓ Paid
2021-08-02 12:36	dddd	0.01	0.01	✓ Paid
2021-08-02 12:09	payment 5	0.01		Expired
2021-08-02 11:38	payment 4	0.01	0.01	✓ Paid
2021-08-02 10:32	payment - incoming - 3	0.01		Expired

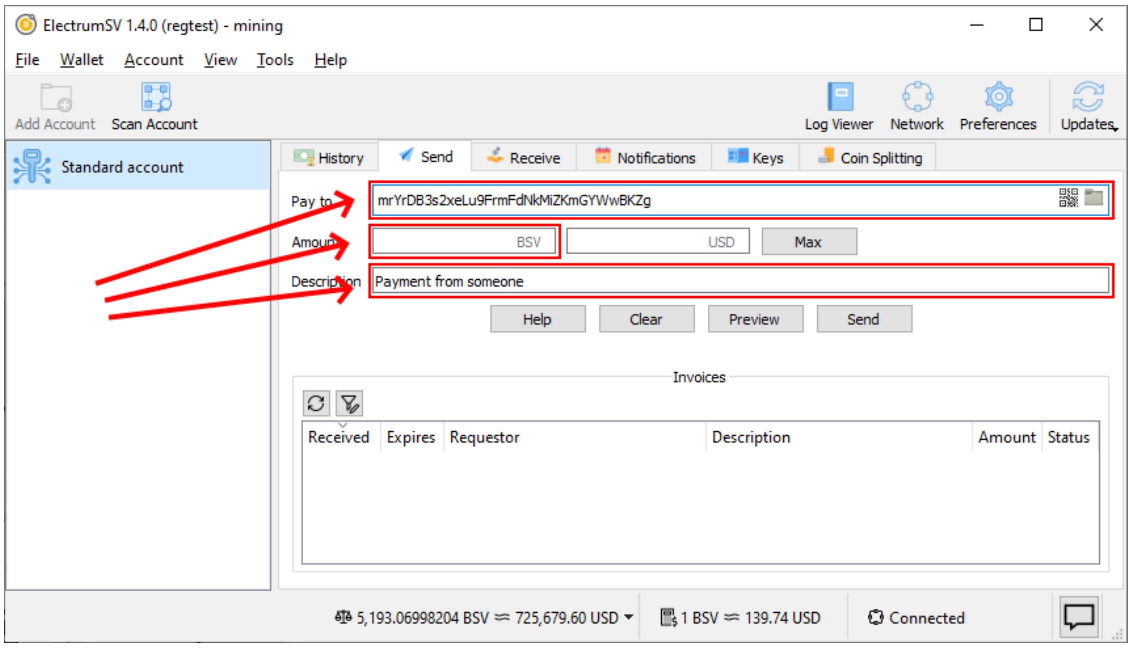
Fig. 33: The incoming payment list at the bottom of the "Receive" tab.



Account:	Standard account
Payment destination:	mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg
Description:	Payment from someone
Requested amount:	BSV USD
Request expires:	About 23 hours ago

5,193.06998204 BSV ≈ 723,394.65 USD · 1 BSV ≈ 139.30 USD · Connected

Fig. 34: The "Copy payment link" button.



Pay to: mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg

Amount: BSV / USD

Description: Payment from someone

5,193.06998204 BSV ≈ 725,679.60 USD | 1 BSV ≈ 139.74 USD | Connected

Fig. 35: The "Send" tab form populated from the copied payment link.

Copying the raw payment destination

In some cases, you might just want to give the payer the raw payment destination which will either be an address or a BIP276 script.

Using a QR code

If the other party is standing there with you, you can show them the expected payment dialog and they can take a photo of the QR code with their wallet. Their wallet will extract the address and streamline the payment process.

1.4.2 Identifying incoming payments

In the legacy model, which is still the most common one, payments are fire and forget. The payer constructs a transaction and broadcasts it to the blockchain. Then when your wallet gets a notification a payment of interest has appeared in the blockchain, it retrieves that transaction and factors it into the related account.

With this model, the wallet has no idea a payment is incoming until it arrives out of the blue. A new and better model is available in the form of Paymail, but ElectrumSV does not have the service infrastructure to support it at this time. We are however working towards it.

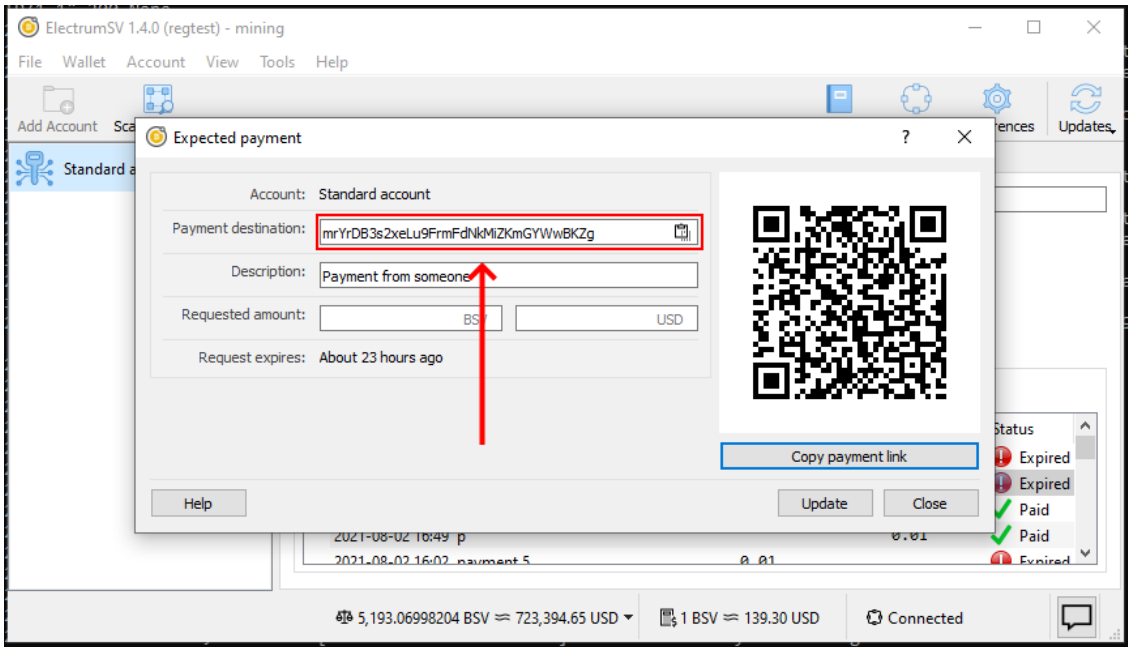

ElectrumSV 1.4.0 (regtest) - mining
File Wallet Account View Tools Help
Add Account
Updates
Expected payment
Account: Standard account
Payment destination: mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg
Description: Payment from someone
Requested amount: BSV USD
Request expires: About 23 hours ago
Copy payment link
Help
Update
Close
Status
Expired
Expired
Paid
Paid
5,193.06998204 BSV ≈ 723,394.65 USD
1 BSV ≈ 139.30 USD
Connected

Fig. 36: Copying the raw payment destination.

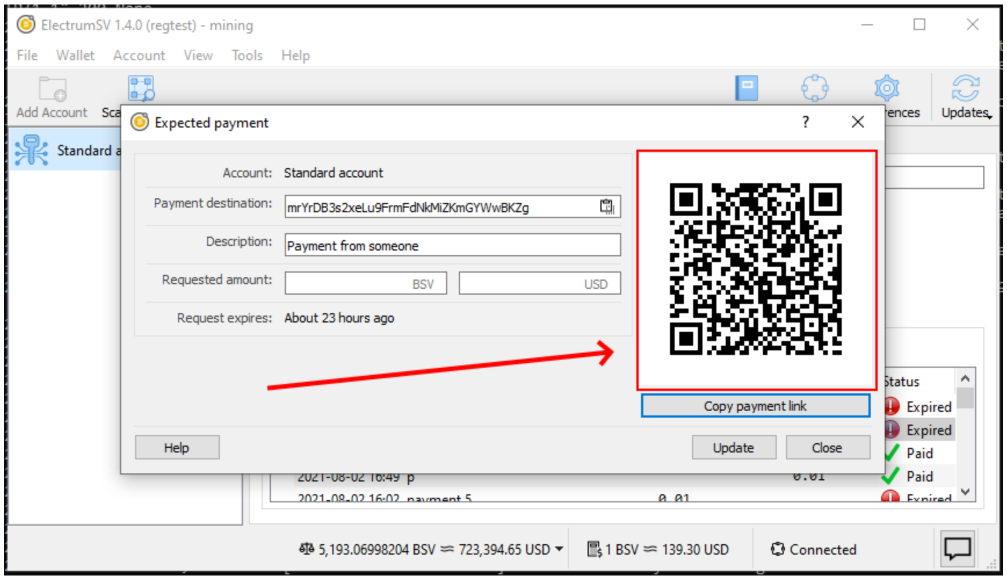
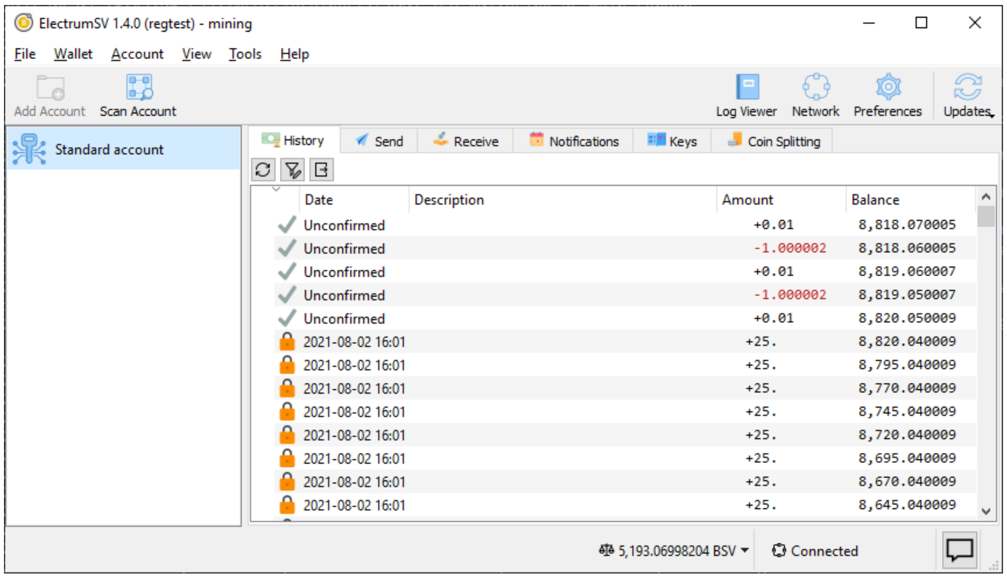

ElectrumSV 1.4.0 (regtest) - mining
File Wallet Account View Tools Help
Add Account
Updates
Expected payment
Account: Standard account
Payment destination: mrYrDB3s2xeLu9FrmFdNkMiZKmGYWwBKZg
Description: Payment from someone
Requested amount: BSV USD
Request expires: About 23 hours ago
Copy payment link
Help
Update
Close
Status
Expired
Expired
Paid
Paid
5,193.06998204 BSV ≈ 723,394.65 USD
1 BSV ≈ 139.30 USD
Connected

Fig. 37: The QR code provided in the expected payment dialog.



Date	Description	Amount	Balance
✓ Unconfirmed		+0.01	8,818.070005
✓ Unconfirmed		-1.000002	8,818.060005
✓ Unconfirmed		+0.01	8,819.060007
✓ Unconfirmed		-1.000002	8,819.050007
✓ Unconfirmed		+0.01	8,820.050009
2021-08-02 16:01		+25.	8,820.040009
2021-08-02 16:01		+25.	8,795.040009
2021-08-02 16:01		+25.	8,770.040009
2021-08-02 16:01		+25.	8,745.040009
2021-08-02 16:01		+25.	8,720.040009
2021-08-02 16:01		+25.	8,695.040009
2021-08-02 16:01		+25.	8,670.040009
2021-08-02 16:01		+25.	8,645.040009

Fig. 38: The history tab when awaiting an incoming payment.

1.5 Making a payment

If you are reading this, you probably want to know how to make a payment. We currently only support making payments in the following ways:

- Payment to a Bitcoin address.
- Payment to a BIP276 address.
- Payment to a Bitcoin script.

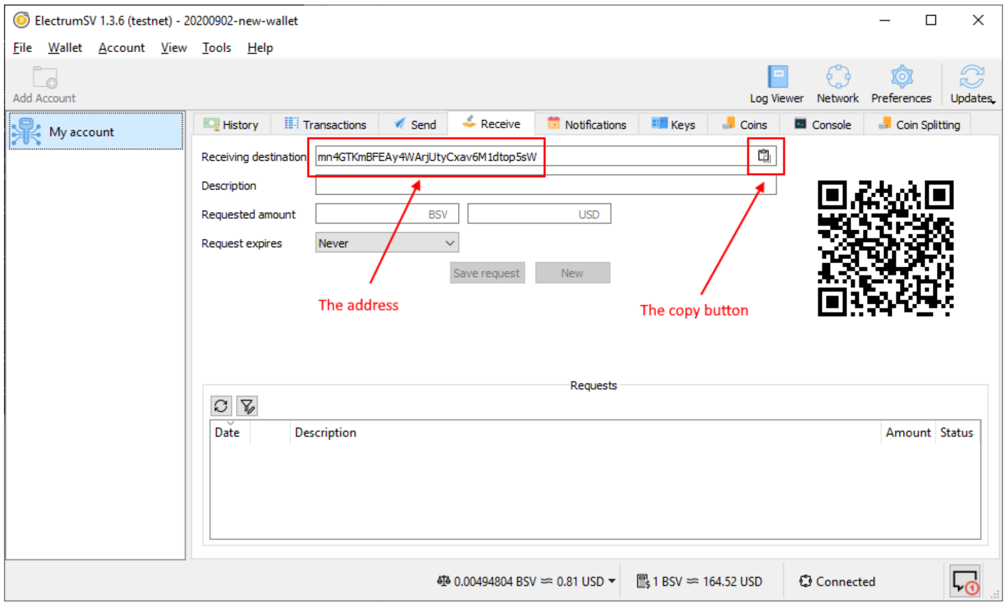
This guide solely covers payment to an address. It is not recommended you pay to a Bitcoin script unless you are an expert.

1.5.1 Paying to yourself

At this point you should have a wallet with a standard account. You should also have an address from another party, that you can make a payment to. However for the purpose of this guide, you can make a payment to yourself, if you have no-one else to currently pay.

Start off on the receiving tab.

As you will be paying to yourself, copy the shown address. The best way to do this is to click on the copy button, which will copy it to the clipboard. You will use this address as you would the address for any other party.



Receiving destination mn4GTKmBFEAy4WArjUtyCxav6M1dtop5sW
Description
Requested amount BSV USD
Request expires Never

0.00494804 BSV ≈ 0.81 USD
1 BSV ≈ 164.52 USD

Fig. 39: Highlighted areas on the receiving tab.

1.5.2 Paying to an address

Ensure your wallet window is now showing the send tab. Select the "Pay to" field and paste in the address you wish to make a payment to.

After pasting in the address, enter a nominal amount of Bitcoin SV to send, where your wallet has sufficient funds to do so.

Important: If you are paying to addresses a good practice is to make what is called a `pilot payment` first, where you pay a small amount you can afford to lose, before paying the larger full amount.

Click the "Send" button to start the payment process.

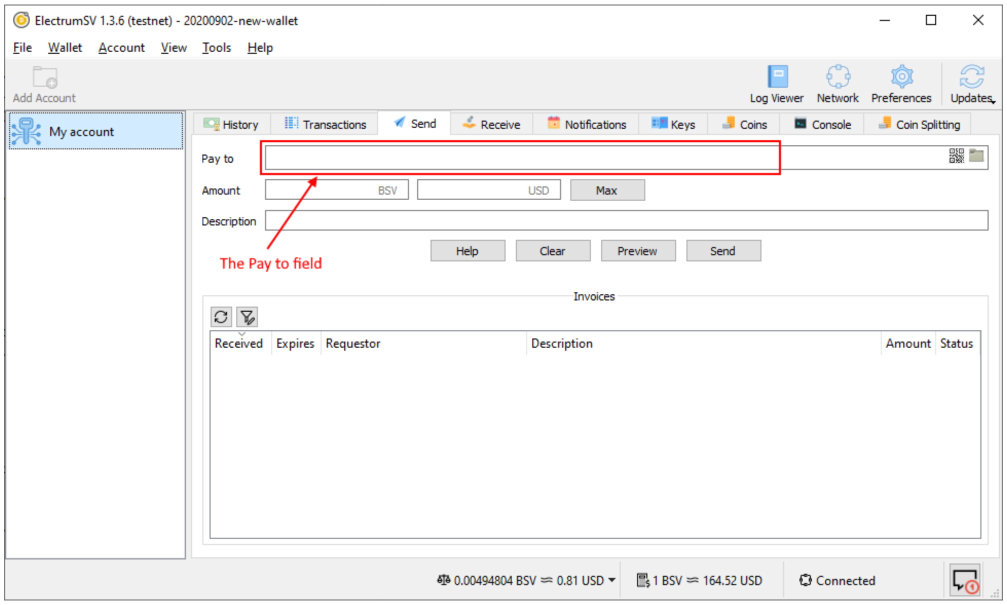
Ensure that both the amount you are sending and the mining fee are the appropriate amounts, then enter your password and click "OK". The "OK" button only becomes enabled when you have entered your password correctly. The transaction will broadcast, and you should receive a confirmation that the payment was made.

The confusing sequence of letters and numbers is actually the ID of the transaction that contained your payment. This can be used to look up your payment, if you were to take it and paste it into a web site that indexes the Bitcoin SV blockchain.

1.5.3 The record of payment

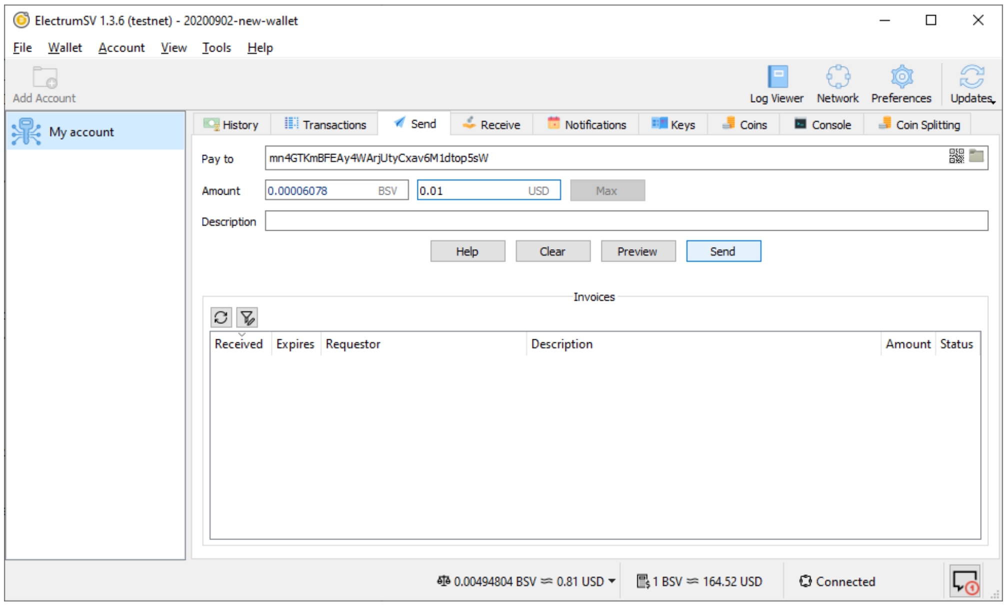
At it's current state of development, the wallet does not have much context about payments made. But you can see the transactions this account is involved in, in the history tab.

If you had provided a description when making the payment, it would appear here in much the same way as the existing transactions with their "ElectrumSV coin splitting: Your split coins" descriptions.


ElectrumSV 1.3.6 (testnet) - 20200902-new-wallet

0.00494804 BSV ≈ 0.81 USD
1 BSV ≈ 164.52 USD
Connected

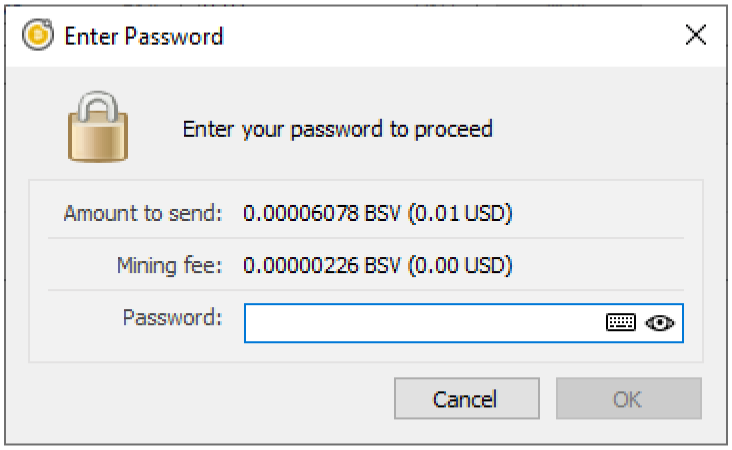
Fig. 40: Highlighted areas on the send tab.


ElectrumSV 1.3.6 (testnet) - 20200902-new-wallet

Pay to: mn4GTKmBFEAy4WArjUtyCxav6M1dtop5sW
Amount: 0.00006078 BSV 0.01 USD Max
Description:

0.00494804 BSV ≈ 0.81 USD
1 BSV ≈ 164.52 USD
Connected

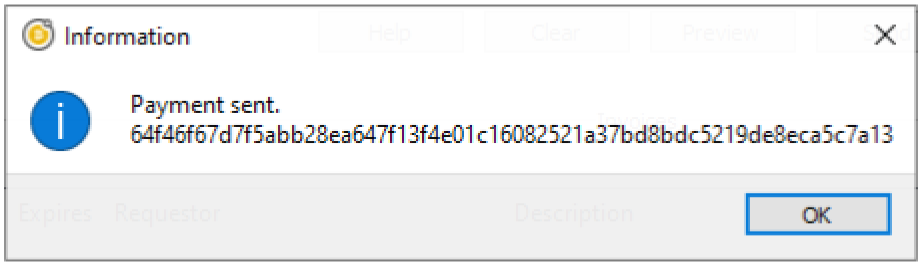
Fig. 41: The filled out send tab.



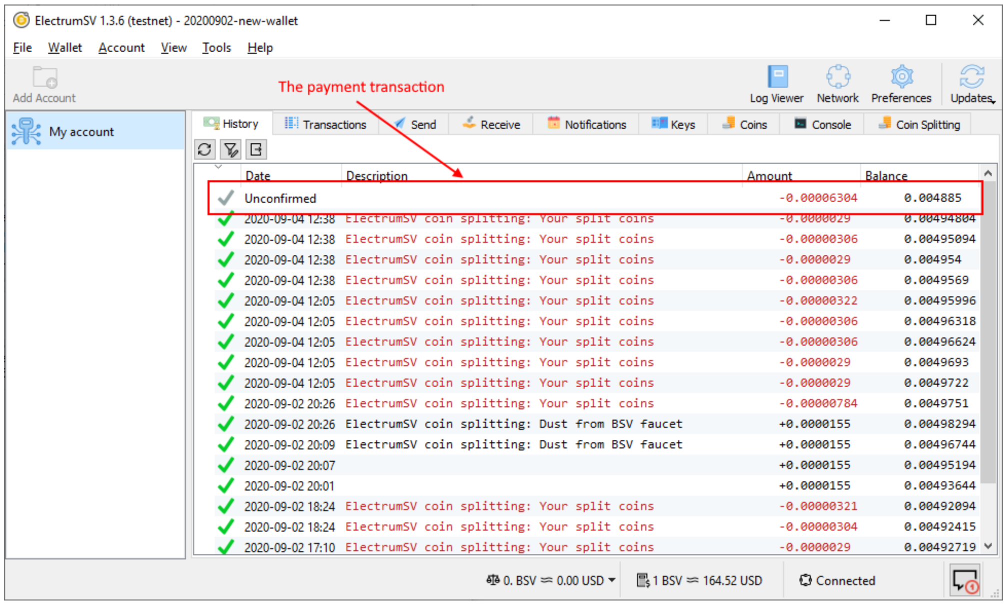
Enter your password to proceed

Amount to send:	0.00006078 BSV (0.01 USD)
Mining fee:	0.00000226 BSV (0.00 USD)
Password:	

Fig. 42: The password confirmation dialog.



Payment sent.
64f46f67d7f5abb28ea647f13f4e01c16082521a37bd8bdc5219de8eca5c7a13



	Date	Description	Amount	Balance
✓	Unconfirmed		-0.00006304	0.004885
✓	2020-09-04 12:38	ElectrumSV coin splitting: Your split coins	-0.0000029	0.00494804
✓	2020-09-04 12:38	ElectrumSV coin splitting: Your split coins	-0.00000306	0.00495094
✓	2020-09-04 12:38	ElectrumSV coin splitting: Your split coins	-0.0000029	0.004954
✓	2020-09-04 12:38	ElectrumSV coin splitting: Your split coins	-0.00000306	0.0049569
✓	2020-09-04 12:05	ElectrumSV coin splitting: Your split coins	-0.00000322	0.00495996
✓	2020-09-04 12:05	ElectrumSV coin splitting: Your split coins	-0.00000306	0.00496318
✓	2020-09-04 12:05	ElectrumSV coin splitting: Your split coins	-0.00000306	0.00496624
✓	2020-09-04 12:05	ElectrumSV coin splitting: Your split coins	-0.0000029	0.0049693
✓	2020-09-04 12:05	ElectrumSV coin splitting: Your split coins	-0.0000029	0.0049722
✓	2020-09-02 20:26	ElectrumSV coin splitting: Your split coins	-0.00000784	0.0049751
✓	2020-09-02 20:26	ElectrumSV coin splitting: Dust from BSV faucet	+0.0000155	0.00498294
✓	2020-09-02 20:09	ElectrumSV coin splitting: Dust from BSV faucet	+0.0000155	0.00496744
✓	2020-09-02 20:07		+0.0000155	0.00495194
✓	2020-09-02 20:01		+0.0000155	0.00493644
✓	2020-09-02 18:24	ElectrumSV coin splitting: Your split coins	-0.00000321	0.00492094
✓	2020-09-02 18:24	ElectrumSV coin splitting: Your split coins	-0.00000304	0.00492415
✓	2020-09-02 17:10	ElectrumSV coin splitting: Your split coins	-0.0000029	0.00492719

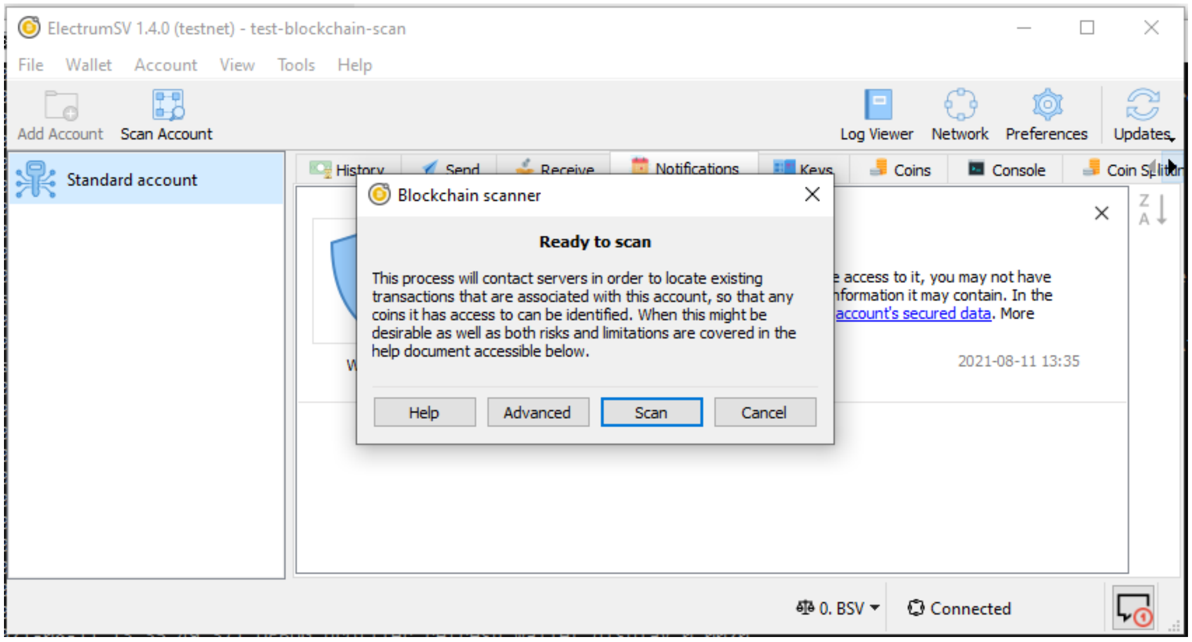
Fig. 44: The highlighted payment transaction in the history tab.

1.6 Scanning the blockchain

As long as there are servers that offer the ability to locate unknown transactions, ElectrumSV will continue to support it. The main use of blockchain scanning, is locating the transactions associated with the seed words or keys that a user imports. Additionally, in the short term before the SPV model is formalised, it can also be used to find unexpected transactions the wallet does not know to look for.

1.6.1 When you import existing seed words or keys

When you create an account by importing existing keys, ElectrumSV should automatically show the blockchain scanning user interface when the wallet window displays that account.



Ready to scan

This process will contact servers in order to locate existing transactions that are associated with this account, so that any coins it has access to can be identified. When this might be desirable as well as both risks and limitations are covered in the help document accessible below.

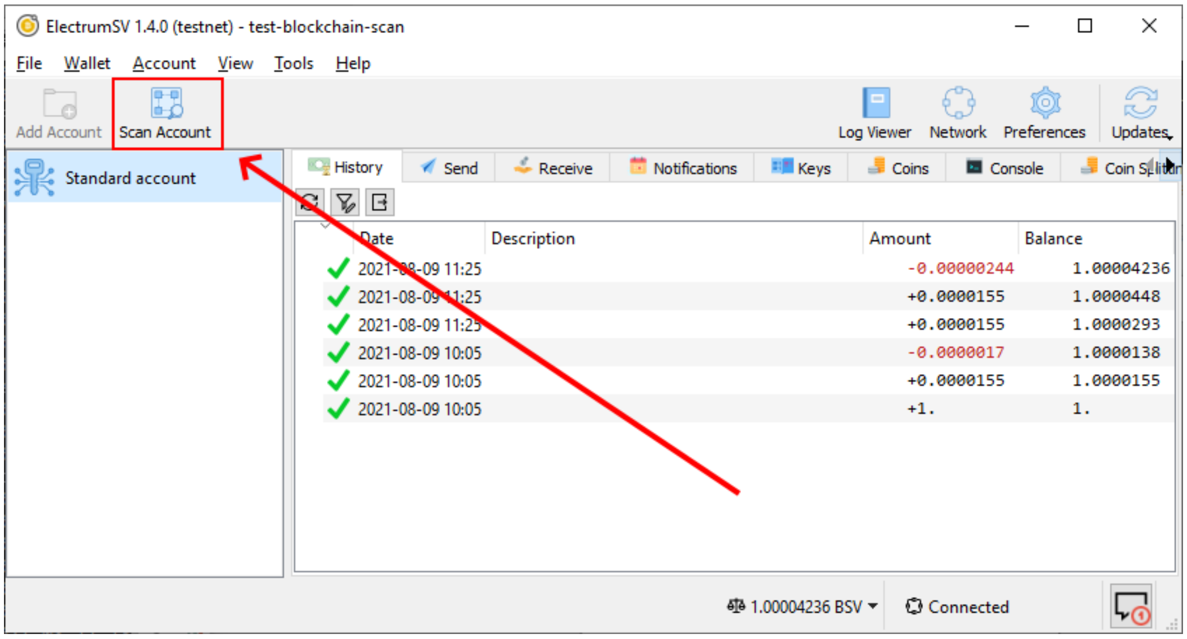
e access to it, you may not have
nformation it may contain. In the
account's secured data. More

2021-08-11 13:35

Fig. 45: The blockchain scanning window.

1.6.2 When you want to run the scanner manually

The button to open the blockchain scanner is featured prominently in the wallet toolbar. It only operates on the currently selected account, but should find any transaction relating to the account that ElectrumSV itself is capable of creating or processing.



Date	Description	Amount	Balance
✓ 2021-08-09 11:25		-0.00000244	1.00004236
✓ 2021-08-09 11:25		+0.0000155	1.0000448
✓ 2021-08-09 11:25		+0.0000155	1.0000293
✓ 2021-08-09 10:05		-0.0000017	1.0000138
✓ 2021-08-09 10:05		+0.0000155	1.0000155
✓ 2021-08-09 10:05		+1.	1.

Fig. 46: The “Scan blockchain” button highlighted in the wallet window.

1.6.3 Operating the scanner

The first step is to ensure you can see the blockchain scanning window. This will either be brought up automatically when you create an account from existing keys, or when you click the button in the toolbar.

Clicking on the “Scan” button will locate any key usage on the blockchain.

By expanding the details section, users can see what transactions were located and decide if they want to import them. At this time it is not possible for users to import some but not all of the transactions, but that would be a good idea for improvement.

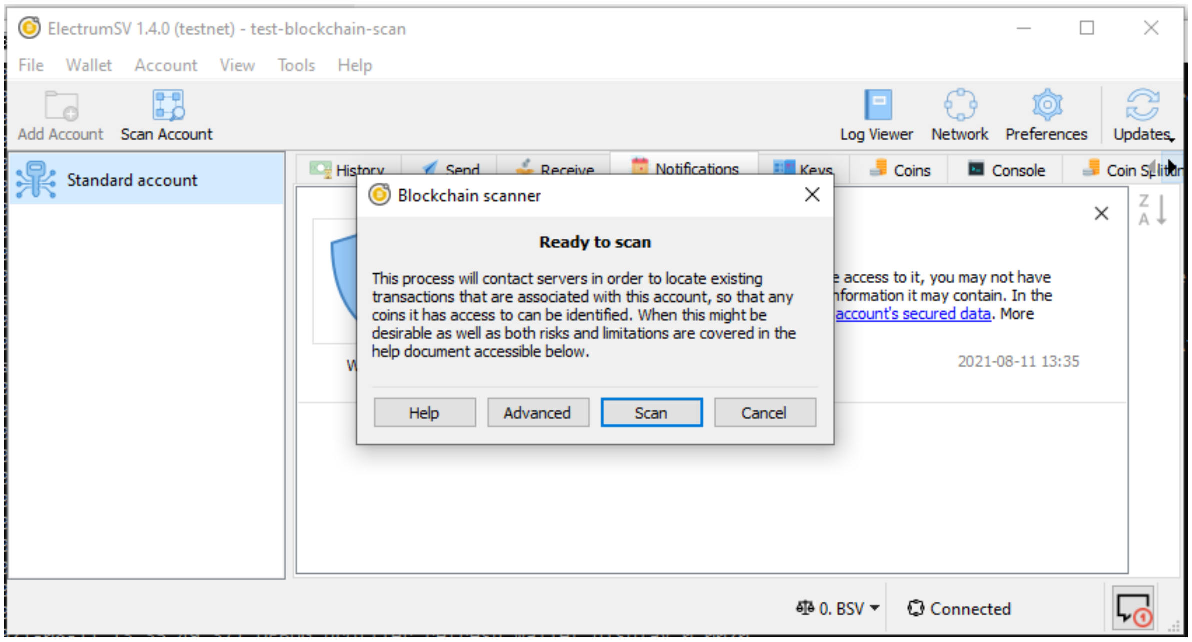
Once the import process is complete, the user will see the results. It is not guaranteed that all the transactions will be imported. The most likely reason that a transaction won’t be imported is the presence of a conflicting transaction in the account. Another potential reason would be bugs in ElectrumSV.

Advanced: Extending the “gap limits”

The BIP32 derivation paths that wallets have standardised on provide consecutive sequences of key usage. An account looks on the blockchain to find transactions related to itself, to work out it’s current balance, available coins to spend and locate historical spends and receipts. This is easily done by starting at the beginning of a derivation path and moving along it until no more transactions are found. When there is a consecutive range at the end of the derivation path of a given length (known as the gap limit) that is known to be unused, it is considered that there are no more transactions.

Standards define both seed words[1], derivation paths[2] and types of payment. One of the benefits of these standards is that with just the seed words a user can change wallets whenever they want. In the short term, while wallets do not do very much besides send and receive coins this works. In the longer term, wallets will do a lot more and it becomes impractical and unsuited. However, there have been bugs in implementation where wallets have gone far beyond the gap limit accidentally. For reasons like this in addition to the simple standardised wallet recovery it is important that ElectrumSV enable users to find transactions beyond the gap limit.

[1] BIP39 seed words standard.
[2] BIP32 derivation paths standard.



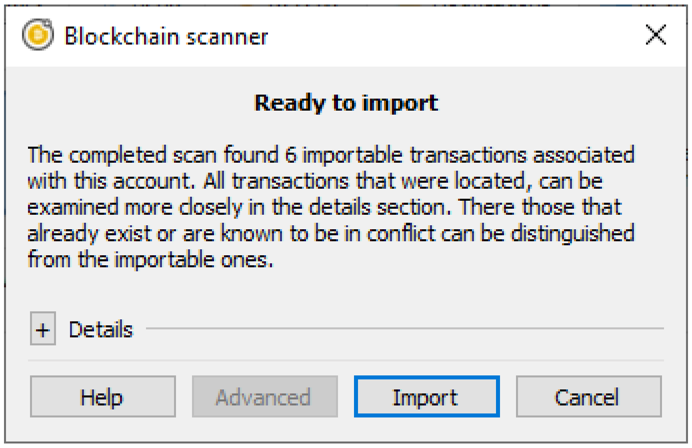
Ready to scan

This process will contact servers in order to locate existing transactions that are associated with this account, so that any coins it has access to can be identified. When this might be desirable as well as both risks and limitations are covered in the help document accessible below.

access to it, you may not have nformation it may contain. In the account's secured data. More

2021-08-11 13:35

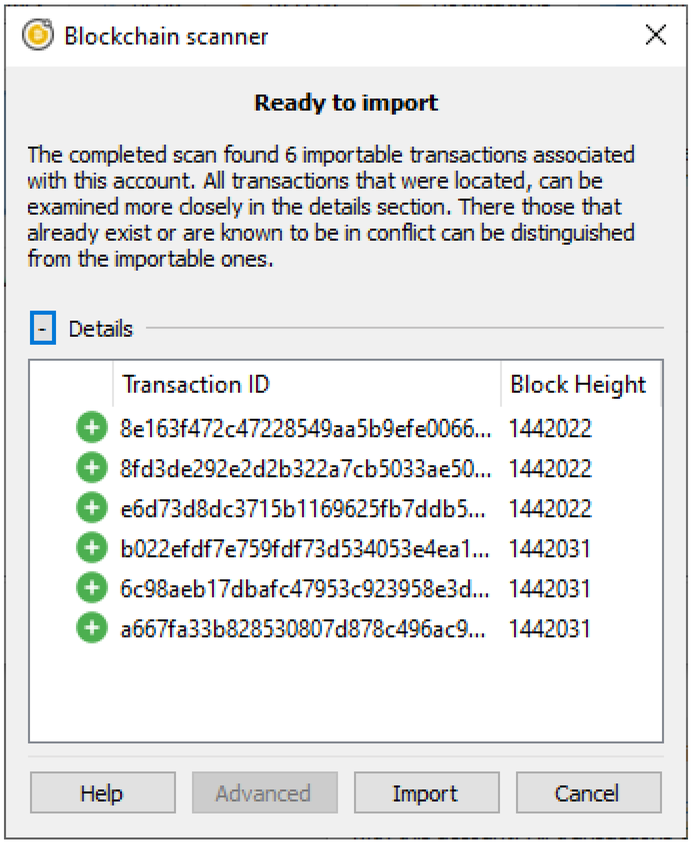
Fig. 47: The blockchain scanning window.



Ready to import

The completed scan found 6 importable transactions associated with this account. All transactions that were located, can be examined more closely in the details section. There those that already exist or are known to be in conflict can be distinguished from the importable ones.

Fig. 48: The located payments after a scan.



Blockchain scanner

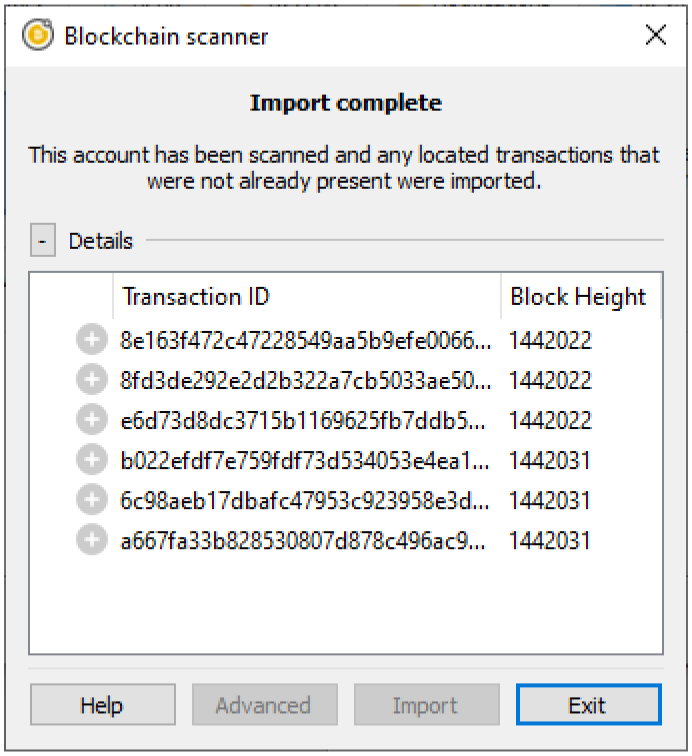
Ready to import

The completed scan found 6 importable transactions associated with this account. All transactions that were located, can be examined more closely in the details section. There those that already exist or are known to be in conflict can be distinguished from the importable ones.

Details

	Transaction ID	Block Height
➕	8e163f472c47228549aa5b9efe0066...	1442022
➕	8fd3de292e2d2b322a7cb5033ae50...	1442022
➕	e6d73d8dc3715b1169625fb7ddb5...	1442022
➕	b022efdf7e759fdf73d534053e4ea1...	1442031
➕	6c98aeb17dbafc47953c923958e3d...	1442031
➕	a667fa33b828530807d878c496ac9...	1442031

Fig. 49: The optionally viewable details of the located payments.

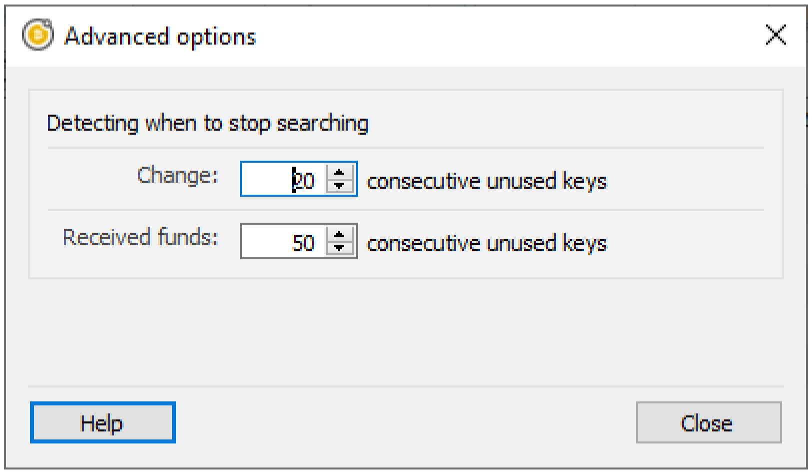


Import complete

This account has been scanned and any located transactions that were not already present were imported.

Transaction ID	Block Height
8e163f472c47228549aa5b9efe0066...	1442022
8fd3de292e2d2b322a7cb5033ae50...	1442022
e6d73d8dc3715b1169625fb7ddb5...	1442022
b022efdf7e759fdf73d534053e4ea1...	1442031
6c98aeb17dbafc47953c923958e3d...	1442031
a667fa33b828530807d878c496ac9...	1442031

Fig. 50: The results of the payment import process.



Detecting when to stop searching

Change: 20 consecutive unused keys

Received funds: 50 consecutive unused keys

Fig. 51: The blockchain scanning advanced options.

CHAPTER

TWO

PROBLEM SOLVING

Why doesn't my hardware wallet work?
Hardware wallet makers do not provide anywhere near enough support for their devices, and some have a history of making breaking changes that stop them working in ElectrumSV. If your hardware wallet does not work then this is where you should look for some pointers, whether the device is a Trezor, a Ledger, a Keepkey or a Bitbox. Read more about *hardware wallet issues*.

How do I split my coins?
If you have coins you have not touched since before Bitcoin SV and Bitcoin Cash split from each other, you might want to make sure that you can send one of these without accidentally sending the other. Read more about *coin splitting*.

How do I deal with problems on MacOS?
Apple have a special operating system which many love. But it comes with some problems. If you use MacOS and something isn't working right, maybe we have documented it here. Read more about *solutions to MacOS problems*.

What if I do not want to upgrade to the latest version?
Users may continue wanting to use the older ElectrumSV as it is, and not use the new releases. Read more about *upgrading concerns*.

2.1 Hardware wallet issues

2.1.1 Ledger

While Ledger as a company do not support Bitcoin SV as a coin on their device, users have been able to use their Ledger devices with ElectrumSV through compatibility with the Bitcoin Cash support.

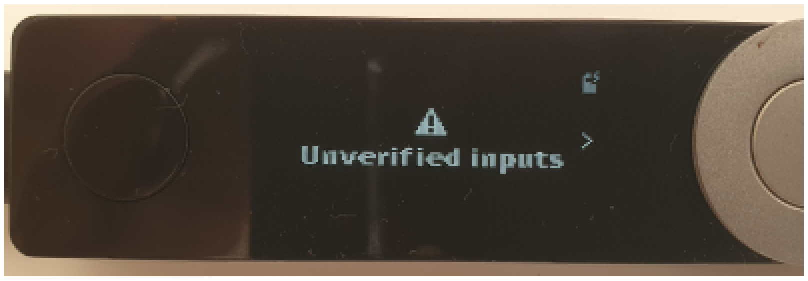
The Ledger device reports "unverified inputs"

You go to sign your transaction and your Ledger device has a confusing series of screens talking about "unverified inputs" and updating your device and/or software. You can simply step through these screens and select continue. These screens will be shown below, and then a detailed explanation of why you are seeing them will be provided.

The short version is that you can continue past these screens to signing your transaction as you signed it before you started seeing these messages, and it will be as secure as it was then. Just make sure you only sign it once, and if ElectrumSV asks you to resign it over and over not recognising that you did it once, you are probably using malware. Again, see below the screens for an explanation of this in more detail.

It should not be necessary to update your Ledger firmware and applications to deal with this.

It should not be necessary to update ElectrumSV, although you should always be using the latest version.



You can cancel the signing of the transaction if you want.

But if you select the “continue” option, the Ledger device will go through the normal transaction signing process.

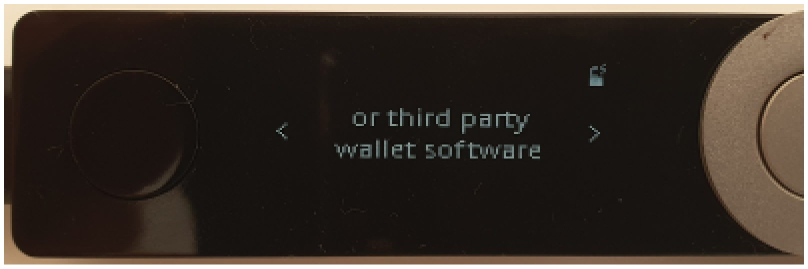
As you might recall, the first step of the correct signing process is to confirm where you are sending funds. And this is where the process is now at. You can go ahead and sign the transaction as you would have in the past before this confusing message.

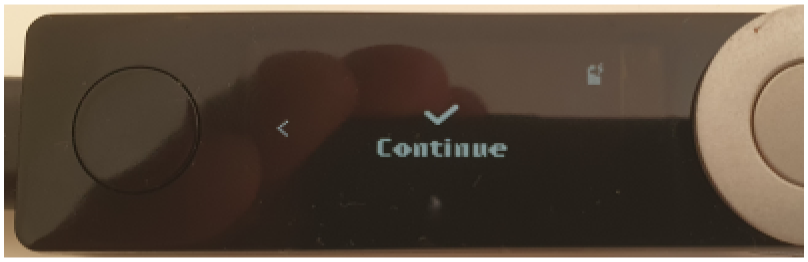
Why do I see this “unverified inputs” message?

A theoretical but unlikely exploit was discovered where wallet malware could direct a user to sign a transaction several times, and extract the signed spends from each and combine them into a new transaction which gave a large fee to miners. Trezor wrote an article about it which you can read if you wish. You see this warning because ElectrumSV is not providing the previous transactions in which the spent coins were originally received to the Ledger device.

The simple reason we do not provide the previous transaction data is because Ledger cannot handle it and will break. You can see in the Trezor hardware issues a “DataError: bytes overflow” error, which their users may encounter. We have to provide these transactions to the Trezor devices but they cannot handle them and they break, this means that Trezor users have to be careful not to spend anything other than the simplest of received payments in their transactions themselves and work out what they can and can’t spend themseves. If any of their coins is not simple and cannot be handled by Trezor, they need to bypass their hardware wallet and spend them in an unsafe way by entering their seed words.

Back to Ledger devices. Ledger allow the transaction to be signed without the spent transaction data, and on detecting they do not have it, they show an “unverified inputs” message. This makes it a little lot for Ledger users. They can still sign a spend they are confident is going to the correct places, and not bypass their hardware wallet to do so. Let’s be





honest, if someone is going to all the effort of writing malware it has never in the history of malware been to give the stolen coins to miners. The chances of downloading malware are slight, and the chances of downloading malware that gives coins to anyone other than the thief are even slighter.

2.1.2 Trezor

While Trezor as a company do not support Bitcoin SV as a coin on their device, generally Bitcoin SV users have been able to use their Trezor devices with ElectrumSV by having it in the Bitcoin Cash coin mode. However, users are encountering situations where the limitations of the Trezor device result in it no longer being sufficient to work with Bitcoin SV transactions. This likely means that if a user is planning to continue to use a Trezor device, it may require them to jump through hoops to do so.

There are two complications:

- Later versions of firmware (starting with 1.9.1 for One and 2.3.1 for Model T) require ElectrumSV to pass in parent transactions with the transaction you are signing. ElectrumSV only started supporting this in ElectrumSV 1.3.8 or newer. What this means is that if you are using these later versions of firmware, you must be using ElectrumSV 1.3.8 or newer - or it will error.
- Bitcoin SV transactions can have large output scripts, larger than what Trezor can handle. Trezor can only sign simple payments and nothing else, but this does not prevent payments from being made into the wallet with additional output scripts added for other reasons that exceed Trezor's size limit of 15 kilobytes. The parent transaction processing in the Trezor device will error when it encounters these.

Trezor devices are becoming problematic for Bitcoin SV users to use. While they are polished and enjoyable devices to use, unless the large output problem is solved by Trezor, we cannot recommend users buy these devices unless they

accept they have to own and deal with these problems. For this reason it is recommended that Trezor users downgrade their devices.

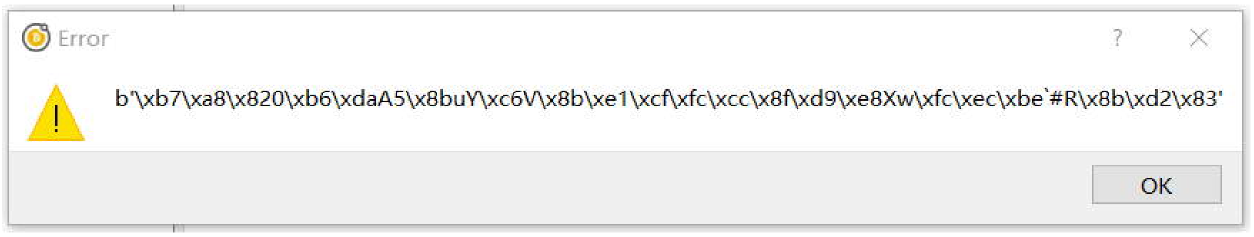
Downgrading your Trezor device

These are Trezor's firmware version pages, for users who plan to downgrade:

- Trezor One: 1.9.0.
- Trezor Model T: 2.3.0.

You will need to visit those pages and download the firmware file. Trezor provide instructions on how to downgrade, and let you know how and where to use the file.

Problem: You see a random looking series of numbers and letters

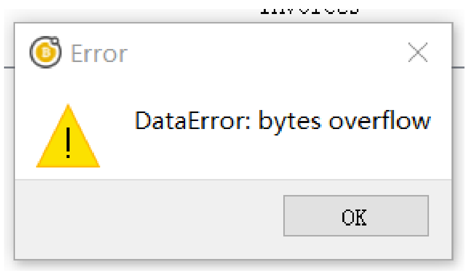


```
b'\xb7\xa8\x820\xb6\xdaA5\x8buY\xc6V\x8b\xe1\xcf\xfc\xcc\x8f\xd9\xe8Xw\xfc\xec\xbe`#R\x8b\xd2\x83'
```

Fig. 1: What this problem looks like..

You are using ElectrumSV 1.3.7 or earlier, and your Trezor device has a later version of the firmware. It expects ElectrumSV to have provided the transaction associated with those numbers and letters, but the ElectrumSV version you are using does not know how to or even that it should. You can take the risk of updating to a more recent version of ElectrumSV that supports these parent transactions, and possibly encounter the "DataError: bytes overflow" problem. Or you can downgrade your Trezor firmware to the version listed above.

Problem: You see the message "DataError: bytes overflow"



DataError: bytes overflow

Fig. 2: What this problem looks like..

One of your parent transactions contains not only the coin you are trying to spend, but a large output script. Your Trezor device has a later version of firmware where parent transactions are required to be provided, and the device is choking on the large output. This is a limit in the device itself, and ElectrumSV can do nothing about this. To spend the coin associated with the problem parent transaction, you need to downgrade your firmware to the versions listed above.

2.2 Coin splitting

Important: ElectrumSV can only be downloaded from electrumsv.io.

When users have coins that existed before Bitcoin Cash became a separate blockchain from Bitcoin SV, those coins are linked on both blockchains. When they are sent in a wallet on one blockchain, that action can also send them on the other blockchain. Users have had this accidentally happen to them, and the recipient has refused to refund the coins from the blockchain the user did not intend to send on.

If you think you have unsplit coins in your wallet, you can use ElectrumSV's coin-splitting feature to split them. But keep in mind that you are responsible for your own coins, you should verify for yourself that the splitting worked. And if you are unsure whether your coins need to be split, you can always split them anyway.

2.2.1 How does splitting work?

The process is simple, if the coins are sent on Bitcoin SV in a way that is incompatible with Bitcoin Cash, then the coins are split. Any usage of those specific coins that have been split will from then on be independent on either blockchain.

In order to keep it simple, we only do the simplest case. We make your wallet do a payment to itself that combines all the available coins within it in a way that should be valid on Bitcoin SV and not Bitcoin Cash. This results in one single split coin combining all the individual coins that you had in your wallet before the split.

2.2.2 How you split your coins

Unfortunately, all the coins in the wallet used here are already split. So the following is just going through the process to show you how it works. You can see that this wallet contains a small amount of Bitcoin SV.

Let's start by changing to the coin-splitting tab:

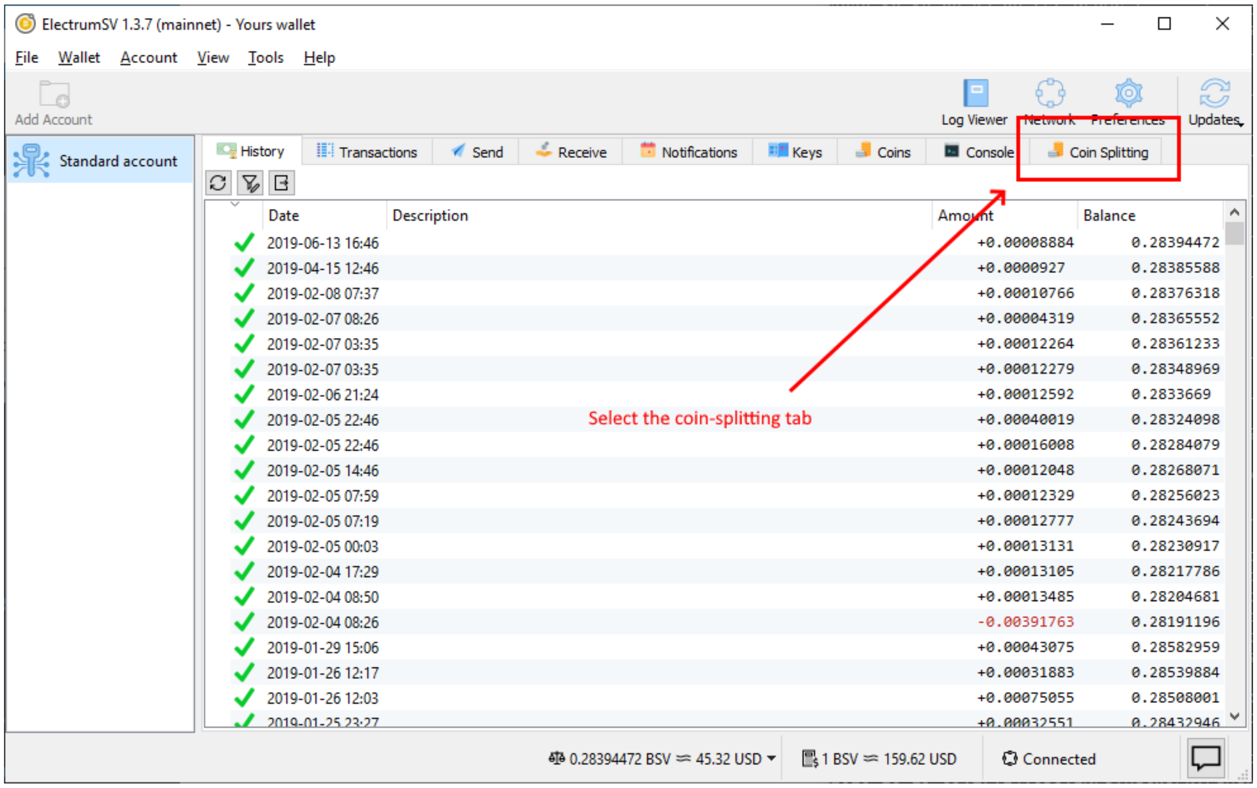
Once you are looking at the coin-splitting tab, you have two options. Either direct splitting or faucet splitting. We recommend the direct splitting, and do not really support the faucet splitting any more. Direct splitting does not work for hardware wallets, which due to inherent limitations can only work in simple ways.

Clicking on the direct splitting button will ask you for your password. You will see that the balance of the splitting transaction is the balance of the available coins in the wallet.

After you enter your password, it will sign and broadcast your transaction. This will happen pretty quickly, and once it is done you will see a dialog letting you know the splitting transaction was broadcast.

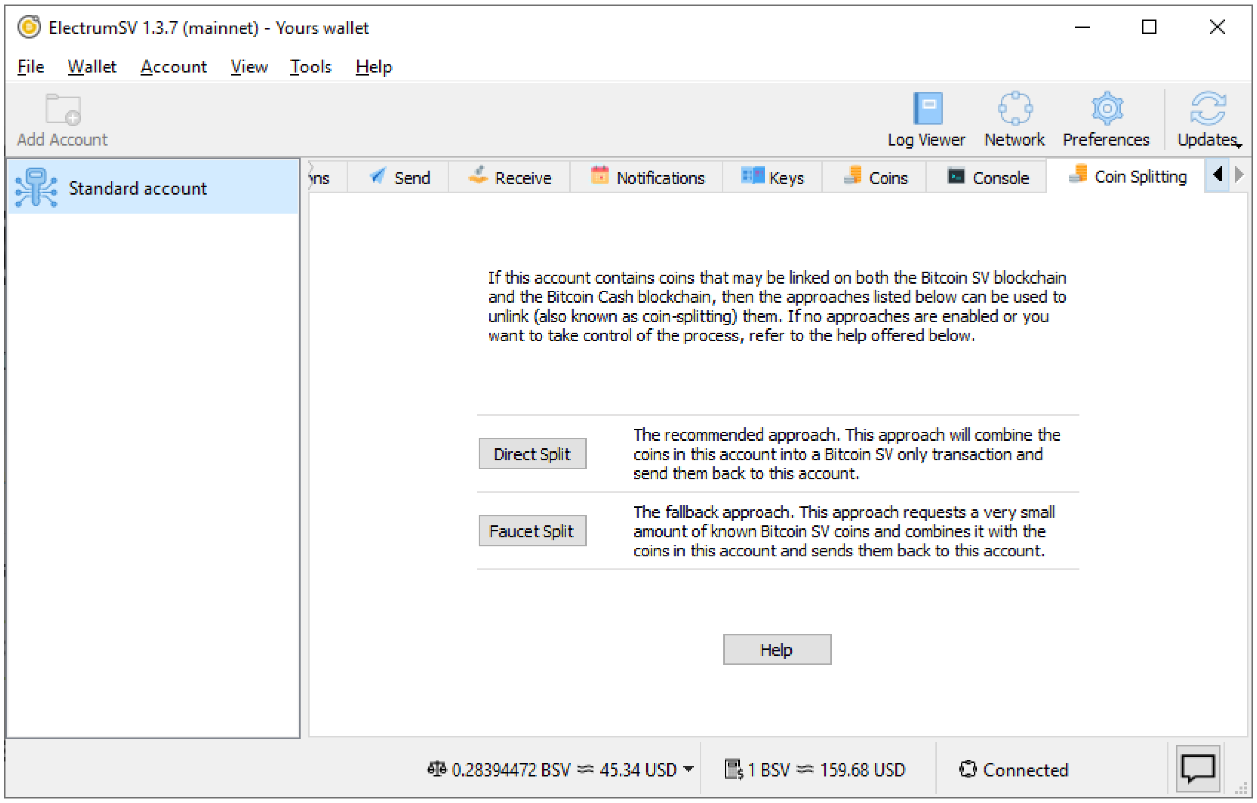
You can now go back to the history tab and see the splitting transaction there, which has an automatic description noting what it was created for.

In theory, your coins should be split. But again, you are responsible for using them safely and you should ensure that they are really split.


ElectrumSV 1.3.7 (mainnet) - Yours wallet
File Wallet Account View Tools Help
Add Account
Log Viewer Network Preferences Updates
Standard account
History Transactions Send Receive Notifications Keys Coins Console Coin Splitting
Select the coin-splitting tab
0.28394472 BSV ≈ 45.32 USD
1 BSV ≈ 159.62 USD
Connected

	Date	Description	Amount	Balance
✓	2019-06-13 16:46		+0.00008884	0.28394472
✓	2019-04-15 12:46		+0.0000927	0.28385588
✓	2019-02-08 07:37		+0.00010766	0.28376318
✓	2019-02-07 08:26		+0.00004319	0.28365552
✓	2019-02-07 03:35		+0.00012264	0.28361233
✓	2019-02-07 03:35		+0.00012279	0.28348969
✓	2019-02-06 21:24		+0.00012592	0.2833669
✓	2019-02-05 22:46		+0.00040019	0.28324098
✓	2019-02-05 22:46		+0.00016008	0.28284079
✓	2019-02-05 14:46		+0.00012048	0.28268071
✓	2019-02-05 07:59		+0.00012329	0.28256023
✓	2019-02-05 07:19		+0.00012777	0.28243694
✓	2019-02-05 00:03		+0.00013131	0.28230917
✓	2019-02-04 17:29		+0.00013105	0.28217786
✓	2019-02-04 08:50		+0.00013485	0.28204681
✓	2019-02-04 08:26		-0.00391763	0.28191196
✓	2019-01-29 15:06		+0.00043075	0.28582959
✓	2019-01-26 12:17		+0.00031883	0.28539884
✓	2019-01-26 12:03		+0.00075055	0.28508001
✓	2019-01-25 23:27		+0.00032551	0.28432946

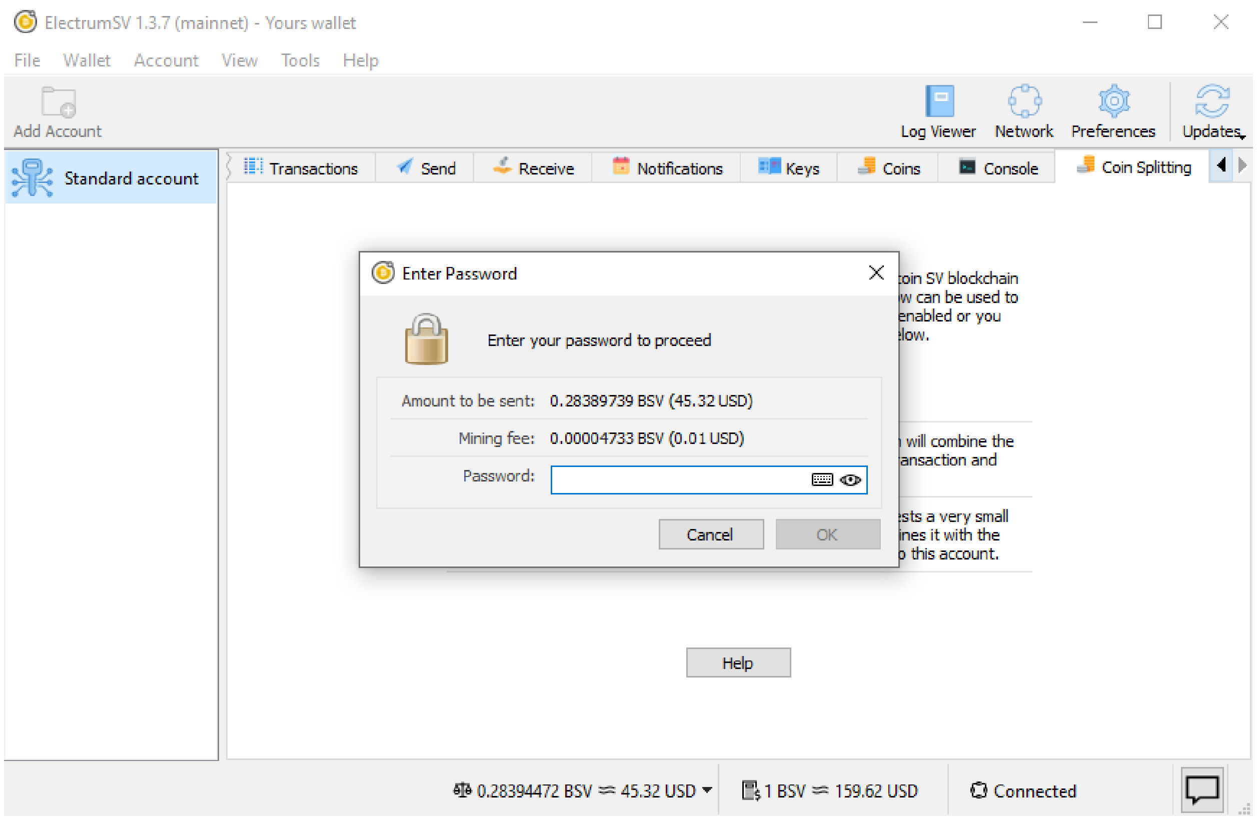
Fig. 3: Selecting the coin-splitting tab.


ElectrumSV 1.3.7 (mainnet) - Yours wallet
File Wallet Account View Tools Help
Add Account
Log Viewer Network Preferences Updates
Standard account
ns Send Receive Notifications Keys Coins Console Coin Splitting
Direct Split
Faucet Split
Help
0.28394472 BSV ≈ 45.34 USD
1 BSV ≈ 159.68 USD
Connected

If this account contains coins that may be linked on both the Bitcoin SV blockchain and the Bitcoin Cash blockchain, then the approaches listed below can be used to unlink (also known as coin-splitting) them. If no approaches are enabled or you want to take control of the process, refer to the help offered below.

Direct Split	The recommended approach. This approach will combine the coins in this account into a Bitcoin SV only transaction and send them back to this account.
Faucet Split	The fallback approach. This approach requests a very small amount of known Bitcoin SV coins and combines it with the coins in this account and sends them back to this account.

Fig. 4: The coin-splitting tab.


ElectrumSV 1.3.7 (mainnet) - Yours wallet

Enter Password

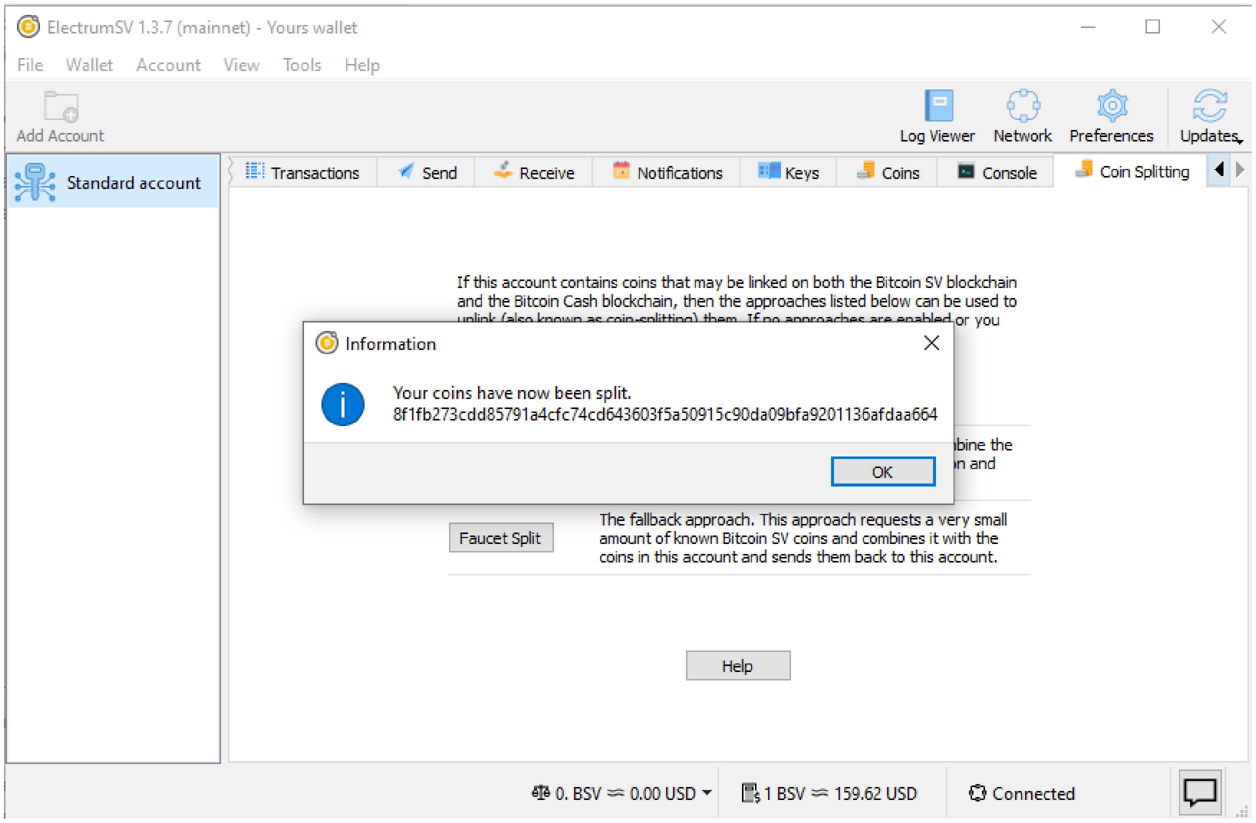
Enter your password to proceed

Amount to be sent: 0.28389739 BSV (45.32 USD)

Mining fee: 0.00004733 BSV (0.01 USD)

Password:

Fig. 5: Approve the splitting payment.



If this account contains coins that may be linked on both the Bitcoin SV blockchain and the Bitcoin Cash blockchain, then the approaches listed below can be used to unlink (also known as coin-splitting) them. If no approaches are enabled or you

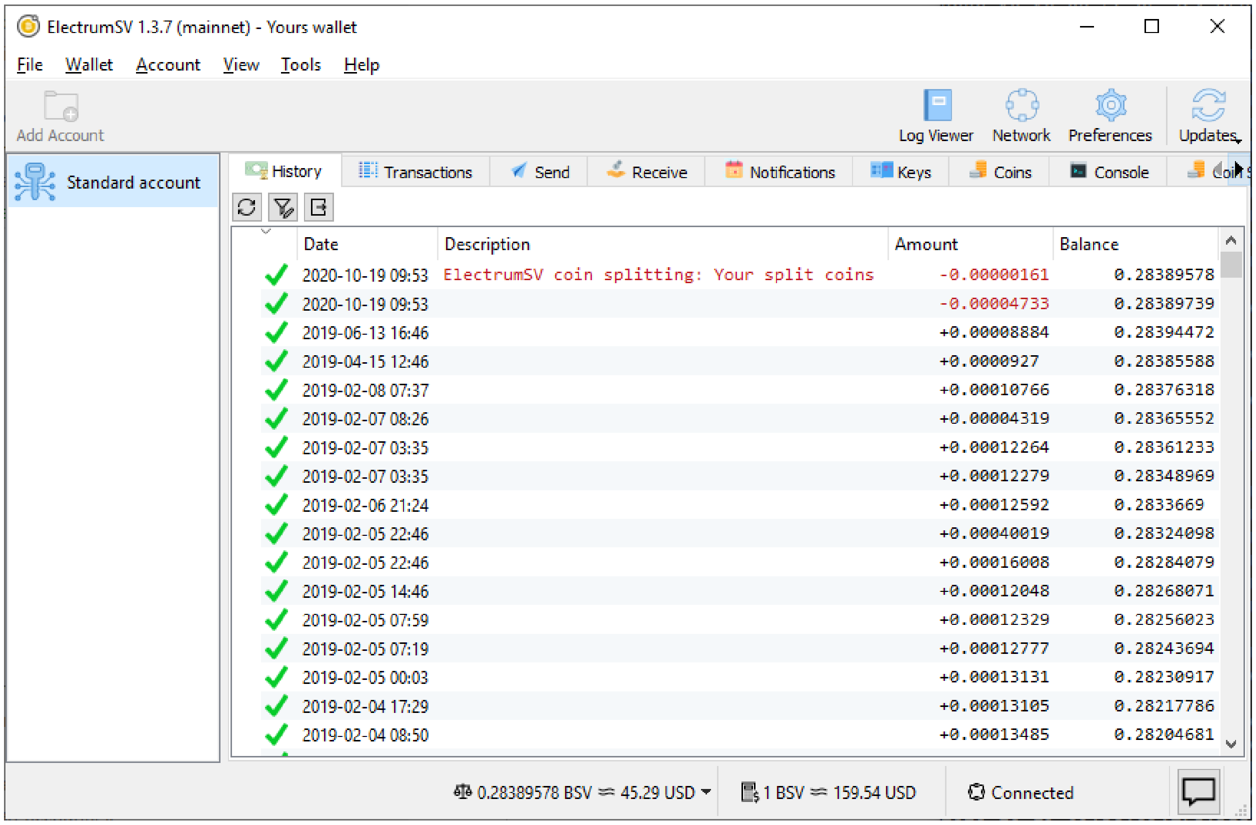
Information

Your coins have now been split.
8f1fb273cdd85791a4cfc74cd643603f5a50915c90da09bfa9201136afdaa664

The fallback approach. This approach requests a very small amount of known Bitcoin SV coins and combines it with the coins in this account and sends them back to this account.

0. BSV ≈ 0.00 USD
1 BSV ≈ 159.62 USD

Fig. 6: The split action completion message.



	Date	Description	Amount	Balance
✓	2020-10-19 09:53	ElectrumSV coin splitting: Your split coins	-0.00000161	0.28389578
✓	2020-10-19 09:53		-0.00004733	0.28389739
✓	2019-06-13 16:46		+0.00008884	0.28394472
✓	2019-04-15 12:46		+0.0000927	0.28385588
✓	2019-02-08 07:37		+0.00010766	0.28376318
✓	2019-02-07 08:26		+0.00004319	0.28365552
✓	2019-02-07 03:35		+0.00012264	0.28361233
✓	2019-02-07 03:35		+0.00012279	0.28348969
✓	2019-02-06 21:24		+0.00012592	0.2833669
✓	2019-02-05 22:46		+0.00040019	0.28324098
✓	2019-02-05 22:46		+0.00016008	0.28284079
✓	2019-02-05 14:46		+0.00012048	0.28268071
✓	2019-02-05 07:59		+0.00012329	0.28256023
✓	2019-02-05 07:19		+0.00012777	0.28243694
✓	2019-02-05 00:03		+0.00013131	0.28230917
✓	2019-02-04 17:29		+0.00013105	0.28217786
✓	2019-02-04 08:50		+0.00013485	0.28204681

0.28389578 BSV ≈ 45.29 USD   1 BSV ≈ 159.54 USD   Connected

Fig. 7: The history tab with the splitting transaction.

2.2.3 Ensuring your coins are split

Bitcoin is complicated, and in order to really know for yourself that your coins are split, you need to have some level of technical understanding. It’s a lot simpler to just send them to different places on both blockchains, especially safe places like your own wallet’s receiving addresses and check that they get there - so just do that!

Here is one way to do it:

1. Do a direct split in ElectrumSV.
2. Open your Bitcoin Cash wallet with the coins that were linked to Bitcoin SV, that you just split in ElectrumSV.
3. Create a new empty Bitcoin Cash wallet.
4. Send the coins in your existing Bitcoin Cash wallet to the new Bitcoin Cash wallet.

You can then observe that your Bitcoin Cash is in a new fresh wallet, and your Bitcoin SV is in the old wallet. Neither moved because the other moved, but rather both were moved by you. You might wonder why you need to create a second Bitcoin Cash wallet, and the reason is that this ensures that your Bitcoin SV and Bitcoin Cash are using different keys and it both helps verify they are unlinked and gives you better security going forward.

2.2.4 Hardware wallets

Hardware wallets are extremely limited devices with not much flexibility. They only allow certain types of transactions to be signed, and this does not include the type that the direct splitting method uses.

If you have a hardware wallet, you can try and use faucet splitting. Faucet splitting works by adding a very small Bitcoin SV coin to your wallet, then combining all the available coins in your wallet with that Bitcoin SV coin. This creates a new Bitcoin SV coin which is of course incompatible with the Bitcoin Cash blockchain, and so the coins in the wallet have been split.

Alternatively, if the faucet is not working you can get someone to send you a very small amount of Bitcoin SV and you can accomplish the same thing yourself by sending all the coins in your wallet to one of your own addresses (including that very small amount of Bitcoin SV).

2.2.5 Increasing differences between blockchains

There are an increasing number of changes between Bitcoin Cash and Bitcoin SV. While it is good practice to split your coins just in case you lose your Bitcoin SV when sending your Bitcoin Cash, or lose your Bitcoin Cash when sending your Bitcoin SV, it is possibly becoming easier to avoid it.

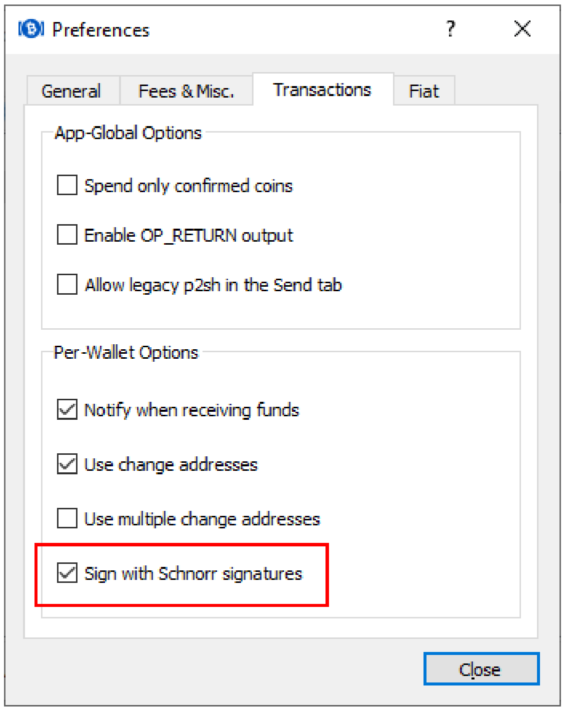
High minimum fee on Bitcoin Cash

The Bitcoin Cash servers for the Electron Cash wallet rejected any attempt to broadcast a transaction containing unsplit coins that had 0.5 satoshis per byte fee as too low. Experiments suggest that it is very difficult to get a transaction at this fee level to propagate, maybe nearing impossible.

As the default fee in ElectrumSV is 0.5 satoshis per byte, this could mean that if you send unsplit coins in ElectrumSV the Bitcoin Cash network will completely ignore them. Should you rely on this? No, but it might provide a coincidental safety net for people who do not know they should split their coins.

Schnorr signatures

By default Electron Cash and likely all Bitcoin Cash wallets now use Schnorr signatures. What this means is that the transactions they make should be incompatible with Bitcoin SV as long as the user has not opted out of using Schnorr. So in theory you can just send your coins on Bitcoin Cash and because those Schnorr signatures are used, the coins on Bitcoin Cash have been sent in a way that is incompatible with Bitcoin SV.



App-Global Options

- [ ] Spend only confirmed coins
- [ ] Enable OP_RETURN output
- [ ] Allow legacy p2sh in the Send tab

Per-Wallet Options

- [✓] Notify when receiving funds
- [✓] Use change addresses
- [ ] Use multiple change addresses
- [✓] Sign with Schnorr signatures

Fig. 8: The default Electron Cash Schnorr setting.

Should you rely on this? Not unless you know for sure that you are using Schnorr signatures in your Bitcoin Cash wallet, and that you have used the correctly.

2.2.6 Thanks

Many thanks to satoshi.io who provided unsplit coins used for testing related to this article.

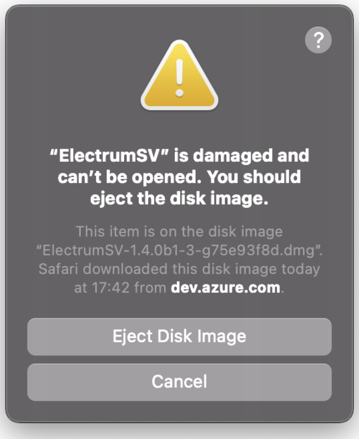
2.3 MacOS issues

2.3.1 Problems launching ElectrumSV

There are various different obstacles users may encounter when they try to run ElectrumSV depending on which version of the operating system they are using. We'll illustrate each below and explain what it means, and what you can do about it.

“damaged and can’t be opened”

This is a bug in the operating system. What it means is that the file you downloaded is unsigned and they won’t run it or give you the standard ways to work around it involving the Security Center. There is a workaround which you can do, but it involves you using the terminal.



“ElectrumSV” is damaged and can’t be opened. You should eject the disk image.

This item is on the disk image “ElectrumSV-1.4.0b1-3-g75e93f8d.dmg”. Safari downloaded this disk image today at 17:42 from **dev.azure.com**.

Workaround

The solution to this is the following steps:

1. Open the terminal. If you do not know how, you can go to Launchpad enter “terminal” as you would any other application name in the search area, and click on it. Note that you do not enter the ” characters around the word when you search for it.
2. It is expected that you have the ElectrumSV dmg file you get this error with in your Downloads directory. You can use the “cd” command to change your directory to get there, using `cd Downloads`.
3. You need to type something close to `xattr -rd com.apple.quarantine <filename>`. However, you need to replace “<filename>” with the filename of the ElectrumSV dmg file you are getting this error with. If for instance you downloaded “ElectrumSV-1.4.0b1.dmg”, then you would need to execute the command `xattr -rd com.apple.quarantine ElectrumSV-1.4.0b1.dmg`.

What this does is it removes the flag Apple put on the file when you downloaded it, to indicate it was not safe. You should be now be able to run it, having applied the workaround.

Startup takes a long time

When you run the dmg and then click on the ElectrumSV logo to start it, does it take a long long time to start? This was never that fast, but it has become slower as we started including the blockchain headers in our application in order to provide a higher quality experience and to prepare for the coming of a new technology called SPV.

Workaround

Install the application and run it as an installed application, rather than launching it from the dmg. This should reduce the time considerably that it takes from when you start the installed application to when you see the first window it opens.

1. Open the dmg file.
2. Observe there is an ElectrumSV icon on the left hand side, and a folder on the right hand side with an arrow pointing from the icon to the folder.
3. Drag the icon into the folder.

ElectrumSV should now be installed and you should be able to use Launchpad to start it or whatever you prefer to do to get applications you have installed to run.

2.4 Using older versions

The ElectrumSV developers only support the latest officially released version and do not provide support for older versions. You can continue to use older versions without our support, but they may stop working unless you are willing to invest a lot of time and effort. Do not contact us about this. We are more than willing to offer support if you upgrade to the latest officially released version and observe the same problems there.

2.4.1 You need a server

Versions of ElectrumSV before 1.4.0 use an open source server project called ElectrumX. Most users do not even pay much attention to the fact that generous community members and businesses run these servers for them to use for free. While a blockchain is small and rarely used, this was feasible. But as blocks get larger and larger, it will become impractical and expensive for people to casually run these servers.

No-one will be running these ElectrumX servers and your wallet will not be able to:

- Detect any incoming or outgoing payments.
- Broadcast any transactions.
- Do pretty much anything other than work as if it were offline.

If you want the wallet to work you will need to run your own ElectrumX server, or pay someone else to run the server for you. This will require additional programming work to deal with all the indexing needs and the indexed blockchain data, as the blocks get bigger and bigger and the blockchain data accrues.

2.4.2 Your responsibility

If you do not upgrade to the latest version of ElectrumSV, then you are responsible for any problems you have. You are responsible for the time, effort and costs in running the required servers when the existing ones finally go down.

CHAPTER

THREE

USING ELECTRUMSV

Where are the wallet files stored?

When the wallet application is run it will store and look for related files like blockchain headers, wallet settings and wallets in either a standard folder location on the user's computer, or in a custom location they have explicitly specified. Read more about *data directories*.

3.1 Data directories

Wallet data is stored in what is called the ElectrumSV data directory. When the application starts up, it looks for the data directory in an expected location, and creates it if it does not already exist.

3.1.1 Standard locations

Unless the user overrides the default behaviour, the data directory will be on a standard location.

On Linux and MacOS, this will be what is called a hidden folder (it starts with the dot character ".") in the user's home directory always named `.electrum-sv`. This should be easy to find.

Listing 1: Linux / MacOS

```
$ ls -a ~/ | grep elec
.electrum-sv
```

On Windows, this will be within the user's application data directories. We currently store most of the application data in the `Roaming` directory and the logs in the `Local` directory. As log files may become quite large in the more verbose debugging levels, these are placed where they won't be synchronised between computers. This is a little harder to find, but by substituting your user name for "Bob" below you should be able to find it.

Listing 2: Windows

```
C:\Users\Bob>dir AppData\Roaming\Elec*
...
 Directory of C:\Users\Bob\AppData\Roaming

28/10/2022  08:54 AM    <DIR>          ElectrumSV


C:\Users\Bob>dir AppData\Local\Elec*
...
 Directory of C:\Users\Bob\AppData\Local
```

(continues on next page)

(continued from previous page)

```
25/10/2022  06:47 AM    <DIR>          ElectrumSV
```

3.1.2 Custom locations

The simplest way to control where your ElectrumSV data directory is located is to use the portable download we provide, this creates and uses an `electrum_sv_data` data directory in the same directory as the portable build executable is located in.

It is also possible to run ElectrumSV from either the source code or our non-portable build, and to tell it where to look for and place it's data directory. This can be done with the `-D` command-line parameter.

Listing 3: Linux / MacOS

```
$ ./electrum-sv -D INSTANCE1
2022-11-25 12:59:34,966:INFO:rest-server:REST API started on http://127.0.0.1:9999
...
$ ls | grep INSTANCE
INSTANCE1
```

Listing 4: Windows

```
C:\Data\Git\electrumsv>py electrum-sv -D INSTANCE1
2022-11-25 12:59:34,966:INFO:rest-server:REST API started on http://127.0.0.1:9999
...
C:\Data\Git\electrumsv>dir INSTANCE
...
 Directory of C:\Data\Git\electrumsv

25/11/2022  12:59 PM    <DIR>          INSTANCE1
```

Note: Only one instance of ElectrumSV can be run at a time. The way this is enforced is through the data directory, and as by default an application instance will use the standard location only the first instance will run and any subsequent instance will exit. However, if each subsequent instance is directed to use a custom data directory, they will run at the same time.

CHAPTER

FOUR

BUILDING ON ELECTRUMSV

How is ElectrumSV implemented?
: This section is intended to be a reference for developers, both the existing ElectrumSV developers and any other developers who might wish to work with the source code or get involved in the project. Refer to the *codebase reference*.

How can I access my wallet using the REST API?
: For most users, accessing their wallet with the user interface will be fine. But if you have a minimal amount of development skill the availability of the REST API gives you a lot more flexibility. The REST API allows a variety of actions among them loading multiple wallets, accessing different accounts, obtaining payment destinations or scripts from any of the accounts. Perhaps you want to add your own interface for your wallet or maybe automate how you use it. Read more about the *REST API*.

How can I access my wallet using the node wallet API?
: Read more about the *node wallet API*.

How would I extend ElectrumSV as a customised wallet server?
: The REST API is limited in what it can do by nature. Getting the ElectrumSV development team to add what you want to it, is not guaranteed to happen, may not even be possible and if it was who knows how long it would take. An alternative is to build your own "daemon application" which is a way of extending ElectrumSV from the inside. Read more about *customised wallet servers*.

Do I have to develop against the existing public blockchains?
: ElectrumSV provides a way for developers to do offline or local development. *customised wallet servers*.

4.1 Codebase reference

4.1.1 Blockchain headers

ElectrumSV is intended to be a P2P application. It should not be required to use external servers provided by an ElectrumSV business. The device the user runs ElectrumSV on is where their wallet state is kept and managed. This has repercussions that other wallets that are not P2P are not subject to.

Possible limitations:

- The network the device is connected to may not allow access to the Bitcoin P2P network.
- The network the device is connected to may not allow non-HTTP internet access.
- The network the device is connected to may not allow incoming connections.
- The user may take their device with them to different networks with different limitations.

These influence ElectrumSV's approach to sourcing blockchain headers.

Header sources

The current policy is:

- If a blockchain service provider is used, use it as the authoritative source of a longest chain to ensure that acquired blockchain state is relevant to the any fork the wallet may be following.
- Connect to all known header servers.

Longer term option we think will be employed:

- Limit connections to perhaps ten concurrent header servers.
- If the Bitcoin P2P network is connectable, use it as a possible source of a longest chain, as long as there is no blockchain server. The user should be able to override it in order to rule out invalid malicious chains, and to ensure they are following a valid chain. If the blockchain server is being used as the authoritative header source, the P2P network can be used for comparison purposes.

The Bitcoin P2P network

At this time ElectrumSV does not connect to the P2P network to listen to headers, or to broadcast transactions. In the longer term this will be the default approach, but there are other priorities for now.

HTTP-based Header APIs

Due to the some of the limitations mentioned above, ElectrumSV cannot rely on users reliably having the ability to access the Bitcoin P2P network. It needs to have the ability for users to get headers from remote HTTP-accessed services, either run by trusted parties or the user themselves.

Wallet behaviour

Following a header source

A wallet follows a header source. When it changes header sources, or the header source switches forks to another chain tip, the wallet performs a reorganisation.

If the wallet is relying on the services provided by a blockchain service provider, it must follow the header API provided by that service. That service will be providing data tied to their chosen chain tip, and when it switches chain tips (reorgs) providing data tied to the new chain tip.

Observing other header sources

If a wallet is following the longest known chain from the P2P network this is straightforward. If a wallet is following a blockchain service provider then it has to follow that provider no matter what to ensure the data it obtains from the service is relevant.

The header state of any other service provider can be observed and used to prompt the user to switch service providers, if it looks like their current one is ill-managed or broken and not fixed in a prompt manner.

The header state of the P2P network can also be observed and used to evaluate the reliability of service providers.

4.1.2 Remote service usage

As part of providing it's user with the information they need, ElectrumSV has to make use of remote servers. The following types of server provide different functionality that use is made of.

Server types

ElectrumSV reference server

These are standard APIs that ElectrumSV expects to be able to use if it is to provide the user with updated wallet state and to react to external events. A reference implementation is provided by the ElectrumSV developers, although it does not provide the more advanced APIs that are related to processing new and old blocks.

Headers API

As an SPV client, ElectrumSV needs up-to-date headers from the blockchain. This API is used to both sychronise existing headers and get notified of new headers as the server makes them available.

The ideal way of obtaining headers is by having an application access the P2P network. However, ElectrumSV cannot guarantee that it has unrestricted access to the network. For this reason we do not use the P2P network, but instead rely on this external REST API. It is possible that in the longer term we will also support direct P2P network access for this, and fallback to the service API if it not usable.

Peer channel API

In the same way that ElectrumSV cannot rely on accessing the P2P network to get headers due to restrictive networking environments, it also cannot and should not rely on being able to connect directly to another person's ElectrumSV application. This is due to restrictive networking environments, privacy and security related concerns and the impracticality of expecting ElectrumSV to be online 24/7. The peer channel API acts as a relaying service, a message box and a privacy aid.

Output spends API (forwarded)

Note: The ElectrumSV project has neither the manpower or the interest in providing a non-regtest open source implementation of this API. Commercial services can if they choose, offer it, as an optional enabled API provided by their servers.

It is useful to be able to ask whether a UTXO is spent. If it is spent then we need a way to find out what transaction it was spent in and what block it was mined in, if not in the mempool. We need to both query the state of outputs, and to register for notifications if there are any changes.

ElectrumSV uses this API to accomplish the following:

- Identify if a transaction is in the mempool.
- Identify if a transaction is mined in a block.
- Identify if a transaction has been malleated.
- Get notified if a transaction gets broadcast.
- Get notified if a transaction gets mined.

- Get notified if a transaction gets malleated.

As we rely on output spends for events related to transactions we did not broadcast ourselves, we just use output spends consistently for the transactions we do as well. We do not use the merchant API for MAPI callbacks.

Restoration API (forwarded)

Note: The ElectrumSV project has neither the manpower or the interest in providing a non-regtest open source implementation of this API. Commercial services can if they choose, offer it, as an optional enabled API provided by their servers.

In the past there was an expectation that seed words could be used to locate all transactions ever associated with a wallet, and that this could be used as both a form of backup and a way to keep a wallet synchronised. As the blockchain becomes larger and larger, there is no-one planning to support this, and if they did it would likely become more and more expensive as the blockchain continue to grow.

The highest priority of the ElectrumSV project is to allow users to retain access to coins where they have been able to access them in the past. One aspect of this is seed-based wallet restoration and we continue to support this by allowing access to blockchain state and indexes that only go up to a fixed height. Past this height, it is expected that users will have to take responsibility for their own wallet data.

The restoration API is effectively a search engine over this limited earlier part of the blockchain, that our users can use to do a limited restore of their seed words.

Transaction API (forwarded)

Note: The ElectrumSV project has neither the manpower or the interest in providing a non-regtest open source implementation of this API. Commercial services can if they choose, offer it, as an optional enabled API provided by their servers.

As the blockchain grows larger and larger, storage and access to arbitrary transaction data becomes a specialised service that will likely require charging for access to data. The transaction API is provided for requesting arbitrary transactions.

Merkle proof API (forwarded)

Note: The ElectrumSV project has neither the manpower or the interest in providing a non-regtest open source implementation of this API. Commercial services can if they choose, offer it, as an optional enabled API provided by their servers.

While we currently broadcast all transactions through a merchant API server, and a callback from that server can notify if the transaction is mined and provide the merkle proof. Not all transactions related to an ElectrumSV user's wallet are broadcast by that user through MAPI. Other parties may broadcast the transaction, and this may not be desirable, expected or have any channel through which the ElectrumSV user can hear about the merkle proofs availability.

It is necessary to have the ability to request arbitrary merkle proofs, and the merkle proof API is used for this purpose. We use it for both the broadcaster who uses MAPI and for the party who has received an event notifying a transaction of interest was included in a block.

Merchant API

The purpose of the merchant API is so that miners can offer a way to both broadcast a transaction and know the fee that must be used in any transaction in order for it to be accepted for broadcast.

Simple indexer API (regtest only)

In order to both develop and test ElectrumSV on the regtest network, we need a simple implementation of the following APIs:

- Merkle proof API.
- Restoration API.
- Transaction API.
- Spent output API.

The simple indexer is a very limited implementation of these APIs that can run against the regtest network. It will never run on any network other than regtest. The amount of work required to make it performant is something a commercial business would have to do, and a commercial business would be required to run and keep that production service going into the future.

Relevant wallet events

There are a small number of wallet events that make use of these remote services.

Loading a wallet

Spent outputs:

- We monitor local transactions that we may not expect to be broadcast. These have the *transaction state* values of `STATE_SIGNED`, `STATE_RECEIVED` or `STATE_DISPATCHED`.
- We monitor transactions that we know have been broadcast but we do not know if they are mined. These have the *transaction state* value of `STATE_CLEARED` with no associated block.

Merkle proofs:

- If we have transactions that have been broadcast and mined, but which we have not obtained the merkle proof for, we pass them in a worker task that will take care of this. These will likely be transactions we received spent output state for, which we have not had the chance to process yet. These have the *transaction state* value of `STATE_CLEARED` with an associated block.
- If we have transactions from before ElectrumSV 1.4.0 they may not have a TSC standardised merkle proof. We pass these to a worker task to take care of obtaining them. These have the *transaction state* value of `STATE_SETTLED` with no associated block. Wallets that were created in ElectrumSV and not earlier versions of the Electrum wallet will most likely have non-TSC proofs which will get converted to the TSC form as part of the wallet file upgrade.

Account restoration

Restoration:

- The restoration process attempts to enumerate known key usage within different script types and locate them in the remote restoration index. It gets metadata about the transactions that use these keys back.

Merkle proof:

- Transactions are fetched through the merkle proof API, taking advantage of the TSC standard providing the ability for transaction data to be wrapped in the merkle proof.

New payment

There are two steps, signing a transaction and broadcasting it. The user can construct a transaction and sign it, then it gets added to their account history. In this case, we want to treat it as a local transaction and monitor it using the spent output API. If it gets signed and then broadcast, we would also monitor it using the spent output API.

Spent outputs:

- We monitor the transaction for a nominal period of time after it is signed. If it is not broadcast we hand it off to the spent output notifications worker task to monitor.

Header sourcing

In order to obtain the latest headers, ElectrumSV connects to several servers offering header APIs. The goal should be to have reliable access to header sources, and to be able to identify servers that do not run reliably.

Headers:

- A web socket is opened to a minimum number of header servers on behalf of the whole application, not any given wallet. The server notifies ElectrumSV of their chain state, and then publishes notifications of new headers. ElectrumSV is expected to reconcile and obtain a copy of the server's main chain and factor it into whether it should be our main chain.

Arbitrary logic should never fetch headers, the tasks that track headers on different servers should be the sole method through which headers are fetched. As new chain tips are obtained, other logic that may be waiting on them, should be notified.

MAPI broadcast

These are the various ways that wallet transactions might be broadcast:

- Existing transactions in the account history list. The user will likely either use the context menu option, or view their transaction dialog and click on the *Broadcast* button. These will have one of the *transaction state* values of `STATE_RECEIVED`, `STATE_DISPATCHED` or `STATE_SIGNED`.
- The payment the user has just entered and opted to send. This can be done in two slight variations. The first is that the user just sends or broadcasts the transaction and they have to perform the signing approval as part of this. The second is that they explicitly sign the transaction and then broadcast it. This adds a period of uncertainty between when the transaction is added to the database in *transaction state* of `STATE_SIGNED` and when it is successfully broadcast via MAPI and is changed to the state `STATE_CLEARED`.
- Background petty cash payments by the wallet that the user might not even know are happening and will not be involved in approving. These may not even be broadcast via MAPI, and the service being paid might take care of the broadcast and merkle proof delivery.

Reliable server usage

The core requirement of ElectrumSV relying on servers for remote state is that we do our best to handle all the reasonable problems in a way where the user is either unaware or presented with minimal complication because of them. Unreasonable problems however, we do not need to be so concerned about. We can try and take measures to prevent the user shooting themselves in the foot but if they do things that are unsafe they may have to pay a third party service to recover their wallet data.

The general approach

There are four possible stages in using a service:

1. Poll for the state of any existing service usage and verify that the state on the service matches the state the wallet has.
2. Establish a web socket connection.
3. Register any per-connection service usage related to that web socket connection.
4. Make on-going requests.

Whether there is a successfully connected web socket or no web socket we will make ongoing requests:

- Requesting data or current state.
- Request short-term notifications for events for the life of the current connection.
- Requesting long-term notifications for events delivered via peer channels.

Handling problems establishing a connection

There are three potential problems that could be encountered when polling for state on the service or establishing a web socket connection:

1. Inability to establish a connection.
2. Unexpected result when accessing an API endpoint.
3. The received state does not match the state the wallet has.

The first failure case when establishing a connection should cancel the whole process and any other concurrently made API calls, and display a “server connection problems” UI to the user. It is not required that the user already has a modal dialog showing connection progress but if they do the problems should be incorporated into that existing UI. If they do not, then a UI should be shown for that purpose.

The second failure case should render the server unusable. The server should be flagged as broken, and the user should be informed of the problem and given options to deal with it.

The third failure case should attempt to reconcile the state. It might be that this is expected because the user has not used the service for a long time, and any prepayment they made to engage long term services like tip filter registrations and peer channels has lapsed. The user would need to go to the payer and obtain the transaction they were watching for, pay extra for any merkle proofs and so on. However it might also be because the user has been operating the server with multiple copies of the wallet which will inevitably confuse one or more of those copies. This will be covered elsewhere.

Handling problems making API requests

An API endpoint should be expected to just work. There are two potential problems:

1. Inability to establish a connection.
2. Unexpected result when accessing an API endpoint.

These can be handled the same as suggested in the establishing a connection section. It may be that the API usage is not done with a specific server, and that it is possible for ElectrumSV to just handle it without bothering the user by switching to another server behind the scenes. If the API usage is with a specific server, then this is problematic and will involve notifying the user and having them explicitly make a choice.

Special case HTTP response status codes

When a request to an API endpoint receives one of these status codes, it is not an unexpected result. It is also not ideal and we should have some standard way of handling them.

401 - UNAUTHORIZED

Use of the given API endpoint requires authentication and the client has not provided that authentication. This response should only be encountered if for some reason the current authentication token is no longer valid and it needs to be renewed.

402 - PAYMENT_REQUIRED

Use of the given API endpoint requires payment and there is insufficient funding remaining to cover the requested server activity. ElectrumSV should automatically fund the channel from the relevant petty cash account, limiting server usage until this is done. It may require user notification or intervention. The implementation should be expected to try and predict this ahead of time and prevent the user doing actions that may be problematic if there is insufficient funding to complete them.

429 - TOO_MANY_REQUESTS

If a server implements a free quota it should return this response when the quota is used up. Ideally ElectrumSV will have some idea of how large the quota is, what it permits and how close it is to being used up. It can then inform the user if desired actions are not possible because the quota is spent.

Indexer services

Most of the indexer services are stateless, and there are not many things that need to be checked for consistency as part of the connection process.

Consistency actions:

- List the tip filter registrations.

Tip filter registrations

Possible problems:

- A tip filter no longer exists due to lack of funding.
- A tip filter no longer exists with no identifiable reason.

Lack of funding

This should not happen as the initial use of this service will be for a specific purpose with a known time limit. The user will be creating a receving address or script to give out to the payer, and wanting to know when a transaction featuring that payment destination is broadcast. ElectrumSV can prepay for that period guaranteeing that any reliable service will monitor for the transaction for the expected amount of time. It can also default to a time period that reflects the longest any reliable payer should have broadcast by and warn the user if they choose a shorter time of the risks.

In the event that the payer does not send in a timely fashion and the payment is not detected this is problematic in theory, but not in practice. In theory ElectrumSV then needs to pay for a costly scan of the blocks that have been mined since they gave out the payment destination. In practice existing businesses that pay this way already show the transaction id in the user's account and the user can use that to cheaply manually instruct ElectrumSV to obtain the transaction.

ElectrumSV should likely do the following:

- Ensure a tip filter is put in place (unless the user has opted not to).
- If a tip filter is put in place ensure the user accepts the expiry time as the latest time the payment transaction will be detected.
- If a payment can be made with no tip filter or after that expiry time that they know whether they can obtain the transaction id directly from the payer.

No identifiable reason

A tip filter no longer being present when it should be would likely only be possible because of a server error or the ElectrumSV user doing things they shouldn't. The user who causes this problem will likely have done something like open two copies of the wallet or an outdated copy of the wallet.

A peer channel no longer being present could be because of a server error. Any service where this happens likely has more widespread problems and will gain a reputation of being unreliable. If it is a reliable service it has a vested interest in scanning recent blocks and detecting missed filter matches, in order to make it right. And it should likely do this proactively as soon as it detects the problem.

An ElectrumSV user may be able to do things that the wallet does not support, which could result in this happening. A possible example is where they open a backup of the wallet file and it has no way of recovering what was missing or even knowing what was missing, and corrupts service usage of the up-to-date version of the wallet. ElectrumSV should do everything it can to detect and disallow this from happening, but it might be that the user chooses to proceed anyway or they find a new way to cause this problem.

Peer channel hosting

Peer channels are a stateful service. The user needs the channels they have created to be alive long enough for the channel to receive any incoming message and for ElectrumSV to identify the presence of that message and fetch it.

Consistency actions:

- List the peer channels on the server.

Peer channel existence

Possible problems:

- A channel no longer exists due to lack of funding.
- A channel no longer exists with no identifiable reason.

Lack of funding

This should not happen unless the user does not open their wallet for a prolonged period of time. If the channel hosting service is professional, then the user should also be able to register for out of band notifications perhaps to their email address in event of low funding.

ElectrumSV should clearly illustrate to the user that there are time limits to when they need to revisit their service usage and top up payments.

No identifiable reason

A peer channel no longer being present when it should be is a server error. The server cannot recover messages it never received nor should it receive messages it thinks it does not expect.

An ElectrumSV user may be able to do things that the wallet does not support, which could result in this happening. A possible example is where they open a backup of the wallet file and it has no way of recovering what was missing or even knowing what was missing, and corrupts service usage of the up-to-date version of the wallet. ElectrumSV should do everything it can to detect and disallow this from happening, but it might be that the user chooses to proceed anyway or they find a new way to cause this problem.

4.1.3 The wallet database

Each wallet is stored in it's own SQLite database. The current version of ElectrumSV at the time of writing, 1.4.0b1, uses the database schema version 29. This schema is include for reference purposes and cannot be used to a create a working wallet.

Each version of ElectrumSV includes migration code that applies any needed changes to older versions of the wallet. This database format is pretty solid at this point, but it is a work in progress. There are many other things ElectrumSV will need to support in the future.

Transactions and atomicity

Between how SQLite works, how the Python sqlite3 module works and how ElectrumSV builds upon both of these some elaboration is needed.

We pass the `isolation_level=None` parameter to the Python sqlite3 function that opens a database connection. This overrides the custom way the Python sqlite3 module overrides how SQLite works and returns it to the autocommit mode. This mode means that statements that modify the database take effect immediately. Use of a `BEGIN` and `SAVEPOINT` statement takes Sqlite out of autocommit mode, and the outermost `COMMIT`, `ROLLBACK` or `RELEASE` statement returns Sqlite to autocommit mode.

ElectrumSV does all of it's database writes in custom transactions starting with the `BEGIN` statement, disabling the autocommit mode, and bundling the writes into groups with the ability to commit them all or roll them all back. Additionally as SQLite does not allow multiple connections to do concurrent writes, ElectrumSV takes a well known approach of having a sequential writer thread. All writes happen in a dedicated writer thread one after the other as managed transactions.

The following logic is used to wrap each ElectrumSV transaction:

```
def __call__(self, db: sqlite3.Connection) -> None:
    if not self._future.set_running_or_notify_cancel():
        return

    db.execute("BEGIN")
    try:
        result = self._fn(db, *self._args, **self._kwargs)
    except BaseException as exc:
        db.execute("ROLLBACK")
        self._future.set_exception(exc)
        # Break a reference cycle with the exception 'exc'
        self = None # type: ignore
    else:
        db.execute("COMMIT")
        self._future.set_result(result)
```

Synchronous writes

The Python `concurrent.futures` module is used in synchronous logic to do database writes in a non-blocking manner. The calling thread can block until the write is complete by calling the Future.result method. Or the calling thread can request a callback through the use of the Future.add_done_callback.

Caution: Futures catch and log exceptions in their callbacks, preventing ElectrumSV from catching and reporting them. This means that the Future callbacks need to be certain they know about all possible exceptions and to catch and handle them all. Developers should be very sure they understand the code they are calling.

Synchronous database calls are performed in this manner:

```
def on_db_call_done(future: concurrent.futures.Future[bool]) -> None:
    # Skip if the operation was cancelled.
    if future.cancelled():
        return
    # Raise any exception if it errored or get the result if completed successfully.
```

(continues on next page)

(continued from previous page)

```
    future.result()
    self.events.trigger_callback(WalletEvent.TRANSACTION_DELETED, self._id, tx_hash)

future = db_functions.remove_transaction(self.get_db_context(), tx_hash)
future.add_done_callback(on_db_call_done)
```

Asynchronous writes

How ElectrumSV wraps asynchronous calls is done in the `DatabaseContext.run_in_thread_async` method. If you wish to see how it works, you can look in the `sqlite_support.py` file.

Asynchronous database calls are performed in this manner:

```
if await update_transaction_flags_async(db_context, [
        (TxFlags.MASK_STATELESS, TxFlags.STATE_SETTLED, tx_hash) ]):
    ...
```

Database schema

This is version 29 of our database schema. It should be correct for the ElectrumSV version this documentation is intended for, but if it is not, please let us know.

```
BEGIN TRANSACTION;

CREATE TABLE IF NOT EXISTS "MasterKeys" (
    "masterkey_id" INTEGER,
    "parent_masterkey_id" INTEGER DEFAULT NULL,
    "derivation_type" INTEGER NOT NULL,
    "derivation_data" BLOB NOT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    "flags" INTEGER NOT NULL DEFAULT 0,
    FOREIGN KEY("parent_masterkey_id") REFERENCES "MasterKeys"("masterkey_id"),
    PRIMARY KEY("masterkey_id")
);

CREATE TABLE IF NOT EXISTS "Accounts" (
    "account_id" INTEGER,
    "default_masterkey_id" INTEGER DEFAULT NULL,
    "default_script_type" INTEGER NOT NULL,
    "account_name" TEXT NOT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    "flags" INTEGER NOT NULL DEFAULT 0,
    FOREIGN KEY("default_masterkey_id") REFERENCES "MasterKeys"("masterkey_id"),
    PRIMARY KEY("account_id")
);

CREATE TABLE IF NOT EXISTS "Transactions" (
    "tx_hash" BLOB,
```

(continues on next page)

(continued from previous page)

```
    "tx_data" BLOB DEFAULT NULL,
    "proof_data" BLOB DEFAULT NULL,
    "block_height" INTEGER DEFAULT NULL,
    "block_position" INTEGER DEFAULT NULL,
    "fee_value" INTEGER DEFAULT NULL,
    "flags" INTEGER NOT NULL DEFAULT 0,
    "description" TEXT DEFAULT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    "locktime" INTEGER DEFAULT NULL,
    "version" INTEGER DEFAULT NULL,
    "block_hash" BLOB DEFAULT NULL,
    PRIMARY KEY("tx_hash")
);

CREATE TABLE IF NOT EXISTS "WalletData" (
    "key" TEXT NOT NULL,
    "value" TEXT NOT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL
);

CREATE TABLE IF NOT EXISTS "WalletEvents" (
    "event_id" INTEGER,
    "event_type" INTEGER NOT NULL,
    "event_flags" INTEGER NOT NULL,
    "account_id" INTEGER,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("account_id") REFERENCES "Accounts"("account_id"),
    PRIMARY KEY("event_id")
);

CREATE TABLE IF NOT EXISTS "Invoices" (
    "invoice_id" INTEGER,
    "account_id" INTEGER NOT NULL,
    "tx_hash" BLOB DEFAULT NULL,
    "payment_uri" TEXT NOT NULL,
    "description" TEXT,
    "invoice_flags" INTEGER NOT NULL,
    "value" INTEGER NOT NULL,
    "invoice_data" BLOB NOT NULL,
    "date_expires" INTEGER DEFAULT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("tx_hash") REFERENCES "Transactions"("tx_hash"),
    FOREIGN KEY("account_id") REFERENCES "Accounts"("account_id"),
    PRIMARY KEY("invoice_id")
);

CREATE TABLE IF NOT EXISTS "AccountTransactions" (
    "tx_hash" BLOB NOT NULL,
```

(continues on next page)

(continued from previous page)

```
    "account_id" INTEGER NOT NULL,
    "flags" INTEGER NOT NULL DEFAULT 0,
    "description" TEXT DEFAULT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("account_id") REFERENCES "Accounts"("account_id"),
    FOREIGN KEY("tx_hash") REFERENCES "Transactions"("tx_hash")
);

CREATE TABLE IF NOT EXISTS "TransactionOutputs" (
    "tx_hash" BLOB NOT NULL,
    "txo_index" INTEGER NOT NULL,
    "value" INTEGER NOT NULL,
    "keyinstance_id" INTEGER DEFAULT NULL,
    "flags" INTEGER NOT NULL,
    "script_type" INTEGER DEFAULT 0,
    "script_hash" BLOB NOT NULL DEFAULT x '',
    "script_offset" INTEGER DEFAULT 0,
    "script_length" INTEGER DEFAULT 0,
    "spending_tx_hash" BLOB,
    "spending_txi_index" INTEGER,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("keyinstance_id") REFERENCES "KeyInstances"("keyinstance_id"),
    FOREIGN KEY("tx_hash") REFERENCES "Transactions"("tx_hash")
);

CREATE TABLE IF NOT EXISTS "TransactionInputs" (
    "tx_hash" BLOB NOT NULL,
    "txi_index" INTEGER NOT NULL,
    "spent_tx_hash" BLOB NOT NULL,
    "spent_txo_index" INTEGER NOT NULL,
    "sequence" INTEGER NOT NULL,
    "flags" INTEGER NOT NULL,
    "script_offset" INTEGER,
    "script_length" INTEGER,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("tx_hash") REFERENCES "Transactions"("tx_hash")
);

CREATE TABLE IF NOT EXISTS "KeyInstances" (
    "keyinstance_id" INTEGER,
    "account_id" INTEGER NOT NULL,
    "masterkey_id" INTEGER DEFAULT NULL,
    "derivation_type" INTEGER NOT NULL,
    "derivation_data" BLOB NOT NULL,
    "derivation_data2" BLOB DEFAULT NULL,
    "flags" INTEGER NOT NULL,
    "description" TEXT DEFAULT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
```

(continues on next page)

(continued from previous page)

```
    FOREIGN KEY("masterkey_id") REFERENCES "MasterKeys"("masterkey_id"),
    FOREIGN KEY("account_id") REFERENCES "Accounts"("account_id"),
    PRIMARY KEY("keyinstance_id")
);

CREATE TABLE IF NOT EXISTS "KeyInstanceScripts" (
    "keyinstance_id" INTEGER NOT NULL,
    "script_type" INTEGER NOT NULL,
    "script_hash" BLOB NOT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("keyinstance_id") REFERENCES "KeyInstances"("keyinstance_id")
);

CREATE TABLE IF NOT EXISTS "MAPIBroadcastCallbacks" (
    "tx_hash" BLOB,
    "peer_channel_id" VARCHAR(1024) NOT NULL,
    "broadcast_date" INTEGER NOT NULL,
    "encrypted_private_key" BLOB NOT NULL,
    "server_id" INTEGER NOT NULL,
    "status_flags" INTEGER NOT NULL,
    PRIMARY KEY("tx_hash")
);

CREATE TABLE IF NOT EXISTS "Servers" (
    "server_id" INTEGER,
    "server_type" INTEGER NOT NULL,
    "url" TEXT NOT NULL,
    "account_id" INTEGER DEFAULT NULL,
    "server_flags" INTEGER NOT NULL DEFAULT 0,
    "api_key_template" TEXT DEFAULT NULL,
    "encrypted_api_key" TEXT DEFAULT NULL,
    "fee_quote_json" TEXT DEFAULT NULL,
    "date_last_connected" INTEGER DEFAULT 0,
    "date_last_tried" INTEGER DEFAULT 0,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
    FOREIGN KEY("account_id") REFERENCES "Accounts"("account_id"),
    PRIMARY KEY("server_id")
);

CREATE TABLE IF NOT EXISTS "PaymentRequests" (
    "paymentrequest_id" INTEGER,
    "keyinstance_id" INTEGER NOT NULL,
    "state" INTEGER NOT NULL,
    "description" TEXT,
    "expiration" INTEGER,
    "value" INTEGER,
    "script_type" INTEGER NOT NULL,
    "pushdata_hash" BLOB NOT NULL,
    "date_created" INTEGER NOT NULL,
    "date_updated" INTEGER NOT NULL,
```

(continues on next page)

(continued from previous page)

```
    FOREIGN KEY("keyinstance_id") REFERENCES "KeyInstances"("keyinstance_id"),
    PRIMARY KEY("paymentrequest_id")
);

CREATE UNIQUE INDEX IF NOT EXISTS "idx_WalletData_unique" ON "WalletData" ("key");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_Invoices_unique" ON "Invoices" ("payment_uri");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_AccountTransactions_unique" ON
↪"AccountTransactions" ("tx_hash", "account_id");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_TransactionOutputs_unique" ON "TransactionOutputs
↪" ("tx_hash", "txo_index");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_TransactionInputs_unique" ON "TransactionInputs" (
↪"tx_hash", "txi_index");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_KeyInstanceScripts_unique" ON "KeyInstanceScripts
↪" ("keyinstance_id", "script_type");

CREATE UNIQUE INDEX IF NOT EXISTS "idx_Servers_unique" ON "Servers" ("server_type", "url
↪", "account_id");

CREATE VIEW TransactionReceivedValues (account_id, tx_hash, keyinstance_id, value) AS
SELECT
    ATX.account_id,
    ATX.tx_hash,
    TXO.keyinstance_id,
    TXO.value
FROM
    AccountTransactions ATX
    INNER JOIN TransactionOutputs TXO ON TXO.tx_hash = ATX.tx_hash
    INNER JOIN KeyInstances KI ON KI.keyinstance_id = TXO.keyinstance_id
WHERE
    TXO.keyinstance_id IS NOT NULL
    AND KI.account_id = ATX.account_id;

CREATE VIEW TransactionSpentValues (account_id, tx_hash, keyinstance_id, value) AS
SELECT
    ATX.account_id,
    ATX.tx_hash,
    PTXO.keyinstance_id,
    PTXO.value
FROM
    AccountTransactions ATX
    INNER JOIN TransactionInputs TXI ON TXI.tx_hash = ATX.tx_hash
    INNER JOIN TransactionOutputs PTXO ON PTXO.tx_hash = TXI.spent_tx_hash
    AND PTXO.txo_index = TXI.spent_txo_index
    INNER JOIN KeyInstances KI ON KI.keyinstance_id = PTXO.keyinstance_id
WHERE
    PTXO.keyinstance_id IS NOT NULL
    AND KI.account_id = ATX.account_id;
```

(continues on next page)

(continued from previous page)

```sql

CREATE VIEW TransactionValues (account_id, tx_hash, keyinstance_id, value) AS
SELECT
    account_id,
    tx_hash,
    keyinstance_id,
    value
FROM
    TransactionReceivedValues
UNION
ALL
SELECT
    account_id,
    tx_hash,
    keyinstance_id,
    - value
FROM
    TransactionSpentValues;

COMMIT;
```

Details

For now various details about the database schema are kept below, but as we flesh it out it should end up being restructured.

Transaction table

block_hash

This column stores the block hash for the block the transaction was mined in. It is expected there is a matching row to the transaction hash and block hash in the *TransactionProofs* table.

block_height / block_position

These columns are intended to track the block height and block position of a transaction, once it has been mined, into the long term future. In theory, it is possible to map the block hash and transaction hash to the *TransactionProofs* table obtain this information. In practice, there are two reasons we may not want to do this.

- We may want to delete proofs for transactions once coins have been spent.
- We may not have the proof for older transactions, which have unspent coins from before proofs were retained. If these are unspent and present in migrated wallets, we will need to obtain the proofs to do an SPV payment.

flags

STATE_SIGNED

A fully signed transaction that is expected to not have been shared with external parties.

STATE_DISPATCHED

A transaction that has been shared with external parties, but is not expected to have been broadcast to the P2P network.

If this is determined to have been broadcast, then additional as yet implemented handling should be done to reconcile how to react to this event.

STATE_RECEIVED

A transaction that an external party has shared, but is not expected to have been broadcast to the P2P network.

If this is determined to have been broadcast, then additional as yet implemented handling should be done to reconcile how to react to this event.

STATE_CLEARED

A cleared transaction is one that is known to have been broadcast to the P2P network.

Nuances:

- *block_hash* will be *NULL* for transactions that have been broadcast. *block_position* and *proof_data* will also be *NULL*.
- *block_hash* will be non-*NULL* to represent knowledge that it has been mined (via non-MAPI channels) and that we should fetch a merkle proof to verify it is in a given block. *block_position* and *proof_data* will be *NULL*.
- *block_hash*, *block_position* and *proof_data* will all have valid unprocessed values if the application headers do not include the given block height yet.

STATE_SETTLED

A settled transaction is one that is known to have been mined in a block, and has been verified as being in that block through checking the merkle proof.

Nuances:

- ***block_hash* will *NULL* for transactions that the legacy ElectrumX proof data was not retained**
  for. These will need to be obtained, if they contain unspent coins (UTXOs).
- *block_hash* will be non non-*NULL* for transactions that have been mined and which we have the proof for, *block_height* will be the height of the block with the given hash and *block_position* will be the index of the transaction in the block. There will be a proof row in the *TransactionProofs* table mapped to the *tx_hash* and *block_hash* columns of the *Transactions* table.

This section aims to provide an overview of how ElectrumSV works. This is useful for us, the ElectrumSV developers to keep track how things are supposed to work and for programmers who wish to get involved in the project.

Blockchain headers
: ElectrumSV is a P2P wallet and is limited by the network access the user running it has. While in the longer term we will obtain headers from the Bitcoin P2P network if possible, in the shorter term we need to focus on consistently reliable ways of obtaining headers. At this time headers are obtained via a standard REST API from remote servers. Read more about *how ElectrumSV obtains and uses headers*.

Remote service usage
: In order to obtain information about the blockchain and the transactions in a wallet, ElectrumSV has to make use of remote services. These provide things like the ability to broadcast transactions, obtain merkle proofs, obtain transactions and hear about relevant changes and data in the blockchain. Read more about what services ElectrumSV uses and *how it uses them*.

The wallet database

Each wallet loaded by ElectrumSV is separate and the data is accordingly stored in a separate wallet file. Read more about the structure and usage of the data in *the wallet database*.

4.2 Local or offline development

A command-line based environment is provided for local Bitcoin SV development. There is no requirement to be online while using it (after the components are installed).

More details about the SDK can be found here: https://electrumsv-sdk.readthedocs.io/

Essentially the SDK allows a developer to run:

- A RegTest Bitcoin node
- A RegTest ElectrumX instance (which is the currently used chain indexing service).
- A RegTest ElectrumSV wallet server with REST API or alternatively in GUI mode.
- The Merchant API which runs alongside the Bitcoin node and will be used for transaction broadcasting and merkle proofs.
- A mock service that acts as an intermediate agent for payment requests, invoices and other SPV / p2p functionalities.

With these processes running it allows for a faster development iteration cycle and the ability to test the correctness of any new changes. This is particularly so for things like processing of confirmed transactions and reorgs - which is not feasible on public testnets (e.g. waiting for a new block to be mined or for a reorg to happen).

A useful workflow for debugging can be to set a "pdb" break point and run the functional tests which then enters the debugger interactive terminal.

For details about formal, automated functional testing, please see the sections:

- functional tests
- stresstesting

The SDK also makes it trivial to reset all services back to a "clean slate" and to perform deterministic (i.e. repeatable) testing - especially for simulated reorgs.

4.3 The REST API

Technically, the restapi is an example 'dapp' (daemon application). But is nevertheless provided in a format that aims to eventually cover the majority of basic use cases.

This RESTAPI may be subject to slight changes but the example dapp source code is there for users to modify to suit your own specific needs.

Possible network options for `{network}`

- `mainnet`
- `testnet`
- `scalingtestnet`
- `regtest`

4.3.1 Endpoints

get_all_wallets

Get a list of all available wallets

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets`

Sample Response

```
{
    "wallets": [
        "worker1.sqlite"
    ]
}
```

get_parent_wallet

Get a high-level information about the parent wallet and accounts (within the parent wallet).

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite`

Sample Response

```
{
    "parent_wallet": "worker1.sqlite",
    "accounts": {
        "1": {
            "wallet_type": "Standard account",
            "default_script_type": "P2PKH",
            "is_wallet_ready": true
        }
    }
}
```

load_wallet

Load the wallet on the daemon (i.e. subscribe to ElectrumX for active keys) and initiate synchronization. Returns a high-level information about the parent wallet and accounts.

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite`

Sample Response

```
{
    "parent_wallet": "worker1.sqlite",
    "accounts": {
        "1": {
            "wallet_type": "Standard account",
            "default_script_type": "P2PKH",
            "is_wallet_ready": true
        }
    }
}
```

get_account

Get high-level information about a given account

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1`

Sample Response

```
{
    "1": {
        "wallet_type": "Standard account",
        "default_script_type": "P2PKH",
        "is_wallet_ready": true
    }
}
```

get_coin_state

Get the count of cleared, settled and matured coins.

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/utxos/coin_state`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/utxos/coin_state`

Sample Response

```
{
    "cleared_coins": 11,
    "settled_coins": 700,
    "unmatured_coins": 0
}
```

get_utxos

Get a list of all utxos.

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/utxos`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/utxos`

Sample Response

```
{
    "utxos": [
        {
            "value": 20000,
            "script_pubkey": "76a91485324d225c81d414fe8a92bf101dba1a59211e8488ac",
            "script_type": 2,
            "tx_hash": "ce7c2fbc25d25d945b4ad539d2b41ead29e1b786a8aa42b2677af28da3f231a0
↪",
            "out_index": 49,
            "keyinstance_id": 13,
            "address": "msfERZdhGaabQmeQ1ks8sHYdCDtxnTfL2z",
            "is_coinbase": false,
```

(continues on next page)

(continued from previous page)

```
            "flags": 0
        },
        {
            "value": 20000,
            "script_pubkey": "76a91488471d45666dadece7f06aca22f1a1cf9a3a534988ac",
            "script_type": 2,
            "tx_hash": "ce7c2fbc25d25d945b4ad539d2b41ead29e1b786a8aa42b2677af28da3f231a0
↪",
            "out_index": 50,
            "keyinstance_id": 12,
            "address": "mswXPFgWJbgvyxkWBFfYjbbaD1DZmFS3ig",
            "is_coinbase": false,
            "flags": 0
        },
    ]
}
```

get_balance

Get account balance (confirmed, unconfirmed, unmatured) in satoshis.

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/balance`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/utxos/balance`

Sample Response

```
{
    "confirmed_balance": 14999694400,
    "unconfirmed_balance": 98000,
    "unmatured_balance": 0
}
```

remove

Removes transactions (currently restricted to 'STATE_SIGNED' transactions.)

Deleting transactions in the 'Dispatched', 'Cleared', 'Settled' states could cause issues with the utxo set and so is not supported at this time (a DisabledFeatureError will be returned). If you require this feature, please make contact via the Atlantis Slack or the MetanetICU slack.

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs`

Sample Body Payload

```
{
    "txids": [
        "96eee07f8e2c96e33d457138496958d912042ff4ed7b3b9c74a2b810fa5c3750",
        "469ddc27b8ef3b386bf7451aebce64edfe22d836ad51076c7a82d78f8b4f4cf9",
        "e81472f9bbf2dc2c7dcc64c1f84b91b6214599d9c79e63be96dcda74dcb8103d"
    ]
}
```

Sample Response

```
{
    "items": [
        {
            "id": "96eee07f8e2c96e33d457138496958d912042ff4ed7b3b9c74a2b810fa5c3750",
            "result": 200
        },
        {
            "id": "469ddc27b8ef3b386bf7451aebce64edfe22d836ad51076c7a82d78f8b4f4cf9",
            "result": 400,
            "description": "DisabledFeatureError: You used this endpoint in a way that␣
↪is not supported for safety reasons. See documentation for details (https://electrumsv.
↪readthedocs.io/ )"
        },
        {
            "id": "e81472f9bbf2dc2c7dcc64c1f84b91b6214599d9c79e63be96dcda74dcb8103d",
            "result": 400,
            "description": "Transaction not found"
        }
    ]
}
```

get_transaction_history

Get transaction history. `tx_flags` can be specified in the request body. This is an enum representing a bitmask for filtering transactions.

The main `TxFlags` are:

STATE_CLEARED
1 << 20 (received over p2p network and is unconfirmed and in the mempool)

STATE_SETTLED
1 << 21 (received over the p2p network and is confirmed in a block)

STATE_RECEIVED
1 << 22 (received from another party and is unknown to the p2p network)

STATE_SIGNED
1 << 23 (not sent or given to anyone else, but are with-holding and consider the inputs it uses allocated)

STATE_DISPATCHED
1 << 24 (a transaction you have given to someone else, and are considering the inputs it uses allocated)

However, there are other flags that can be set. See `electrumsv/constants.py:TxFlags` for details.

In the example below, (1 << 23 | 1 << 21) yields 9437184 (to filter for only STATE_SIGNED and STATE_CLEARED transactions)

An empty request body will return all transaction history for this account. Pagination is not yet implemented.

Request

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/history`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/history`

Sample Body Payload

```
{
    "tx_flags": 9437184
}
```

Sample Response

```
{
    "history": [
        {
            "txid": "64a9564588f9ebcce4ac52f4e0c8fe758b16dfd6fdb5bd8db5920da317aa15c8",
            "height": 0,
            "tx_flags": 1052720,
            "value": -10200
        },
        {
            "txid": "a6ec24243a79de1b51646d1a46ece854a8f682ff23b4d4afabaebc2bc10ef110",
            "height": 0,
            "tx_flags": 1052720,
            "value": -10200
        }
    ]
}
```

fetch_transaction

Get the raw transaction for a given hex txid (as a hex string) - must be a transaction in the wallet's history.

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/fetch`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/fetch`

Sample Request Payload

```
{
    "txid": "d45145f0c2ff87f6cfe5524d46d5ba14932363e927bd5a4af899a9b8fc0ab76f"
}
```

Sample Response

```
{
    "tx_hex":
↪"0100000001e59dd2992ed46911bea87af1b4f7ab1edce8e038520f142d2aa219492664d993160000006b483045022100ec97
↪"
}
```

create_tx

Create a locally signed transaction ready for broadcast. A side effect of this is that the utxos associated with the transaction are allocated for use and so cannot be used in any other transaction.

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/create`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/create`

Sample Request Payload This example is of a single "OP_FALSE OP_RETURN" output with "Hello World" encoded in Hex. The preceeding 0x0b byte represents a pushdata op code to push the next 11 bytes onto the stack ("68656c6c6f20776f726c64").

Additional outputs for leftover change will be created automatically.

```
{
    "outputs": [
        {"script_pubkey":"006a0b68656c6c6f20776f726c64", "value": 0}
```

(continues on next page)

(continued from previous page)

```
    ],
    "password": "test"
}
```

Sample Response

```
{
    "txid": "96eee07f8e2c96e33d457138496958d912042ff4ed7b3b9c74a2b810fa5c3750",
    "rawtx":
↪"0100000001cfdec4ce0f10c4148b44163bf6205f53e5ab31f04a57fcaaeb33ef6487e08511000000006b483045022100873bb
↪"
}
```

broadcast

Broadcast a rawtx (created with the previous endpoint).

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/broadcast`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/broadcast`

Sample Request Payload This example is of a single "OP_FALSE OP_RETURN" output with "Hello World" encoded in Hex. The preceeding 0x0b byte represents a pushdata op code to push the next 11 bytes onto the stack ("68656c6c6f20776f726c64").

Additional outputs for leftover change will be created automatically.

```
{
    "rawtx":
↪"0100000001ab9aff89a92c011b5436a0c02eb53cf6328286e5cf5767f309cde5414f657661000000006a473044022050750e
↪"
}
```

Sample Response

```
{
    "txid": "7ff0fcf6de91ffa71ef145e31d0bffe31467ecaa125a8db307cf9066fea55db5"
}
```

create_and_broadcast

Atomically creates and broadcasts a transaction. If any errors occur, the intermediate step of creating a signed transaction will be reversed (i.e. the transaction will be deleted and the utxos freed for use).

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/create_and_broadcast`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/create_and_broadcast`

Sample Request Payload This example is of a single “OP_FALSE OP_RETURN” output with “Hello World” encoded in Hex. The preceeding 0x0b byte represents a pushdata op code to push the next 11 bytes onto the stack (“68656c6c6f20776f726c64”).

Additional outputs for leftover change will be created automatically.

```
{
    "outputs": [
        {"script_pubkey":"006a0b68656c6c6f20776f726c64", "value": 0}
    ],
    "password": "test"
}
```

Sample Response

```
{
    "txid": "469ddc27b8ef3b386bf7451aebce64edfe22d836ad51076c7a82d78f8b4f4cf9"
}
```

split_utxos

Creates and broadcasts a coin-splitting transaction i.e. it breaks up existing utxos into a specified number of new utxos with the desired “split_value” (satoshis). “split_count” represents the maximum number of splitting outputs for the transaction. “desired_utxo_count” determines when the desired utxo count has been reached (i.e. if you have 200 utxos but “desired_utxo_count” is 220 then the next coin splitting transaction will create 20 more utxos.

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/txs/split_utxos`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/txs/split_utxos`

Sample Request Payload

```
{
    "split_value": 10000,
    "split_count": 100,
    "password": "test",
    "desired_utxo_count": 1000
}
```

Sample Response

```
{
    "txid": "42329848db94cb16379b0c8898eb2b98542fb25d9257a47663c3fac7b0f49938"
}
```

4.3.2 Regtest only endpoints

If you try to access these endpoints when not in RegTest mode you will get back a 404 error because the endpoint will not be available.

generate_blocks

Tops up the RegTest wallet from the RegTest node wallet (new blocks may be generated to facilitate this process).

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/generate_blocks`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/generate_blocks`

Sample Request Payload

```
{
    "nblocks": 3
}
```

Sample Response

```
{
    "txid": [
        "72d1270d0b3ad4c71d8257db8d6f88018610815253465 8ae6a127b616795530d"
    ]
}
```

create_new_wallet

This will create a new wallet - in this example "worker1.sqlite".

Method
POST

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/create_new_wallet`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/create_new_wallet`

Sample Request Payload

```
{
    "password": "test"
}
```

Sample Response

```
{
    "new_wallet": "G:\\electrumsv_official\\electrumsv1\\regtest\\wallets\\worker1.sqlite
↪"
}
```

transaction state websocket

This websocket is for tracking transaction state changes. One main use case might be to wait on the websocket pending transaction confirmation (i.e. 'StateSettled'). But it is not limited to this transaction state.

Supported States:

StateCleared
1 << 20 (received over p2p network and is unconfirmed and in the mempool)

StateSettled
1 << 21 (received over the p2p network and is confirmed in a block)

May be supported later:

StateReceived
1 << 22 (received from another party and is unknown to the p2p network)

Request

Method
GET

Content-Type
application/json

Endpoint
`http://127.0.0.1:9999/v1/{network}/dapp/wallets/{wallet_name}/{account_id}/websocket/text-events`

Regtest example
`http://127.0.0.1:9999/v1/regtest/dapp/wallets/worker1.sqlite/1/websocket/text-events`

Sample Websocket message

```
{
    "txids": ["3c26c76acebffdd614d6a829bc014114803ba650710652d67837718e467a94ab"]
}
```

Sample Response

```
{
    "txid": "3c26c76acebffdd614d6a829bc014114803ba650710652d67837718e467a94ab",
    "tx_flags": 2109552
}
```

Sample Error Response

```
{
    'code': 40000,
    'message': "some error message goes here"
}
```

4.4 The Node wallet API

The Bitcoin SV node bitcoind has provided a JSON-RPC based wallet API. However running a node just to operate a wallet is becoming prohibitive as blocks and the blockchain get larger and larger on Bitcoin SV.

ElectrumSV aims to provide a replacement option to running the node. This is a special mode where ElectrumSV is run as a wallet server without a GUI, that has to be explicitly activated with the required command-line options. Where possible we aim to try and present the same API, errors and experience that the node API does. This replacement option is the only JSON-RPC API that ElectrumSV provides.

Warning: Before integrating this into your stack, check the documentation for each endpoint you plan to use for any incompatibilities or outstanding issues between the ElectrumSV RPC API and the Node RPC API. As much as possible, the APIs are equivalent but there are some unavoidable differences. A prime example is the *account* parameter, which does not map well onto ElectrumSV so is expected to be omitted or set to null for all RPC requests.

4.4.1 Command-line options

Outside of `--enable-node-wallet-api`, these command-line options are provided to match those provided by *bitcoind*. They are not used for any other purpose outside of configuring the node wallet API server.

`--enable-node-wallet-api`

This command-line argument must be specified to direct ElectrumSV to run a server providing the JSON-RPC wallet API.

In the following example, the operator provides this command-line argument and the JSON-RPC wallet API can be seen to be available by the logged entry.

Listing 1: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcuser=bob -rpcpassword=weakpassword
2022-11-07 12:28:54,983:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:28:54,983:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

Listing 2: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcuser=bob -
↪rpcpassword=weakpassword
2022-11-07 12:28:54,983:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:28:54,983:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

In the following example, the operator does not provide this command-line argument and the JSON-RPC wallet API can be seen as not available by the absence of the logged entry.

Listing 3: Linux / MacOS

```
$ ./electrum-sv
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

Listing 4: Windows

```
electrumsv>py electrum-sv
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

`-rpcuser`

This is the basic authorization user name. It is required for the server to run, except when credentials are disabled by an empty *rpcpassword* value.

Listing 5: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcuser=bob -rpcpassword=weakpassword
2022-11-07 12:28:54,983:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:28:54,983:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

Listing 6: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcuser=bob -
↪rpcpassword=weakpassword
2022-11-07 12:28:54,983:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

(continues on next page)

(continued from previous page)

```
2022-11-07 12:28:54,983:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

A value for this argument must be provided for the server to run, given that credentials have not been disabled with a blank password. An error will be logged indicating why the server is not running, if the operator does not provide this argument.

Listing 7: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcpassword=weakpassword
2022-11-07 12:43:29,313:ERROR:daemon:JSON-RPC wallet API server not running: invalid␣
↪user name or password
2022-11-07 12:43:29,313:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

Listing 8: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcpassword=weakpassword
2022-11-07 12:43:29,313:ERROR:daemon:JSON-RPC wallet API server not running: invalid␣
↪user name or password
2022-11-07 12:43:29,313:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

`-rpcpassword`

This is the basic authorization password. Passing an empty password whether as *-rpcpassword=* or *-rpcpassword ""* will disable authorization and allow anyone who can access the host it is running on to freely make any API calls.

Providing a blank password disables credential checking and will log a warning.

Listing 9: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcpassword=
2022-11-07 10:03:24,375:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 10:03:24,381:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

Listing 10: Windows

```
electrumsv>py electrum-sv daemon  --enable-node-wallet-api -rpcpassword=
2022-11-07 10:03:24,375:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 10:03:24,381:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

A value for this argument must be provided for the server to run. An error will be logged indicating why the server is not running, if the operator does not provide this argument.

Listing 11: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcuser=bob
2022-11-07 12:43:29,313:ERROR:daemon:JSON-RPC wallet API server not running: invalid␣
↪user name or password
2022-11-07 12:43:29,313:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

Listing 12: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcuser=bob
2022-11-07 12:43:29,313:ERROR:daemon:JSON-RPC wallet API server not running: invalid␣
↪user name or password
2022-11-07 12:43:29,313:INFO:rest-server:REST API started on http://127.0.0.1:9999
```

`-rpcport`

The server will default to using port *8332* to serve the API. Using this command-line argument the operator can direct the JSON-RPC API to be served on a different port.

Specifying a custom port of *18332* will result in the server using that port instead.

Listing 13: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcpassword= -rpcport=18332
2022-11-07 12:49:22,204:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 12:49:22,204:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:49:22,204:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:18332
```

Listing 14: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcpassword= -rpcport=18332
2022-11-07 12:49:22,204:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 12:49:22,204:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:49:22,204:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:18332
```

`-walletnotify`

The way that external notifications are provided about changes in wallet state by *bitcoind* is by providing a value to the *walletnotify* command-line argument. ElectrumSV also accepts this command-line argument in order to aid in a clean switch. The provided value should be the full command to execute and the *%s* placeholder will be replaced with the id of the transaction for which there has been a state change.

Supported events:

- A transaction is added to the wallet.
- An external transaction is added to the wallet.
- The wallet broadcasts a transaction.

- The wallet is notified that a transaction has been broadcast.
- A transaction is associated with a block on the favoured tip (mined).
- A transaction is disassociated with a block on the favoured tip (reorged).

Here we specify the `contrib/scripts/jsonrpc_wallet_event.py` sample script provided with ElectrumSV for debugging. It logs all events to a *tx.log* file in the same directory as the script as a testing aid.

Listing 15: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcpassword= -walletnotify="python3␣
↪contrib/scripts/jsonrpc_wallet_event.py %s"
2022-11-07 12:49:22,204:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 12:49:22,204:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:49:22,204:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:18332
```

Listing 16: Windows

```
electrumsv>py electrum-sv daemon --enable-node-wallet-api -rpcpassword= -walletnotify=
↪"py contrib\scripts\jsonrpc_wallet_event.py %s"
2022-11-07 12:49:22,204:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 12:49:22,204:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 12:49:22,204:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:18332
```

4.4.2 Setup

Once you are satisfied the ElectrumSV daemon is running correctly, there are several tasks that need to be performed to get a working wallet and to be able to make use of the JSON-RPC API to do things like solicit payments for it.

1. Create a compatible wallet.
2. Start the ElectrumSV daemon.
3. Load the wallet you created.
4. Link that wallet to a blockchain server.

Wallet creation

In order to create a wallet that is compatible with the node wallet API, a special command `create_jsonrpc_wallet` has to be used. The file name to be used should be provided with the `-w` option and the wallet will be created in the "wallets" folder in the *ElectrumSV data directory*.

Listing 17: Linux / MacOS

```
$ ./electrum-sv create_jsonrpc_wallet -w my_new_wallet
Password:
Confirm:
Wallet saved in '/home/bob/.electrum-sv/wallets/my_new_wallet.sqlite'
NOTE: This wallet is ready for use with the node wallet API.
```

Listing 18: Windows

```
electrumsv>py electrum-sv create_jsonrpc_wallet -w my_new_wallet
Password:
Confirm:
Wallet saved in 'C:\Users\bob\AppData\Roaming\ElectrumSV\regtest\wallets\my_new_wallet.
↪sqlite'
NOTE: This wallet is ready for use with the node wallet API.
```

Warning: Wallets can only be used with the node wallet API if there is one and only one account in the wallet. Existing ElectrumSV wallets that have no accounts or more than one account will not be usable with the node wallet API.

Blockchain server access

The advantage the wallet integrated into the Bitcoin node has is that it listens to and processes all blocks, and knows what in them relates to the wallet. This is however why it is now problematic to run, because the resource requirements to receive and process all those blocks is prohibitive.

In order to detect incoming payments the ElectrumSV JSON-RPC wallet needs to replace that prohibitive block processing with something much much lighter weight. This is done by registering the addresses those payments will come in on with a remote blockchain server. That blockchain server also notifies us when transactions are broadcast and other events of interest that were discerned directly from block data by the node wallet.

The wallet you created with the `create_jsonrpc_wallet` command needs to set up an account on the blockchain server Bitcoin Association provides. This is what is described below.

The first step is to start the wallet server.

Listing 19: Linux / MacOS

```
$ ./electrum-sv daemon --enable-node-wallet-api -rpcpassword=
2022-11-07 10:03:24,375:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 10:03:24,381:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

Listing 20: Windows

```
electrumsv>py electrum-sv daemon  --enable-node-wallet-api -rpcpassword=
2022-11-07 10:03:24,375:WARNING:daemon:No password set for JSON-RPC wallet API. No␣
↪credentials required for access.
2022-11-07 10:03:24,380:INFO:rest-server:REST API started on http://127.0.0.1:9999
2022-11-07 10:03:24,381:INFO:nodeapi-server:JSON-RPC wallet API started on http://127.0.
↪0.1:8332
```

Next open another console/terminal and load your wallet with the daemon subcommand `load_wallet`. This asks the wallet server to load that wallet. If there is an error, it will display in place of the `true` that is otherwise returned.

Listing 21: Linux / MacOS

```
$ ./electrum-sv daemon load_wallet -w my_new_wallet
Password:
true
```

Listing 22: Windows

```
electrumsv>py electrum-sv daemon  load_wallet -w my_new_wallet
Password:
true
```

The final step is to setup the wallet's account with the blockchain server. This requires network access by the wallet server and the `service_signup` daemon subcommand is used for this. You need to specify the wallet you are signing up.

A successful signup will result in the following output:

Listing 23: Linux / MacOS

```
$ ./electrum-sv daemon service_signup -w my_new_wallet
Password:
Registering..
For services:
    Blockchain.
    Message box.
With server:
    http://127.0.0.1:47124/
Done.
```

Listing 24: Windows

```
electrumsv>py electrum-sv daemon service_signup -w my_new_wallet
Password:
Registering..
For services:
    Blockchain.
    Message box.
With server:
    http://127.0.0.1:47124/
Done.
```

If the wallet is already signed up for the services, the output will indicate this:

Listing 25: Linux / MacOS

```
$ ./electrum-sv daemon service_signup -w my_new_wallet
Password:
All services appear to be signed up for.
```

Listing 26: Windows

```
electrumsv>py electrum-sv daemon service_signup -w my_new_wallet
```

(continues on next page)

(continued from previous page)

```
Password:
All services appear to be signed up for.
```

It is also possible to use the `status` daemon subcommand to verify what servers you are connected to and which services they are handling. This can be seen in the `wallets` section under the `servers` key:

Listing 27: Linux / MacOS

```
$ ./electrum-sv daemon status
{
    "blockchain_height": 116,
    "fee_per_kb": 500,
    "network": "online",
    "path": "/home/bob/.electrum-sv",
    "version": "1.4.0",
    "wallets": {
        "/home/bob/.electrum-sv/wallets/my_new_wallet.sqlite": {
            "servers": {
                "http://127.0.0.1:47124/": [
                    "USE_BLOCKCHAIN",
                    "USE_MESSAGE_BOX"
                ]
            }
        }
    }
}
```

Listing 28: Windows

```
electrumsv>py -3.10 electrum-sv daemon status
{
    "blockchain_height": 116,
    "fee_per_kb": 500,
    "network": "online",
    "path": "c:\\Users\\bob\\AppData\\Roaming\\ElectrumSV",
    "version": "1.4.0",
    "wallets": {
        "c:\\Users\\bob\\AppData\\Roaming\\ElectrumSV\\wallets\\my_new_wallet.sqlite": {
            "servers": {
                "http://127.0.0.1:47124/": [
                    "USE_BLOCKCHAIN",
                    "USE_MESSAGE_BOX"
                ]
            }
        }
    }
}
```

4.4.3 API usage

Authorization

Requests made on the JSON-RPC API are required to provide basic authorization credentials.

- If *rpcuser* is provided and *rpcpassword* is not, the server will not run.
- If *rpcpassword* is provided with an empty value, the server will run and will not check credentials.
- If both *rpcuser* and *rpcpassword* are provided, the server will run and expect those values to authorize access.

Curl can be used to make manual or scripted API calls, and will take care of encoding the basic authorization user name and password for the request.

In the following example the arguments were `-rpcuser bob` and `-rpcpassword weakpassword`. This enforced basic authorization credential checking for that user name and password combination.

```
curl --user bob:weakpassword --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method
↪": "getnewaddress", "params": [] }' -H 'content-type: text/plain;' http://127.0.0.
↪1:8332/
```

In the following example the arguments were just `-rpcpassword ""`. This disabled the checking of credentials for API access.

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getnewaddress",
↪"params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Base errors

These errors are high level ones that happen outside of the handling of any call. They are modelled on and should be identical to those returned by the node JSON-RPC implementation. Developers should be able to come here when they encounter an error that is obviously not specific to the call they are making and match the status code to a possible reason they are getting it.

- 400 (Bad request).
  - If a call entry from a single or batch request is not an object. The response body is:

    ```
    {
        id: null,
        result: null,
        error: {
            code: -32600, // RPC_INVALID_REQUEST
            message: "Invalid Request object"
        }
    }
    ```

  - If the *id* field is not a string, numeric or *null*. The response body is:

    ```
    {
        id: null,
        result: null,
        error: {
            code: -32600, // RPC_INVALID_REQUEST
            message: "Id must be int, string or null"
    ```

(continues on next page)

(continued from previous page)

```
    }
}
```

Warning: The node itself places no constraints on what the *id* value can be. This is a custom ElectrumSV constraint. We can relax it if we need to.

   - If the *method* field is not present. The response body is:

```
{
    id: incoming_call.id,
    result: null,
    error: {
        code: -32600, // RPC_INVALID_REQUEST
        message: "Missing method"
    }
}
```

   - If the *method* field value is not a string. The response body is:

```
{
    id: incoming_call.id,
    result: null,
    error: {
        code: -32600, // RPC_INVALID_REQUEST
        message: "Method must be a string"
    }
}
```

   - If the *params* field value is not an object or an array. The response body is:

```
{
    id: incoming_call.id,
    result: null,
    error: {
        code: -32600, // RPC_INVALID_REQUEST
        message: "Params must be an array or object"
    }
}
```

- 401 (Unauthorized).
   - If the *Authorization* header is required but not present.
   - If the authorization type is not *Basic*.
   - If the authorization value cannot be converted into a valid username and password.
- 404 (Not found).
   - If the *method* field value is not a recognized method name. The response body is:

```
{
    id: incoming_call.id,
    result: null,
```

(continues on next page)

(continued from previous page)

```
    error: {
        code: -32601, // RPC_METHOD_NOT_FOUND
        message: "Method not found"
    }
}
```

- 500 (Internal server error).
  - If the JSON in the body cannot be deserialized correctly. The response body is:

```
{
    id: null,
    result: null,
    error: {
        code: -32700, // RPC_PARSE_ERROR
        message: "Parse error"
    }
}
```

  - If the deserialized body is not an object (a single call) or an array (a batch call). The response body is:

```
{
    id: null,
    result: null,
    error: {
        code: -32700, // RPC_PARSE_ERROR
        message: "Top-level object parse error"
    }
}
```

  - If the */wallet/<wallet-name>* path form is used and no wallet with the name *<wallet-name>* exists. The response body is:

```
{
    id: incoming_call.id,
    result: null,
    error: {
        code: -18, // RPC_WALLET_NOT_FOUND
        message: "Requested wallet does not exist or is not loaded"
    }
}
```

Supported endpoints

createrawtransaction

Act as a library of functionality and piece together an incomplete (not fully signed and final) transaction using the provided list of inputs, an object containing outputs and optionally lock time. The transaction created here is not persisted by the wallet in any way.

Parameters:

1. `inputs` (array of input objects, required). Each item in the array is an object containing fields relating to the given input. The structure of an input object is described below.
2. `outputs` (object of outputs, required). The amount in BSV to send (e.g. 0.1).
3. `locktime` (numeric, optional). The transaction locktime value to use.

Each object in the `inputs` array has the following structure:

- `txid` (string, required). The canonically hexadecimal encoded transaction hash of the transaction being spent.
- `vout` (numeric, required). The index of the output in the given transaction being spent.
- `sequence` (numeric, optional, default varies). The sequence value to be specified in the input. This can of course be used to differentiate between final and non-final inputs. If the transaction `locktime` value is non-zero, it will default to `0xFFFFFFFE` (last possible non-final value) otherwise it default to `0xFFFFFFFF` (final).

Listing 29: Example input object.

```
{
    "txid": "0df80206d8c30046d1fbf0f19959b81cef72a9d01fe4fe831520cfee361d2a8a",
    "vout": 0
}
```

The `outputs` object has the following structure:

- `"<address>"` (string, required). The address to send an amount to. - `<amount>` (numeric). The amount to send to the address.
- `"data"` (literal string, optional). One optional "op return" data output.
  - `<hexadecimally encoded data>` (string). The bytes of data to put in the `OP_FALSE OP_RETURN` 0-value data output.

Listing 30: Example outputs object.

```
{
    "mneqqWSAQCg6tTP4BUdnPDBRanFqaaryMM": 200,
    "mineSVDRCrSg2gzBRsY4Swb5QHFgdnGkis": 500,
    "data": "6e6f77206973207468652074696d65"
}
```

Returns:

The hexadecimally encoded serialised transaction bytes. `scriptSig` values in serialised inputs will be empty, represented by `0` which is a zero-length push.

For example, we can force outputs that are considered to be unsafe to be returned:

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "createrawtransaction",
↪ "params": [[{ "txid":
↪"f6a5a25e297a40aafec9ad948efda26597945adf93a4b726ad32a656d73743df", "vout": 1 }], {
↪"mneqqWSAQCg6tTP4BUdnPDBRanFqaaryMM": 0.5 }] }' -H 'content-type: text/plain;' http://
↪127.0.0.1:8332/
```

Returns a result similar to the following:

```

↪"0100000001df4337d756a632ad26b7a493df5a949765a2fd8e94adc9feaa407a295ea2a5f60100000000ffffffff0180f0fa
↪"
```

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 500 (Internal server error)
  - **Code**
    -3 `RPC_TYPE_ERROR`

    Message

    `Expected array, got <other type>`
    The `inputs` argument is not the array type.

    Message

    `Expected object, got <other type>`
    The `outputs` argument is not the object type.

    Message

    `Expected number, got <other type>`
    The `outputs` argument is not the numeric type.

    Message

    `Amount is not a number or string`
    An output object value is not a number or string.

    Message

    `Invalid amount`
    An output object value is a string that cannot be parsed as a value.

    Message

    `Amount out of range`
    An output object amount is an invalid amount of satoshis.
  - **Code**
    -5 `RPC_INVALID_ADDRESS_OR_KEY`

    Message

    `Invalid Bitcoin address:  <address>`
    The provided address parameter is not a valid P2PKH address.
  - **Code**
    -8 `RPC_INVALID_PARAMETER`

    Message

    `Invalid parameter, arguments 1 and 2 must be non-null`
    One or both of `inputs` or `outputs` parameter were `null`.

Message

`Invalid parameter, locktime out of range`

The value of `locktime` was not equal to or between 0 and 0xFFFFFFFF.

Message

`txid must be hexadecimal string (not '<whatever it is>') and length of it must be divisible by 2`

A `txid` field in an entry in the `inputs` parameter, was either missing, `null` or not an string that was a valid hexadecimal data.

Message

`Invalid parameter, missing vout key`

A `vout` field in an entry in the `inputs` parameter, was either missing, `null` or not an integer.

Message

`Invalid parameter, vout must be positive`

A `vout` field in an entry in the `inputs` parameter, was an integer but was negative which is invalid.

Message

`Invalid parameter, sequence number is out of range`

A `sequence` field in an entry in the `inputs` parameter, was an integer but was outside of the valid range of values. It cannot be less than 0 or greater than `0xFFFFFFFF`.

Message

`Invalid parameter, duplicated address:  <address>`

An entry in the `addresses` parameter was specified twice. The node wallet errors on this, so do we.

– **Code**

-32700 `RPC_PARSE_ERROR`

Message

`JSON value is not an object as expected`

The type of an entry in the `inputs` parameter was not an object.

getbalance

Get the balance of the default account in the loaded wallet.

Parameters:

1. `account` (string, optional). Not supported. Use *null* placeholder if necessary.
2. `minconf` (integer, optional, default=1). Limit the results to the UTXOs with this number or more confirmations.
3. `include_watchonly` (bool, optional, default=false). Not supported. Use *null* placeholder if necessary.

Returns:

A numeric value representing how many unspent Bitcoin there are in the wallet.

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getbalance", "params
↪": {} }' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Returns a result similar to the following:

```
4.00001000
```

Incompatibilities:

1. Parameter The `account` parameter is ignored as the node wallet API only operates on one account. If the user has interfered with the wallet and created more accounts, the API call will error. Specifying a non-null value will raise an error.
2. Parameter: The `include_watchonly` parameter is ignored as the node wallet API does not support watch-only accounts at this time. Specifying a non-null value will raise an error.
3. Error: The `RPC_PARSE_ERROR` for `account` is customised and reflects that we do not accept non-null values.
4. Error: The `RPC_PARSE_ERROR` for `include_watchonly` is customised and reflects that we do not accept non-null values.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    ```
    Method not found (wallet method is disabled because no wallet is
    loaded)
    ```

    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    ```
    Ambiguous account (found <count>, expected 1)
    ```

A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by "<count>").

- **Code**
  -8 `RPC_INVALID_PARAMETER`

  Message

  `Invalid parameter, unexpected utxo type: <number>`
  An unspent output was encountered that does not have a supported key type for the JSON-RPC API. This would be if the user is accessing an externally created account with this API.

- **Code**
  -32602 `RPC_INVALID_PARAMS`

  Message

  `Invalid parameters, see documentation for this call`
  Either too few or too many parameters were provided.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a null as expected`
  This is an intentional incompatibility as we do not support this parameter.

  Message

  `JSON value is not an integer as expected`
  The type of the `minconf` parameters are expected to be integers and one or more were interpreted as another type.

  Message

  `JSON value is not a null as expected`
  This is an intentional incompatibility as we do not support this parameter.

getnewaddress

Reserve the next unused receiving address (otherwise known as external key) and return it as a P2PKH address.

Unlike the node wallet, this application does not receive and process all blocks. As such for an address to be reserved and returned, a remote blockchain service needs to be successfully provisioned to monitor this address for a set period of time.

Parameters:

None.

Returns:

The base58 encoded address for the reserved key (string).

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    `Method not found (wallet method is disabled because no wallet is loaded)`

    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `No connected blockchain server`

    No address can be provided until the wallet has signed up with a server and it is currently connected to that server.
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Blockchain server address monitoring request not successful`

    It was not possible to get a successful acknowledgement from the blockchain server that it would monitor the address. It might be that the server has lost connection or it might be that some unexpected error occurred provisioning the monitoring of the address from the server. See the logs.
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`

    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by the *count*).
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `<other error messages>`

    An error occurred attempting to register the address with the blockchain server to be monitored for incoming payments. The error message provides some indication of what happened, but the wallet logs will be needed to diagnose further.

getrawchangeaddress

Reserve the next unused change address (otherwise known as external key) and return it as a P2PKH address.

Parameters:

None.

Returns:

The base58 encoded address for the reserved key (string).

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    `Method not found (wallet method is disabled because no wallet is loaded)`
    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`
    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by the *count*).

gettransaction

Get detailed information about a transaction in the wallet.

Parameters:

1. `txid` (string, required). The transaction id.
2. `include_watchonly` (bool, optional, default=``false``). Not supported. Use *null* placeholder if necessary.

Returns:

The gettransaction method returns an object detailing the matched transaction. The returned transaction object has the following fields:

- `amount`: (numeric). The number of bitcoin sent (negative value) or received (positive value).
- `blockhash`: (string, only present if `confirmations` > 0). The block hash containing the transaction.
- `blockindex`: (numeric, only present if `confirmations` > 0). The index of the transaction in the block that includes it.

- `blocktime`: (numeric, only present if `confirmations` > 0). The block time in seconds since epoch (1 Jan 1970 GMT).
- `confirmations`: (integer). Number of mined blocks including the transaction and on top of it.
- `details`: (array of objects). The sends and receives associated with the transaction.
- `fee`: (numeric, only present for send). The total number of bitcoin paid as a fee (negative value).
- `generated`: (bool, always `true`, only present if transaction is coinbase).
- `hex`: (string). The transaction bytes encoded as a hexadecimal representation.
- `time`: (numeric). The transaction time in seconds since epoch (midnight Jan 1 1970 GMT). Intentionally incompatible, and the same as `timereceived` for now.
- `timereceived`: (numeric). The time received in seconds since epoch (midnight Jan 1 1970 GMT).
- `trusted`: (bool, only present if `confirmations` == 0). Indicates if this coin is considered safe or not.
- `txid`: (string). The transaction id.
- `walletconflicts`: (array of strings, always `[]`).

Each object in the `details` array reflects any coins received (inputs) or coins spent (outputs) in the transaction and has the following fields.

- `abandoned`: (bool, only present for send). `true` if the transaction has been abandoned (inputs are respendable). Currently will always be `false`.
- `account`: (string). The account name associated with the transaction. As we do not support this feature the value will always be `""`.
- `address`: (string, only present if P2PKH). The address of this transaction.
- `amount`: (numeric). The number of bitcoin sent (negative value) or received (positive value).
- `category`: (string). - For outgoing funds this will always be `send`.
  - For incoming funds this will be `receive` unless the transaction is a coinbase transaction. In which case, it might be `immature` (standard heuristic), `orphan` (in a block not on the wallet's chain) or otherwise it will be `generate`.
- `fee`: (numeric, only present for send). The number of bitcoin paid as a fee (negative value). Will be the same for all send detail objects.
- `vout`: (integer). The index of the output with the given amount in the transaction.

Incompatibilities:

1. Parameter: The `include_watchonly` parameter is ignored as the node wallet API does not support watch-only accounts at this time. Specifying a non-null value will raise an error.
2. Response: The details `account` property has no meaning in wallets created for use with the node API.
3. Response: The details `comment` property relates to data that cannot be modified. There are therefore no plans to support this property.
4. Response: The details `label` property relates to data that cannot be modified. There are therefore no plans to support this property.
5. Error: The `RPC_PARSE_ERROR` for `include_watchonly` is customised and reflects that we do not accept non-null values.
6. Returned value: The `abandoned` field in details objects object is always `false` as we always exclude deleted transactions from results in the wallet proper.

7. Returned value: The `account` field in details objects is always `""` as we do not support this feature in any way. Node API wallets explicitly must only ever have one account to be allowed for use.
8. Returned value: The `involvesWatchonly` field in details objects is never included as the node wallet API does not support watch-only accounts at this time.
9. Returned value: The `label` field in details objects is never included as node API wallets do not support setting this field, and the original bitcoind API does not add this property for vouts with no label/comment.
10. Returned value: The `time` field in the main transaction object is always the same as the `timereceived` value. bitcoind computes a "smart time" but we do not support that at this time.
11. Returned value: The `walletconflicts` field in the main transaction object is always `[]` as we do not currently support this field.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    `Method not found (wallet method is disabled because no wallet is loaded)`
    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`
    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by "<count>").
  - **Code**
    -5 `RPC_INVALID_ADDRESS_OR_KEY`

    Message

    `Invalid or non-wallet transaction id`
    The `txid` parameter value is not recognised as the id of a transaction in the wallet database.
  - **Code**
    -32602 `RPC_INVALID_PARAMS`

    Message

    `Invalid parameters, see documentation for this call`
    Either too few or too many parameters were provided.

– **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a string as expected`
  The type of the `txid` parameter is expected to be a string and was interpreted as another type.

  Message

  `JSON value is not a null as expected`
  This is an intentional incompatibility as we do not support this parameter. The type of the entries in the `include_watchonly` parameter are expected to be strings and one or more were interpreted as another type.

listtransactions

Return a selection of the most recent transactions from the wallet ordered from the oldest of these to the most recent of these. Unconfirmed and local transactions are considered most recent and beyond that ordering is by descending block height and block position of transactions.

Warning: Outstanding issues: To maintain compatibility with the node API, the *address* and *vout* fields are limited to a single address per transaction. This doesn't seem to make much sense for the ElectrumSV wallet because each transaction could have many relevant outputs. For now, the addresses are filtered for ones that are "owned" (the wallet has the keys for it) and the first one is selected.

Parameters:

1. `account` (string, optional). Not supported. Use *null* placeholder if necessary.
2. `count` (integer, optional, default=10). Limit the results to this number of transactions.
3. `skip` (integer, optional, default=0). Skip this many transactions before starting the list.
4. `include_watchonly` (bool, optional, default=false). Not supported. Use *null* placeholder if necessary.

Returns:

An array is returned listing the requested number (as modified by the `count` and `skip` parameters) of the newest spends or receipts by the wallet. These are returned in order of the newest to the oldest.

The structure of each entry in the array is as follows:

- `abandoned`: (bool, only present for send). `true` if the transaction has been abandoned (inputs are respendable). Currently will always be `false`.
- `account`: (string). The account name associated with the transaction. As we do not support this feature the value will always be `""`.
- `address`: (string, only present if P2PKH and owned by the wallet - i.e. we have the key for it). **See warning message above**.
- `amount`: (numeric). The number of bitcoin sent (negative value) or received (positive value).
- `blockhash`: (string, only present if `confirmations` > 0). The block hash containing the transaction.

- `blockindex`: (numeric, only present if `confirmations` > 0). The index of the transaction in the block that includes it.
- `blocktime`: (numeric, only present if `confirmations` > 0). The block time in seconds since epoch (1 Jan 1970 GMT).
- `category`: (string). - For outgoing funds this will always be `send`.
  - For incoming funds this will be `receive` unless the transaction is a coinbase transaction. In which case, it might be `immature` (standard heuristic), `orphan` (in a block not on the wallet's chain) or otherwise it will be `generate`.
- `confirmations`: (integer). Number of mined blocks including the transaction and on top of it.
- `fee`: (numeric, only present for send). The number of bitcoin paid as a fee (negative value).
- `generated`: (bool, always `true`, only present if transaction is coinbase).
- `time`: (numeric). The transaction time in seconds since epoch (midnight Jan 1 1970 GMT). Intentionally incompatible, and the same as `timereceived` for now.
- `timereceived`: (numeric). The time received in seconds since epoch (midnight Jan 1 1970 GMT).
- `trusted`: (bool, only present if `confirmations` == 0). Indicates if this coin is considered safe or not.
- `txid`: (string). The transaction id.
- `vout`: (integer, only present if address is present). **See warning message above**.
- `walletconflicts`: (array of strings, always `[]`).

Incompatibilities:

1. Parameter The `account` parameter is ignored as the node wallet API only operates on one account. If the user has interfered with the wallet and created more accounts, the API call will error. Specifying a non-null value will raise an error.
2. Parameter: The `include_watchonly` parameter is ignored as the node wallet API does not support watch-only accounts at this time. Specifying a non-null value will raise an error.
3. Response: The `comment` property is not going to be supported.
4. Response: The `label` property is not going to be supported.
5. Response: The `address` field only selects the first 'owned' address (see details above)
6. Response: The `vout` field is only for the first 'owned' address (see details above)
7. Error: The `RPC_PARSE_ERROR` for `account` is customised and reflects that we do not accept non-null values.
8. Error: The `RPC_PARSE_ERROR` for `include_watchonly` is customised and reflects that we do not accept non-null values.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

`Method not found (wallet method is disabled because no wallet is loaded)`
The implicit wallet access failed because no wallets are loaded.

- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`
    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by "<count>").
  - **Code**
    -8 `RPC_INVALID_PARAMETER`

    Message

    `Negative count`
    The `count` parameter was less than zero.

    Message

    `Negative from`
    The `skip` parameter was less than zero.
  - **Code**
    -32602 `RPC_INVALID_PARAMS`

    Message

    `Invalid parameters, see documentation for this call`
    Either too few or too many parameters were provided.
  - **Code**
    -32700 `RPC_PARSE_ERROR`

    Message

    `JSON value is not a null as expected`
    This is an intentional incompatibility as we do not support this parameter. The type of the entries in the `account` parameter are expected to be strings and one or more were interpreted as another type.

    Message

    `JSON value is not an integer as expected`
    The type of the `count` parameter is expected to be an integer and was interpreted as another type.

    Message

`JSON value is not an integer as expected`
The type of the `skip` parameter is expected to be an integer and was interpreted as another type.

Message

`JSON value is not a null as expected`
This is an intentional incompatibility as we do not support this parameter. The type of the entries in the `include_watchonly` parameter are expected to be strings and one or more were interpreted as another type.

listunspent

List the unspent outputs within the wallet. This can optionally be filtered by number of confirmations or specific addresses.

Parameters:

1. `minconf` (integer, optional, default=1). Limit the results to the UTXOs with this number or more confirmations.
2. `maxconf` (integer, optional, default=9999999). Limit the results to the UTXOs with this number or less confirmations.
3. `addresses` (array of strings, optional, default=null). Limit the results to the UTXOs locked to the provided addresses.
4. `include_unsafe` (bool, optional, default=false). Safe coins are either confirmed, or unconfirmed and fully funded by ourselves. By default they are not included in the set of returned coins.

Returns:

An array of objects, where each object details a matched UTXO. Each object has the following fields:

- `txid`: The canonically encoded hexadeximal transaction id.
- `vout`: The output index in that transaction.
- `scriptPubKey`: The hexadecimal encoded output locking script.
- `amount`: The number of bitcoin locked in the output.
- `confirmations`: Number of mined blocks including the transaction and on top of it.
- `spendable`: Whether we have the keys to spend this coin.
- `solvable`: Whether we know how to spend this coin regardless of whether we have the keys to do so.
- `safe`: Indicate if this coin is considered safe or not, if unsafe coins were included.
- `address`: Only present if this address was included in the `addresses` array and filtered on.

For example, we can filter for a specific address:

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "listunspent", "params
↪": { "addresses": ["mmne6bSrjwRZk16Y7TkwrrWysiUXZfd9ZY"] } }' -H 'content-type: text/
↪plain;' http://127.0.0.1:8332/
```

Returns a result similar to the following:

```
[
    {
        "txid": "023d74ad33de138ef8b98cfd9950dc1b69c5855146e6a64f502a5be92fd626af",
        "vout": 0,
        "scriptPubKey":"76a91444c838328b3b9ab6e0ee1f021e281c46fb2804ca88ac",
        "amount": 50.00000553,
        "confirmations": 2,
        "spendable": false,
        "solvable": true,
        "safe": true,
        "address": "mmne6bSrjwRZk16Y7TkwrrWysiUXZfd9ZY"
    }
]
```

Note that in this case the UTXO is an unspent immature coinbase output, and is not spendable.

For example, we can force outputs that are considered to be unsafe to be returned:

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "listunspent", "params
↪": [0,null,null,true] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Returns a result similar to the following:

```
[
  {
      "txid":"f6a5a25e297a40aafec9ad948efda26597945adf93a4b726ad32a656d73743df",
      "vout":1,
      "address":"mrUu747bPcGuEgGqnRy7LwtWz9phNpTzgF",
      "scriptPubKey":"76a9147845e39d07817a8415d5741893018e4204c2394388ac",
      "amount":1.0,
      "confirmations":0,
      "spendable":true,
      "solvable":true,
      "safe":false
  }
]
```

Common problems:

- If you are not seeing an incoming payment from another wallet or another party, this will likely be because you are not passing the `"include_unsafe"=true` parameter. For a unspent output to be included without this flag, the transaction it is in has to be confirmed or both unconfirmed and for all funding to come from this wallet.

Incompatibilities:

1. Parameter: The node wallet does redundant type checking on the `minconf` and `maxconf` parameters which would otherwise override the `RPC_PARSE_ERROR` with a `RPC_TYPE_ERROR`. As existing API usage should not be erroring with incorrectly typed parameters, this should not be an important point of compatibility.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

Message

`Method not found (wallet method is disabled because no wallet is loaded)`
The implicit wallet access failed because no wallets are loaded.

- 500 (Internal server error)
  - **Code**
    -3 `RPC_TYPE_ERROR`

    Message

    `Expected type list, got <other type>`
    The type of the `addresses` parameter was not a javascript array, and was instead some other type here substituted as "<other type>".
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`
    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by "<count>").
  - **Code**
    -5 `RPC_INVALID_ADDRESS_OR_KEY`

    Message

    `Invalid Bitcoin address:  <details>`
    An entry in the `addresses` parameter was a string but not a valid address. "<details>" will be substituted for contextual text on the reason the address is not valid.
  - **Code**
    -8 `RPC_INVALID_PARAMETER`

    Message

    `Invalid parameter, duplicated address:  <address>`
    An entry in the `addresses` parameter was specified twice. The node wallet errors on this, so do we.

    Message

    `Invalid parameter, unexpected utxo type:  <number>`
    A unspent output was encountered that does not have a supported key type for the JSON-RPC API. This would be if the user is accessing an externally created account with this API.
  - **Code**
    -32602 `RPC_INVALID_PARAMS`

Message

`Invalid parameters, see documentation for this call`
Either too few or too many parameters were provided.

– **Code**
-32700 `RPC_PARSE_ERROR`

Message

`JSON value is not a string as expected`
The type of the entries in the `addresses` parameter are expected to be strings and one or more were interpreted as another type.

Message

`JSON value is not an integer as expected`
The type of the `minconf` or `maxconf` parameters are expected to be integers and one or more were interpreted as another type.

Message

`JSON value is not a boolean as expected`
The type of the `include_unsafe` parameter is expected to be a boolean and was interpreted as another type.

sendrawtransaction

Broadcast the given transaction via a MAPI server. This will not add a related but unknown transaction to the wallet, and it is at this time assumed that any related transactions should be created using the other API methods this API provides.

The Payd wallet is different from a node and the bitcoind wallet is built into a node. When you submit a transaction to bitcoind, it can validate the transaction directly and then add it to the mempool and maybe propagate it if other nodes have compatible settings for transaction acceptance and propagation. Payd instead has to submit the transaction to a remote MAPI server to get direct acceptance from a miner, and the it is running on has no connectivity issues relays the response from there.

Parameters:

1. `hexstring` (string, required). The serialised complete transaction in hexadecimal encoding.
2. `allowhighfees` (bool, optional, default=false). Allow the transaction to have a fee higher than the wallet's designated reasonable fee level.
3. `dontcheckfee` (bool, optional, default=false). Not supported. This would prioritise the transaction in the mempool.

Returns:

The canonically encoded hexadecimal id of the broadcast transaction (string).

Incompatibilities:

1. Parameter: We cannot support the `dontcheckfee` parameter. A node accepts this as an indication the caller wishes the fee to be ignored and for the transaction to be prioritised in it's local mempool.

2. Error: Payd returns `RPC_INVALID_PARAMETER` if a `true` value is given for the `dontcheckfee` parameter.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    `Method not found (wallet method is disabled because no wallet is loaded)`

    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `Ambiguous account (found <count>, expected 1)`

    A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by the *count*).
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

    `No suitable MAPI server for broadcast`

    The wallet tried to obtain fee quotes from MAPI servers and failed. As it chooses the fee for the payment you are askign it to make based on available MAPI server quotes, this means it cannot proceed.
  - **Code**
    -8 `RPC_INVALID_PARAMETER`

    Message

    `dontcheckfee parameter not currently supported`

    This is an intentional incompatibility. Payd is not a node and has no way to implement the feature of prioritsing a transaction in the mempool. We only return this error if a value of `true` is provided by the caller in order to maximise compatibility with existing applications.
  - **Code**
    -22 `DESERIALIZATION_ERROR`

    Message

    `Tx decode failed`

    The `hexstring` argument value was successfully decoded to bytes, but the processing of it as a validly formed transaction failed.

- **Code**
  -26 `VERIFY_REJECTED`

  Message

  `<whatever error MAPI returned>`
  Any of a range of reasons for why the MAPI broadcast of the signed transaction failed.

- **Code**
  -27 `VERIFY_ALREADY_IN_CHAIN`

  Message

  `Transaction already in the mempool`
  The transaction is already known to have been broadcast.

- **Code**
  -32602 `RPC_INVALID_PARAMS`

  Message

  `Invalid parameters, see documentation for this call`
  Either too few or too many parameters were provided.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a string as expected`
  The type of the `hexstring` parameter was expected to be a string but was interpreted as another type.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a boolean as expected`
  The type of the `allowhighfees` parameter was expected to be a boolean but was interpreted as another type.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a boolean as expected`
  The type of the `dontcheckfee` parameter was expected to be a boolean but was interpreted as another type.

sendtoaddress

Construct and broadcast a payment transaction to the given address for the given amount.

Broadcast of the transaction happens through a MAPI endpoint, and the fee is based on the quote returned by that selected endpoint.

`TODO:` Retry broadcast for failed broadcasts?

Parameters:

1. `address` (string, required). The P2PKH address of the recipient.
2. `amount` (numeric or string, required). The amount in BSV to send (e.g. 0.1).
3. `comment` (string, optional). A note to be attached to the transaction in the wallet, for reference purposes.
4. `commentto` (string, optional). The node wallet used this to allow the user to specify the name of a person or organisation who is the recipient. If provided this will be appended to the preceding comment parameter.
5. `subtractfeefromamount` (bool, optional). Not supported.

Returns:

The transaction id of the broadcast transaction (string).

Incompatibilities:

1. Parameter: We do not currently support the `subtractfeefromamount` parameter, which the node accepts as an indication the caller wishes the fee to be subtracted from the payment amount. This is an executive decision in order to limit the scope of work to the necessary parts.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    ```
    Method not found (wallet method is disabled because no wallet is
    loaded)
    ```
    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -3 `RPC_TYPE_ERROR`

    Message

    ```
    Invalid amount for send
    ```
    The specified amount is zero or less.
  - **Code**
    -4 `RPC_WALLET_ERROR`

    Message

`Ambiguous account (found <count>, expected 1)`
A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by the *count*).

– **Code**
  -4 `RPC_WALLET_ERROR`

  Message

  `No suitable MAPI server for broadcast`
  The wallet tried to obtain fee quotes from MAPI servers and failed. As it chooses the fee for the payment you are askign it to make based on available MAPI server quotes, this means it cannot proceed.

– **Code**
  -4 `RPC_WALLET_ERROR`

  Message

  `<A succinct reason for why broadcast failed>`
  There may be a range of reasons for why the broadcast of the signed transaction failed. The 'succinct reason' detailed in the response should make it clear why, and if not give a pointer to a path to follow up.

– **Code**
  -5 `RPC_INVALID_ADDRESS_OR_KEY`

  Message

  `Invalid address`
  The provided address parameter is not a valid P2PKH address.

– **Code**
  -6 `RPC_WALLET_INSUFFICIENT_FUNDS`

  Message

  `Insufficient funds`
  There is not enough money in the wallet available to meet the specified payment amount.

– **Code**
  -8 `RPC_INVALID_PARAMETER`

  Message

  `subtractfeefromamount not currently supported`
  This is an intentional incompatibility. The wallet application does not currently support deducting the fee from the payment amount.

– **Code**
  -13 `RPC_WALLET_UNLOCK_NEEDED`

  Message

```
Error:  Please enter the wallet passphrase with walletpassphrase
first.
```

In order to send funds from this wallet to the provided address, access to the signing keys is required. This is given by unlocking the wallet, if it is not already unlocked.

– **Code**
  -32602 `RPC_INVALID_PARAMS`

  Message

  ```
  Invalid parameters, see documentation for this call
  ```

  Either too few or too many parameters were provided.

– **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  ```
  JSON value is not a string as expected
  ```

  The type of the *comment* or *comment to* parameters are expected to be strings and one or more were interpreted as another type.

signrawtransaction

Take a transaction serialised in hexadecimal encoding and sign it using the keys available to the account.

Parameters:

1. `hexstring` (string, required). The serialised incomplete transaction in hexadecimal encoding. It is expected that this will have one or more absent signatures, that the account will be able to sign as the purpose of this call. Multiple versions of this transaction can be concatenated into the value passed as this parameter and the pre-existing signatures will be extracted from each.
2. `prevtxs` (array of objects, optional). An array of objects containing the relevant parts of each parent transaction.
3. `privkeys` (array of strings, optional). An array of private keys that are not currently used. If we supported this, they would be used to sign instead of any of the keys available to the account.
4. `sighashtype` (string, optional, default="ALL|FORKID"). A specified sighash name out of twelve possible options accepted by bitcoind. Only the default value is currently accepted, see the Incompatibilities section.

The `hexstring` parameter is expected to encode unsigned or partially signed inputs in the standard way accepted by bitcoind. `scriptSig` values in single signature serialised inputs will be empty, represented by `OP_0` which is in effect a zero-length push. Multi-signature serialised inputs are expected to have the correct `scriptSig` structure with absent signatures substituted with `OP_0` (multi-signature support is currently disabled).

Each object in the `prevtxs` array has the following structure:

- `txid` (string, required). The canonically hexadecimal encoded transaction hash of the transaction being spent.
- `vout` (numeric, required). The index of the given transaction output being spent.
- `scriptPubKey` (string, required). The script from the given transaction output being spent.
- `redeemScript` (string, optional). If the script from the output in the given transaction being spent is an older P2SH script, then this is the redeem script of that P2SH script. See the Incompatibilities section.
- `amount` (numeric, required). The value stored in the given transaction output, denominated in units of bitcoin as per bitcoind convention.

Listing 31: Example prevtx object.

```
{
    "txid": "0df80206d8c30046d1fbf0f19959b81cef72a9d01fe4fe831520cfee361d2a8a",
    "vout": 0,
    "scriptPubKey": "",
    "redeemScript": "",
    "amount": 1.2
}
```

The `privkeys` array can contain base 58 encoded private keys. If these are provided these will be the only keys used to sign the transaction and the keys in the wallet will not be used. These parameter is not currently supported, see the Incompatibilities section.

The `sighashtype` value can in theory be one of twelve possible options accepted by bitcoind. Note that currently we only accept `ALL|FORKID` which is also the default, see the Incompatibilities section.

- `ALL`
- `ALL|ANYONECANPAY`
- `ALL|FORKID`
- `ALL|FORKID|ANYONECANPAY`
- `NONE`
- `NONE|ANYONECANPAY`
- `NONE|FORKID`
- `NONE|FORKID|ANYONECANPAY`
- `SINGLE`
- `SINGLE|ANYONECANPAY`
- `SINGLE|FORKID`
- `SINGLE|FORKID|ANYONECANPAY`

Returns:

The returned object has the following structure:

- `"hex"` (string, required). The serialised processed transaction in hexadecimal encoding.
- `"complete"` (boolean, required). Indicates whether the processed transaction is fully signed. It is not a requirement of successful signing for the processed transaction to have all inputs fully signed.
- `"errors"` (list of objects, optional). This is only present if there are entries to include.
  - `"txid"` (string, required). The canonically hexadecimal encoded hash of the transaction being spent.
  - `"vout"` (string, required). The index of the transaction output being spent.
  - `"scriptSig"` (string, required). The hexadecimally encoded signature script.
  - `"sequence"` (string, required). The sequence of the spending input.
  - `"error"` (string, required). Any text describing why the verification or spend failed.

Listing 32: Example returned object with no errors.

```
{
    "hex": "<serialised hexadecimally encoded transaction>",
    "complete": true
}
```

In theory this endpoint can return a variety of errors encountered. The returned transaction should be unchanged from the base transaction originally provided for signing.

Listing 33: Example returned object with inline errors.

```
{
    "hex": "<serialised hexadecimally encoded transaction>",
    "complete": false,
    "errors": [
        {
            "error": "<some message>",
            "scriptSig": "",
            "sequence": 4294967295,
            "txid": "2222222222222222222222222222222222222222222222222222222222222222",
            "vout": 0
        }
    ]
}
```

In reality the `errors` property only ever returns errors for one specific situation. This is where the wallet does not have the spent coin metadata for an input. Every input in the transaction must have matching spent coin metadata. This is normally automatically retrieved from the account database, but for inputs the wallet is not signing must be provided using the `prevtxs` parameter intended for this purpose. If any coin metadata obtained from the account database is known to be spent, this error entry will also be added for the given input for that reason.

Listing 34: Example returned object with inline errors.

```
{
    "hex": "<serialised hexadecimally encoded transaction>",
    "complete": false,
    "errors": [
        {
            "error": "Input not found or already spent",
            "scriptSig": "",
            "sequence": 4294967295,
            "txid": "2222222222222222222222222222222222222222222222222222222222222222",
            "vout": 0
        }
    ]
}
```

For example, here we are taking the result of `createrawtransaction` and using that as input:

```
curl --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "signrawtransaction",
↪"params": [
↪"0100000001df4337d756a632ad26b7a493df5a949765a2fd8e94adc9feaa407a295ea2a5f60100000000ffffffff0180f0fa
↪"] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Returned the following result:

```
{
  "hex":
↪"0100000001df4337d756a632ad26b7a493df5a949765a2fd8e94adc9feaa407a295ea2a5f6010000006a47304402206d2e8a
↪",
  "complete":true
}
```

Incompatibilities:

If a user requires any of these disabled functionalities, they should get in touch with their contact at the Bitcoin Association and request the ones they need.

1. Parameter: Only one transaction is currently accepted in the `hexstring` parameter. If more than one transaction is provided then a compatibility related error will be raised.
2. Parameter: Only spending of P2PKH coins is supported. While it is in theory to support P2PK, P2SH and bare multi-signature, spending of these coins are not currently enabled. Any attempt to do so will result in a compatibility related error.
    1. Related to this: If any `prevouts` entry contains a `redeemScript` property, it will be ignored as we do not currently handle P2SH spends.
3. Parameter: External private keys are not currently accepted. If any are provided a compatibility related error will be raised.
4. Parameter: Only the `ALL|FORKID` sighash name is accepted. As with the bitcoind implementation sighash names that do not include `FORKID` will result in a standard error. However, valid sighash names other than `ALL|FORKID` will raise a compatibility related error.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
    - **Code**
      -32601 `RPC_METHOD_NOT_FOUND`

      Message

      ```
      Method not found (wallet method is disabled because no wallet is
      loaded)
      ```

      The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
    - **Code**
      -3 `RPC_TYPE_ERROR`

      Message

      ```
      Expected string, got <other type>
      ```

      The `hexstring` argument is not the string type.

      Message

      ```
      Expected array, got <other type>
      ```

The `prevtxs` argument is not the array type.

Message

```
Expected array, got <other type>
```

The `privkeys` argument is not the array type.

Message

```
Expected string, got <other type>
```

The `sighashtype` argument is not the string type.

Message

```
Missing txid
```

The `prevtxs` array was found to contain at least one entry with a missing or `null` value in the `txid` field.

Message

```
Expected type string for txid, got <other type>
```

The `prevtxs` array was found to contain at least one entry with a non-string value for the `txid` field.

Message

```
Missing vout
```

The `prevtxs` array was found to contain at least one entry with a missing or `null` value in the `vout` field.

Message

```
Expected type integer for vout, got <other type>
```

The `prevtxs` array was found to contain at least one entry with a non-integer value for the `vout` field.

Message

```
Missing scriptPubKey
```

The `prevtxs` array was found to contain at least one entry with a missing or `null` value in the `scriptPubKey` field.

Message

```
Expected type string for scriptPubKey, got <other type>
```

The `prevtxs` array was found to contain at least one entry with a non-string value for the `scriptPubKey` field.

Message

```
Amount is not a number or string
```

The `prevtxs` array was found to contain at least one entry with a value for the `amount` property that was not a string or number.

Message

```
Invalid amount
```

The `prevtxs` array was found to contain at least one entry with a string value for the `amount` property that cannot be parsed as a value.

Message

```
Amount out of range
```

The `prevtxs` array was found to contain at least one entry with a string value for the `amount` property that was an invalid amount of satoshis.

– **Code**
  -4 `RPC_WALLET_ERROR`

Message

```
Ambiguous account (found <count>, expected 1)
```

A wallet used by the JSON-RPC API must only have one account so that the API code knows which to make use of. The given wallet has either no accounts or more than one account (the current number indicated by the *count*).

– **Code**
  -8 `RPC_INVALID_PARAMETER`

Message

```
hexstring must be hexadecimal string (not '<whatever it is>') and
length of it must be divisible by 2
```

The `hexstring` parameter, was either `null` or not an string that was valid hexadecimal data.

Message

```
scriptPubKey must be hexadecimal string (not '<whatever it is>') and
length of it must be divisible by 2
```

The `prevtxs` list was provided and had at least one object within it that had a `scriptPubKey` value of either `null` or not an string that was valid hexadecimal data.

Message

```
txid must be hexadecimal string (not '<whatever it is>') and length of
it must be divisible by 2
```

The `prevtxs` list was provided and had at least one object within it that had a `txid` value of either `null` or not an string that was valid hexadecimal data.

Message

```
txid must be of length 64 (not <actual hexadecimal string length>)
```

The `prevtxs` list was provided and had at least one object within it that had a `txid` value that was a valid hexadecimal string but was not the required 32 bytes in size.

Message

`Missing amount`

The `prevtxs` list was provided and had at least one object within it that had a missing `amount` property.

Message

`Invalid sighash param`

The provided `sighashtype` parameter value is not one of the twelve accepted by bitcoind.

Message

`Signature must use SIGHASH_FORKID`

The provided `sighashtype` parameter value is not one of the few that include `FORKID`, and `FORKID` is currently required for valid Bitcoin SV signatures.

Message

`Compatibility difference (only ALL|FORKID sighash accepted)`

See the Incompatibilities section.

– **Code**

-13 `RPC_WALLET_UNLOCK_NEEDED`

Message

`Error:  Please enter the wallet passphrase with walletpassphrase first.`

In order to send funds from this wallet to the provided address, access to the signing keys is required. This is given by unlocking the wallet, if it is not already unlocked.

– **Code**

-22 `DESERIALIZATION_ERROR`

Message

`Tx decode failed`

The `hexstring` argument value was successfully decoded to bytes, but the processing of it as a validly formed transaction failed.

Message

`Missing transaction`

The `hexstring` argument value is empty and contains no transaction to sign.

Message

`Compatibility difference (multiple transactions not accepted)`

See the Incompatibilities section.

Message

`Compatibility difference (non-P2PKH spends not accepted)`

See the Incompatibilities section.

Message

`expected object with {"txid","vout","scriptPubKey"}`
The `prevtxs` array was found to contain at least one non-object.

Message

`vout must be positive`
The `prevtxs` array was found to contain at least one object with a `vout` property that was correctly an integer, but less than zero.

Message

`Previous output scriptPubKey mismatch:\n<wallet asm>\nvs\n<prevtxs asm>`
If a `prevtxs` entry is provided for a coin managed by the wallet, the provided `scriptPubKey` bytes must exactly match those the wallet has for that coin. The disassembled script text will be substituted for the `<wallet asm>` and `<prevtxs asm>` placeholders in the message text.

walletpassphrase

This call provides the ElectrumSV daemon with the password for the given wallet which allows the daemon to perform secure operations without requiring user intervention. All private keys are encrypted with the wallet password and without it available they cannot be accessed and operations like signing cannot be performed.

Parameters:

1. Passphrase (string, required). The wallet passphrase.
2. Timeout (numeric, required). The time to keep the wallet passphrase cached in seconds.

Returns:

`null`.

Error responses:

These errors are the custom errors returned from within this call. Base errors that occur during call processing are described above.

- 404 (Not found)
  - **Code**
    -32601 `RPC_METHOD_NOT_FOUND`

    Message

    `Method not found (wallet method is disabled because no wallet is loaded)`
    The implicit wallet access failed because no wallets are loaded.
- 500 (Internal server error)
  - **Code**
    -32602 `RPC_INVALID_PARAMS`

Message

`Invalid parameters, see documentation for this call`

For this error the node would return documentation for this call as the response. We do not. This error is seen when the two required parameters are not passed.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not a string as expected`

  The type of the *passphrase* parameter is expected to be a string and was interpreted as another type.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `JSON value is not an integer as expected`

  The type of the *timeout* parameter is expected to be a integer and was interpreted as another type.

- **Code**
  -32700 `RPC_PARSE_ERROR`

  Message

  `Invalid parameters, see documentation for this call`

  This error is seen when the passphrase is an empty string.

- **Code**
  -14 `RPC_WALLET_PASSPHRASE_INCORRECT`

  Message

  `Error:  The wallet passphrase entered was incorrect`

  This error is seen when the passphrase is not the correct passphrase for the wallet being accessed.

4.5 Wallet as a service

As the Bitcoin SV ecosystem matures services will become available that allow businesses to outsource the wallet management and the services necessary for their products. There is a sizeable advantage to this, as it allows the business to focus on the products and avoid complicated and rather standard work that there is a benefit to outsourcing.

For the businesses that want or even need to be in control of their own wallet and infrastructure, they can treat ElectrumSV as a common open source base, and extend it with their own proprietary functionality. An starting point for this approach can be found in the example application on Github.

This documentation will be fleshed out as time allows.

4.6 Functional tests

The functional tests are a set of tests that use the REST API to manipulate the state of a 'live' RegTest wallet and check that the state changes are matching what is expected. Eventually, the REST API should mature to cover all functionality available through the GUI, allowing automation of any wallet tasks as well as full end-to-end integration testing coverage.

The functional tests are run as part of the Azure pipeline for any commits to any new feature branch or pull requests but can also be run locally (and offline) provided a few dependencies are installed.

A few examples of functional tests are:

- A simulated reorg test (via the RegTest SDK) in which wallet transactions are affected and move to a new block height with a new merkle branch. The database and general wallet state is checked for consistency before and after the reorg.
- Loading the wallet on the daemon.
- Getting the account details.
- Getting utxos before and after topping up the wallet with new coins.
- Getting utxos before and after splitting a coin into smaller outputs.
- Concurrent transaction broadcasting to check the broadcast pathway and as a basic check for data races.

These tests give a broad-brush coverage of many different code paths and as the REST API grows to cover all GUI interactions will give assurances about:

- Correctness of wallet state.
- Regressions at the wallet server level that could in turn affect the user experience.

4.6.1 Installation of dependencies

1. Install pytest pre-requisites:

```
python3 -m pip install pytest pytest-cov pytest-asyncio pytest-timeout electrumsv_
↪node openpyxl
```

2. Install the ElectrumSV-SDK (follow instructions here: https://electrumsv-sdk.readthedocs.io/) and then do:

```
electrumsv-sdk install node
electrumsv-sdk install simple_indexer
electrumsv-sdk install reference_server
electrumsv-sdk install --repo=$PWD electrumsv
```

This will install the repositories and dependencies for these components.

4.6.2 Run the functional tests

The SDK components should be stopped before running the tests as the tests automate resetting and starting these services - it will fail if they are already running.

Run the functional tests with pytest like this:

```
python3 -m pytest -v -v -v contrib/functional_tests/functional
```

Which should output something like (but with verbose logging output):

```
contrib\functional_tests\functional\test_reorg.py::TestReorg::test_reorg PASSED
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_get_all_wallets␣
↪PASSED                                                              [ 25%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_load_wallet␣
↪PASSED                                                                  [ 33%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_websocket_wait_
↪for_mempool PASSED                                                   [ 41%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_websocket_wait_
↪for_confirmation PASSED                                              [ 50%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_get_parent_wallet␣
↪PASSED                                                            [ 58%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_get_account␣
↪PASSED                                                                  [ 66%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_get_utxos_and_top_
↪up PASSED                                                          [ 75%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_get_balance␣
↪PASSED                                                                  [ 83%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_concurrent_tx_
↪creation_and_broadcast PASSED                                          [ 91%]
contrib\functional_tests\functional\test_restapi.py::TestRestAPI::test_create_and_
↪broadcast_exception_handling PASSED

============================================================= 11 passed, 1 skipped, 0␣
↪failed in 49.53s ==========================================================
```

4.6.3 Logging

There is a `pytest.ini` file in `contrib/functional_tests/pytest.ini` with these settings:

```
[pytest]
log_cli=true
log_level=INFO
```

If you are finding the logging details distracting or you want more verbose logging you can refer to the pytest documentation and change the `pytest.ini` settings as needed.

4.7 Benchmarks

The benchmarks use the REST API running on a 'live' RegTest wallet server to produce global metrics about performance.

Currently this includes the rate of transaction broadcast and processing and its interaction with the size and composition of the utxo set. It is likely that the benchmarks will be added to and changed from what they are today (perhaps with an ergonimic way of gathering profiling data at runtime).

The benchmarks are not run in the Azure pipeline in order to avoid slowing it down. Furthermore, the results would not be consistent across time because the underlying hardware and strain on the Azure agent will vary over time. Therefore, the benchmarks should be run on your local development machine where these factors can be controlled for.

4.7.1 Installation of dependencies

1. Install pytest pre-requisites:

```
python3 -m pip install pytest pytest-cov pytest-asyncio pytest-timeout electrumsv_
↪node openpyxl
```

2. Install the ElectrumSV-SDK (follow instructions here: https://electrumsv-sdk.readthedocs.io/) and then do:

```
electrumsv-sdk install node
electrumsv-sdk install simple_indexer
electrumsv-sdk install reference_server
electrumsv-sdk install --repo=$PWD electrumsv
```

This will install the repositories and dependencies for these components.

4.7.2 Settings

The SDK components should be stopped before running the tests as the tests automate resetting and starting these services - it will fail if they are already running.

The current stresstest automates the preparatory measure of splitting utxos up to a predefined count: `DESIRED_UTXO_COUNT` which defaults to 5000 utxos.

There is also a parameter for how many coin splitting outputs there are per coin splitting transaction: `SPLIT_TX_MAX_OUTPUTS` which defaults to 2000. This is included because prior experience found that this affected throughput.

The number of worker tasks: `N_TX_CREATION_TASKS` (which run as coroutines on the asyncio event loop) defaults to 100. I recommend leaving this unchanged.

The total number of transactions that are created and broadcast is defined by the environment variable: `STRESSTEST_N_TXS` and defaults to 2000. If you want to perform a prolonged stresstest you could raise this significantly.

The timer starts when the initial coin splitting has completed and stops when all transactions have been:

- Created and signed
- Broadcast to the network
- Fully confirmed and processed (there is an automated background task that mines blocks and a websocket for waiting on transaction state changes)

In summary; The default settings (as environment variables) are:

```
N_TX_CREATION_TASKS = 100
DESIRED_UTXO_COUNT = 5000
SPLIT_TX_MAX_OUTPUTS = 2000
STRESSTEST_N_TXS = 2000
```

4.7.3 Run with pytest

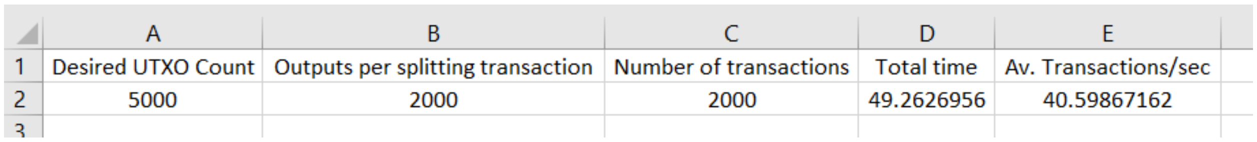
Run the benchmark:

```
python3 -m pytest -v contrib/functional_tests/stresstesting
```

The result is exported to:

```
contrib/functional_tests/stresstesting/.benchmarks/bench_result.xlsx
```

With this format:



	A	B	C	D	E
1	Desired UTXO Count	Outputs per splitting transaction	Number of transactions	Total time	Av. Transactions/sec
2	5000	2000	2000	49.2626956	40.59867162
3					

4.7.4 Run with a matrix of settings

There is also a python script for running the benchmark multiple consecutive times with a matrix of different settings and appending the results to the excel spreadsheet:

```
python3 contrib/functional_tests/stresstesting/run_stresstest_matrix.py
```

The initial output will look like this:

```
============================== Test Params Matrix ==============================
TestParams(number_txs=2000, total_utxo_count=2000, max_split_tx_outputs=500)
TestParams(number_txs=2000, total_utxo_count=2000, max_split_tx_outputs=2000)
TestParams(number_txs=2000, total_utxo_count=5000, max_split_tx_outputs=500)
TestParams(number_txs=2000, total_utxo_count=5000, max_split_tx_outputs=2000)
TestParams(number_txs=2000, total_utxo_count=10000, max_split_tx_outputs=500)
TestParams(number_txs=2000, total_utxo_count=10000, max_split_tx_outputs=2000)
TestParams(number_txs=5000, total_utxo_count=2000, max_split_tx_outputs=500)
TestParams(number_txs=5000, total_utxo_count=2000, max_split_tx_outputs=2000)
TestParams(number_txs=5000, total_utxo_count=5000, max_split_tx_outputs=500)
TestParams(number_txs=5000, total_utxo_count=5000, max_split_tx_outputs=2000)
TestParams(number_txs=5000, total_utxo_count=10000, max_split_tx_outputs=500)
TestParams(number_txs=5000, total_utxo_count=10000, max_split_tx_outputs=2000)
================================================================================
```

This can be thought of as an example script that can be tweaked to your own needs.

4.7.5 Logging

There is a `pytest.ini` file in `contrib/functional_tests/pytest.ini` with these settings:

```
[pytest]
log_cli=true
log_level=INFO
```

If you are finding the logging details distracting or you want more verbose logging you can refer to the pytest documentation and change the `pytest.ini` settings as needed.

CHAPTER

FIVE

THE ELECTRUMSV PROJECT

Perhaps you are a developer who already helps out on the ElectrumSV project, or you who would like to get involved in some way, or you are just curious about the processes and information related to project management and development. If so, this is the information you want.

How can you contribute?
There are many ways that you can help the ElectrumSV project improve. If you want something to work in a different way, you can work on making it different and offer us the changes. If you feel the documentation could be better, you can improve it and offer us the changes. If you want ElectrumSV or anything related to it in your native language, you can offer to do the work to translate it. And that's just a few of the possibilities. Read more about *contributing*.

What platforms and platform versions do we make releases for?
Our builds are created using both third-party dependencies and the Azure Devops services. There are certain limitations that each of these two things imposes on the releases we can make. Read more about the relevant choices and limitations involved in *our releases*.

Where is the continuous integration and how is it used?
We use Microsoft's Azure DevOps services for continuous integration. Microsoft provide generous levels of free usage to open source projects hosted on Github. This is used to do a range of activities for every change we make to the source code, from running the unit tests against each change on each supported operating system, to creating a packaged release for each system that can be manually tested. Read more about our use of *continuous integration*.

What is the process of releasing a new version?
Because we generate packaged releases for every change we make, with a bit of extra work we can generate properly prepared public releases. This involves changing the source code so that the release has the content changes required for new version, and also publishing the release and updating the web site to have the content changes required to offer it for download. Read more about the *release process*.

5.1 How you can contribute

What are some of the ways you might contribute to ElectrumSV?

- Contributing translations.
- Contributing new features.
- Contributing bug fixes.
- Reporting problems.

5.1.1 Translations

Anyone wishing to contribute translations of the text in the ElectrumSV user interface, can do so by the ElectrumSV project on Crowdin. Once you've done entering translations let us know, and we'll do the process of exporting the latest data from Crowdin so that your transaction work gets used.

5.1.2 New features

Be aware that you should check with us before starting work on a feature you are hoping we will accept into ElectrumSV. If we accept a new feature, we then have to maintain it and accept the extra work involved on top of that required for support requests, current features and bug fixes. And it may be that depending on the feature we cannot remove it later, if users become reliant on it or have data that requires it to be present.

5.1.3 Bug fixes

We welcome bug fixes for existing problems, whether they are problems you encounter yourself or ones that you see others have reported that have not already been fixed. You can find our existing bugs in our issue tracker on Github.

5.1.4 Reporting problems

Even if you do not have the experience, skill or inclination to attempt to fix problems you encounter, it would be appreciated if you could report them to us. And if you can take the time to describe what you were doing when you encountered the problem, it helps us fix the problems much more easily. You can report bugs using our template in our issue tracker on Github.

5.2 Continuous integration

As Microsoft provide generous levels of free usage to open source projects hosted on Github through their Azure DevOps service, ElectrumSV makes use of it for a range of purposes. Every time changes are pushed to Github, the following tasks are run:

- Unit tests on Windows, MacOS and Linux.
- Linting.
- Type checking.
- Code coverage analysis.
- Producing releases.

While Azure DevOps will do these things against each individual commit, we have configured the project to only do it against the latest commit.

5.2.1 Releases

There are two goals in having CI produce build files:

- We can use it to produce the build files we release publically.
- Members of the public can access and download build files for any build.

Using CI to produce official release files

By having CI produce the build files, this allows a developer to offload the processing work from their own computer and carry on working on other tasks. In addition there is some security in having the build files made within CI, where the CI obtains the source code directly from the latest commit on Github. And on generating the build files, also produces SHA256 hashes that can be used to validate the content at any later time.

Benefits of public build access

If a user is experiencing a bug, a developer can fix it and push the fix to Github. This will result in an automatic build on Azure DevOps, and if it succeeds will produce build files. The developer can point the user to the build, and although the user may not have an account with Azure DevOps they still have enough access that they can download build artifacts like the build files.

5.3 Releases

Currently ElectrumSV only builds releases for Windows and MacOS. It is expected that Windows users are using at least Windows 10, and MacOS users are using 10.15 or later. We intentionally do not provide either Linux builds or support any form of packaging, and Linux users are expected to get it running from source.

Some of the reasons we do this are intentional, and others are technical limitations that are either imposed by our build environment or the dependencies we use. This document is intended to detail those reasons, both for reference by our users and developers.

5.3.1 Platforms

Developer resources are limited, and we need to focus it where it matters. This is the main reason relating to our platform-related release choices. Even if a community member contributes changes to add support for dated platform versions, accepting those changes can impose heavy ongoing costs on developer time and even unacceptable limitations on development for recent platform versions.

Windows

ElectrumSV Windows builds are for Windows 10 or above. It is possible they work on earlier versions of Windows, but we will neither test that it works or make unreasonable changes to keep it working.

MacOS

ElectrumSV MacOS builds are currently limited to 10.15 and above.

Build environment

Our releases are made in our CI environment provided by Microsoft. The release build the CI environment creates currently requires MacOS 10.15.

Dependency: Qt

ElectrumSV uses the Qt user interface package. Each updated version of Qt requires more and more recent versions of MacOS. At the time of writing, we use 5.15 but we plan to update to 6.1 when we get the time.

- Qt 5.15 needs MacOS 10.13 or later.
- Qt 6.0 needs MacOS 10.14 or later.
- Qt 6.1 needs MacOS 10.14 or later.

Linux

ElectrumSV does not provide any builds or packages for Linux.

People have offered to contribute code to support various Linux packaging systems, but we have had to refuse that. It is very little work to take in that code and produce those packages, but it too much work to test them and verify they work on all the different Linux distributions. We will never accept Linux packaging support for this reason.

What we would be willing to accept, is AppImage support, where the AppImage build runs on at least all mainstream Linux distributions without any extra work. Unfortunately, there has been no interest from Linux users on working on this and contributing that code. The ElectrumSV developers will need to produce the builds, test them and polish them - so there are quality requirements.

Important: Do you want ElectrumSV to have AppImage support for Linux? Get in touch with the ElectrumSV developers and work out what we require in any acceptable solution.

5.4 Release process

There are a lot of steps to releasing a new version of ElectrumSV. This document is intended to lay out the entire proces and some of the reasoning behind it, so that any developer can jump in and do a release if necessary. In addition, formalising the release process ensures that nothing is accidentally left out due to any informal and casually documented process leading to an oversight of various steps.

5.4.1 Initial preparation

When it is time to release a new version, the first step is to freeze the release branch in Github and prevent introduction of any changes that introduce new functionality or change existing functionality. This is an exercise in self restraint, rather than anything that is done to programmatically disallow these changes to be made.

Writing an article

An initial outline of a release article is written, including the featured changes that will be highlighted. Mostly this involves taking the last article, removing all the changes that were included in the previous version, and putting the new version's changes in their place using the same format. The key goal of these articles is to illustrate these changes and help users visualise them even if they skim through the article, and it should include screenshots at every possible opportunity.

For each change featured in a release article:

- A link should be provided to any issue that exists in relation to that change.
- A link should be provided to every code change made to the source code in the making of the given change.

Updating the version

The version number is increased to the new version number, and the approximate release date is updated to be approximately what it will be when the release is made. If the release process is protracted over many days due to the testing, and any subsequent changes they require, then the date may be modified later.

If the last version was `1.3.6`:

- Find all `1.3.6` references and replace them with the new version `1.3.7`.
- Find all `1-3-6` references. These will be in links to the release article for the previous version. The link should be replaced with the link to the new article.

Writing release notes

There are two places that changes are documented in the source code. The first is a HTML-based summary that is accessible from the splash screen that ElectrumSV shows when it starts up. The second is the text-based formal `RELEASE-NOTES` file in the top level of the source code.

The HTML-based summary is intended to be a list of user focused descriptions of the main changes in the release. It lists the same changes as those chosen for the release article.

The `RELEASE-NOTES` file is intended to be developer oriented, and should attempt to list all the changes made and included in the release.

5.4.2 Pre-build testing

There are two different kinds of pre-build testing, both manual and automatic. The manual tests are primarily those which involve a user checking the user interface works as it should. The automatic tests ensure the code is correct as it is possible for such a tool to detect, and that when asked to perform processes the outcomes of those processes are as they should be.

User interface testing

There is a checklist of common use cases for ElectrumSV that the user interface is manually stepped through. New accounts are created, keys and seeds are imported, invoices are paid, hardware wallets are plugged in and out and most if not all of the menu options are used in order to ensure they still work.

Note: TODO: Reference manual user interface testing documents.

As bugs, problems or small aspects that can be improved are identified, they are fixed and the relevant user interfaces are retested. Along these lines, if intuitively something does not quite seem like it is working, time is spent to work out why.

Code analysis

As a part of normal development, before code changes are committed to the Github source code repository, developers are expected to run code quality tools. If they push the changes to Github and they have made changes that do not meet code quality standards, then the CI process will do those same checks and error. The changes made to both prepare the release and fix any problems observed in the user interface should be tested by the developer.

mypy

Python is a programming language with optional typing. For users who choose to use typing, this tool can then try and work out if the code that uses those types is buggy or incorrect.

Running mypy on Windows, Linux or MacOS:

```
mypy --config-file mypy.ini --python-version 3.7
```

pylint

This tool checks for general code correctness and common errors, and warns the developer if it finds any.

Running pylint on Windows, Linux or MacOS:

```
pylint --rcfile=.pylintrc electrum-sv electrumsv contrib/scripts
```

Unit testing

The existing collection of unit tests ensure that a range of processes work correctly. This includes how the code handles different kinds of accounts, migration of wallets from older versions to newer versions, old Electrum seed words, new Electrum seed words, BIP39 seed words, different key types and so on. Running these against lower level changes can often help detect regressions or oversights made in implementing those changes.

Running the unit tests on Windows:

```
pytest electrumsv\tests
```

Running the unit tests on Linux or MacOS:

```
pytest electrumsv/tests
```

5.4.3 Building the release

The continuous integration (CI) service is hooked up to Github. Every time a set of changes are pushed to Github it automatically triggers the CI to test and build those changes. Every build results in what are called a set of artifacts, which are the executables and archives produced as a result of that build. If the developer adds a Git tag structured in a way to designate a release version to the changes they push, then this modifies the build process and produces an official versioned set of build artifacts.

Tagging the latest code as a potential stable release of a `1.3.7` version:

```
git tag sv-1.3.7
```

The developer than pushes both the latest code and the tag to Github, both separately, and in that order:

```
git push
git push --tags
```

A build is only triggered if unpushed code changes are pushed. And the build only looks for the release tag at the start. So the developer needs to push unpushed code changes, and then the new release tag in quick succession.

Build errors

The build runs all the tests that the developer should run before they push the final changes. If they fail, or their development tools are out of date, this might mean that either the developer did not run the tests correctly or that the developer needs to update their tools.

Recapping the automated tests employed:

- The unit tests.
- The functional tests.
- Pylint for style and correctness checking.
- Mypy for type checking.

If there are build errors or the build needs to be rerun, the developer needs to delete the tag and recreate it, and push a new tag with additional code changes to trigger a new build.

Deleting the local tag for a `1.3.7` release:

```
git tag --delete sv-1.3.7
```

Deleting the remote tag for a `1.3.7` release:

```
git push origin --delete sv-1.3.7
```

Testing the build

Once a successful candidate build has been made, the build artifacts are downloaded. One artifact is deleted, the Windows installer which is named with the `-setup.exe` suffix. At this time we do not support this or test it, and in the longer term we will provide this in the form of a Windows Store application.

The build testing is not extensive. If a build executable runs and the wallet user interface appears, then all testing of both functionality and user interface within the pre-build testing will represent how the build behaves.

Linux

There are no Linux builds at this time, so there is no need for testing at this stage.

Note: If a member of the community creates an AppImage build process that is of sufficient quality, we would be willing to help them maintain it and use it in producing official Linux builds.

MacOS

The build is downloaded to a MacOS device, and run.

The following trivial steps are tested:

1. Funds are sent to the wallet on the MacOS device.
2. The funds are then sent back out to an external wallet.

Windows

There are two builds on Windows, a portable build and a non-portable build. A quick recap on the difference is that the portable build stores it's data in a directory local to the portable build executable. The non-portable build stores it's data in the user's application data directory.

The following trivial steps are tested for the non-portable build:

1. Funds are sent to the wallet on the MacOS device.
2. The funds are then sent back out to an external wallet.

The non-portable build is merely started, and if the user interface appears and the wallet selection screen can be reached, it is deemed sufficient.

5.4.4 Deployment

There are a range of steps to doing the deployment.

Build files

The build files are currently hosted for download on Amazon S3 storage rather than on the web site. This was initially done in order to try and reduce the false positive flagging for Malware that ElectrumSV gets on Windows, because of it's use of Pyinstaller. The process of uploading these is intended to be paranoid to ensure that the files uploaded are the actually the ones the CI process produced.

After the build artifacts are uploaded to Amazon S3 storage, they are re-downloaded and the SHA256 hash of each is compared to those that CI produced by redownloading the build hashes from CI.

Web site

Besides reflecting the latest release, another function of the web site is that it hosts a JSON file with signatures from at least one developer for the given release version and date. This is used by the update checker to alert users that there is a new release. The web site also hosts the GPG signatures from at least one developer, which need to be added before it is generated.

Update signatures

The keys used to verify that a release has been signed by a known developer are hard-coded into each build. This makes it difficult to add new signing developers, as users with older builds will lack the keys for those new developers, those builds will appear illegitimate. It is probably a good idea for the process to change sooner rather than later to prepare for working around this.

One or more of the developers can sign to announce the release of the build, and each should do the following:

1. Take the release version which might be `1.3.7`.
2. Take the release date which might be `2020-10-08T20:00:00.000000+13:00`.
3. Combine them which in this case will result in `1.3.72020-10-08T20:00:00.000000+13:00`.
4. Go into the signing wallet and select the signing key.
5. Select the *Sign/verify message* menu.
6. Enter the combined text.
7. Click the *Sign* button and enter the wallet password.
8. Copy the signature and place in the *release.json* file.

The existing *release.json* file is included in the web site generation content, and should be updated and it will automatically be included in the generated web site.

GPG signatures

In addition to hashes proving the integrity of downloaded build files, there are also GPG signatures that indicate who they came from. The public keys of the developers who might sign the build files are in Github much like the SHA256 hashes for each build file.

A sub-directory should be made within the *download* web site content directory for the release version, and the GPG signatures for each new build file placed in there.

Generation

With GPG signatures and release version signatures in place, and also updated for the new version and build files, the final web site can be generated and put in place on the ElectrumSV web host. The generation instructions documented in the web site directory. Assuming that the developer has already been generating the web site in the past, the following commands are all they need to do one final generation.

```
cd docs
cd website
pelican -s pelicanconf.py
```

Standard deployment steps need to be followed and the new uploaded *html* directory needs to match the existing one in the following ways:

1. The same owner using `chown -R`.
2. The same permissions using `chmod -R`.

Documentation

The documentation is hosted on the Read the Docs service. As changes are pushed to the Github repository, Read the Docs is notified and they fetch the changes and trigger an update of the documentation. This mostly benefits users being able to view development documentation. The deployed documentation for a given release cannot change any time post-release development changes are made.

After the tag for the release changes is pushed to Github, a developer needs to add it to the list of tags that Read the Docs is hosting documentation for. And then they need to make it the default tag so that the documentation URL `electrumsv.readthedocs.io` goes there by default.

Github

At this point the documentation, the web site, and almost all other changes should be present in Github. The one thing that may be missing is the SHA256 hashes for the build files, which need to be added to the file `build-hashes.txt` in the source code, and pushed as well. Beyond that they need to be merged into the master branch, which is the place we recommend users go to find them.

Github releases

Github has it's own system for projects to make releases, and we do use that, but we do not use it to release build files. It's primary used to formally designate the release tag as a new release, and associate it with a list of the changes in the release. The changes listed there are taken directly from the `build-hashes.txt` file.

Release article publication

This should just be a matter of applying any final polish to the already prepared release article and pressing whatever resembles the *Publish* button.

Announcements

The link to the release article should be posted to the following places with some additional decorative text.

- Twitter.
- The Metanet.ICU slack.
- The Atlantistic Unwriter slack.
- Anywhere else.

Note: TODO: Guidelines to how we write the standard decorative text should be added here.

5.4.5 The release checklist

It is not realistic for developers to read this document when they want to make a release and step through the description of the process. Instead, they should refer to the following checklist and where necessary refer to the description of the process for context and further details.

Note: TODO: Formalise the above as a list of concrete steps.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex