

# account-based-addresses.md:

---

sidebarlabel: Account based addresses

title: Rootstock Accounts

sidebarposition: 6

tags: [rsk, rootstock, architecture, checksum, derivation path, contract addresses, smart contracts]

description: "EIP-1191 chainId is used in Rootstock addresses as a checksum. m/44'/137'/0'/0 is the derivation path used for BIP-44 compatible wallets."

---

Rootstock Addresses incorporate an optional blockchain identifier (also known as chainId). If the chainId is not present, it is assumed the address refers to the Rootstock main network.

:::info[Info]

See contract addresses for the list of contract addresses on Rootstock or how to verify address ownership.

:::

How to get an address

Check out the already integrated wallets on Rootstock.

Derivation path info

When using

BIP-44-compatible

wallet software, you will need to specify a derivation path.

text

Mainnet: m/44'/137'/0'/0/N

Testnet: m/44'/37310'/0'/0/N

- The first level of the hierarchy is for purpose.  
This is always 44', as per the BIP44 specification.
- The second level of the hierarchy is for the registered coin type.
  - For Rootstock Mainnet, this should be 137', as per the SLIP-44 specification.
  - For Rootstock Testnet, this should be 37310', as per the RSKIP-57 specification.
- The final level of the hierarchy is for index: Addresses are numbered from index 0 in sequentially increasing manner. This number is used as child index in BIP32 derivation. Public derivation is used at this level.

Checksum

Rootstock implements EIP-1191 to protect users from losing funds by mixing addresses of different Ethereum based networks.

In this document, you can find out how to apply the checksum and validate an address. This EIP is also

supported by Web3 and hardware wallets.

## ChainId

To avoid a replay attack by using an already-signed transaction, originally broadcast in “network A”, and subsequently replayed it in “network B”, the EVM-based networks use chainId as part of the transaction properties.

All chainIds can be found at [chainid.network](https://chainid.network).

Rootstock Mainnet: 30

Rootstock Testnet: 31

See EIP-155 for more information.

We strongly recommend the following:

1. Add the chainId in the Rootstock integration (and every time you integrate EVM-based blockchains)
2. Use a different account to hold value for each blockchain (do not share the same account among Rootstock, ETH, and others)

# index.md:

---

sidebarposition: 1

title: Concepts Overview

sidebarlabel: Overview

tags: [rsk, rootstock, concepts, glossary, resources]

description: "This section of the documentation covers the core concepts about the Rootstock blockchain. Working with Rootstock requires an understanding of blockchain technology, bitcoin and smart contracts."

---

Rootstock is the first and longest-lasting Bitcoin sidechain. It is the only layer 2 solution that combines the security of Bitcoin's proof of work with Ethereum's smart contract capabilities. The platform is open-source, EVM-compatible, and secured by over 60% of Bitcoin's hashing power, This robust security model empowers developers to build trustless, innovative dApps within a thriving ecosystem.

This section equips you with the fundamental knowledge required to navigate the Rootstock blockchain. Familiarity with blockchain technology, Bitcoin, and smart contracts will be beneficial as you navigate deeper.

## Navigating Core Concepts

| Resource                      | Description   |
|-------------------------------|---|
| Rootstock Blockchain Overview | Gain a comprehensive understanding of the Rootstock platform.   |
| Rootstock Stack               | Learn about how Rootstock combines the security of Bitcoin PoW with Ethereum's smart contract functionality.  |
| RBTC Token                    | The RBTC token fuels transactions on the Rootstock network. Converting BTC to RBTC is straightforward using various methods. Visit the RBTC section for a comprehensive list of exchanges |

and applications facilitating RBTC acquisition. Visit the RBTC section for a list of exchanges and apps to get RBTC.|

| RIF Suite | Learn about the Rootstock Infrastructure Framework, a comprehensive set of Open-source tools and technologies designed to streamline and incentivize development on Bitcoin.|

| Rootstock Security | The Rootstock platform uses a security mechanism called the Powpeg, it is based on a layered security model, called “defence-in-depth”.|

| Powpeg HSM Firmware | Learn how to verify Powpeg nodes using the HSM Firmware Attestation. |

| Account Based Addresses | EIP-1191 chainId is used in Rootstock addresses as a checksum.

m/44'/137'/0'/0 is the derivation path used for BIP-44 compatible wallets. |

## Next Steps

Ready to embark on your Rootstock development journey? Explore these sections tailored to your specific interests:

## Developers

The Developers section provides all the necessary guides and information for building secure and scalable dApps on Bitcoin with Rootstock. Leverage your existing knowledge of Solidity and tools like Rust, Hardhat, and Wagmi to deploy and scale your dApps on the pioneering layer 2 solution that combines the best of Bitcoin security and Ethereum Smart Contract capabilities.

## Node Operators

Rootstock’s Merged mining offers bitcoin miners an additional revenue stream at no additional cost by using the same mining infrastructure and work to secure the Rootstock sidechain.

The Node Operators section caters specifically to node miners and developers interested in running and managing a Rootstock node.

## Developer Tools

The tools section curates all the essential developer tools available on Rootstock. Find comprehensive resources on tool configuration, usage guides, reference materials, and informative tutorials.

## Resources

Expand your knowledge base with the comprehensive Resources section. Explore tutorials, courses, FAQs, and valuable information on contributing to the Rootstock ecosystem.

# index.md:

---

sidebarlabel: Rootstock Fundamentals

sidebarposition: 2

title: Rootstock Fundamentals

tags: [rsk, rootstock, beginner, concepts]

description: Rootstock is the first and longest-lasting Bitcoin sidechain. It is the only layer 2 solution that combines the security of Bitcoin’s proof of work with Ethereum’s smart contract capabilities.

---

## What is Rootstock?

Rootstock is the first and longest-lasting Bitcoin sidechain. It is the only layer 2 solution that combines the security of Bitcoin’s proof of work with Ethereum’s smart contract capabilities. The platform is

open-source, EVM-compatible, and secured by over 60% of Bitcoin's hashing power, making it the gateway to a vibrant ecosystem of dApps that continues to evolve to become fully trustless.

See the Rootstock Stack.

How is Rootstock connected to bitcoin?

Merged mining with Bitcoin

The first point of contact is through mining.

The bitcoin miners do what is known as merged mining, securing both networks with the same infrastructure and energy consumption.

They create blocks on the bitcoin network every 10 minutes, including transfer of bitcoin from different addresses and in the process they create new bitcoins.

On Rootstock, blocks are created every 30 seconds, to secure the execution of smart contracts. This does not mint any new coins in the process, but does earn a reward from the merged mining.

> Check out <https://rootstock.io/mine-btc-with-rootstock/> to learn more about mining.

Powpeg with Bitcoin

The second point of contact is the Powpeg, also known as the bridge.

This component connects both networks to allow the transfer of bitcoins to Rootstock, thereby allowing developers to interact with smart contracts. They pay gas using the same bitcoin, the smart bitcoin.

To do so, you send bitcoin to a special address, where they are locked in the bitcoin network. Next, in the same address over in the Rootstock network, that same bitcoin is released to the user for use in the Rootstock network. This is called peg-in.

You can do the reverse operation called peg-out, by sending your bitcoin to a special address in the Rootstock network, and receiving your bitcoin back in the bitcoin network.

# index.md:

---

sectionlabel: The Stack

title: Rootstock Stack

sidebarlabel: The Stack

sidebarposition: 200

tags: [rsk, rootstock, stack, architecture]

description: "Learn about how Rootstock combines the security of Bitcoin PoW with Ethereum's smart contract functionality to build dApps on Bitcoin and also how RIF's Open-source tools and technologies designed to streamline and incentivize development on Bitcoin."

---

Rootstock virtual machine (RVM) is the core of the Smart Contract platform. Smart Contracts are executed by all network full nodes. The result of the execution of a Smart Contract can be the processing of inter-contract messages, creating monetary transactions and changing the state of contract-persistent memory. The RVM is compatible with EVM at the op-code level, allowing Ethereum contracts to run flawlessly on Rootstock.

Currently, the VM is executed by interpretation. In a future network upgrade, the Rootstock community is aiming to improve the VM performance substantially. One proposal is to emulate the EVM by dynamically retargeting EVM opcodes to a subset of Java-like bytecode, and a security-hardened and memory restricted Java-like VM will become the new VM (RVM2). This may bring Rootstock code execution to a performance close to native code.

Main features:

Independent virtual machine, that is highly compatible with EVM at the opcode level

Run Ethereum DApps with the security of the Bitcoin network

Performance improvement pipeline documented in numerous RSKIPs created by the Rootstock community

See the Rootstock Improvement Proposals.

!Rootstock Technology Stack - High Level

<section>

<div class="row">

<div class="col two-x-card">

<div class="header-div">

<h2 class="zg-text-bg fs-28">Bitcoin</h2><h3 class="fp-title-color fp-title-color-sm"><span class="ml-1 zg-label">BTC</span></h3>

</div>

<p> Is a store and transfer of value.

The blockchain is secure because miners

with high infrastructure and energy costs

create the new blocks to be added to the blockchain every 10 minutes.

The more hashing power they provide, the more secure the network is.</p>

</div>

<div class="col two-x-card">

<div class="header-div"><h2 class="zg-text-bg fs-28">Rootstock</h2><h3 class="fp-title-color fp-title-color-sm"><span class="ml-1 zg-label">RBTC</span></h3></div>

<p> Is the first open source smart contract platform that is

powered by the bitcoin network.

Rootstock's goal is to add value and functionality to the

bitcoin ecosystem by enabling smart-contracts,

near instant payments, and higher-scalability.</p>

<p>The Smart Bitcoin (RBTC) is the native currency in Rootstock and it is used to pay for the gas required for the execution of transactions. It is pegged 1:1 with Bitcoin, which means in Rootstock there are exactly 21M RBTC. A Powpeg allows the transfer of bitcoins from the Bitcoin blockchain to the

Rootstock blockchain and vice-versa.</p>

</div>

</div>

</section>

# index.md:

---

sidebarposition: 7

title: What is Merged Mining?

sidebarlabel: Merged Mining

tags: [rsk, rootstock, concepts, merged mining]

description: "How merge mining Rootstock with Bitcoin works, and its benefits."

---

Merged mining is the process that allows Rootstock blockchain to be mined simultaneously with Bitcoin blockchain. This can be done because both chains use the same proof-of-work (PoW) algorithm, double SHA-256.

## Get Started

### How it works

Bitcoin mining pools include a reference to Rootstock's block in every mining job they deliver to miners. Every time miners find a solution, it is compared to both networks' difficulties (Bitcoin and Rootstock), delivering three possible outcomes:

- Solution satisfies Bitcoin network difficulty. Hence, a block is assembled and sent to the network. Rootstock's merged mining reference will be included and ignored by Bitcoin network. Since Rootstock's network difficulty is lower than Bitcoin, this solution will also work for Rootstock and can be submitted to the network.
- Solution does not satisfy Bitcoin network difficulty, but does satisfy Rootstock network difficulty. As a consequence, solution will be submitted to the Rootstock network, but not to the Bitcoin network.
- Solution only satisfies pool difficulty, which is many times lower than Bitcoin or Rootstock network difficulty, and it is not submitted to any network.

Solution submitted to the network allows the node to build an SPV proof. If the proof is valid, it is included as part of the block that will be sent to the network.

<div class="video-container">

<iframe width="949" height="534" src="https://youtube.com/embed/l3DkV2tkjU0" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

</div>

### What are the benefits?

Miners earn a high percentage of transaction fees from the Rootstock block they mine. This mining process is done with the same hashing power used in Bitcoin mining, and has no additional cost or impact.

What is the current Rootstock network's hashing power?

You can see Rootstock network hashing power in the Rootstock Stats Website.

## Implementation details for mining software pools

Check out the Getting Started Implementation Guide.

# hsm-firmware-attestation.md:

```
---
title: "Powpeg HSM Firmware Attestation"
sidebarposition: 250
sidebarlabel: Verify Powpeg Nodes
tags: [rsk, rootstock, rbtc, btc, peg, powpeg, hsm]
description: "Learn how to verify Powpeg nodes using the HSM Firmware Attestation."
renderfeatures: 'powpeg-hsm-attestation-frame'
---
```

To verify the Powpeg nodes, follow the HSM firmware attestation process using the steps below. See the Attestation README.

### Powpeg HSM Firmware Attestation - Sovryn

```
<iframe class="w-100 rounded-4" src="/img/rsk/architecture/powpeg-hsm-attestation/sovryn.html"
title="Sovryn" height="400"></iframe>
```

### Powpeg HSM Firmware Attestation - pNetwork

```
<iframe class="w-100 rounded-4" src="/img/rsk/architecture/powpeg-hsm-attestation/pnetwork.html"
title="pNetwork" height="400"></iframe>
```

## Frequently Asked Questions

```
<Accordion>
  <Accordion.Item eventKey="1">
    <Accordion.Header as="h3">What is the multisig scheme for the powHSM? It is a M of N multisig.
    What is M and what is N?</Accordion.Header>
    <Accordion.Body>
      > - A: The best way to get this information is by querying the Bridge directly, since the number of
      members of the PowPeg may change after a PowPeg composition change.
      > - You can use the following methods to query the bridge: getFederationSize,
      getFederationThreshold.
      > - By consensus the required amount of signers (M) will always be half plus one the total amount of
      pignatories  $M = N / 2 + 1$ . See the signatories and attestation information in PowPeg HSM Firmware
      Attestation.
    </Accordion.Body>
  </Accordion.Item>
</Accordion>
```

# index.md:

```
---
title: "Building the Most Secure, Permissionless and Uncensorable Bitcoin Peg"
sidebarposition: 4
sidebarlabel: Powpeg
tags: [rsk, rootstock, rbtc, btc, architecture, powpeg, 2 way peg]
description: "Transfer BTC to RBTC, and RBTC to BTC through the Powpeg."
```

renderfeatures: 'powpeg-hsm-attestation-frame'

---

Rootstock's 2-way peg protocol, called "the Powpeg", has matured from its inception in 2018 as a federation to now include many decentralized qualities. The new Rootstock Powpeg protects private keys stored in special purpose PowHSMs based on tamper-proof secure elements (SE). Each PowHSM runs a Rootstock node in SPV mode, and so signatures can only be commanded by chain cumulative proof of work. Security is established in the Powpeg through the simplicity of a layered design we refer to as defence-in-depth.

:::note[Info]

- The 2 Way Peg Application is available on Testnet and Mainnet.
- For general information about the design and architecture, how to perform a peg-in transaction using Ledger and Trezor, Frequently asked questions and advanced operations you can perform on the 2 way peg app, please refer to the 2 way peg app user guide.
- Get information on the signatories and attestation in the Powpeg HSM Firmware Attestation section.

...

## The History of the Powpeg

Two blockchains with distinct block formats can communicate in a fully decentralized manner if each one can evaluate the other blockchain's consensus rules, and if cross-chain messages are not censored for long periods of time. Currently, only platforms with "Turing-complete" smart contracts can evaluate other blockchain consensus rules. Bitcoin, for better or for worse, lacks the ability to unlock coins over arbitrary predicates. Therefore, when Rootstock was created, it had to use the only existing technology in Bitcoin to distribute trust among parties: multi-signatures. With a multi-signature it is possible to give a group of notaries the task to protect locked bitcoins, tolerating a certain amount of malicious, hacked or unavailable parties.

When the Rootstock genesis block was mined, the Rootstock Federation, an autonomous set of functionaries aimed at protecting the multi-signature, was born. The federation was controlled by the Rootstock Bridge, an unstoppable smart-contract running on Rootstock, and has been successfully working since its creation. In 2020 the Rootstock community decided it was time for the Rootstock peg to grow, both in security and in censorship resistance, evolving from a federated system to the Powpeg. The Powpeg is a unique 2-way peg system that secures the locked bitcoins with the same Bitcoin hashrate that establishes consensus. The set of functionaries still exists, but their role is mainly to keep their hardware and nodes connected and alive at all times; they do not directly control the Bitcoin multisig private keys. See PowPeg HSM Firmware Attestation

## The Powpeg in Rootstock

The Rootstock researchers and developers strategy when designing the Powpeg differs from the one adopted by other teams that have built 2-way peg protocols. The Rootstock Powpeg is based on a layered security model, a practice we call "defence-in-depth". Most other pegs rely on a single all-encompassing cryptographic protocol that solves a multi-party custody problem in an intricate way. These complex cryptographic protocols are delicate and very few entities can audit them thoroughly. Often these types of protocols become compromised, resulting in a sudden loss of security for users.

Other recent 2-way peg designs focus on crypto-economic incentives that take advantage of high collateralization in a new token. However, using a different token for the core sidechain functionality is not aligned with Bitcoin values. The Rootstock Powpeg bridge, instead, relies on multiple defences, or layers, with each layer relatively simple to understand and test. This defence-in-depth approach is what has allowed Rootstock to grow from genesis to the current state without major problems, and without



downtime. Since there is no collateral, the Rootstock Powpeg members are incentivized to participate by receiving a small portion of Rootstock transaction fees that is automatically channeled to them. As seen in the Ethereum ecosystem, transaction fees can eventually provide a sustained income for miners and sometimes even higher than the blockchain subsidy.

## Powpeg Functionaries

Functionaries participating in the Rootstock PowPeg keep specialized hardware called PowHSMs active and connected to special types of Rootstock full nodes (the “Powpeg Node”). A PowHSM is an external tamper-proof device that creates and protects one of the private keys required for the Bitcoin multi-signature protocol, only signing transactions proven valid by enough cumulative work. The Powpeg node is designed to have maximal connectivity and to communicate information about the Rootstock blockchain, specifically cumulative work, to the PowHSM.

The functionary’s role is to ensure that only valid multi-signature transactions are signed by the PowHSM through auditing changes in the PowHSM, the Powpeg node and the communication between them. Functionaries themselves are not actively involved in the signing of transactions in any way, and do not participate in the production of blocks on the Rootstock blockchain.

## Merged-miners and the Armadillo Monitor

A large portion of Bitcoin miners participate in Rootstock merge-mining, providing the persistence and liveness blockchain properties required for effectively securing the Rootstock network. The role of merged-miners in the Powpeg protocol is the largest and most crucial layer of Rootstock’s defence-in-depth approach in securing the bridge between Rootstock and Bitcoin. Functionaries rely on the stability of merge-mining to ensure valid multi-signature transactions are signed and validated in a secure and timely manner.

## Economic Actors and the Bridge Contract

Economic actors such as merchants and exchanges, interact with the Rootstock 2-way peg by sending and receiving peg-in and peg-out transactions (described in more detail below) to the Bridge smart contract through the Rootstock network. The Bridge is a pre-compiled smart contract living in the Rootstock blockchain. The role of the Bridge is to maintain an up-to-date view of the Bitcoin blockchain, verify peg-in requests and command peg-outs. To achieve this functionality, the Bridge contract manages a Bitcoin wallet in SPV (Simple Payment Verification) mode. In this mode, transactions are confirmed by block headers and block headers are minimally validated, but the validation includes the expected proof of work. These validations ensure the Bridge wallet follows the Bitcoin chain which has the highest chain work, but does not check that the chain is valid.

Normally the chain with the highest chain work is the network’s best chain. In the history of Bitcoin there was only a single unintended network fork where one branch was invalid according to pre-established consensus rules. The fork length was 24 blocks. Therefore, in order to prevent intended or unintended invalid forks, the Bridge is designed to wait for 100 confirmations before confirming a peg-in transaction.

## Peg-in/Peg-out and Other Properties of Rootstock Powpeg

We use the now standardized terms peg-in for the process that transfers bitcoins to the sidechain, and peg-out to the process that returns them back to Bitcoin. Performing a peg-in is as easy as sending the bitcoins to the Powpeg address and informing the Bridge about the Bitcoin transaction. The Powpeg functionaries provide a “watch tower” service on behalf of users and inform the Bridge of any peg-in as well.

The Rootstock Powpeg is an asset migration protocol and cannot abort a peg-in in case of network delays. The inability to abort a peg-in during network delays is what generally distinguishes asset

migration protocols from exchange protocols. In exchange protocols, there is always a risk that the counterparty fails to unlock funds, and a user is forced to inform this failure within a bounded delay. Only in a special case does Rootstock refund the bitcoins of a peg-in operation, and this is when a cap, which gradually increases over time, is surpassed.

Technically, the Rootstock Powpeg is a hybrid peg. Peg-ins work in a fully decentralized manner using SPV proofs with the Powpeg members acting only as watchtowers to make sure bitcoin deposits are correctly informed to Rootstock. The user issuing the peg-in transaction can inform Rootstock if the Powpeg members fail to, assuming a worst-case scenario where the user is eventually online to inform Rootstock of the transaction. Since Rootstock assumes a user is the sender and receiver of a 2-way peg transaction, it is highly advised that users inform the Rootstock network.

To perform peg-outs, the Bridge accepts requests from Rootstock accounts, and after thousands of confirmation blocks, the Bridge builds a Bitcoin peg-out transaction commanding the PowHSMs to sign this transaction. The Bridge selects the transaction inputs (or UTXOs) to include in the peg-out transactions, preventing selective censorship of UTXOs of any kind. The Bridge also coordinates and applies forced-delays to all treasury operations required when the Powpeg composition changes. Finally the Bridge serves as an Oracle to expose the Bitcoin blockchain to Rootstock smart-contracts. Rootstock peg-outs rely on the participation of the PowHSMs and collaboration of the majority of Powpeg members, as the PowHSMs need to sign every peg-out transaction. Assuming the practical security provided by PowHSMs, Powpeg peg-outs are also trustless.

## Rootstock Powpeg Security

Rootstock peg is becoming one of the most secure multi-signature systems in existence. Technically, the security of the Powpeg relies on several concurrent strategies: Defence-in-depth, coordination transparency, and public attestation, but a peg's security does not only rely on its technical features. The real-world security must be analysed from several points of view: technical, operational and reputational. In the following, we focus on the Powpeg technical design decisions.

### Defence-in-Depth

Defence-in-depth is realized by a careful separation of responsibilities so that compromising the system requires more than just compromising one element or one actor. The miners alone cannot steal the funds of the peg, neither can the functionaries, nor the PowHSM manufacturer, nor the developers. The peg process is governed by consensus rules enforced in software and firmware, each protecting the other from bugs and vulnerabilities. Furthermore, the Rootstock community protects the code from mistakes. The community goal is to improve the Powpeg by adding more protective layers, each layer adding more security.

As described above, each functionary not only runs a Powpeg node, but also a PowHSM. In the coming months, all existing Powpeg members will have finished upgrading to the PowHSM version 2.0. As explained before, each PowHSM runs a consensus node in SPV mode, so commands need to be backed-up by real hashrate. Cheating the PowHSM becomes too difficult if not impossible without hacking several Bitcoin mining pools.

The term "vetocracy" is very useful in this context. A vetocracy is a system of governance whereby no single entity can acquire enough power to make decisions and take effective charge. Rootstock's defence-in-depth approach to security of the Powpeg follows such an ideology, rendering attacks ineffective. A good question to ask when designing a 2-way peg system should be: "how closely does the protocol resemble a vetocracy", saving many from endless religious debates over federated vs. decentralized systems.

### Coordination Transparency

All communications between functionaries occur over the Rootstock blockchain. There are no hidden messages between functionaries and there is no pre-established subsystem that allows them to communicate secretly. All exchanged messages are public. While we can't prevent hidden communication by hypothetical attackers in full control of the Powpeg node executable code, we do prevent hidden collusion for long periods. As coordination is carried out over the public network, the system forces the PowHSMs to be exposed to the blockchain honest best chain, and allows all network participants to periodically know the PowHSM internal state. As for external hackers, the existence of a pre-established system for hidden coordination would be a powerful tool for privilege escalation as it can be used to obtain functionaries IPs and attempt targeted attacks. Powpeg functionaries could connect to the network over Tor, or change their IPs daily without problem.

Finally the bridge smart-contract builds the peg-out transaction and won't let any of the PowHSMs pick anything related to the transaction to sign. The whole transaction content is decided by Rootstock consensus.

## Firmware Attestation

Rootstock PowHSM firmwares, as well the full node and Powpeg nodes, are generated using deterministic builds, yet currently the firmware installation on PowHSMs cannot be fully trust-free. An auditing group must attest for the correctness of the process of firmware installation on each new device or batch of devices. But we're improving this area with a new defence: the next iteration of the PowHSM firmware (version 2.1) is capable of providing firmware attestation using security features provided by the device. Therefore, the next objective is to include firmware attestation as part of Rootstock's deployment procedures, or even periodically as keepalive messages. Soon attestation messages will be stored in the blockchain and every member of the community will be able to validate PowHSM firmwares.

## Proof of Work is Proof of Time

The cumulative work required by the PowHSM also works as a rate limiter or forced time delay for any attack: Given the fact that Rootstock has a large portion of the Bitcoin hashrate through merge-mining, the amount of cumulative difficulty required to "cheat" the PowHSM into confirming a peg-out over a malicious forked branch implies a large scale collusion by some of the major Bitcoin mining pools for a duration of multiple days. Such an attack would be transparent and visible to both the Bitcoin and Rootstock communities. As in banking vault opening procedures, the PowHSM is actually enforcing a time-delay that lets humans enter the loop if an attack is suspected.

## Peg-in and Peg-out Finality

Since the Bitcoin blockchain and the Rootstock sidechain are not entangled in a single blockchain or in a parent-child relation as in a syncchain, the transfers of bitcoins between them must at some point in time be considered final. If not, bitcoins locked on one side would never be able to be safely unlocked on the other. Therefore, peg-in and peg-out transactions require a high number of block confirmations. Peg-ins require 100 Bitcoin blocks (approximately 2000 RSK blocks), and peg-outs require 4000 Rootstock blocks (approximately 200 Bitcoin blocks). Transactions signed by federation nodes are considered final by Rootstock: these transactions are broadcast and assumed to be included sooner or later in the Bitcoin blockchain. Due to the need for finality, Rootstock consensus does not attempt to recover from an attack that manages to revert the blockchain deep enough to revert a final peg-in or peg-out transaction. If a huge reversal occurs, Powpeg nodes halt any future peg-out, and the malicious actors should not be able to double-spend the peg.

### :::note[IRIS 3.0.0]

Since the IRIS 3.0.0 upgrade, minimum required values for peg-in and peg-out have been halved, Peg-in (BTC) minimum is now 0.005 and Peg-out (RBTC) minimum is now 0.004. Besides this minimum, the Bridge will estimate the fees required to pay for the pegout, if the remainder after paying the fees is too low (not enough to be spent in BTC) the pegout will be rejected. The funds will be reimbursed if the

pegout is rejected by any of the conditions described above.

...

## Decentralization - Building a Vetocracy

The use of PowHSMs in a federation is a step forward in decentralization, because a remotely compromised functionary does not compromise the main element for the security of the peg: a multisig private key. Since Rootstock has a large portion of the Bitcoin merge-mined hashrate, currently surpassing 51%, it seems extremely unlikely that a new group of merge-miners can hijack consensus long enough to force PowHSMs to perform a malicious peg-out. But the Rootstock community should never rest on its laurels. Instead, the Rootstock community is planning to apply once again a layered approach leading to more “additive security”.

## The Powpeg Censorship-Resistance

The Rootstock Powpeg is also unique in the limited set of responsibilities delegated to each Powpeg node. In particular, Powpeg functionaries cannot apply selective censorship on peg-in and peg-out transactions. If one Powpeg functionary attempts to censor a particular transaction, the others functionaries sign and execute the peg-out transaction, causing the censorship to fail. If all functionaries attempt to censor a transaction, then the functionaries cannot continue to perform other peg-outs, as peg-outs are linked with UTXOs, and functionaries cannot choose the UTXOs for the peg-out transactions. The peg-out UTXOs, including “change” UTXOs, are selected by the Bridge contract, forming a consensus-enforced chain. Therefore, selectively banning a transaction leads eventually to a complete halt of the Powpeg, and that’s why selective censorship is not possible.

Regarding the complete shutdown of the Powpeg by a single government, it would be very difficult to pull off as the functionaries are geographically distributed all over the world. To protect from powerful worldwide coordinated attacks or attacks coming from three-letter agencies, Rootstock plans to add an emergency recovery multisig time-lock to activate one year after the Powpeg is proven dismantled. A shutdown attempt would only make Rootstock stronger and more resilient to subsequent attacks, as a new Rootstock Powpeg would rapidly expand and decentralize itself into a hundred individual users around the world, each running an PowHSM device and a Powpeg node over Tor.

## Conclusion

The Rootstock peg has matured from a federation to a Powpeg. As the peg grows over time, more bitcoins are being moved into Rootstock. Developers can find a unique opportunity to build their dApps on our secure and efficient money vault. Compared to alternatives, the Powpeg combines strong security based on layered protections, with maximum decentralization within the constraints established by the Bitcoin scripting system.

# security-model.md:

---

title: Security model

sidebarposition: 200

sidebarlabel: Security model

tags: [Rootstock, security, powpeg, architecture, federation, 2-way peg]

description: "Achieving security in a Powpegged sidechain using proofs of payment"

---

A sidechain is an independent blockchain whose native currency is pegged to the value of another blockchain currency. The peg can be enforced by a protocol or it can be synthetic. A 2-way peg is a protocol-enforced system allowing two currencies to be exchanged freely, automatically, and without incurring in a price negotiation. In Rootstock, the asset that can be freely moved is Bitcoin. When the

network where the bitcoins exist is not clear from the context, we refer to RBTC to bitcoins existing in Rootstock.

In practice, when BTC is exchanged for RBTC, no currency is “transferred” between blockchains in a single transaction. The transfer operation is split into two transactions. In the first, some BTCs are locked in Bitcoin and in the second the same amount of RBTC is unlocked in Rootstock. The whole process is called peg-in. When RBTC needs to be converted back into BTC, the reverse process occurs: the RBTC gets locked again in Rootstock and the same amount of BTC is unlocked in Bitcoin. The process is called peg-out.

Fully trust-minimized and third-party-free two-way pegs can be created if two platforms have Turing-complete smart-contracts. But since Bitcoin currently does not support Turing-complete smart-contracts nor native opcodes to validate external SPV proofs, part of the 2-way peg system in Rootstock relies on an autonomous system called Powpeg. This system comprises a smart-contract called Bridge that controls every operation, and a set of third-parties called pignatories, each one running a software called Powpeg node and a hardware security module called PowHSM. The PowHSM is a tamper-proof device responsible for storing a private key that is part of a multi-signature scheme. In the peg-in process, users send bitcoins to this multi-signature address. The PowHSMs are also responsible for choosing the Rootstock best chain based on cumulative proof-of-work, in a security model called SPV, and for signing Bitcoin peg-put transactions in case the Rootstock blockchain consensus requires it. No single pignatory can control the locked BTCs, nor access the multi-sig private key stored in the PowHSM. Not even the majority of pignatories has the ability to release BTC funds. The PowHSM only proceeds to sign a peg-out transaction upon receiving commands from the Rootstock blockchain, backed by 4000 confirmation blocks, with a cumulative proof-of-work currently equivalent to approximately 100 bitcoin blocks. Note that if a user transfers BTC into RBTC and back, they will normally not receive bitcoins that are directly connected by UTXOs with the original BTC sent. The bitcoins are not locked for specific users, and instead they are locked for use across the entire Rootstock network.

The locking and unlocking of funds is done by the Powpeg without any human intervention. A requirement for being part of the Powpeg is the ability to maintain the PowHSM device online and connected to the Rootstock network with high up-time. It's also a requirement that pignatories are capable of auditing, or review third party audits that attest that the software that powers the node behaves as expected. The PowHSM device is manufactured by a top hardware security company and the firmware was developed by RootstockLabs. The PowHSM provides state-of-the-art maximum security for their private keys using a Secure Element (SE).

As of January 2020, the Powpeg comprises 12 well-known, and highly secure pignatories. Leading Blockchain companies are currently members of the Powpeg. In exchange for their work, the pignatories are awarded 1% of the transaction fees generated on Rootstock, in order to cover the hardware and maintenance costs.

## Powpeg Members Update

The Powpeg is governed by a written protocol that establishes when it is possible or required to add or remove a member. If the conditions to change the composition are met, a pignatory can send a message to the Bridge contract requiring the beginning of a Powpeg composition change. The change involves three phases: a voting period, a delay period and a funds migration period. All phases are automated and coordinated by the Bridge contract, so the process is open, public, and leaves a cryptographic audit trail. During the voting phase, each pignatory can either accept or reject a composition change. Only if the majority of pignatories accept the change, the next phase begins. This phase is a consensus enforced delay of one week. The delay allows users to transfer the Bitcoins back to the Bitcoin network in case they do not trust the new Powpeg composition. Finally, the composition change is activated and the last phase starts, which is responsible for the migration of the funds from the old Powpeg to the new one.

## The Future

One of the features that has been accepted by the community is adding public attestation of the PowHSM firmware for all users to verify the correctness of the PowHSMs. Another upcoming feature is the introduction of frequent keep-alives to detect as early as possible if a pegnatory is down. Several competing community proposals exist on how to improve the security of the Powpeg. If Bitcoin adds special opcodes or extensibility to validate SPV proofs, and once the new system is proven to be secure and trust-free, the Powpeg role will no longer be necessary, and the Rootstock community may implement the changes to adapt Rootstock to the new trust-free system. For example, members of the Rootstock community proposed in 2016 a drivechain BIP, which represents a trust-minimized alternative to the Powpeg.

# conversion-with-ledger.md:

```
---
title: Conversion using a Ledger hardware wallet
sidebarlabel: Ledger
tags: [rsk, rootstock, rbtc, conversion, peg, peg-in, peg-out, federation, ledger]
description: 'How to perform the Powpeg mechanism using Ledger.'
sidebarposition: 304
---
```

In this section, we will go over the steps of converting BTC to RBTC using Ledger hardware wallet, and vice versa on the Bitcoin and Rootstock (RSK) networks.

## General Requirements

- You need a Ledger with Bitcoin and Rootstock Apps installed. We recommend you to have Ledger Live and review this tutorial:
- You need to have Electrum. Install it and configure it to be used with Ledger.
- Node >= 10.16.0

## BTC to RBTC conversion

Instructions on how to do a Mainnet peg-in.

Get a BTC address with balance

We recommend to use Electrum BTC wallet for connecting to BTC Mainnet using Ledger hardware wallet.

- Download the wallet from Electrum Website
- Install Electrum
- Connect and unlock your Ledger device.
- Open the Bitcoin app
- Start Electrum
- Once Electrum starts, create or import a wallet
- At the keystore screen, select Use a hardware device and click Next.
- Select your Ledger device and click next.
- Choose the right derivation path for your account and click Next:
  - Legacy for an account that has addresses starting with a 1

- Go to the third tab "Receive". You will see a Bitcoin address.

:::info[Note]

The Bitcoin wallet needs to be legacy (not Segwit)  
whose public key starts with either m or n,  
and private key starting with p2pkh:

:::

Find a BTC address with balance

You will need to find the corresponding BTC address derived  
from the BTC derivation path in Electrum "Receive" tab.

- Check the derivation path for BTC to be used:
  - Mainnet: 44'/0'/0'/0/0  
BIP 44 Legacy
- Unlock Ledger and open the Bitcoin App
- To get the BTC address derived from the derivation path that you have specified. Run the following script:

```
js
const Transport = require("@ledgerhq/hw-transport-node-hid").default;
const AppBtc = require("@ledgerhq/hw-app-btc").default;

const getBtcAddress = async (derivationPath = "44'/0'/0'/0/0") => {
  try{
    const transport = await Transport.create();
    const btc = new AppBtc(transport);
    const result = await btc.getWalletPublicKey(derivationPath);

    console.log('BTC Address');
    console.log(result.bitcoinAddress);
    console.log('Derivation Path: ' + derivationPath);
  }
  catch(err){
    console.log(err);
  }
};

(async () => {
  await getBtcAddress("44'/0'/0'/0/0");
})();
```

- After that you should get a result similar to:

text

BTC Address

12dAR91ji1xqimzdTQYHDtY....ppSR

Derivation Path: 44'/0'/0'/0/0

:::tip[Tip]

This is the address that you have to use in order to do the transfer to the federation.

:::

Send Bitcoin to Rootstock Federation address

:::warning[Warning]

You need to send a minimum amount of 0.01 BTC or maximum amount, not more than 10 BTC for conversion.

...

To get the Rootstock Federation address you can run the following script:

javascript

```
const Web3 = require('web3');
```

```
const precompiled = require('@rskmart/rsk-precompiled-abis');
```

```
const getFederationAddress = async function(){
```

```
  const bridge = precompiled.bridge.build(new Web3('https://public-node.rsk.co'));
  const address = await bridge.methods.getFederationAddress().call();
  console.log('Federation Address:');
  console.log(address);
}
```

```
(async () => {
```

```
  await getFederationAddress();
})();
```

Once you have the Rootstock Federation address, you can send Bitcoin to it from your Bitcoin address.

Use Electrum to send BTCs to the Rootstock Federation Address. To do that:

- Open Electrum
- Go to Addresses Tab
- Right click over it
- Select the option "Spend From":  
!Spend from
- Finally make a payment to the RSK Federation Address  
!Sending Payment

4 Wait for BTC confirmations

To ensure the transaction, we need to wait 100 BTC confirmations, be patient :

:::tip[Tip]

100 blocks \ 10 minutes/block = 1000 minutes = 16.667 hours approx.

...

5 Get RBTC address from Ledger hardware wallet

Get the corresponding RBTC address from your Ledger hardware wallet, following these steps:

- Connect and unlock your Ledger device.
- Open the RSK app.
- Get RSK derived address running this scripts:

javascript

```
const Transport = require("@ledgerhq/hw-transport-node-hid").default;
```

```
const AppEth = require("@ledgerhq/hw-app-eth").default;
```

```
const getRskAddress = async (derivationPath = "44'/0'/0'/0/0") => {
```



```
try{
  const transport = await Transport.create();
  const eth = new AppEthereum(transport);
  const result = await eth.getAddress(derivationPath);

  console.log('RSK Address');
  console.log(result.address);
  console.log('Derivation Path: ' + derivationPath);
}
catch(err){
  console.log(err);
}
};

(async () => {
  await getRskAddress("44'/0'/0'/0/0");
})();
```

- Go to MyCrypto and connect to Ledger hardware wallet.
- Select Custom Address and put the derivation path  $m/44'/0'/0'/0$ . Then choose the address that you got from the previous step.

## 6 Check RBTC balance

You can check balance of RBTC address on MyCrypto or MEW setting the corresponding derivation path and selection the address.

```
:::info[Note]
```

You have to wait a minimum of 100 confirmations + a minimum of 5 minutes for checking your RBTC balance

• • •  
• • •

## RBTC to BTC conversion

## Instructions on how to do a Mainnet peg-out.

## 1. Get BTC address with Ledger hardware wallet

If you forgot your BTC public address, you can check section 1. The important thing is that the receiving is BTC address will be the same that it was used to send to the federation.

## 2. Send RBTC to Rootstock Bridge Contract

Open MyCrypto or MEW.

Set the corresponding derivation path and selection the address. \

This address has to be the same as that from section 6.

Then do a transaction to the Bridge Contract.

[illegible]

:::info[Note]

- The minimum amount to send in a peg-out transaction must be greater than or equals to 0.004 RBTC for

Mainnet and the minimum amount to send in a peg-in transaction must be greater than or equals to 0.005 BTC for Mainnet.

- Gas Limit of the transaction needs to be manually set at 100,000 gas; otherwise the transaction will fail.
- Gas Price can be set to 0.06 gwei.

...

!Customize Gas in Metamask before send transaction on Rootstock

### 3. Check balance of BTC address

You can either use Electrum wallet downloaded earlier or any Bitcoin explorer to check the balance.

:::info[Note]

The release process on Bitcoin network takes 4000 RSK block confirmations and at least 10 more minutes.

...

# conversion-with-node-console.md:

---

title: Conversion with node and console

sidebarlabel: Node and Console

tags: [rsk, rbtc, conversion, peg, 2-way, peg-in, peg-out, federation, node, cli]

description: 'How to perform the Powpeg mechanism using node and console.'

sidebarposition: 305

---

This section explains how to try the Powpeg mechanism using your Rootstock node and a command line.

#### General Requirements

- You need to be in full control of your BTC private key.
- You need a BTC Wallet properly configured using said private key.
- [Only for release process] You need an Rootstock node up and running, with the RPC interface enabled, and the personal and eth modules enabled
- See how do I run an Rootstock Node?.

#### BTC to RBTC conversion

How to perform a peg-in.

:::warning[Warning]

Read the lock requirements

...

1. With your Bitcoin address, send a BTC transaction to the Rootstock Federation Address.
2. Using your preferred BTC block explorer (e.g. Blocktrail), follow your transaction and wait the stipulated time.
3. Convert the private key to Rootstock format with this tool:

<https://github.com/rsksmart/utils>),  
and write down your Rootstock account information.

4. Then use the Rootstock Testnet Explorer  
or Rootstock Mainnet Explorer  
to see your RBTC balance.  
Remember that Rootstock addresses must start with 0x.

RBTC to BTC conversion

How to perform a peg-out.

:::warning[Warning]

Read the release requirements

:::

1. Add your obtained Rootstock private key to your Rootstock node.  
Replace RSKConvertedPrivateKey, RSKNode and RSKNodePort  
and run this command:  
shell  
\$ curl -X POST --data '{"method":"personalimportRawKey", "params":["<RSKConvertedPrivateKey>", "<passPhraseToEncryptPrivKey>"], "jsonrpc":"2.0", "id":1}' http://<RSKNode>:<RSKNodePort>

2. Unlock your account for transfers.  
Replace RSKAddress, passPhraseJustUsedToEncryptPrivKey, RSKNode  
and RSKNodePort and run:  
shell  
\$ curl -X POST --data '{"method":"personalunlockAccount", "params":["<RSKAddress>", "<passPhraseJustUsedToEncryptPrivKey>", ""], "jsonrpc":"2.0", "id":1}' http://<RSKNode>:<RSKNodePort>

3. Transfer your desired amount.  
Replace RSKAddress, valueToReleaseInWeis, RSKNode and RSKNodePort  
and run:  
shell  
\$ curl -X POST --data '{"method":"ethsendTransaction", "params":[{"from": "<RSKAddress>", "to": "0x00", "gasPrice": 59240000, "gas": 44000, "value": "<valueToReleaseInWeis>"}], "jsonrpc":"2.0", "id":1}' http://<RSKNode>:<RSKNodePort>

4. Wait the stipulated time and check your BTC balance.

# conversion-with-trezor.md:

---

title: Accessing and using funds that are not in accounts derived with Rootstock (RSK) dpath in Trezor T  
sidebarlabel: With Trezor T

tags: [rsk, rbtc, conversion, peg, 2-way, peg-in, peg-out, federation, trezor, dpath]

description: 'How to configure a Trezor T hardware wallet to derive with a custom dpath.'

sidebarposition: 306

---

How to solve the problem of moving your funds when they are in an account that needs to be derived with a custom derivation path (dpath) using Trezor T.

## Context

If you made a BTC to RBTC conversion using Trezor T, you need to access your account by using a custom dpath (44'/0'/0'/0/0 for Mainnet). With the last firmware versions, Trezor T is checking that the dpath matches with the expected one as a safety feature and this is a blocker when you intend to use a different dpath.

You may also want to access your account with a different dpath if you made a mistake; for example, receiving RBTC at an address derived using the Ethereum dpath instead of the Rootstock dpath.

In MyCrypto or MyEtherWallet you may have received this message: "Forbidden key path".

## Solution

To allow custom derivation paths, you will need to turn off safety checks (see Pavol Rusnak message).

To do this, you need to install python-trezor:

```
shell
pip3 install --upgrade setuptools
pip3 install trezor
```

Once you are ready, run this command:

```
shell
trezorctl set safety-checks prompt
```

(you need to have your Trezor T unlocked and accept the configuration in the device)

After moving your funds, you can turn them on again:

```
shell
trezorctl set safety-checks strict
```

# conversion.md:

```
---
title: "RBTC Conversion: Peg in and Peg Out"
sidebarlabel: Peg In & Out
tags: [rsk, rootstock, rbtc, conversion, peg, 2-way, peg-in, peg-out, federation, powpeg]
description: 'Converting RBTC to BTC (peg-in) and BTC to RBTC (peg-out), for both Mainnet and Testnet.'
sidebarposition: 301
---
```

In this article, we explain step by step on how to convert from BTC to RBTC, and vice versa.

The process of conversion utilises a Powpeg mechanism. Thus, these conversions are referred to as peg-ins and peg-outs.

- Peg-in:
  - A conversion from BTC to RBTC
  - Locks BTC in the BTC Federation address
  - Releases RBTC in the Rootstock derived address
- Peg-out:

- A conversion from RBTC to BTC
- Locks RBTC on the Rootstock network
- Releases BTC on the Bitcoin network

## Compatibility

The types of addresses that are accepted for the Federation are:

- Legacy (P2PKH)
- Segwit Compatible (P2SH-P2WPKH)

:::info[Note]

On the Testnets, the token symbols are prefixed with a lowercase t.

Thus, we have BTC and RBTC on the Mainnets, which correspond to tBTC and tRBTC of the Testnets.

...

## Address verifier

Enter your BTC address below to verify whether it may be used to peg in from BTC to RBTC.

## User Guide

- Mainnet Guide
- Testnet Guide

You can try the conversion process using either options below;

- Using a ledger hardware wallet
- Using a software

## Video

Watch this explainer video on How to do BTC & R-BTC Conversions using the Rootstock Powpeg.

```
<div class="video-container">
  <iframe width="949" height="534" src="https://youtube.com/embed/XTpQW9Rw838" frameborder="0"
  allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
  allowfullscreen></iframe>
</div>
```

## FAQs

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">How often does the Rootstock Federation address change?</Accordion.Header>

<Accordion.Body>

Rootstock Federation address has changed several times since Rootstock mainnet launch.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">Do I lose my Bitcoin if the Rootstock Federation address change during my transfer?</Accordion.Header>

<Accordion.Body>

There is a grace period for the Rootstock Federation address change. You will still be able to lock Bitcoin and get RBTC during the grace period. However, any Bitcoin sent to the old Rootstock Federation

address will be lost post to the grace period.

</Accordion.Body>  
</Accordion.Item>  
</Accordion>

## Feedback

Join the Rootstock Global Discord Community, to ask questions and get answers.

# gas.md:

---

title: "RBTC Gas Fees: Optimizing Transaction Costs"

sidebarlabel: Gas

tags: [gas, transactions, rbtc, mainnet, rsk, rootstock, conversion, bitcoin, gas-price]

sidebarposition: 302

---

Gas is the internal pricing for running a transaction or contract.

When you send tokens, interact with a contract, send RBTC, or do anything else on the blockchain, you must pay for that computation. That payment is calculated as gas. In Rootstock, this is paid in RBTC.

What is gas?

There are four important concepts:

- Gas price: The cost of the operation.
- Gas limit: The maximum gas the operation can afford. It's an upper limit the user sets to prevent losing gas.
- Total gas: The gas the operation consumed. Also referred to as gas used.
- Unit: Gas is paid in RBTC.

Let's start with a simple analogy: A car.

To drive a car you need gas. Gas price is the money you pay for each gallon. Gas limit is the max amount of gas you accept to consume, the gas you charge. The total gas is the amount you've spent at the end of the trip.

You can calculate the total gas and set an appropriate gas limit so that your trip does not expend more than expected.

Transactions are quite similar:

Gas price is the price you set for operations. The gas limit is the maximum price you are going to pay for the transaction when operated. Then, when transaction is executed, the total gas is the price you finally pay.

Gas is the fee collected by the miner who mines the block that includes the transaction.

The resulting fee is:

$$\text{fee} = \text{totalGas} \times \text{gasPrice}$$

How do I choose an appropriate gas price and limit?

If you want to spend less on a transaction, you can do so by lowering the amount you pay per unit of gas (gas price). Similar to Bitcoin, the price you pay for each unit increases or decreases how quickly your transaction will be mined.

## Appropriate gas price

Gas price changes with time. To choose an appropriate gas price you should consider 2 concepts:

- What is minimum gas price and how it changes
- How to get that minimum gas price

## Minimum Gas Price

The `minimumGasPrice` is written in the block header by miners and establishes the minimum gas price a transaction should have in order to be included in that block. It can change with time, by up to 1% of the `minimumGasPrice` of the previous block. The latest block's minimum gas price can be obtained using this Web3 method:

The means by which minimum gas price is negotiated by miners is described in RSKIP09.

```
javascript
web3.eth.getBlock('latest').minimumGasPrice
```

tip[Gas Limit]

The transaction gas limit per block is 6,800,000 units.

...

Here are some practical approaches to this subject:

### 1. Optimistic approach (not recommended):

You can set `minimumGasPrice` as gas price parameter for the transaction but if minimum gas price is under negotiation and it gets higher, your transaction could be rejected.

### 2. Sensible approach:

Instead of using `minimumGasPrice` as it is, you may add 10% to its value.

### 3. Network average approach:

You can obtain the average gas price that is being paid in the network:

```
javascript
web3.eth.gasPrice()
```

Even though this value is greater than or equal to minimum gas price. (`gasPrice >= minimumGasPrice`), it is recommended to add a small percentage to increase the priority of your transaction.

## Appropriate gas limit

Total gas can be estimated using this Web3 method:

javascript

```
myContract.methods.myMethod(param1, param2, ...).estimateGas(options, callback)
```

> Go here for Web3 documentation.

More information

How does gas price change over time?

Each miner can vote to increase or decrease the minimumGasPrice up to 1%. This allows miners to increase the minimumGasPrice 100% in approximately 50 minutes, assuming a block every 30 seconds. Nodes that forward transactions could check that the advertised gas price in a transaction is at least 10% higher than the minimum. This assures the transaction a lifetime of 10 blocks assuming a constantly increasing block minimumGasPrice.

Negotiated minimum gas price is described in RSKIP09.

What happen if my transaction fails?

You are paying for the computation, regardless of whether your transaction succeeds or fails. Even if it fails, the miners must validate and execute your transaction (computation request) and therefore you must pay for that computation just like you would pay for a successful transaction.

What happen if I run out of gas?

If a transaction reaches the gas limit, all changes will be reverted but the fee is still paid.

Gas in smart contracts

When you compile smart contracts (commonly written in Solidity), they get converted to operation codes, known as 'opcodes'.

These codes (opcodes) are shown with mnemotechnic names as ADD (addition) or MUL (multiplication). Here you can see the price of each opcode.

As you can guess, it is important to write smart contracts using the best (cheaper) combination of opcodes.

Examples of good practices to write smart contracts:

Avoid declaring variables as var

javascript

```
function payBonus() {  
  for (uint i = 0; i < employees.length; i++) {  
    address employee = employees[i];  
    uint bonus = calculateBonus(employee);  
    employee.send(bonus);  
  }  
}
```

In the code above, the problem is that if the type of i was declared as var, it would be taken as uint8 because this is the smallest type that is required to hold the value 0. If the array has more than 255 elements, the loop will not finish successfully, resulting in wasted gas. You'd better use the explicit type uint for no surprises and higher limits. Avoid declaring variables using var if possible.

Looping large arrays



```
javascript
function soDifficultLooper() {
  for (uint i = 0; i < largeArray.length; i++) {
    address person = largeArray[i];
    uint payment = difficultOperation(largeArray);
    person.send(payment);
  }
}
```

Every function call that modifies state of the smart contract has a gas cost. A loop could spend a lot of gas, which could easily reach the gas limit of a transaction or block. If a transaction reaches the gas limit, all changes will be reverted but the fee is still paid. Be aware of variable gas costs when using loops.

# index.md:

```
---
sidebarposition: 2
title: Smart Bitcoin RBTC Token on RBTC - BTC to RBTC
tags: [mainnet, testnet, tokens, rbtc, gas, rsk, rootstock]
sidebarlabel: RBTC Token
---
```

RBTC is the token used to pay for the execution of transactions in Rootstock. You can convert BTC into RBTC by sending BTC through the Powpeg (both in Testnet and Mainnet), or by using the faucet in Testnet, or via decentralized exchanges.

See supported wallets.

Get RBTC

RBTC (Smart Bitcoin in Mainnet)

```
<table >
  <tbody>
    <tr>
      <td scope="row">Token Name</td>
      <td><a href="https://coinmarketcap.com/currencies/rsk-smart-bitcoin/"
target="blank">RBTC</a></td>
    </tr>
    <tr>
      <td scope="row">Total Supply</td>
      <td>21,000,000 RBTC</td>
    </tr>
    <tr>
      <td scope="row">Circulating supply</td>
      <td><a href="https://be.explorer.rootstock.io/circulating" target="blank">API</a></td>
    </tr>
    <tr>
      <td scope="row">Contract Type</td>
      <td>Native (<a
href="https://explorer.rootstock.io/address/0x0000000000000000000000000000000010000006"
target="blank">Bridge contract</a></td>
```

```

</tr>
<tr>
  <td scope="row">How to get</td>
  <td>
    <ul>
      <li><a href="https://rootstock.io/rbtc/#get-rbtc" target="blank">Exchanges and Bridges to get
RBTC</a></li>
    </ul>
  </td>
</tr>
<!-- <tr>
  <td scope="row">Logo</td>
  <td>
    
    <a href="/img/rsk/RBTC-logo.png" target="blank">RBTC</a>
  </td>
</tr> -->
</tbody>
</table>

```

tRBTC (Smart Bitcoin in Testnet)

```

<table >
  <tbody>
    <tr>
      <td scope="row">Token Name</td>
      <td>tRBTC</td>
    </tr>
    <tr>
      <td scope="row">Total Supply</td>
      <td>21,000,000 tRBTC</td>
    </tr>
    <tr>
      <td scope="row">Circulating supply</td>
      <td><a href="https://be.explorer.testnet.rootstock.io/circulating" target="blank">API</a></td>
    </tr>
    <tr>
      <td scope="row">Contract Type</td>
      <td>Native (<a
href="https://explorer.testnet.rootstock.io/address/0x0000000000000000000000000000000000000000000000000000000000000000"
target="blank">Bridge contract</a>)</td>
    </tr>
    <tr>
      <td scope="row">How to get</td>
      <td>
        <ul>
          <li><a href="https://faucet.rootstock.io" target="blank">Faucet</a></li>
        </ul>
      </td>
    </tr>
  </tbody>
</table>

```

# networks.md:

---

title: Converting BTC to RBTC and vice versa  
sidebarlabel: Networks  
tags: [rsk, rootstock, rbtc, conversion, peg, 2-way, peg-in, peg-out, federation]  
description: 'Converting BTC to RBTC (peg-in) and RBTC to BTC (peg-out) on Mainnet and Testnet.'  
sidebarposition: 303  
---

## Mainnet Conversion

In this section we will go over the steps of converting BTC to RBTC and vice versa in Bitcoin and Rootstock (RSK) Networks.

:::tip[Tip]  
The minimum amount of Bitcoin to convert is 0.005 BTC for Mainnet.  
:::

### BTC to RBTC conversion

Instructions on how to do a Mainnet peg-in.

```
<Accordion>
  <Accordion.Item eventKey="0">
    <Accordion.Header as="h3">1. Get a BTC address with balance</Accordion.Header>
    <Accordion.Body>
      - Any Bitcoin wallet that supports legacy (p2pkh) private key works for this step. In this section, we
      use the Electrum BTC wallet for connecting to BTC Mainnet.
      1. Download the wallet from Electrum Website
      2. Install Electrum
      3. Start Electrum
      4. Once Electrum starts, create or import a wallet
      5. Go to the third tab "Receive". You will see a Bitcoin Testnet address like below:
      <div align="left"></div>
      > Note: Use a legacy Bitcoin wallet (not Segwit) with a public key beginning with m or n, and a private
      key prefixed by p2pkh.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="2">
    <Accordion.Header as="h3">2. Send Bitcoin to Rootstock Federation address</Accordion.Header>
    <Accordion.Body>
      Send Bitcoin to Rootstock Federation address
      - The Rootstock Federation address is retrieved by making a Smart Contract call on Rootstock
      Mainnet. To make the call, you need to have MyCrypto installed:
      1. Select Rootstock (RSK) Network.
      2. Navigate to MyCrypto -> Contracts.
      3. Select Existing Contracts and choose Bridge from the drop-down menu.
      4. Click getFederationAddress to execute the call.
      It should look like the screenshot below:
      <div align="left"></div>
      Once you have the Rootstock Federation address, you can send Bitcoin to it from your Bitcoin
      address.
      > Note: You must send a minimum amount of 0.005 BTC.
```

```

</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="3">
  <Accordion.Header as="h3">3. Wait for BTC confirmations</Accordion.Header>
  <Accordion.Body>
    - To ensure the transaction is successful, we need to wait for 100 BTC network confirmations.
      > 100 blocks \ 10 minutes/block = 1000 minutes = 16.667 hours. That is, this will take approximately
17 hours.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="4">
  <Accordion.Header as="h3">4. Get RBTC address with BTC private key</Accordion.Header>
  <Accordion.Body>
    - You can get a corresponding RBTC address from your BTC private key by using the Rootstock
Utils. If you do not want to compile the utility, you can download the latest release.
      > Note: when entering Bitcoin private key do not include p2pkh: in the front.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="5">
  <Accordion.Header as="h3">5. Check RBTC balance</Accordion.Header>
  <Accordion.Body>
    You can check balance of RBTC address on Metamask, MyCrypto, or any Rootstock compatible
wallets.
      > Note: You have to wait a minimum of 100 confirmations + a minimum of 5 minutes for checking your
RBTC balance.
  </Accordion.Body>
</Accordion.Item>
</Accordion>

```

## RBTC to BTC conversion

## Instructions on how to do a Mainnet peg-out.

[illegible]

- You can either use Electrum wallet downloaded earlier or from any Bitcoin explorer to check the balance.

> Note: The release process on Bitcoin network takes 4000 Rootstock block confirmations and at least 10 more minutes.

</Accordion.Body>

</Accordion.Item>

</Accordion>

## Testnet Conversion

In this section we will go over the steps of converting t-BTC to tRBTC, and vice versa on the Bitcoin and Rootstock Testnets.

:::tip[Tip]

The minimum amount of Bitcoin to convert is 0.005 tBTC for Testnet.

:::

### tBTC to tRBTC conversion

Instructions on how to do a Testnet peg-in.

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">1. Connect a wallet to Bitcoin Testnet</Accordion.Header>

<Accordion.Body>

We recommend to use Electrum BTC wallet for connecting to Bitcoin Testnet.

- Download the wallet from

Electrum Website

- Install Electrum

- Start Electrum in Testnet mode

- For example on MacOS:

/Applications/Electrum.app/Contents/MacOS/Electrum --testnet

- After Electrum starts, create or import a wallet

- Go to the third tab, "Receive".

You will see a Bitcoin Testnet address like below.

!Create a Legacy (p2pkh) wallet

- Note: The Bitcoin wallet needs to be legacy (not Segwit) whose public key starts with either m or n, and private key starting with p2pkh:

!Get a Bitcoin Testnet address in Electrum Wallet

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">2. Get test Bitcoin from Testnet Faucet</Accordion.Header>

<Accordion.Body>

There are a few options to get Bitcoin on Testnet. We use <https://testnet-faucet.mempool.co/>.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">3. Send Bitcoin to Rootstock Federation address</Accordion.Header>

<Accordion.Body>

- The Rootstock Federation address is retrieved by making a Smart Contract call on Rootstock Testnet.

- In order to make the call, you will need to have

MyCrypto



```
!Customize Gas in Metamask before send transaction on Rootstock
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="2">
  <Accordion.Header as="h3">3. Check balance of tBTC address on Bitcoin
Testnet</Accordion.Header>
  <Accordion.Body>
    You can either use Electrum wallet downloaded earlier or from
    any Bitcoin explorer to check the balance.
  </Accordion.Body>
</Accordion.Item>
</Accordion>
```

# index.md:

```
---
sidebarlabel: RIF Suite
sidebarposition: 3
title: "RIF Suite - Open Source Tools"
tags: [rif, token, dApps, products]
description: "Open-source tools and technologies that make it faster, easier and more rewarding to build
on Bitcoin."
---
```

Open-source tools and technologies that make it faster, easier and more rewarding to build on Bitcoin.

Meet the Suite

| Product | Description   |
|---------|---|
| Flyover | Flyover is a fast and secure way for users to transfer BTC in and out of the Rootstock Ecosystem where it can be used to interact with a range of applications to send, save and spend money.   |
| RNS     | RNS (RIF Name Service) replaces complicated cryptocurrency addresses with easy-to-remember nicknames, simplifying digital asset transactions. It also facilitates the integration of a Self Sovereign Identity protocol into your products, which enhances user security and flexibility. . |
| Wallet  | Bring Bitcoin DeFi to your users with RIF Wallet, an open source Bitcoin wallet with smart contract capabilities. Open-source, fully programmable and customizable.   |
| Relay   | RIF Relay simplifies gas fee payments by allowing users to pay transaction fees with any ERC20 token. This enables end users to transact entirely using one asset, removing complexity and improving onboarding.  |

# index.md:

```
---
sidebarlabel: RIF Token
sidebarposition: 400
title: "RIF Token: Empowering Decentralized Applications"
tags: [rif, token, erc677]
description: "Information about the RIF token, where to obtain it, how to transfer it, and technical details
on its token standard"
---
```

The Rootstock Infrastructure Framework (RIF) Token allows any token holder to consume the services that are compatible with RIF Tools.

#### RIF (RIF Token in Mainnet)

|                  |  |
|------------------|--|
| Token Name       | <a href="https://coinmarketcap.com/currencies/rsk-infrastructure-framework/" target="blank">RIF</a>  |
| Total Supply     | 1,000,000,000 RIF  |
| Contract Address | <a href="https://explorer.rootstock.io/address/0x2acc95758f8b5f583470ba265eb685a8f45fc9d5" target="blank">0x2acc95758f8b5f583470ba265eb685a8f45fc9d5</a> |
| Contract Type    | ERC677   |
| How to get       | <ul style="list-style-type: none"><li><a href="#exchanges" target="blank">Exchanges</a></li></ul>  |

#### tRIF (RIF Token in Testnet)

|                          |  |
|--------------------------|--|
| Token Name               | tRIF   |
| Total Supply             | 1,000,000,000 tRIF   |
| Contract Testnet Address | <a href="https://explorer.testnet.rootstock.io/address/0x19f64674d8a5b4e652319f5e239efd3bc969a1fe" target="blank">0x19f64674D8a5b4e652319F5e239EFd3bc969a1FE</a> |



```

<td scope="row">Contract Type</td>
<td>ERC677</td>
</tr>
<tr>
<td scope="row">How to get</td>
<td>
<ul>
<li><a href="https://faucet.rifos.org/" target="blank">Faucet</a></li>
</ul>
</td>
</tr>
</tbody>
</table>

```

## Wallets

See supported wallets.

## Technical information

### ERC677 token standard

#### An ERC20

token transaction between a regular/non-contract address and contract are two different transactions: You should call approve on the token contract and then call transferFrom on the other contract when you want to deposit your tokens into it.

#### ERC677

simplifies this requirement and allows using the same transfer function. ERC677 tokens can be sent by calling transfer function on the token contract with no difference if the receiver is a contract or a wallet address, since there is a new way to notify the receiving contract of the transfer.

An ERC677 token transfer will be the same as an ERC20 transfer. On the other hand, if the receiver is a contract, then the ERC677 token contract will try to call tokenFallback function on receiver contract. If there is no tokenFallback function on receiver contract, the transaction will fail.

### RIF transfer methods

#### - Approve and transfer:

```

js
function approve(address spender, uint256 value) public returns (bool)
function transfer(address to, uint256 value) public returns (bool)

```

#### - Transfer and call:

```

js
function transfer(address to, uint256 value, bytes data)

```

#### Parameters

- to: address: Contract address.
- value: uint256: Amount of RIF tokens to send.
- data: bytes: 4-byte signature of the function to be executed, followed by the function parameters to be executed with encoded as a byte array.

# index.md:

---

sidebarposition: 1

title: Developers Overview

sidebarlabel: Overview

tags: [rsk, rootstock, developers, web3, starter kits, smart contracts, how-tos,]

description: "Leverage your existing knowledge of Solidity and tools like Rust, Hardhat, and Wagmi to deploy and scale your dApps on the pioneering layer 2 solution that combines the best of Bitcoin security and Ethereum Smart Contract capabilities."

---

Welcome to the Rootstock Developers Overview section.

This section enables developers getting started with the Rootstock blockchain. Developers can install a local development environment using Hardhat, etc, create and test contracts with the libraries provided and use the libraries to build decentralized applications

:::info[Info]

Looking to quickly test your dApp on testnet before deploying to mainnet? Use the RPC API or view the json-rpc methods available on the RPC API.

Dive right in with step-by-step guides to get your development environment set up and deploy your first dApp.

:::

Navigating the Developer Section

| Resource                   | Description   |
|----------------------------|---|
| -----                      | -----   |
| Hardware Requirements      | Set up your local environment.  |
| Blockchain Essentials      | Rootstock Blockchain Essentials.  |
| Quick Starts               | Dive right in with step-by-step guides to get your development environment set up and deploy your first dApp.             |
| Smart Contract Development | Explore in-depth resources on building secure and scalable smart contracts on Rootstock.                                  |
| Integration Guides         | Deepen your knowledge with detailed guides and informative tutorials that cover various aspects of Rootstock development. |
| JSON-RPC                   | Test your dApps on testnet in minutes before deploying to mainnet using the RPC API.                                      |
| Libraries                  | Access essential tools and libraries to streamline your development process.  |

# index.md:

---

sidebarlabel: Development Prerequisites

sidebarposition: 2

title: Prerequisites

tags: [rsk, rootstock, prerequisites, setup, requirements]

description: "Minimum hardware requirements for Rootstock."

---

This guide provides clear instructions for developers on the supported Solidity versions and the necessary configurations needed to ensure your smart contracts are deployed on the Rootstock network. See the developer tools section for a list of tools to build on Rootstock.

### Solidity Version

- Supported solc version: 0.8.19

### Node RPC

- Interact with Rootstock using the RPC API

:::tip[Get an API Key]

See how to setup the RPC API and get an API Key.

...

### Network Configuration

Fill these values to connect to the Rootstock Mainnet or Testnet.

| Field              | Rootstock Mainnet   | Rootstock Testnet  |
|--------------------|---|--|
| Network Name       | Rootstock Mainnet   | Rootstock Testnet  |
| RPC URL            | <a href="https://rpc.mainnet.rootstock.io/{YOURAPIKEY}">https://rpc.mainnet.rootstock.io/{YOURAPIKEY}</a> | <a href="https://rpc.testnet.rootstock.io/{YOURAPIKEY}">https://rpc.testnet.rootstock.io/{YOURAPIKEY}</a>  |
| ChainID            | 30  | 31   |
| Symbol             | RBTC  | tRBTC  |
| Block explorer URL | <a href="https://explorer.rootstock.io/" target="blank">https://explorer.rootstock.io/</a>                | <a href="https://explorer.testnet.rootstock.io/" target="blank">https://explorer.testnet.rootstock.io/</a> |

</table>

## Contract Addresses

See the List of Contract Addresses on Rootstock

## Derivation path

When using BIP-44-compatible wallet software, you will need to specify a derivation path.

text

Mainnet: m/44'/137'/0'/0/N

Testnet: m/44'/37310'/0'/0/N

:::info[Info]

See Account based addresses section for more information or how to verify address ownership.

:::

## Install Hardhat

bash

npm install --save-dev hardhat

:::tip[Recommended]

- Install hh autocomplete to use hh shorthand globally.

bash

npm i -g hardhat-shorthand

- Use the Hardhat Starter Kit

- Learn how to write, interact, deploy, and test smart contracts on Rootstock using Hardhat or Foundry.

:::

## Command Line Tools

### POSIX Compliant Shell

<Tabs>

<TabItem value="windows" label="Windows">

Standard terminals like cmd or PowerShell may not support some commands. We recommended installing Git for Windows for Git Bash, which provides a more UNIX-like experience. Here's a tutorial on Git Bash.

</TabItem>

<TabItem value="macos" label="MacOS/Linux">

Standard terminal.

</TabItem>

</Tabs>

## Installing Node.js and NPM

```

<Tabs>
<TabItem value="nvm" label="NVM" default>
  - Node v18 or later.
    - For installation, use NVM install script.
</TabItem>
<TabItem value="windows" label="Windows">
  1. Download the Node.js Installer from Node.js Downloads.
  2. Run the installer and follow the on-screen instructions.
  3. Open Command Prompt or PowerShell and check versions with node -v and npm -v.
    - See Posix Compliant Shell.
</TabItem>
<TabItem value="macos" label="MacOS">
  1. Install Homebrew (if not installed):
    bash
    /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)

  2. Install Node.js and npm with brew install node
  3. Check versions in Terminal with node -v and npm -v
</TabItem>
<TabItem value="linux" label="Linux">
  1. Open a terminal.
  2. Update package manager with sudo apt update
  3. Install Node.js and npm with sudo apt install nodejs npm
  4. Check versions in the terminal with node -v and npm -v
</TabItem>
</Tabs>

```

## Optional Setup

- Foundry
- Remix

# index.md:

```

---
sidebarlabel: Blockchain Essentials
sidebarposition: 3
title: "Rootstock Blockchain Essentials"
tags: [quick-starts, rsk, rootstock, blockchain, browser wallets, developers, beginners]
description: "Learn how to interact with Rootstock in your web browser, how to look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network."
---

```

Learn about Rootstock, how it enables smart contract on Bitcoin, and its compatibility with Ethereum and other platforms.

| Title   | Description |
|---|-------------|
| -----   |             |
| -----   |             |
| Rootstock Overview   Learn about Rootstock and its Ecosystem.   |             |
| Using Rootstock in the Browser   Learn how to interact with Rootstock in your web browser, how to look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network. |             |
| Exploring Rootstock Transactions   Learn how to interact with Rootstock in your web browser, how to   |             |

look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network.|

# index.md:

---

sidebarlabel: Blockchain Overview

sidebarposition: 200

title: 'Getting Started with Rootstock (RSK) Development'

description: 'Learn how to interact with Rootstock in your web browser, how to look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network.'

tags: [quick-starts, rsk, rootstock, blockchain, browser wallets, developers, beginners]

---

Learn about Rootstock, how it enables smart contract on Bitcoin, and its compatibility with Ethereum and other platforms.

## What is Rootstock?

Rootstock's full technology stack is built on top of Bitcoin:

From Rootstock smart contracts to the Rootstock Infrastructure Framework.

The stack is designed to create a more fair and inclusive financial system.

> See The Stack

Bitcoin, is a store and transfer of value.

The blockchain is secure because miners with high infrastructure and energy costs create new blocks to be added to the blockchain every 10 minutes.

The more hashing power they provide, the more secure the network is.

Rootstock is the first open source smart contract platform that is powered by the bitcoin network.

Rootstock's goal is to add value and functionality to the bitcoin ecosystem by enabling smart-contracts, near instant payments, and higher-scalability.

RIF is an all-in-one suite of open and decentralized infrastructure applications and services that enable faster,

easier and scalable development of distributed applications (dApps) within a unified blockchain environment.

Rootstock is connected to Bitcoin in terms of how its blocks are mined,

and also in terms of a common currency.

Rootstock is also compatible with Ethereum in terms of its virtual machine (which executes smart contracts),

as well as the RPC (external API) that it exposes.

Let's briefly look at each of these areas.

## Powpeg

The second point of contact is the Powpeg.

This component connects both networks to allow the transfer of bitcoins to Rootstock, thereby allowing developers to interact with smart contracts.

They pay gas using the same bitcoin, the smart bitcoin.

```
<div class="sprite-transform-animation-wrapper rsk-peg">
```

```
  <div class="sprite-transform-animation rsk-peg"></div>
```

```
</div>
```

To do so, you send bitcoin to a special address,

where they are locked in the bitcoin network.

Next, in the same address over in the Rootstock network, that same bitcoin is released to the user for use in the Rootstock network.

This is called peg-in.

You can do the reverse operation called peg-out, by sending your bitcoin to a special address in the Rootstock network, and receiving your bitcoin back in the bitcoin network.

## Differences with Rootstock and Ethereum

Rootstock is not 100% compatible with Ethereum: It has differences in the way checksums are calculated, the derivation path it uses, and how gas is calculated.

### Checksum differences

- Different Ethereum-compatible networks differentiate themselves using “chain IDs”.
- Each blockchain network has its own unique chain ID.
- Rootstock uses the chain ID when calculating checksums for its addresses, whereas Ethereum does not take this into account.
- Checksums in both networks are represented using capitalisation (uppercase and lowercase letters), so the “same” address will not pass checksum validations on both Rootstock and Ethereum.

### Derivation path differences

Remembering or storing private keys for your crypto wallets can be super challenging, even for technical people.

This is because these keys are essentially extremely large numbers.

So to make things easier, the crypto community has come up with a technique called “HD wallets”, where using a seed phrase (a set of randomly chosen dictionary words), plus a “derivation path”. Rootstock and Ethereum have different derivation paths, therefore, the same seed phrase results in a different set of keys and addresses between Rootstock and Ethereum.

### Gas differences

The EVM and RVM are compatible in that they support the same op-codes, and therefore can run the same smart contracts.

However, the price of each op-code (measured in units known as gas) is different between EVM and RVM, thus the total gas consumed in various transactions is different.

Further to that, gas units are multiplied by gas price to calculate the transaction cost.

Since Rootstock’s gas price is denominated in RBTC and Ethereum’s gas price is denominated in Ether, there is another difference between gas prices on Rootstock and Ethereum.

### EVM Compatible Smart Contracts

If you are familiar with smart contract development or dApp development using solidity, web3, and other compatible technologies; you might be excited to know that the Rootstock Virtual Machine (RVM) is compatible with the Ethereum Virtual machine (EVM).

So you can use the same code, tools, and libraries when developing with Rootstock too.

Thus, the smart contract/dApp development skills that you’re used to will transfer across quite nicely too!

> See supported Solidity version in requirements

### Tools

- Hardhat is an Ethereum development environment designed for professionals. It's primarily used in the development of smart contracts for the Ethereum blockchain.

Refer to the Hardhat Overview for an overview of how it's used on Rootstock.

- Metamask is a browser extension cryptocurrency wallet or mobile app, enabling users to interact with the Rootstock blockchain, including sending RBTC, sending Rootstock-based tokens such as RIF, and interacting with smart contracts deployed to the Rootstock network. See how to configure MetaMask to connect to Rootstock.

- Mocha is a popular JavaScript test framework running on Node.js.

See Testing Smart Contracts to see how to use it to test your smart contracts on Rootstock.

- Solidity is the most popular programming language for implementing smart contracts.

The bytecode and ABI that the Solidity compiler, solc, outputs can be used to deploy and interact with smart contracts on Rootstock, thanks to the compatibility between RVM and EVM.

> For a comprehensive list of tools, see Dev Tools section.

## Ethereum Compatible JSON RPC

The set of remote procedure calls (RPCs) that Rootstock supports is largely the same as the RPCs supported by Ethereum.

This is another layer of compatibility, in addition to the virtual machine implementation, which allows the same tools and libraries to be used.

## Merged Mining

The bitcoin miners do what is known as merged mining, securing both networks with the same infrastructure and energy consumption.

```
<div class="sprite-transform-animation-wrapper rsk-mining">
  <div class="sprite-transform-animation rsk-mining"></div>
</div>
```

They create blocks on the bitcoin network every 10 minutes, including transfer of bitcoin from different addresses, and in the process they create new bitcoins.

On Rootstock, blocks are created every 30 seconds, to secure the execution of smart contracts.

This does not mint any new coins in the process, but does earn a reward from the merged mining.

Check out [rootstock.io/mine-btc-with-rootstock](https://rootstock.io/mine-btc-with-rootstock) to learn more about mining.

:::info[Note]

The time between blocks on each network listed above are approximate values.

:::

# index.md:

---

sidebarlabel: Using Rootstock in Browser

sidebarposition: 300

title: 'Using Rootstock with a Browser Extension'

description: 'Learn how to interact with Rootstock in your web browser, how to look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network.'

tags: [quick-starts, rsk, rootstock, blockchain, browser wallets, developers, beginners]

---



As Rootstock is a blockchain with smart contract capabilities, it is possible to build decentralised applications (dApps) with it.

Most dApps are web applications that you access with a regular Internet browser, such as Chrome. However, the blockchain interactions require some additional software, which comes in the form of browser extensions.

These browser extensions insert a web3 provider object, with the Javascript parts of the web application used to interact with the blockchain, forming an integral part of dApp architecture.

> Note that these browser extensions store your private keys,  
> and use them to sign transactions. So keep them secure.

:::note[Rootstock Wallets]

There are several browser extensions that you can use to interact with the Rootstock blockchain, this includes: MetaMask. For a full list of wallets, see the Dev Tools section.

:::

Since this is a quick start, we will not go through all of them - just MetaMask.

There are some hidden complexity that we've glossed over in the content above so you can set up and get running as quickly as possible.

If you would like to delve deeper, here are some resources that we recommend.

## Install Metamask

MetaMask is the most popular browser extension with web3 provider capabilities. It enables users to buy, store, send and swap tokens.

Metamask also equips you with a key vault, secure login, token wallet, and token exchange—everything you need to manage your digital assets.

Open up Chrome browser, and install the extension from the Chrome store.

This short video demonstrates how to download and install MetaMask on your browser, and also how to create a wallet to store your crypto assets.

```
<div class="video-container">  
  <iframe width="949" height="534" src="https://www.youtube.com/embed/VlyqXD1TjJk" frameborder="0"  
  allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"  
  allowfullscreen></iframe>  
</div>
```

## Cryptography

### Private Keys and Public Keys

In wallet software, you generally see “accounts” represented by addresses on the blockchain network. In the case of Rootstock, this is 0x followed by a series of hexadecimal characters, for example, 0xdfc0e6361fd1846a223e2d7834a5ebd441a16dd4.

There is some hidden complexity behind that, to do with cryptography, which is necessary to secure the account, and all the blockchain transactions it makes.

- You start off with a private key, which is essentially an extremely large number, and should be randomly generated.

You should keep the private key secret, because that is what is used to sign transactions.

- A public key is generated from the private key, and this is also a very large number.  
This does not need to be kept secret, because others in the blockchain network use it to verify transactions.
- An address is generated from the public key, and is the hexadecimal string that you see in your wallet software.

## Seed Phrases

When you open up MetaMask for the first time after installing it, you will be asked to initialise it using a seed phrase.

If you have done this before, you can use your own seed phrase. Otherwise, let's generate a new one!

- > To generate a new seed phrase, you will need to create a new wallet.
- > See the above steps to create a new wallet.

Most blockchain users operate one or more accounts, and it can be quite difficult to remember the value of cryptographic keys - those very large numbers - you'll need superhuman memory!  
The seed phrase is presently the most popular method used to generate, store, remember, and recover keys for crypto wallets, and is something that is approachable for the average user.

It also is the default method used by MetaMask (and many other wallets).  
In a nutshell, it takes a randomly generated sequence of dictionary words.  
The wallet then uses this sequence of words to generate not one, but multiple sets of cryptographic keys.  
This is how MetaMask is able to support multiple accounts using a single seed phrase.

This process is described in detail in the BIP-44 technical standard.  
This ensures that the way that seed phrases work is the same between multiple crypto wallets, enabling the same phrase to be portable.

## Configure custom network for Rootstock Testnet

MetaMask comes pre-configured with connections for Ethereum networks.  
Let's use its custom networks feature to add a connection to an Rootstock network.

```
<div class="video-container">  
  <iframe width="949" height="534" src="https://www.youtube.com/embed/VyPewQoWhn0"  
  frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"  
  allowfullscreen></iframe>  
</div>
```

After creating the custom network for the Rootstock Testnet, you should be able to interact with smart contracts deployed on the Rootstock Testnet!  
You should also see your balances in tRBTC (Testnet RBTC).  
This is currently zero, which means that we cannot send any transactions to the blockchain, so let's get some using the RBTC faucet.

```
<div class="video-container">  
  <iframe width="949" height="534" src="https://www.youtube.com/embed/twfK8Rd5hak" frameborder="0"  
  allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"  
  allowfullscreen></iframe>  
</div>
```

Now you should have a balance of tRBTC, and you will be able to send transactions on the Rootstock Testnet!

## Configure Custom Token for tRIF

The Rootstock Infrastructure Framework (RIF) includes multiple services for decentralised applications. These services may be paid for using the RIF token. Let's configure MetaMask to be aware of the RIF token. We'll use tRIF as the token symbol, since we're on the Rootstock Testnet.

```
<div class="video-container">
  <iframe width="949" height="534" src="https://www.youtube.com/embed/QCabRPfr2Zs"
frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
</div>
```

Now that MetaMask has the RIF token configured, let's get some test tokens using the RIF faucet.

```
<div class="video-container">
  <iframe width="949" height="534" src="https://www.youtube.com/embed/ttb8EOTWey8"
frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
</div>
```

Now you should have a balance of tRIF, and you will be able to use RIF services on the Rootstock Testnet!

### Further Reading

- How to configure Metamask
- Account based addresses on Rootstock
- About the RIF token
- About the RBTC cryptocurrency
- About Gas
- About RIF Services
- About BIP-44
- About EIP-20
- Asymmetric Key Generation

# index.md:

```
---
title: 'Exploring Rootstock Transactions'
sidebarposition: 400
sidebarlabel: Rootstock Transactions
description: 'Learn how to interact with Rootstock in your web browser, how to look at Rootstock transactions, develop and deploy your very first smart contract to the Rootstock network.'
tags: [quick-starts, rsk, rootstock, blockchain, browser wallets, developers, beginners]
---
```

In the previous section, we set up a browser extension that is a crypto wallet, MetaMask. We connected to the Rootstock Testnet, and loaded this up with Rootstock's cryptocurrency, RBTC, and an Rootstock-based token, RIF.

> Note, if you are yet to do the above, we encourage you to go back and complete that step first. See: Using Rootstock in the browser.

## Block Explorer

Now that we are set up, let's explore some transactions!

The Rootstock network is an immutable public ledger.

Let's dissect that phrase:

- Ledger: An ordered list of transactions recorded in some form
- Immutable: The way this ledger is recorded and stored means that any existing transactions may not be deleted or modified. You may also think of it as being an "append-only" ledger.
- Public: The contents of this ledger are open and transparent, therefore anyone connected to this network can view every single transaction in history.

This is where block explorers come in.

They are a special type of software that connect to a blockchain network, and display the data from this immutable public ledger.

Since it is open and transparent, there is nothing stopping multiple block explorers from displaying the data in a single blockchain. This is certainly true for Rootstock, and there are multiple block explorers. We'll use the canonical one here, however, feel free to use other block explorers too!

View account in the block explorer

Watch this short video demonstrating how to view an account in the block explorer.

```
<div class="video-container">  
  <iframe width="949" height="534" src="https://www.youtube.com/embed/p-q7NBmEqBo"  
  frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"  
  allowfullscreen></iframe>  
</div>
```

For the Rootstock Mainnet, we would go to [explorer.rootstock.io](https://explorer.rootstock.io).

However, since we are currently connected to the Rootstock Testnet, we go to [explorer.testnet.rootstock.io](https://explorer.testnet.rootstock.io) instead.

## Transfer tRBTC

So far, you have not made any transactions from your address. The transactions that you see when you view the address in the block explorer were made from other addresses (in this case, a couple of Testnet faucets). Now, it's time for you to initiate your own transactions!

Watch this short video demonstrating how to transfer tRBTC from one account to another.

```
<div class="video-container">  
  <iframe width="949" height="534" src="https://www.youtube.com/embed/fMdiNeDLKo0"  
  frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"  
  allowfullscreen></iframe>  
</div>
```

We'll start by transferring cryptocurrency from your address, back to the faucet's address.

## Transfer tRIF

Watch this short video demonstrating how to transfer tRIF from one account to another.

```
<div class="video-container">
```

<iframe width="949" height="534" src="https://www.youtube.com/embed/ncCzQnnMVR8" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>  
</div>

## RBTC Balance Decrease

You may have noticed that when you sent tRBTC, the tRBTC balance decreased by slightly more than the amount that you sent. You may also have noticed that when you sent tRIF, the tRBTC balance also decreased by a small amount, even though only tRIF were sent in that transaction.

You would have seen this in the transaction confirmation screens when you confirmed each transaction.

This is not an error, it is simply a fundamental aspect of how blockchain networks function - any time you add a transaction to the blockchain, you must pay the network a fee to compensate them for their computational costs.

## View Transactions

When you performed each of the transactions, you should have received notifications in popups.

However, if you missed this, not to worry, you can also find this within the transaction history within MetaMask. To do so, within the main screen of MetaMask, click on the “Activity” tab. You’ll see the list of the transactions.

Then you click on any transaction, and click on the arrow button beside copy button named transaction ID, this takes you to the Testnet explorer

If you clicked on the popup notification, or if you find it within the “Activity” tab, either way, this should open up the block explorer with the selected transaction selected.

For the transaction of the tRBTC transfer, you should see this

You will notice that this transaction has an amount.

For the transaction of the tRIF transfer, you should see this

You will notice that this transaction has a zero amount, but it does emit some events, which is because the smart contract of the RIF token does this.

## View Network Stats

So far we have checked out individual addresses and transactions. These are very detailed and specific information. What if you were after the big picture instead? A bird’s eye view of the Rootstock blockchain as a whole?

For this, we will not use the Rootstock Block explorer, and instead use the Rootstock Stats page.

## Rootstock Stats

]

Here, we can see some very important numbers such as the average block duration, and the merged mining hash rate - and several other important technical indicators of the Rootstock network.

A key indicator to look for is the average block time, which should be approximately 33s. Another key indicator to look for is the percentage of the Bitcoin network’s hash rate that is merge mining Rootstock.

# hardhat.md:

---

sidebarlabel: Using Hardhat

sidebarposition: 400

title: Rootstock Hardhat Starter dApp

description: 'Whether you are a seasoned developer or just starting your journey into smart contract development, the hardhat starter kit provides a solid foundation for building decentralized applications (dApps) on the Rootstock network.'

tags: [rsk, rootstock, tutorials, developers, hardhat, quick starts, dApps, smart contracts]

---

Whether you're a seasoned developer or just starting your journey into smart contract development, the hardhat starter kit provides a solid foundation for building decentralized applications (dApps) on the Rootstock network.

Rootstock is fully compatible with Ethereum. It brings the power of smart contracts to Bitcoin, allowing developers to leverage Bitcoin's security while benefiting from Ethereum's ecosystem.

## Prerequisites

Before starting the dApp, make sure to have the following prerequisites:

### 1. Familiarity with Smart Contracts:

- If you're new to smart contracts, consider learning the basics. Understanding how smart contracts work will enhance your experience with Rootstock development.

### 2. Node.js and Hardhat Installed:

- Ensure you have Node.js installed on your system. See the prerequisites section.

### 3. Install Hardhat Shorthand:

- We recommend installing hh autocomplete to use hh shorthand globally.

```
bash
```

```
npm i -g hardhat-shorthand
```

- For more details, refer to their official guide.

### 4. Metamask Setup with Rootstock:

- Install the Metamask browser extension if you haven't already.

- Configure Metamask to connect to the Rootstock network. Visit the MetaMask Integration on the Rootstock Dev Portal.

### 5. Basic Knowledge of Hardhat:

- Familiarity with Hardhat's core concepts and functionalities is recommended. If you're new to Hardhat, refer to the Rootstock Hardhat Guide.

:::tip[Rootstock Blockchain Developer Course]

Learn how to write, test, secure, deploy and verify smart contracts on the Rootstock blockchain network. Enroll for the Rootstock Blockchain Developer Course.

...

## Setting Up the Sample dApp

## Clone the Repository

Open your terminal or command prompt and run the following command to clone the repository from GitHub:

```
bash
git clone https://github.com/rsksmart/rootstock-hardhat-starterkit.git
```

## Install Dependencies

Navigate to the cloned repository folder:

```
bash
cd rootstock-hardhat-starterkit
```

Install all required dependencies using npm:

```
bash
npm install
```

## Obtain Rootstock Testnet and Mainnet RPC URLs

This section will walk you through adding Rootstock Testnet and Mainnet RPC URLs to your development environment. These URLs are essential for connecting your application to the Rootstock network and interacting with smart contracts.

There are two ways to obtain RPC URLs:

### Using Public RPC URLs

- Visit the MetaMask Integration on the Rootstock Dev Portal. This guide provides instructions on setting up MetaMask for Rootstock. While following these steps, pay close attention to the sections on adding custom networks. You'll find the RPC URLs for Rootstock Testnet and Mainnet listed.

### Using RPC API

- Create an account at the Rootstock RPC API. Once logged in, navigate to your dashboard and copy the API Key.

## Adding the URLs to your project

After obtaining the RPC URLs, create a file named `.env` in your project's root directory (important: this file should not be committed to version control). Add the necessary environment variables to the `.env` file:

`PRIVATEKEY`: Your private key (e.g., from your Metamask account details).

`RSKMAINNETRPCURL`: The RPC URL for the Rootstock mainnet.

`RSKTESTNETRPCURL`: The RPC URL for the Rootstock testnet.

## Deploying an ERC721 Token Contract

This section uses the Hardhat development framework to deploy an ERC721 token (a non-fungible token) on the Rootstock network.

Run the following command, replacing <network> with either rskTestnet or rskMainnet depending on your desired deployment environment:

```
bash
hh deploy --network <network> --tags 721
```

Example command:

```
bash
hh deploy --network rskTestnet --tags 721
```

This command will compile your Solidity contracts, generate type information, and deploy your ERC721 contract to the specified Rootstock network. The output will display the deployed contract address and the amount of gas used.

The above command will return an output similar to the following:

```
bash
Generating typings for: 36 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 106 typings!
Compiled 34 Solidity files successfully (evm target: paris).
deploying "MockERC721" (tx:
0x9ad1dbc047b78594cf2cad105ded54c851fc0895ae69e4381908feceddd0ee3fc)...: deployed at
0x2E027a3a05f3de6777B23397a50a60ecd04fe34C with 2849621 gas
```

### Interacting with the Contract - Minting a Token

On contract deployment, you can interact with it using Hardhat's `erc721-mint` command. This command allows you to mint (create) new ERC721 tokens.

#### Minting a Token:

In your terminal, run the following command, replacing the placeholders with actual values:

```
bash
hh erc721-mint \
  --contract <ContractAddress> \
  --recipient <RecipientAddress> \
  --network rskTestnet
```

Example command:

```
bash
hh erc721-mint --contract 0x2E027a3a05f3de6777B23397a50a60ecd04fe34C --recipient
0xB0f22816750851D18aD9bd54c32C5e09D1940F7d --network rskTestnet
```

- <ContractAddress>: Replace this with the address of your deployed ERC721 contract obtained from the previous step.
- <RecipientAddress>: Replace this with the wallet address to receive the newly minted token.
- <network>: Replace this with either rskTestnet or rskMainnet, depending on the network where your contract is deployed.

This command will initiate a transaction to mint a new ERC721 tokens and send it to the specified recipient address.

The output will display the transaction details:



bash

Transaction Hash: 0xa127ff008e20d8b3944cecb374f28535cd84555881cde157708ec5545603a4e4

Transaction confirmed

# index.md:

---

sidebarlabel: Quick Starts

sidebarposition: 4

title: Quick Starts

tags: [rsk, rootstock, beginner, quick starts, developers, advanced, port to rootstock, tutorials]

description: "Quick starts, demos and starter kits to develop on Rootstock."

---

```
<Filter
values=[
  {label: 'Beginner', value: 'beginner'},
  {label: 'Advanced', value: 'advanced'},
  {label: 'Hardhat', value: 'hardhat'},
  {label: 'Wagmi', value: 'wagmi'},
  {label: 'Smart Contracts', value: 'sc'},
  {label: 'On-chain data', value: 'data'},
  {label: 'Port to Rootstock', value: 'port-dapps'}
]>
<FilterItem
  value="wagmi, sc, beginner"
  title="Wagmi Starter Kit"
  subtitle="quickstart"
  color="orange"
  linkHref="/developers/quickstart/wagmi/"
  linkTitle="Use the Kit"
  description="This starter kit provides a foundation for building decentralized applications (dApps) on the
Rootstock blockchain using React, Wagmi and Shadcn libraries."
/>
<FilterItem
  value="hardhat, sc, beginner"
  title="Hardhat Starter Kit"
  subtitle="quickstart"
  color="orange"
  linkHref="/developers/quickstart/hardhat/"
  linkTitle="Use the Kit"
  description="Smart Contract examples, Tests, Deployments and Tasks for Common ERC Standards
(ERC20, ERC721, ERC1155)."
/>
<FilterItem
  value="wagmi, sc, advanced"
  title="Account Abstraction Kit"
  subtitle="quickstart"
  color="orange"
  linkHref="/developers/quickstart/rootstock-etherspot/"
  linkTitle="Use the Kit"
  description="Account Abstraction Starter dApp using Etherspot."
/>
<FilterItem
```

```

    value="sc, advanced"
    title="dApp Automation with Cucumber"
    subtitle="quickstart"
    color="orange"
    linkHref="/resources/tutorials/dapp-automation-cucumber/"
    linkTitle="Automate dApps"
    description="Learn how to automate dApps using Cucumber Agile Automation Framework."
  />
<FilterItem
  value="sc, advanced"
  title="RIF Relay Starter Kit"
  subtitle="quickstart"
  color="orange"
  linkHref="/developers/integrate/rif-relay/sample-dapp/"
  linkTitle="Use Kit"
  description="Starter kit to develop on RIF Relay."
/>
<FilterItem
  value="sc, data, advanced"
  title="Get Started with The Graph"
  subtitle="quickstart"
  color="orange"
  linkHref="/dev-tools/thegraph/"
  linkTitle="Get Started"
  description="Easily query on-chain data through a decentralized network of indexers"
/>
<FilterItem
  value="sc, beginner"
  title="Get Started with Web3.py"
  subtitle="Web3.py"
  color="orange"
  linkHref="/developers/quickstart/web3-python/"
  linkTitle="Get Started"
  description="Get started with deploying and interacting with smart contracts on Rootstock using Web3.py."
/>
<FilterItem
  value="sc, beginner, advanced, port-dapps"
  title="Port an Ethereum dApp to Rootstock"
  subtitle="Port dApps"
  color="orange"
  linkHref="/resources/port-to-rootstock/ethereum-dapp"
  linkTitle="Get Started"
  description="Learn how to port an Ethereum dApp to Rootstock."
/>
</Filter>

```

# rootstock-etherspot.md:

---

sidebarlabel: Account Abstraction

sidebarposition: 500

title: Account Abstraction using Etherspot Prime SDK

description: 'In this guide, you will learn how to use the Etherspot Prime SDK to deploy an Account

Abstraction dApp on the Rootstock network. By following these steps, you will empower your users to interact with your dApp without managing private keys directly.'

tags: [rsk, rootstock, developers, quick starts, etherspot, dApps, account abstraction]

---

In this guide, you will learn how to use the Etherspot Prime SDK to deploy an Account Abstraction dApp on the Rootstock network.

By following these steps, you'll empower your users to interact with your dApp without managing private keys directly.

## Prerequisites

- Ensure git is installed
- Basic understanding of React and JavaScript
- Node.js and npm (or yarn) installed on your machine
- A code editor of your choice (e.g., Visual Studio Code)
- Familiarity with the Wagmi starter kit

:::info[Info]

This guide assumes you have a Wagmi starter kit already set up.

:::

## Understanding Account Abstraction

Abstraction involves hiding unnecessary data about an "object" to simplify the system and improve efficiency. When applied to Ethereum's blockchain technology, Account Abstraction aims to create a single account type that includes only relevant aspects.

There are two main types of Ethereum accounts: User Accounts (EOA) and Contracts. User Accounts are designed for individuals and are controlled by private keys. These accounts, also known as externally owned accounts (EOA), can hold a balance in Ether and conduct transactions with other EOAs using Ether and other ERC-supported tokens.

On the other hand, Contracts are controlled by code and can perform various functions, including interacting with external accounts and initiating activities such as exchanging tokens or creating new contracts.

With account abstraction, a single account can hold both code and Ether, enabling it to execute transactions and smart contract functions. This eliminates the need for a separate EOA to manage transactions, allowing contracts to handle funds directly.

Etherspot Prime, an open-source SDK, simplifies the implementation of Account Abstraction for dApp developers. Using an Etherspot smart wallet, users can enjoy a seamless web2-like experience through social logins or transaction batching.

## Getting Started

To explore Account Abstraction with Etherspot, follow these steps:

Using a Different Branch:

1. Clone the Wagmi starter kit repository:

```
sh
git clone https://github.com/wagmi-dev/wagmi-starter-kit.git
```

2. Navigate to the project directory:

```
javascript  
cd wagmi-starter-kit
```

3. Instead of using the main branch, switch to the branch containing the Account Abstraction functionalities:

```
javascript  
git checkout aa-sdk
```

4. Run the project:

Now that you've cloned the repository and installed dependencies, it's time to run the project. Execute the following command:

```
javascript  
yarn dev
```

This will start your Rootstock Wagmi dApp locally, allowing you to develop and test your smart contracts. You can access the Vite server at <http://localhost:5173>.

Interact with Account abstraction



1. Generate a Random Account:

- Click the "Generate" button to create a random account.

2. Generate a Payment Address:

- Click the "Generate" button to obtain a payment address.

3. Check Account Balance:

- Clicking the Get Balance will show the balance of the payment address.

4. Estimate and Send a Transaction:

- This section has two fields:

- Receipt Address: This field is where you specify the recipient's Ethereum address. It's the address where you want to send the transaction. Think of it as the destination for your funds. Make sure you enter a valid Ethereum address here.

- Value (in Eth): In this field, you indicate the amount of Ether (ETH) you want to send in the transaction. Enter the value you wish to transfer. For example, if you want to send 0.5 ETH, input "0.5" in this field.

- Click the "Estimate and Send" button to initiate the transaction.

Understanding the codebase

This code defines a React component named Demo, which provides a user interface for interacting with blockchain functionalities through the Etherspot SDK.

The component allows users to generate a random externally owned account (EOA), generate an Etherspot wallet, check the balance of the Etherspot wallet, and estimate and send transactions using the Arka Paymaster.

The component handles various states and interactions, making it easier to manage wallets and perform blockchain transactions without directly dealing with private keys.

### 1. generateRandomEOA

- This function generates a random externally owned account (EOA).

- Function:

This Asynchronously generates a private key and derives an account address from it, setting the EOA wallet address and private key state variables.

### 2. getBalance

- This function fetches the balance of the current Etherspot wallet.

- Function:

This Asynchronously uses the SDK to retrieve the native balance of the account and updates the balance state variable.

### 3. generateEtherspotWallet

- This function generates a counterfactual address for the Etherspot wallet.

- Function:

This Asynchronously interacts with the SDK to generate an Etherspot wallet address and fetches its balance.

### 4. estimateAndTransfer

- This function estimates the transaction cost and sends a specified value to a recipient using the Arka Paymaster.

- Function:

This Validates recipient address and value inputs.

Uses the SDK to set up the transaction, estimate the gas cost, send the transaction, and waits for the transaction receipt.

### 5. useEffect Hook

- This hook initializes the Prime SDK when the EOA private key is set.

Parameters:

eoaPrivateKey: The private key of the externally owned account (EOA).

- Function:

useEffect:

Sets up the Prime SDK instance with the eoaPrivateKey.

Configures the SDK with the specified bundler provider.

## Resources

- Rootstock Account Abstraction Starter Kit
- Using Prime SDK Examples
- Etherspot Prime SDK Repo

# wagmi.md:

---

sidebarlabel: Using Wagmi and React Hooks

sidebarposition: 300

title: Rootstock Wagmi Starter dApp

description: 'The Rootstock Wagmi Starter Kit provides a solid foundation for developing decentralized applications (dApps) on the Rootstock blockchain. It streamlines development by leveraging the React, Wagmi, and Shadcn libraries.'

tags: [rsk, rootstock, developers, wagmi, quickstart, dApps, Smart Contracts]

---

The Rootstock Wagmi starter kit provides a foundation for building decentralized applications (dApps) on the Rootstock blockchain.

It leverages the security of Bitcoin and the flexibility of Ethereum.

The kit uses Wagmi, a React Hooks library, to simplify smart contracts and blockchain network interactions and and Shadcn libraries.

> This starter kit is designed to help developers jump-start their dApp development journey on Rootstock.

### Prerequisites

- Node.js and Git: Ensure to have Node.js and Git installed on your system.
  - See the Prerequisites section for how to download Node.js using NVM.
- Yarn: Install Yarn, a package manager for Node.js projects. You can do this by running the following command in your terminal:

```
bash
npm install -g yarn
```

- Basic Knowledge:
  - React (a JavaScript library for building user interfaces)
  - Solidity (a programming language for Ethereum smart contracts).

:::tip[Rootstock Blockchain Developer Course]

Learn how to write, test, secure, deploy and verify smart contracts on the Rootstock blockchain network. Enroll for the Rootstock Blockchain Developer Course.

...

## Setup

### 1. Clone the Repository

First, you'll need to clone the Rootstock Wagmi Starter Kit repository. Open your terminal and run the following commands:

```
bash
git clone https://github.com/rsksmart/rsk-wagmi-starter-kit
cd rsk-wagmi-starter-kit
```

## 2. Get Project ID

Every dApp that relies on WalletConnect now needs to obtain a projectId from WalletConnect Cloud. This is free and only takes few minutes.

To get the key:

1. Go to Walletconnect and sign up.
2. Create a new project by clicking on Create Project.
3. Add a Name and Link to your project, select a product (AppKit or WalletKit), select WalletKit.
4. Now you will see the project ID, copy it.

## 3. Environment Setup

To set up your environment, follow these steps:

1. Create a .env file and add environment variables.

text

VITEWCPROJECTID=Your projectid from cloud Walletconnect

## 4. Install Dependencies

Before running the project, make sure to have the necessary dependencies installed. We recommend using the yarn package manager due to potential conflicts with npm packages. Run the following command to install dependencies:

bash

yarn

## 5. Run the Project

Now that you've cloned the repository and installed dependencies, it's time to run the project. Execute the following command:

bash

yarn dev

This will start the Rootstock Wagmi Starter dApp locally, allowing you to develop and test your smart contracts. You can access the Vite server at <http://localhost:5173>.

## Result



:::info[Info]

After successfully running your project using the command above, do the following:

- Click the "Connect Wallet" button to log in. Once connected, you can:
- Switch Networks: Easily switch between Mainnet and Testnet.
- View and Copy Your Address: Access your wallet address.
- Check Your tRBTC Balance: See your tRBTC balance.
- Disconnect: Log out from the project.

:::

## Test Project

To test the Wagmi project, follow these simple steps:

1. Connect Your Wallet: Click the "Connect Wallet" button.

2. Navigate to the Wagmi Section: Scroll down and find the card labeled “Contract Interaction with Wagmi Starter Kit.” Click on it.
3. Explore the Tabs: In the Wagmi section, you’ll see three tabs: ERC-20, ERC-721, and ERC-1155. Click on any of these tabs to explore further.



Provide a section on the document about how the contracts are called in the code, folder structure for ease of finding codes and the main features from Wagmi and Rainbowkit used in the codebase.

## Understanding the Codebase

### Folder Structure

Public  
Src  
.env  
.env.example

The src folder is organized to streamline the development process and make it easy to locate specific code or assets. Here’s a detailed breakdown:

#### .src Folder Structure

- Assets: Contains the ABIs (Application Binary Interfaces) for ERC20, ERC721, and ERC1155.
- Components:
  - AccountAbstraction: Contains code related to account abstraction.
  - Home: Holds components specific to the homepage.
  - Icons: Contains various icon components.
  - Tokens: Includes components for different token types.
  - UI: General UI components used across the application.
  - Footers.tsx: Footer component.
  - Navbar.tsx: Navbar component.
- Config:
  - provider.tsx: Configuration for providers.
  - rainbowkitConfig.ts: Configuration for RainbowKit.
  - wagmiProviderConfig.ts: Configuration for WAGMI providers.
- Lib: Contains various utility folders for easy organization:
  - Constants: Application constants.
  - Functions: General functions used across the app.
  - Types: Type definitions.
  - Utils: Utility functions.
- Pages:
  - index.ts: Main entry point.
  - Etherspot.tsx: Page component for Etherspot.
  - Home.tsx: Homepage component.
  - Wagmi.tsx: Wagmi-related page component.

#### Code for ERC20, ERC721, and ERC1155 Tabs

The code responsible for the tabs corresponding to ERC20, ERC721, and ERC1155 can be found within the components folder:

- ERC20: Located in the components/tokens/ERC20 directory.
- ERC721: Located in the components/tokens/ERC721 directory.



- ERC1155: Located in the components/tokens/ERC1155 directory.

This structured approach ensures that code and assets are logically grouped, facilitating ease of navigation and maintainability.

### Understanding the ERC20 Tab Code

The code interacts with a smart contract to mint tRSK tokens. Here's a detailed breakdown of how this is achieved:

#### 1. Smart Contract Reference:

- Address: The smart contract's address is specified by the ERC20ADDRESS constant.
- ABI: The contract's ABI (Application Binary Interface), which defines the contract functions and their parameters, is provided by the abi constant.

#### 2. Reading Contract Data:

```
javascript
const { data, isLoading, isError, refetch } = useReadContract({
  abi,
  address: ERC20ADDRESS,
  functionName: "balanceOf",
  args: [address],
});
```

#### 3. Writing to the Contract:

The useWriteContract hook from the wagmi library is used to interact with the contract's write functions (functions that modify the state).

#### 4. Minting Tokens:

The mintTokens function calls writeContractAsync to mint tRSK tokens.

##### - Arguments:

- abi: Defines the contract functions and their parameters.
- address: The address of the deployed ERC-20 contract.
- functionName: The name of the function to call, which is "mint" in this case.
- args: An array containing the user's wallet address and the amount to mint (100 in this case).

```
javascript
const mintTokens = async () => {
  setLoading(true);
  try {
    const txHash = await writeContractAsync({
      abi,
      address: ERC20ADDRESS,
      functionName: "mint",
      args: [address, 100],
    });
    await waitForTransactionReceipt(rainbowkitConfig, {
      confirmations: 1,
      hash: txHash,
    });
    setLoading(false);
    toast({
      title: "Successfully minted tRSK tokens",
    });
  } catch (error) {
    console.error(error);
  }
};
```

```

        description: "Refresh the page to see changes",
    });
    refetch();
  } catch (e) {
    toast({
      title: "Error",
      description: "Failed to mint tRSK tokens",
      variant: "destructive",
    });
    setLoading(false);
    console.error(e);
  }
};

```

This sends a transaction to the blockchain to execute the "mint" function on the smart contract, thereby minting tRSK tokens and depositing them into the user's wallet.

## Understanding the ERC721 Tab Code

This code defines a React component named ERC721Tab, which provides a user interface for interacting with an ERC-721 smart contract.

The Key Functions Within This Component:

### 1. useReadContract:

This hook is used to read data from the ERC-721 contract. It fetches the balance of NFTs held by the connected user's address.

- Parameters:

- abi: The ABI (Application Binary Interface) of the ERC-721 contract.
- address: The address of the ERC-721 contract.
- functionName: The name of the function to call on the contract (balanceOf).
- args: The arguments to pass to the contract function ([address]).

### 2. useWriteContract:

This hook is used to write data to the ERC-721 contract, specifically to mint a new NFT.

Function:

- writeContractAsync: Asynchronously writes to the contract by calling the safeMint function of the ERC-721 contract.

### 3. mintNFT:

This is an asynchronous function that handles the minting process of a new NFT.

- Steps:

- Sets the loading state to true.
- Attempts to call the safeMint function on the ERC-721 contract using writeContractAsync.
- Waits for the transaction to be confirmed using waitForTransactionReceipt.
- Displays a success toast message if the minting is successful.
- Refetches the user's NFT balance by calling refetch.
- Catches any errors, logs them, and displays an error toast message.
- Sets the loading state to false.

#### 4. refetch:

This function is part of the useReadContract hook and is used to refresh the balance of NFTs after a successful minting operation.

#### 5. toast:

This function is used to display toast notifications for success or error messages.

The rest of the component contains JSX to render the UI elements, including a button to mint the NFT, a balance display, and a link to view the minted NFTs on a block explorer.

### Understanding the ERC1155 Tab Code

The code provided is a React component that interacts with a smart contract using the ERC-1155 standard. It allows users to mint tokens and check their balances.

#### The Key Functions Within This Component:

##### 1. ERC1155Tab Component:

###### State Variables:

- loading: Boolean to manage the loading state during token minting.
- value: Number to store the selected token type for minting.
- address: The user's wallet address obtained from the useAccount hook.

##### 2. useReadContract Hooks:

These hooks are used to read data from the smart contract.

- useReadContract for checking the balance of Type A tokens (with ID 1).
- useReadContract for checking the balance of Type B tokens (with ID 2).

##### 3. mintTokens Function:

An asynchronous function that handles the minting of tokens.

- Steps:
  - Calls writeContractAsync to interact with the smart contract and mint tokens.
  - Waits for the transaction receipt using waitForTransactionReceipt.
  - Displays success or error toasts based on the outcome.
  - Refetches the balance data after minting.

### Join the Community

Building dApps can be challenging, but you're not alone.

Join the Rootstock Discord community for help, questions, and collaboration.

# web3-python.md:

---

sidebarlabel: Using Web3.py

sidebarposition: 200

title: Deploy and Interact with a Smart Contract using Web3.py

description: 'Deploy and Interact with a Smart Contract Using Web3.py.'

tags: [quick starts, rsk, rootstock, ethereum, python, web3.py, developers, smart contracts]

---

Web3.py is a Python library that allows developers to interact with Ethereum-based blockchains with Python. Rootstock has an Ethereum-like API available that is fully compatible with Ethereum-style JSON-RPC invocations. Therefore, developers can leverage this compatibility and use the Web3.py library to interact with Rootstock similar to how developers interact with smart contracts on Ethereum.

In this guide, you'll learn how to use the Web3.py library to deploy and interact with smart contracts on Rootstock.

:::tip[Interact with Rootstock using Rust]

See tutorial on how to interact with Rootstock using Rust

:::

## Prerequisites

- A testnet account with tRBTC funds.
  - Get tRBTC.
- An API KEY from the Rootstock RPC Service.
- Set up the project
- A Solidity Compiler installed -> see solidity compiler installation instructions

Set up the project and install dependencies:

```
bash
create a directory for the project
mkdir web3-python-guide && cd web3-python-guide
```

```
install python 3.10
brew install python@3.10
```

```
set up the development virtual environment
python3.10 -m venv env
source env/bin/activate
```

```
install dependencies
pip install Web3 py-solc-x
```

Solidity compiler installation instructions for MacOS:

```
bash
brew install solc-select
solc-select use 0.8.19 --always-install
```

```
solc --version
Version: 0.8.19+commit.7dd6d404.Darwin.appleclang
```

## Set Up Secrets for the Project

We will be using sensitive data that doesn't have to be stored in the code, and instead we will store them in a .env file.

For that, first lets install the package to read data from the .env file:

```
python
pip install python-dotenv
```

Then, we will create a .env file and add the secrets:

```
bash
touch .env
```

add the following variables to the file:

Replace YOURAPIKEY with the API key from your dashboard.

```
bash
get this YOURAPIKEY from the Rootstock RPC Service.
RPCPROVIDERAPIKEY = '{YOURAPIKEY}'
```

this is the private key of the account from which you will deploy the contract  
ACCOUNTPRIVATEKEY = '{YOURPRIVATEKEY}'

Deploy a smart contract

Write the smart contract

The contract to be compiled and deployed in the next section is a simple contract that stores a message, and will allow for setting different messages by sending a transaction.

You can get started by creating a file for the contract:

```
bash
touch Greeter.sol
```

Next, add the Solidity code to the file:

```
s
// SPDX-License-Identifier: MIT

pragma solidity >0.5.0;

contract Greeter {
    string public greeting;

    constructor() public {
        greeting = 'Hello';
    }

    function setGreeting(string memory greeting) public {
        greeting = greeting;
    }

    function greet() view public returns (string memory) {
```

```
        return greeting;
    }
}
```

The constructor function, which runs when the contract is deployed, sets the initial value of the string variable stored on-chain to “Hello”. The setGreeting function adds the greeting provided to the greeting, but a transaction needs to be sent, which modifies the stored data. Lastly, the greet function retrieves the stored value.

## Compile the smart contract

We will create a script that uses the Solidity compiler to output the bytecode and interface (ABI) for the Greeter.sol contract. To get started, we will create a compile.py file by running:

```
bash
touch compile.py
```

Next, we will create the script for this file and complete the following steps:

- Import the solcx package, which will compile the source code
- Compile the Greeter.sol contract using the solcx.compilefiles function
- Export the contract’s ABI and bytecode

Code and paste the code below into compile.py;

```
s
import solcx
solcx.install_solc('0.8.19')
```

```
Compile contract
tempfile = solcx.compilefiles(
    'Greeter.sol',
    outputvalues=['abi', 'bin'],
    solcversion='0.8.19'
)
```

```
Export contract data
abi = tempfile['Greeter.sol:Greeter']['abi']
bytecode = tempfile['Greeter.sol:Greeter']['bin']
```

You can now run the script to compile the contract:

```
python
python compile.py
```

## Deploy the smart contract

With the script for compiling the Greeter.sol contract in place, you can then use the results to send a signed transaction that deploys it. To do so, you can create a file for the deployment script called deploy.py:

```
bash
touch deploy.py
```

Next, you will create the script for this file and complete the following steps:

1. Add imports, including Web3.py and the ABI and bytecode of the Greeter.sol contract
2. Set up the Web3 provider

In order to set up the Web3 Provider, we have to read the environment variables that we previously added to the .env file.

```
text
Add the Web3 Provider
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

3. Define the accountfrom. The private key is required to sign the transaction. Note: This is for example purposes only. Never store your private keys in your code

```
text
Set the default account
PRIVATEKEY = os.getenv('ACCOUNTPRIVATEKEY')
accountfrom = {
    'privatekey': PRIVATEKEY,
    'address': web3.eth.account.fromkey(PRIVATEKEY).address
}
```

4. Create a contract instance using the web3.eth.contract function and passing in the ABI and bytecode of the contract
5. Set the gas price strategy using the web3.eth.setgaspricestrategy function, which will allow us to fetch the gasPrice from the RPC Provider. This is important because otherwise the Web3 library will attempt to use ethmaxPriorityFeePerGas and ethfeeHistory RPC methods, which are only supported by post-London Ethereum nodes.
6. Build a constructor transaction using the contract instance. You will then use the buildtransaction function to pass in the transaction information including the from address and the nonce for the sender. To get the nonce you can use the web3.eth.getTransactioncount function
7. Sign the transaction using the web3.eth.account.signtransaction function and pass in the constructor transaction and the privatekey of the sender
8. Using the signed transaction, you can then send it using the web3.eth.sendrawtransaction function and wait for the transaction receipt by using the web3.eth.waitFortransactionreceipt function

Code and paste the code below into deploy.py;

```
bash
from compile import abi, bytecode
from web3 import Web3
from web3.gasstrategies.rpc import rpcgaspricestrategy
from dotenv import load_dotenv
import os
```

```
load_dotenv()
```

Add the Web3 Provider

```
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

Set the default account

```
PRIVATEKEY = os.getenv('ACCOUNTPRIVATEKEY')
accountfrom = {
    'privatekey': PRIVATEKEY,
    'address': web3.eth.account.from_key(PRIVATEKEY).address
}
```

```
print("Attempting to deploy from account: ", accountfrom['address'])
```

Create contract instance

```
Greeter = web3.eth.contract(abi=abi, bytecode=bytecode)
```

Set the gas price strategy

```
web3.eth.set_gas_price_strategy(rpc_gas_price_strategy)
```

Build the transaction

```
construct_txn = Greeter.constructor().build_transaction({
    'from': accountfrom['address'],
    'nonce': web3.eth.get_transaction_count(accountfrom['address']),
    'gasPrice': web3.eth.generate_gas_price()
})
```

Sign the transaction that deploys the contract

```
signed_txn = web3.eth.account.sign_transaction(construct_txn, accountfrom['privatekey'])
```

Send the transaction that deploys the contract

```
txn_hash = web3.eth.send_raw_transaction(signed_txn.raw_transaction)
```

Wait for the transaction to be mined, and get the transaction receipt

```
txn_receipt = web3.eth.wait_for_transaction_receipt(txn_hash)
print(f"Transaction successful with hash: { txn_receipt.transactionHash.hex() }")
print(f"Contract deployed at address: { txn_receipt.contractAddress }")
```

Now you can run the script and get the result.

```
python
python deploy.py
```

```
>> Attempting to deploy from account: 0x3b32a6463Bd0837fBF428bbC2A4c8B4c022e5077
>> Transaction successful with hash:
0x98a256c106bdb65e4de6a267e94000acdf0d6f23c3dc1444f14dccf00713a69
>> Contract deployed at address: 0xba39f329255d55a0276c695111b2edc9250C2341
```

Note: Save the contract address, as we will use it later in the guide.



Interact with a smart contract

## Read Contract Data (Call Methods)

Call methods are the type of interaction that don't modify the contract's storage (change variables), meaning no transaction needs to be sent. They simply read various storage variables of the deployed contract.

To get started, you can create a file and name it getMessage.py:

```
text
touch getMessage.py
```

Then you can take the following steps to create the script:

1. Add imports, including Web3.py and the ABI of the Greeter.sol contract
2. Set up the Web3 provider and replace YOURAPIKEY
3. Define the contractaddress of the deployed contract
4. Create a contract instance using the web3.eth.contract function and passing in the ABI and address of the deployed contract
5. Using the contract instance, you can then call the greet function

Code and paste the code below into getMessage.py;

```
bash
from compile import abi
from web3 import Web3
from dotenv import load_dotenv
import os
```

```
load_dotenv()
```

Add the Web3 Provider

```
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

Create address variable (use the address of the contract you just deployed)

```
contractaddress = '0xba39f329255d55a0276c695111b2edc9250C2341'
```

```
print(f"Making a call to contract at address: { contractaddress }")
```

Create contract instance

```
Greeter = web3.eth.contract(address=contractaddress, abi=abi)
```

Call the contract

```
callresult = Greeter.functions.greet().call()
print(f"Contract returned: { callresult }")
```

If successful, the response will be displayed in the terminal:

```
python
```

```
python getMessage.py
```

```
>> Making a call to contract at address: 0xba39f329255d55a0276c695111b2edc9250C2341
```

```
>> Contract returned: Hello
```

## Write data to the contract (Write Methods)

Write methods are the type of interaction that modify the contract's storage (change variables), meaning a transaction needs to be signed and sent. In this section, you'll create the script to change the text stored in the Greeter contract.

To get started, you can create a file for the script and name it `setMessage.py`:

```
bash
touch setMessage.py
```

Open the `setMessage.py` file and take the following steps to create the script:

1. Add imports, including `Web3.py` and the ABI of the `Incrementer.sol` contract
2. Set up the Web3 provider
3. Define the `accountfrom` variable, including the `privatekey`, and the `contractaddress` of the deployed contract. The private key is required to sign the transaction. Note: This is for example purposes only. Never store your private keys in your code
4. Create a contract instance using the `web3.eth.contract` function and passing in the ABI and address of the deployed contract
5. Set the gas price strategy using the `web3.eth.setgaspricestrategy` function, which will allow us to fetch the `gasPrice` from the RPC Provider. This is important because otherwise the Web3 library will attempt to use `ethmaxPriorityFeePerGas` and `ethfeeHistory` RPC methods, which are only supported by post-London Ethereum nodes.
6. Build the `setGreeting` transaction using the contract instance and passing in the new message. You'll then use the `buildtransaction` function to pass in the transaction information including the from address and the nonce for the sender. To get the nonce you can use the `web3.eth.gettransactioncount` function
7. Sign the transaction using the `web3.eth.account.signtransaction` function and pass in the `setGreeting` transaction and the `privatekey` of the sender
8. Using the signed transaction, you can then send it using the `web3.eth.sendrawtransaction` function and wait for the transaction receipt by using the `web3.eth.waitfortransactionreceipt` function

Code and paste the code below into `setMessage.py`;

```
bash
from compile import abi
from web3 import Web3
from web3.gasstrategies.rpc import rpcgaspricestrategy
from dotenv import load_dotenv
import os
```

```
load_dotenv()
```

Add the Web3 Provider

```
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

Set the default account

```
PRIVATEKEY = os.getenv('ACCOUNTPRIVATEKEY')
accountfrom = {
    'privatekey': PRIVATEKEY,
    'address': web3.eth.account.fromkey(PRIVATEKEY).address
}
```

Create address variable

```
contractaddress = '0xba39f329255d55a0276c695111b2edc9250C2341'
```

Create contract instance

```
Greeter = web3.eth.contract(address=contractaddress, abi=abi)
```

Set the gas price strategy

```
web3.eth.setgaspricestrategy(rpcgaspricestrategy)
```

Build the transaction

```
txn = Greeter.functions.setGreeting('Hello, World!').buildtransaction({
    'from': accountfrom['address'],
    'nonce': web3.eth.getTransactioncount(accountfrom['address']),
    'gasPrice': web3.eth.generategasprice()
})
```

Sign the transaction

```
signedtxn = web3.eth.account.signtransaction(txn, accountfrom['privatekey'])
```

Send the transaction

```
txnhash = web3.eth.sendrawtransaction(signedtxn.rawTransaction)
txnreceipt = web3.eth.waitfortransactionreceipt(txnhash)
```

```
print(f"Transaction successful with hash: { txnreceipt.transactionHash.hex() }")
```

If successful, the transaction hash will be displayed in the terminal.

```
python
```

```
python setMessage.py
```

```
>> Transaction successful with hash:
```

```
0x95ba4e13269aba8e51c3037270c0ee90f4872c36e076fc94e51226c1597f6d86
```

You can now run the getMessage.py script to get the new value stored at the contract.

```
python
```

```
python getMessage.py
```

```
>> Making a call to contract at address: 0xba39f329255d55a0276c695111b2edc9250C2341
```

```
>> Contract returned: Hello, World!
```

## Sending transactions

Here you will understand how to check the balance of an account, and how to send tBTC from one

account to another.

Check the balance of an account

Here you will create a script that checks the balance of an account.

First, start by creating a file for the script.

```
python  
touch balances.py
```

Next, you will create the script for this file and complete the following steps:

1. Set up the Web3 provider
2. Define the addressfrom and addressto variables
3. Get the balance for the accounts using the web3.eth.getbalance function and format the 3. results using the web3.fromwei

Code and paste the code below into balances.py;

```
bash  
from web3 import Web3  
from dotenv import load_dotenv  
import os  
  
load_dotenv()
```

Add the Web3 Provider

```
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')  
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY  
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

Create address variables

```
addressfrom = '0x3b32a6463Bd0837fBF428bbC2A4c8B4c022e5077'  
addressto = '0xcff73226883c1cE8b3bcCc28E45c3c92C843485c'
```

Get the balance of the sender

```
balancefrom = web3.fromwei(web3.eth.getbalance(addressfrom), 'ether')  
print(f"Balance of sender address {addressfrom}: { balancefrom } TRBTC")
```

Get the balance of the receiver

```
balanceto = web3.fromwei(web3.eth.getbalance(addressto), 'ether')  
print(f"Balance of receiver address {addressto}: { balanceto } TRBTC")
```

Run the script:

```
python  
python balances.py
```

```
>> Balance of sender address 0x3b32a6463Bd0837fBF428bbC2A4c8B4c022e5077:  
0.192538506119378425 TRBTC  
>> Balance of receiver address 0xcff73226883c1cE8b3bcCc28E45c3c92C843485c:  
0.407838671951567233 TRBTC
```

## Send TRBTC

Here you will create a script to send tBTC from one account to another.  
First, start by creating a file for the script.

```
bash
touch transaction.py
```

Next, you will create the script for this file and complete the following steps:

1. Add imports, including Web3.py and the rpcgaspricestrategy, which will be used in the following steps to get the gas price used for the transaction
2. Set up the Web3 provider
3. Define the accountfrom, including the privatekey, and the addressto variables. The private key is required to sign the transaction. Note: This is for example purposes only. Never store your private keys in your code
4. Use the Web3.py Gas Price API to set a gas price strategy. For this example, you'll use the imported rpcgaspricestrategy. This is important because otherwise the Web3 library will attempt to use ethmaxPriorityFeePerGas and ethfeeHistory RPC methods, which are only supported by post-London Ethereum nodes.
5. Create and sign the transaction using the web3.eth.account.signtransaction function. Pass in the nonce, gas, gasPrice, to, and value for the transaction along with the sender's privatekey. To get the nonce you can use the web3.eth.getTransactioncount function and pass in the sender's address. To predetermine the gasPrice you'll use the web3.eth.generategasprice function. For the value, you can format the amount to send from an easily readable format to Wei using the web3.twei function
6. Using the signed transaction, you can then send it using the web3.eth.sendrawtransaction function and wait for the transaction receipt by using the web3.eth.waitForTransactionReceipt function

Code and paste the code below into transaction.py;

```
bash
from web3 import Web3
from web3.gasstrategies.rpc import rpcgaspricestrategy
from dotenv import load_dotenv
import os
```

```
load_dotenv()
```

Add the Web3 Provider

```
RPCPROVIDERAPIKEY = os.getenv('RPCPROVIDERAPIKEY')
RPCPROVIDERURL = 'https://rpc.testnet.rootstock.io/' + RPCPROVIDERAPIKEY
web3 = Web3(Web3.HTTPProvider(RPCPROVIDERURL))
```

Set the default account

```
PRIVATEKEY = os.getenv('ACCOUNTPRIVATEKEY')
accountfrom = {
    'privatekey': PRIVATEKEY,
    'address': web3.eth.account.from_key(PRIVATEKEY).address
}
addressto = '0xcff73226883c1cE8b3bcCc28E45c3c92C843485c'
```

```
print(f"Attempting to send transaction from { accountfrom['address'] } to { addressto }")
```

Set the gas price strategy

```
web3.eth.setgaspricestrategy(rpcgaspricestrategy)
```

Build the transaction

```
txn = {  
    'to': addressto,  
    'value': web3.towei(0.0001, 'ether'),  
    'gas': 21000,  
    'gasPrice': web3.eth.generategasprice(),  
    'nonce': web3.eth.getTransactioncount(accountfrom['address'])  
}
```

Sign the transaction

```
signedtxn = web3.eth.account.signtransaction(txn, accountfrom['privatekey'])
```

Send the transaction

```
txnhash = web3.eth.sendrawtransaction(signedtxn.rawTransaction)
```

Wait for the transaction to be mined, and get the transaction receipt

```
txnreceipt = web3.eth.waitfortransactionreceipt(txnhash)
```

```
print(f"Transaction successful with hash: { txnreceipt.transactionHash.hex() }")
```

Run the script:

```
python  
python transaction.py
```

```
Attempting to send transaction from 0x112621448Eb148173d5b00edB14B1f576c58cCEE to  
0xcff73226883c1cE8b3bcCc28E45c3c92C843485c
```

```
Transaction successful with hash:
```

```
0x79ab8be672b0218d31f81876c34321ee7b08e6a4ec8bfff5249f70c443cbce00
```

## Summary

In this guide, we learnt how to use the Web3.py library to deploy, interact with a smart contract and send transactions on Rootstock.

## Troubleshooting

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">1. Error message: ethsendTransaction method does not exist</Accordion.Header>

<Accordion.Body>

- When deploying a smart contract, or when trying to interact with it, you may receive the “method not found” message:

```
bash
```

```
web3.exceptions.MethodUnavailable: {'code': -32601, 'message': 'The method  
ethsendTransaction does not exist/is not available. See available methods at  
https://dev.rootstock.io/developers/rpc-api/methods'}
```

- Note: The cause of the error on the deployment is that the Web3.py module is set to use the private keys of the RPC provider (Hosted Keys), which is a legacy way to use accounts, and is not supported by modern RPC providers, as they do not store private keys.

- Methods like web3.eth.sendtransaction do not work with RPC providers, because they rely on a node state and all modern nodes are stateless, which underneath make JSON-RPC calls to methods like ethaccounts and ethsendTransaction. You must always use local private keys when working with nodes hosted by someone else.

- If unfamiliar, note that you can export your private keys from Metamask and other wallets. Remember to never share your private keys, and do not put it on your code or repository.

- In order to successfully deploy the contract, the developer needs to set up Web3.py to use his Local Private Keys, and to build and pre-sign the transaction before sending it, so the module uses ethsendRawTransaction instead.

- To allow Web3.py to use the local keys, we have to use the Signing middleware to add the Private Key to the signing keychain.

```
bash
import os
from ethaccount import Account
from ethaccount.signers.local import LocalAccount
from web3 import Web3, EthereumTesterProvider
from web3.middleware import constructsignandsendrawmiddleware

w3 = Web3(EthereumTesterProvider())

privatekey = os.environ.get("PRIVATEKEY")
assert privatekey is not None, "You must set PRIVATEKEY environment variable"
assert privatekey.startswith("0x"), "Private key must start with 0x hex prefix"

account: LocalAccount = Account.fromkey(privatekey)
w3.middlewareonion.add(constructsignandsendrawmiddleware(account))

print(f"Your hot wallet address is {account.address}")
```

- Now you can use web3.eth.sendtransaction(), Contract.functions.xxx.transact() functions with your local private key through middleware and you no longer get the error "ValueError: The method ethsendTransaction does not exist/is not available."

```
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="1">
  <Accordion.Header as="h3">2. Error message: ethfeeHistory or ethmaxPriorityFeePerGas method
  does not exist</Accordion.Header>
  <Accordion.Body>
```

- Web3.js will try to use these methods because the Ethereum London fork introduced maxFeePerGas and maxPriorityFeePerGas transaction parameters that can be used instead of gasPrice, which Rootstock uses. For that reason, we have to define Web3's behavior for populating the gas price. This is done using a "Gas Price Strategy" - a method which takes the Web3 object and a transaction dictionary and returns a gas price (denominated in wei).

- A gas price strategy is implemented as a python method with the following signature, and by setting the gas price strategy by calling setgaspricestrategy().

- Setting a specific gas price:

```
bash
from web3 import Web3, HTTPProvider

specify Gas Price in wei
GASPRICE = 60000000
```

```
def gaspricestrategy(web3, transactionparams=None):  
    return GASPRICE
```

```
set the gas price strategy  
w3.eth.setgaspricestrategy(gaspricestrategy)
```

- Using ethgasPrice method:
- Makes a call to the JSON-RPC ethgasPrice method which returns the gas price configured by the connected Ethereum node.

```
bash  
from web3.gasstrategies.rpc import rpcgaspricestrategy  
from web3 import Web3, HTTPProvider  
  
RPCPROVIDER = 'https://rpc.testnet.rootstock.io/{APIKEY}'  
  
w3 = Web3(HTTPProvider(RPCPROVIDER))  
w3.eth.setgaspricestrategy(rpcgaspricestrategy)  
  
gasPrice = w3.eth.generategasprice()  
  
print('gasPrice: ', gasPrice)
```

```
</Accordion.Body>  
</Accordion.Item>  
</Accordion>
```

## Resources

- Web3.py: Gas Price Strategy
- Infura: ethaccounts
- Infura: ethsendTransaction
- Web3.py: Working with Local Private Keys
- Web3.py: Contract Deployment Example
- Web3.py: Sign a Contract Transaction
- Web3.py: Setting up an RPC Provider

# contract-addresses.md:

```
---  
sidebarposition: 250  
title: Rootstock Contract Addresses  
sidebarlabel: Contract Addresses  
tags: [rsk, rootstock, developers, quick starts, smart contracts, contract addresses]  
description: "All Contract Addresses on Rootstock."  
---
```

Here, you can find a list of contracts addresses on Rootstock.  
For info on derivation paths, see Account based addresses or verify address ownership section.

```
:::note[Note]  
- Rootstock Contract Metadata  
- Rootstock Testnet Contract Metadata  
- Rootstock - DefiLlama  
:::
```



## List of Contract Addresses

| Symbol | Name            | Token Standard         | Network   | Contract Address |
|--------|-----------------|------------------------|-----------|------------------|
| ---    | ---             | ---                    | ---       | ---              |
| RIF    | RIF Token       | ERC677                 | Rootstock | 0x2acc95...      |
| DOC    | Dollar on Chain | ERC20                  | Rootstock | 0xe70069...      |
| USDRIF | RIF US Dollar   | ERC20, ERC165, ERC1967 | Rootstock | 0x3a15461...     |
| RIFP   | RIFPro          | ERC20                  | Rootstock | 0xf4d27c5...     |
| BPro   | BitPro          | ERC20                  | Rootstock | 0x440cd83...     |
| BTCX   | BTCX            |                        | Rootstock | 0xf773b5...      |
| RIFX   | RIFX            |                        | Rootstock | 0xcff3fca...     |
| WRBTC  | Wrapped RBTC    | ERC20                  | Rootstock | 0x542FDA3...     |

# index.md:

```
---
sidebarlabel: Smart Contracts Development
sidebarposition: 5
title: Deploy Smart Contracts on Bitcoin
tags: [rootstock, hardhat, smart contracts, dApps]
description: "Learn how to write, interact and deploy smart contracts on Bitcoin."
---
```

Get started with deploying dApps on Rootstock using Hardhat, Wagmi, Remix and other EVM-compatible tools.

| Resource   | Description  |
|--|--|
| -----  | -----  |
| Getting Started with Hardhat                           | Get started with creating a dApps on Rootstock using Hardhat.                          |
| Getting Started with Foundry                           | How to write, test, and deploy smart contracts with Foundry                            |
| Contract Addresses                                     | List of contract addresses on Rootstock.   |
| Verify Address Ownership                               | Verify Address Ownership with Metamask Wallet.   |
| Interface Registry                                     | See the ERC1820 standard interface, address support and smart contract implementation. |
| Verify Smart Contracts using the Hardhat Verify Plugin | Configuring Hardhat Verification plugin for Rootstock.                                 |

# verify-address-ownership.md:

```
---
sidebarposition: 300
sidebarlabel: Verify address ownership
title: 'Verify Address Ownership with Metamask Wallet'
description: 'Confirm that you own an Rootstock address using RIF Identity Manager'
tags: [metamask, address, account, smart contracts]
---
```

Let's say that you need to receive a transfer of RBTC, or tokens on the Rootstock network, for the very first time. To do this you need to set up a wallet and connect it to the Rootstock network.

However, you may be unsure if you actually "control" the addresses in the wallet.

Understandably so, because it is your first time using it.  
That concern has a technical basis too -  
you need to be sure that you are able to sign transactions at this address,  
before you ask others to send you cryptocurrency or tokens at this address.

Here we will demonstrate exactly how to do this,  
and be sure that you truly "control" a particular address.  
All you need is Chrome (web browser) and MetaMask (browser extension).  
You do not need any RBTC balance to do so.

## Getting Started

:::tip[Install MetaMask]

You can either use the [metamask-landing.rifos.org](https://metamask-landing.rifos.org) tool to download/install Metamask, and add Rootstock custom network or follow the steps listed in [metamask.io](https://metamask.io).

...

In Chrome, visit [metamask.io](https://metamask.io),  
and follow the instructions to install this extension in your browser.  
If you are doing this for the first time,  
you will need to generate a seed phrase,  
and it is extremely important that you record this somewhere.

## Enable only one Web3 browser extension

If you have more than one Web3 browser extension installed,  
for example, if you have either MetaMask, Liquidity or Nifty,  
be aware that they can conflict with each other.

Paste `chrome://extensions/` in your address bar,  
to see all the browser extensions that you have installed.  
Verify that you only have MetaMask installed, or  
if you have other Web3 browser extensions,  
you should disable all of the others by clicking on the toggle buttons.



:::info[Optional]

For a better user experience, you may also wish to

- Click on the extensions icon (jigsaw shape), and in the dropdown,
- Click the pin icon next to MetaMask to ensure it is always visible.

...



## Unlock MetaMask

After installing the extension or starting your browser,  
MetaMask should display a popup asking you to unlock the account.  
Enter your MetaMask password.  
(Note that this is not the same as your seed phrase.)

If it does not popup, you can manually enter  
chrome-extension://nkbihfbeogaeaoehlefnkodbefgpgknn/home.html#unlock  
in your address bar to navigate there in "expanded view",  
instead of within a popup.



Add custom network for Rootstock

MetaMask only contains network configurations to connect to Ethereum by default.  
To connect to Rootstock you will need to add Rootstock Network configurations.

You have the option to manually add  
Rootstock Mainnet network configuration to MetaMask.

Alternatively, you can do this automatically,  
by visiting [identity.rifos.org](https://identity.rifos.org),  
and when you attempt to connect using MetaMask,  
you will get presented with the following:



Click "RSK Mainnet". MetaMask will then show this popup:



Click "Approve". This will automatically fill out the network configuration for you.



Then click "Switch Network" to connect to the Rootstock Mainnet.

Verifying your Rootstock account

At this point, you should have everything set up:  
You have a wallet installed,  
that wallet is connected to the RSK Mainnet,  
and you have addresses inside that wallet.

You're ready to verify that you can use your wallet to sign messages!

View transaction history

In MetaMask, you can view your transaction history for a particular address  
by selecting the "Activity" tab in the main screen.

> "Expanded view": <chrome-extension://nkbihfbeogaeaoehlefnkodbefgpgknn/home.html#>



If your activity tab is empty, like the one above,  
it means that there are zero transactions at this address.  
Let's copy the address by clicking on it.  
It is located near the top, begins with 0x,

and should be under a label similar to "Account 1".

Visit block explorer

Let's check the address that you've just copied on the Rootstock block explorer.

Visit [explorer.rsk.co/address/\\${YOURADDRESS}](https://explorer.rsk.co/address/${YOURADDRESS}).

Replace `${YOURADDRESS}` with the address copied from MetaMask earlier.

For example, if you copied `0xdfc0e6361fd1846a223e2d7834a5ebd441a16dd4`, the URL will be <https://explorer.rsk.co/address/0xdfc0e6361fd1846a223e2d7834a5ebd441a16dd4>



Here you may see "Not Found".

This does not necessarily mean that the account does not exist.

Instead, it means that there simply are no transactions on the blockchain at this address.

Visit RIF Identity Manager

So far, not so good, right?

... Nothing we've seen thus far assures you that you do indeed control this address.

This is where the RIF Identity Manager comes in!

This DApp allows you to verify whether we control this address.

You'll do this by signing a message that is not a blockchain transaction.

Visit [identity.rifos.org](https://identity.rifos.org).



Click on "Connect your Wallet"



Select "MetaMask"

- > Note that if you have multiple Web3 browser extensions installed,
- > disable all of them except for one.
- > If not, this confuses most DApps including RIF Identity Manager,
- > and you may not see MetaMask here as a result.
- > See the "before you begin" section for details.

MetaMask site connection permission

You will be presented with a popup from MetaMask, which essentially is asking you whether you trust RIF Identity Manager.



Click "Next".

This allows MetaMask to interact with RIF Identity Manager

MetaMask will then show another popup, asking you whether you want to allow RIF Identity Manager to see your account addresses.



Click "Connect".

This allows MetaMask to see your account addresses.

## RIF Identity Authentication

Upon granting these permissions, the RIF Identity Manager DApp presents you with yet another MetaMask popup.



This time, it asks to sign a text message, which should look similar to the following:

text

Are you sure you want to login to the RIF Data Vault?

URL: <https://data-vault.identity.rifos.org>

Verification code: \${SOMERANDOMVALUE}

Click "Sign".

When you do this, the crucial part happens!

- MetaMask uses the private key corresponding to the address to sign that message.
- The signed message is transmitted to RIF Identity Manager's backend, which performs digital signature verification, which it uses to confirm whether it has indeed been signed by this particular address.
- Since this is a plain text message, and does not involve adding a transaction to the blockchain, no gas fees need to be paid, and therefore your RBTC balance can be zero.

This is perfect for newly generated accounts!

Check the dashboard

Once you have signed the message and it has been verified, you will see the dashboard for the RIF Identity Manager.



Check that the "Persona Address" field that is displayed here matches the address of your account in MetaMask.



That's all - now you can be confident that you do control this address on the Rootstock Mainnet!

## Resources

- Verify Smart Contracts with SolidityScan
- Developer Tools
- Verify Smart Contracts using Hardhat Verify Plugin for Rootstock

# configure-hardhat-rootstock.md:

---

sidebarlabel: Configure Hardhat for Rootstock

sidebarposition: 102

title: Configure Hardhat for Rootstock

description: "Learn how to configure your Hardhat project for development on Rootstock testnet and mainnet"

tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]

---

## Prerequisites

1. Rootstock-compatible accounts/address.
  - You can use existing accounts or create new ones. See Account Based Addresses.
2. Wallet
  - Set up a Metamask wallet and get a private key.

## Getting Started

### Step 1: Set up Your Hardhat Environment

- Install dotenv

To manage environment variables, install dotenv using the following command:

shell

```
npm install dotenv
```

- Create a .env file

- In the rootstock-quick-start-guide project root, create a .env file and add your private keys (do not share this file):

shell

```
ROOTSTOCKMAINNETPRIVATEKEY="yourmainnetprivatekey"  
ROOTSTOCKTESTNETPRIVATEKEY="yourtestnetprivatekey"
```

:::info[Note]

Depending on your desired network, using a Testnet and Mainnet private key is optional, as you're not required to have separate private keys in your environment variable.

:::

### Step 2: Configure Private Keys

To configure your rskMainnet and rskTestnet private keys, you'll need to update your hardhat.config.js file in the root directory with your private keys.

- Copy the code snippet below and replace the existing code in your hardhat.config.js file. See diff file for initial code.

```
js
require("@nomiclabs/hardhat-ethers");
require('dotenv').config();

<!-- Hardhat configuration -->
module.exports = {
  solidity: "0.8.20",
  networks: {
    rskMainnet: {
      url: "https://rpc.mainnet.rootstock.io/{YOURAPIKEY}",
      chainId: 30,
      gasPrice: 60000000,
      accounts: [process.env.ROOTSTOCKMAINNETPRIVATEKEY]
    },
    rskTestnet: {
      url: "https://rpc.testnet.rootstock.io/{YOURAPIKEY}",
      chainId: 31,
      gasPrice: 60000000,
      accounts: [process.env.ROOTSTOCKTESTNETPRIVATEKEY]
    }
  }
};
```

> See how to Get an API Key from the RPC API

> Replace "yourmainnetprivatekey" and "yourtestnetprivatekey" with your private keys. For information on how to retrieve your private keys, see How to export an account's private key.

### Step 3: Fund Your Accounts

- Mainnet
  - You'll need RBTC, which you can obtain from an exchange. See Get RBTC using Exchanges.
- Testnet
  - You can get tRBTC from the Rootstock Faucet.

# create-hardhat-project.md:

```
---
sidebarlabel: Create a Hardhat Project
sidebarposition: 101
title: Create a Hardhat Project
description: "Learn how to set up your environment for development using Hardhat"
tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]
---
```

In this section, you will learn how to create a hardhat project and verify hardhat installation.

### Clone the Project Repository

To get started, clone the rootstock-quick-start-guide repository:

```
shell
git clone https://github.com/rsksmart/rootstock-quick-start-guide.git
```

## Install Dependencies

Run the following command in the project root.

```
shell
npm install
```

> The quick start repo already comes pre-installed with hardhat. The master branch has the initial setup and barebones project and the feat/complete branch has the complete state of the hardhat project. You can view the diff in the initial and complete state branches of the repo at any point in time while going through this material. To run the full project, checkout into feat/complete branch, install the dependencies and run the command: `npx http-server`.

## Verify Hardhat Installation

Here, we will verify the installation of hardhat in your project.

- To verify hardhat installation:
  - The quickstart repository comes with Hardhat pre-installed. To check if Hardhat is installed, execute `npx hardhat` in the rootstock-quick-start-guide directory.
  - `npx hardhat` not only verifies installation but also allows you to initiate a new Hardhat project if it doesn't exist. For a new project, you'll be prompted to choose from several options. To create a blank project, select Create an empty `hardhat.config.js`, or pick one of the other options to begin with a pre-set template.

Once setup is complete, you can verify Hardhat is installed correctly by running `npx hardhat` again. It should display a help message with available tasks, indicating that Hardhat is installed and ready to use.

# deploy-smart-contracts.md:

```
---
sidebarlabel: Deploy Smart Contracts
sidebarposition: 105
title: Deploy Smart Contracts
description: "Learn how to deploy your Rootstock smart contract on your local environment and the Rootstock network."
tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]
---
```

In this section, we'll deploy your token contract to your local environment and also deploy and interact with the contract on the Rootstock network.

## Step 1: Configure Deployment File

To configure your deployment file:

- Navigate to the scripts directory in the root directory of the quick start repo:

```
shell
cd scripts
```



- In the scripts directory, open the deploy.js deployment file:

To deploy myToken contract, copy the deployment script below and paste it in your deployment file or see the deploy.js file on GitHub.

```
js
async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying contracts with the account:", deployer.address);

  const MyToken = await ethers.getContractFactory("MyToken");
  const myToken = await MyToken.deploy(1000);

  console.log("Token address:", myToken.address);
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

## Step 2: Run the Hardhat Network Locally

> Note: You need to have sufficient RBTC in your deploying account for gas fees. See section on Fund your account.

To run the Hardhat network locally:

- Start the Hardhat network
  - Hardhat comes with a built-in Ethereum network for development. Run the following command in your project's root directory to start it.

```
shell
npx hardhat node
```

This command will start a local blockchain network and display a list of available accounts and private keys:

```
!Rootstock Node Running
```

- Deploy your contract to the local network
  - Deploy your contract to the local Hardhat network, in another terminal or command prompt, run the command below in the root directory:

```
shell
npx hardhat run --network hardhat scripts/deploy.js
```

This should give a result similar to the following:

```
shell
npx hardhat run --network hardhat scripts/deploy.js
```

```
Deploying contracts with the account: 0xf39Fd6e51aad88F6F4ce6aB8827279cFfFb92266
Token address: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

### Step 3: Deploy Your Contract on Rootstock Network

Follow these steps to deploy your contract on Rootstock network:

- Use Hardhat's run command to deploy your contract, depending on the desired network. You can choose to deploy to either Rootstock's Testnet or Mainnet.

To deploy to the Rootstock Testnet, run:

```
shell
npx hardhat run --network rskTestnet scripts/deploy.js
```

This should return the following:

```
shell
% npx hardhat run --network rskTestnet scripts/deploy.js
Deploying contracts with the account: 0xA210D04d707f6beBF914Cb1a57199Aebe7B40380
Token address: 0xc6EcBe0F6643825FD1AAfc03BEC999014759a279
```

- To deploy to the Rootstock Mainnet, run:

```
shell
npx hardhat run --network rskMainnet scripts/deploy.js
```

### Configure MetaMask

:::note[Install Metamask]

If you haven't already, you can use the [metamask-landing.rifos.org](https://metamask-landing.rifos.org) tool to download/install Metamask, and add Rootstock custom network or follow the steps in Configure Network and Token.

...

### Step 4: Connect Remix to Rootstock Testnet (Optional)

#### 1. Open Remix IDE

- Go to Remix IDE in your browser.

#### 2. Connect MetaMask to Remix:

- In Remix, go to the Deploy & run transactions plugin.
- In the Environment dropdown, select Injected Provider.
- This will connect to MetaMask. Make sure MetaMask is on the RSK Testnet network that you configured earlier.

### Interact with the Deployed Contract on Remix

To interact with your deployed contract on Rootstock network:

- Load Your Deployed Contract

- Import the myToken.sol file into remix and compile.

### !Import Solidity File and Compile

- Once compiled, you should see the checkmark and solidity file loaded into Remix.

### !Successful Compile

- Choose Deploy and Run Transactions and in Environment, Choose "Injected Provider - Metamask".

This loads the Metamask wallet.

### !Deploy and Run Transactions

Now click on Transactions recorded interact with the Smart Contract! Call its functions, send transactions, and observe the results. Ensure you have enough tRBTC in your MetaMask wallet for transaction fees.

- Monitor Transactions

- Use Remix and MetaMask to monitor transaction confirmations and results.
- You can also use a Rootstock Testnet Explorer to view transactions and contract interactions.

# index.md:

```
---
sidebarlabel: Getting Started with Hardhat
sidebarposition: 100
title: Getting Started with Hardhat
description: "Get started with creating a dApps on Rootstock using Hardhat."
tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]
---
```

:::note[Before you begin]

> If you're new to Web3 and Smart Contract Development, begin by exploring the Rootstock network. Then progress step by step to the quick start Guide with Hardhat for a comprehensive understanding of the network and getting started with writing, testing, and deploying smart contracts on Rootstock.

> For your convenience, we've established a GitHub repository dedicated to this guide. The master branch contains the initial project state, while the feat/complete branch features the complete project, equipped with all the necessary installations for your reference.

> Note: This guide is optimized for Node.js version 18 or earlier. If you're using a later version, consider using a version manager like NVM to switch to a compatible version.

> Need to ramp up fast and get started with Hardhat? Use the Hardhat Starter Kit or use the Wagmi Starter Kit

:::

### Navigating the Guide

| Resource      | Description  |
|---------------|--|
|               |  |
| -----         |  |
| -----         |  |
| Prerequisites | Learn about the tools you need to have in place to follow along with this guide. |

| Create a Hardhat Project | Learn how to set up your environment for development using Hardhat. |  
| Configure Hardhat for Rootstock | Learn how to configure your Hardhat project for development on Rootstock testnet and mainnet. |  
| Write Smart Contracts | Learn how to write a smart contracts. |  
| Test Smart Contracts | Learn how to test your smart contract to ensure it's working as expected. |  
| Deploy Smart Contracts | Learn how to deploy your smart contract to your local environment and the Rootstock network. |  
| Interact with the Frontend | Learn how to interact with the smart contract from the front-end application. |  
| Debugging and Troubleshooting Tips | Learn about the common issues you can come across while building following this guide and how you can solve them. |

# interact-with-frontend.md:

```
---
sidebarlabel: Interact with the Front-end
sidebarposition: 106
title: Interact with the Front-end
description: "Learn how to integrate your Rootstock smart contract with front-end applications."
tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]
---
```

Creating a user-friendly web interface for smart contracts on the Rootstock network can enhance user interaction. Here, we'll focus on using ethers.js, a popular Ethereum library, for connecting your smart contracts to a web front-end.

## Project Setup

1. Create a new folder called frontend and navigate to the directory:

```
shell
mkdir frontend
cd frontend
```

> Note: If you use the quick start repo on master, there's already a frontend folder. You can cd into the frontend directory.

2. In the frontend directory, initialize a Node.js Project:

```
shell
npm init -y
```

3. Install Ethers.js:

```
shell
npm install --save ethers
```

## Create HTML File

- Update HTML File
  - In the frontend directory, open the index.html file:
  - Copy the code snippet below and paste it in your html file:

```
html
<!DOCTYPE html>
```

```

<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Web3 App with Ethers.js</title>
</head>

<body>
</body>

</html>

```

#### - Import Ethers

- To import the Ethers library to interact with the wallet to the network, copy the code snippet below and paste it in the <head> section of your html file:

```

html
<script src="https://cdn.ethers.io/lib/ethers-5.2.umd.min.js"></script>

```

#### - Create HTML elements inside the body

1. Create a button to trigger the function for connecting the wallet.
2. Create a button to trigger the function to get balance.
3. Create a div element to show the answer for the address connected.
4. Create a div element to show the answer for wallet balance.

```

html
<body>
  <div>
    <h1>Connect to Rootstock Network</h1>
    <button id="connectButton">Connect Wallet</button>
    <button id="getBalanceButton" disabled>Get MTK Balance</button>
    <div id="walletAddress"></div>
    <div id="walletBalance"></div>
  </div>
</body>

```

#### - Import javascript file

- Finally, to import the javascript library that we will create in a further step, copy the code snippet below and paste it in the <body> section of your html file::

```

html
<script src="app.js"></script>

```

Your index.html file should now look like the index.html file on GitHub.

### Create JavaScript Functions

#### - Create basic javascript function

1. In the frontend directory, open the app.js file.
2. Copy the MyToken.json artifact file generated when building the contracts in /artifacts/contracts/MyToken.sol/MyToken.json.
3. Copy the networks.json file. You can get the file from this link.
4. Create the function to wait until the DOM is loaded, instance the HTML elements (buttons and divs), and declare some variables:

```

js
document.addEventListener('DOMContentLoaded', function () {

```

```

// Instantiating HTML elements
const connectButton = document.getElementById('connectButton');
const getBalanceButton = document.getElementById('getBalanceButton');
const walletAddressDiv = document.getElementById('walletAddress');
const walletBalanceDiv = document.getElementById('walletBalance');
// Instantiating variables
let provider, account, myTokenContract;
let contractABI = [];
let networks = {};
const contractAddress = 'Replace with your contract\'s address'; // E.g.
0xa6fb392538BaC56e03a900BFDE22e76C05fb5122
});

```

- Add a function that fetches the ABI and stores it in a variable

```

js
async function fetchExternalFiles() {
  // Place MyToken.json generated in artifacts after compiling the contracts
  let response = await fetch('MyToken.json');
  const data = await response.json();
  contractABI = data.abi;
  // Place networks.json to set the network automatically with the checkNetwork() function
  // You can set it manually instead following this guide
https://dev.rootstock.io/resources/tutorials/rootstock-metamask/
  response = await fetch('networks.json');
  networks = await response.json();
}

```

- Add a function that checks the wallet is connected to the Rootstock network

```

js
async function checkNetwork() {
  try {
    // Make sure Metamask is installed
    if (!window.ethereum){
      alert('Please install Metamask!');
      return;
    }
    // Switch network
    await window.ethereum.request({
      method: 'walletswitchEthereumChain',
      params: [{ chainId: networks.rskTestnet.chainId }],
    });
  } catch (error) {
    // This error code indicates that the chain has not been added to Metamask
    if (error.code === 4902) {
      // Trying to add new chain to Metamask
      await window.ethereum.request({
        method: 'walletaddEthereumChain',
        params: [networks.rskTestnet],
      });
    } else {
      // Rethrow all other errors
      throw error;
    }
  }
}

```

- Call the fetchABI function that loads the ABI and connects the wallet to the network

```
js
// Get the required data and set the events
fetchExternalFiles().then(() => {
  // Connect button event
  connectButton.addEventListener('click', async function () {
    // Check the network is set properly
    await checkNetwork();
    if (typeof window.ethereum !== 'undefined') {
      try {
        // Get the account from Metamask
        const accounts = await window.ethereum.request({ method: 'ethrequestAccounts' });
        account = accounts[0];
        // Update the front with the account address
        walletAddressDiv.innerHTML = Connected account: ${account};
        // Get the network provider
        provider = new ethers.providers.Web3Provider(window.ethereum);
        // Get the signer for network interaction
        signer = provider.getSigner();
        // Activates the getBalanceButton
        connectButton.disabled = true;
        getBalanceButton.disabled = false;
      } catch (error) {
        console.error("Error connecting to MetaMask", error);
        walletAddressDiv.innerHTML = Error: ${error.message};
      }
    } else {
      walletAddressDiv.innerHTML = 'Please install MetaMask!';
    }
  });
});
```

- Add a function responding to the click event on the get balance button.

```
js
// Get balance button event
getBalanceButton.addEventListener('click', async function () {
  // Verify contractAddress is a valid address
  if (!ethers.utils.isAddress(contractAddress)){
    alert('Please verify that contractAddress is set');
    return;
  }
  // Instantiate the contract
  myTokenContract = new ethers.Contract(contractAddress, contractABI, signer);
  // Check if the contract is instantiated properly
  if (myTokenContract) {
    // Obtains the user balance
    const balance = await myTokenContract.balanceOf(account);
    // Show the user balance
    walletBalanceDiv.innerHTML = MyToken Balance: ${balance} MTK;
  }
});
```

- View the Complete Code

- GitHub Link

```
js
```

```

document.addEventListener('DOMContentLoaded', function () {
  // Instantiating HTML elements
  const connectButton = document.getElementById('connectButton');
  const getBalanceButton = document.getElementById('getBalanceButton');
  const walletAddressDiv = document.getElementById('walletAddress');
  const walletBalanceDiv = document.getElementById('walletBalance');
  // Instantiating variables
  let provider, account, myTokenContract;
  let contractABI = [];
  let networks = {};
  const contractAddress = 'Replace with your contract\'s address'; // E.g.
0xa6fb392538BaC56e03a900BFDE22e76C05fb5122

  /
  Load data from external JSON files
  /
  async function fetchExternalFiles() {
    // Place MyToken.json generated in artifacts after compiling the contracts
    let response = await fetch('MyToken.json');
    const data = await response.json();
    contractABI = data.abi;
    // Place networks.json to set the network automatically with the checkNetwork() function
    // You can set it manually instead following this guide
https://dev.rootstock.io/resources/tutorials/rootstock-metamask/
    response = await fetch('networks.json');
    networks = await response.json();
  }

  /
  Check and set network automatically in case it is not already done
  /
  async function checkNetwork() {
    try {
      // Make sure Metamask is installed
      if (!window.ethereum){
        alert('Please install Metamask!');
        return;
      }
      // Switch network
      await window.ethereum.request({
        method: 'walletswitchEthereumChain',
        params: [{ chainId: networks.rskTestnet.chainId }],
      });
    } catch (error) {
      // This error code indicates that the chain has not been added to Metamask
      if (error.code === 4902) {
        // Trying to add new chain to Metamask
        await window.ethereum.request({
          method: 'walletaddEthereumChain',
          params: [networks.rskTestnet],
        });
      } else {
        // Rethrow all other errors
        throw error;
      }
    }
  }

```



```

    }
  }

  // Get the required data and set the events
  fetchExternalFiles().then(() => {
    // Connect button event
    connectButton.addEventListener('click', async function () {
      // Check the network is set properly
      await checkNetwork();
      if (typeof window.ethereum !== 'undefined') {
        try {
          // Get the account from Metamask
          const accounts = await window.ethereum.request({ method: 'ethrequestAccounts' });
          account = accounts[0];
          // Update the front with the account address
          walletAddressDiv.innerHTML = Connected account: ${account};
          // Get the network provider
          provider = new ethers.providers.Web3Provider(window.ethereum);
          // Get the signer for network interaction
          signer = provider.getSigner();
          // Activates the getBalanceButton
          connectButton.disabled = true;
          getBalanceButton.disabled = false;
        } catch (error) {
          console.error("Error connecting to MetaMask", error);
          walletAddressDiv.innerHTML = Error: ${error.message};
        }
      } else {
        walletAddressDiv.innerHTML = 'Please install MetaMask!';
      }
    });

    // Get balance button event
    getBalanceButton.addEventListener('click', async function () {
      // Verify contractAddress is a valid address
      if (!ethers.utils.isAddress(contractAddress)){
        alert('Please verify that contractAddress is set');
        return;
      }
      // Instantiate the contract
      myTokenContract = new ethers.Contract(contractAddress, contractABI, signer);
      // Check if the contract is instantiated properly
      if (myTokenContract) {
        // Obtains the user balance
        const balance = await myTokenContract.balanceOf(account);
        // Show the user balance
        walletBalanceDiv.innerHTML = MyToken Balance: ${balance} MTK;
      }
    });
  });
});

```

Run the frontend

To run the frontend, execute a local web server to test the HTML file using the following command:

```
shell
npx http-server
```

Navigate to the URL: <http://127.0.0.1:8080> to test the code in the browser and you should get a result similar to the image below:

!Smart Contract Frontend

:::tip[Tip]

- Ensure the local hardhat network is running. Run `npx hardhat node` in the root directory to start the local network. See section on Troubleshooting and Common Errors to fix common errors.
- You can view and run the complete project from the `feat/complete` branch. To do so, git checkout into the `feat/complete` branch, run `cd frontend`, run `npm install`, then run `npx http-server` to view and interact with the smart contract from the frontend.

...

## Resources

These tools are specifically tailored for Web3 development, and they can simplify the integration of blockchain functionality into web interfaces. Here are a few recommended tools and libraries that are popular in the Web3 space, along with brief descriptions:

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">1. RainbowKit</Accordion.Header>

<Accordion.Body>

- RainbowKit is a React library offering a comprehensive wallet connection solution. It provides a beautiful, easy-to-use wallet connection interface that supports multiple wallets and networks.

- Why Use It:

It is great for projects where you want a seamless and user-friendly wallet connection experience. It's easy to integrate and manage, especially in React-based applications.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">2. Web3Modal</Accordion.Header>

<Accordion.Body>

- Web3Modal is a JavaScript library that provides a simple, unified wallet connection modal for Web3 applications. It supports various wallet providers and can be used with different Web3 libraries.

- Why Use It: If you need to start using React or want a framework-agnostic wallet connection solution, Web3Modal is an excellent choice. It's customizable and works well with both `web3.js` and `ethers.js`.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">3. Wagmi</Accordion.Header>

<Accordion.Body>

- Wagmi is a React Hooks for Ethereum set that simplifies interactions with `ethers.js`. It provides hooks for wallet connection, contract interaction, balances, and more.

- Why Use It: For React developers who prefer a hooks-based approach, Wagmi offers an elegant way to integrate Ethereum functionality. It makes managing state and blockchain interactions more intuitive.

</Accordion.Body>

```

</Accordion.Item>
<Accordion.Item eventKey="3">
  <Accordion.Header as="h3">4. Moralis</Accordion.Header>
  <Accordion.Body>
    - Moralis is a fully managed backend platform for Web3 and blockchain applications. It offers a suite
of tools for authentication, real-time databases, cloud functions, and syncing blockchain data.
    - Why Use It: It can be a time-saver to build a more comprehensive application with backend support.
It handles much of the backend complexity and lets you focus on front-end development.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="4">
  <Accordion.Header as="h3">5. Foundry</Accordion.Header>
  <Accordion.Body>
    - Foundry is a smart contract development toolchain, and user-friendly development environment
used for writing and advanced smart contracts testing in Solidity.
  </Accordion.Body>
</Accordion.Item>
</Accordion>

```

# test-smart-contracts.md:

```

---
sidebarlabel: Test Smart Contracts
sidebarposition: 104
title: Test Smart Contracts
description: "Learn how to test your Rootstock smart contract"
tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]
---

```

In this section, you'll set up a smart contract test and test your contract using Mocha and Chai testing frameworks. See DApps Automation using Cucumber and Playwright.

Follow these steps below to test the smart contract.

### Step 1: Install Dependencies

We'll install the Mocha and Chai testing dependencies.

Mocha is a JavaScript test framework running on Node.js. Chai is an assertion library for the browser and Node that can be paired with any JavaScript testing framework.

- Before writing tests for your token contract, ensure Mocha and Chai is installed. To install the required testing dependencies:

```

shell
npm install --save-dev mocha@10.2.0 chai@4.2.0 @nomiclabs/hardhat-ethers@2.2.3

```

### Step 2: Create Tests

1. Navigate to the test directory in the root directory of your project, this is recommended for storing all test files:

```

shell
cd test

```

2. In the test directory, open the MyToken.test.js file, we'll write tests for the token contract using Mocha and Chai:

Copy the code snippet below and paste it in your test file or see the MyToken.test.js file on GitHub.

```
js
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("MyToken", function () {
  it("Should deploy MyToken and assign the total supply to the owner", async function () {
    const [owner] = await ethers.getSigners();

    const MyToken = await ethers.getContractFactory("MyToken");
    const myToken = await MyToken.deploy(1000);
    await myToken.deployed();

    expect((await myToken.totalSupply()).toString()).to.equal('1000');
    expect((await myToken.balanceOf(owner.address)).toString()).to.equal('1000');
  });
});
```

### Step 3: Run the Tests

To execute tests, run the following command in your project's root directory. This will run the written tests, confirming that the contract works as expected.

```
shell
npx hardhat test
```

You should get a response like below:  
!Test Success

By following these steps, you'll have the necessary testing frameworks installed and be well prepared to write effective tests for your smart contract.

### Alternative Testing Approaches and Frameworks

In addition to Mocha and Chai, you can use several other frameworks and approaches in your Hardhat project. Each has its unique features and benefits.

- Jest - JavaScript Testing Framework
  - Jest is popular for its delightful syntax and focus on simplicity. It works well for testing both frontend and backend JavaScript applications.
- Waffle - Ethereum Smart Contract Testing Library
  - Waffle is a library for writing and testing smart contracts. It is often used with ethers.js and is known for its fluent syntax.
- Cucumber DApps Automation
  - dApp Automation with Cucumber

# troubleshooting.md:

---

sidebarlabel: Debugging and Troubleshooting

sidebarposition: 106

title: Common Errors and Tips

description: "Learn about some potential issues you can run into and tips on how to resolve them."

tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]

---

This section provides help on some potential issues you may run into and tips on how to resolve them.

## Errors

- Error HH8: There's one or more errors in your config file

shell

% npx hardhat compile

Error HH8: There's one or more errors in your config file:

Invalid account: #0 for network: rskMainnet - Expected string, received undefined

Invalid account: #0 for network: rskTestnet - Expected string, received undefined

To learn more about Hardhat's configuration, please go to <https://hardhat.org/config/>

For more info go to <https://hardhat.org/HH8> or run Hardhat with --show-stack-traces

> - FIX 1: Ensure the values in the environment variables matches with the hardhat network configuration hardhat.config.js file. For bash, run source .env in the root directory for dotenv to enable the environment variables.

- Error: Nothing to Compile

shell

% npx hardhat compile

Nothing to compile

> - FIX 2: Delete artifacts folder and run the npx hardhat compile command to generate new artifacts.

- Error: "GET /MyToken.json" Error (404): "Not found"

- Check that contracts were compiled successfully, and artifacts folder was generated.

- Check that all the steps in interacting with frontend were followed sequentially.

- Error: HH601: Script scripts/deploy.js doesn't exist.

- Ensure that you're running the npx hardhat run --network hardhat scripts/deploy.js command from the root directory.

# write-smart-contracts.md:

---

sidebarlabel: Write a Smart Contract

sidebarposition: 103

title: Write a Smart Contract

description: "Learn how to write a smart contract using Solidity and OpenZeppelin"

tags: [guides, developers, smart contracts, rsk, rootstock, hardhat, dApps, ethers]

---

In this section, we'll learn how to write a smart contract using the OpenZeppelin library and Solidity. OpenZeppelin is widely used for its secure, community-vetted, and standardized codebase, which simplifies developing robust and secure smart contracts.

## Create a Smart Contract

### Step 1: Install OpenZeppelin Contracts

Run the following command to install the OpenZeppelin's library of reusable smart contracts.

```
shell
  npm install @openzeppelin/contracts
```

### Step 2: Create a Token Contract

- Navigate to the contracts directory in the root directory of quick start project:

```
shell
  cd contracts
```

- In the contracts directory, open the MyToken.sol file for your token contract:

To configure an ERC20 token, copy the code snippet below and paste it in your token file or view the complete MyToken.sol file on [GitHub](#).

```
shell
  // SPDX-License-Identifier: MIT
  pragma solidity ^0.8.20;

  import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

  contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {
      mint(msg.sender, initialSupply);
    }
  }
```

This contract defines an ERC20 token named MyToken with the symbol MTK, using OpenZeppelin's ERC20 standard implementation.

### Compile the Contract

To build the contract, run the following command in the project's root directory.

```
shell
  npx hardhat compile
```

This will compile your smart contracts and generate artifacts:

```
shell
% npx hardhat compile
Downloading compiler 0.8.20
Compiled 6 Solidity files successfully (evm target: paris).
```

---

Next

- Test your Smart Contract to ensure it's working as expected.

# index.md:

---

sidebarlabel: Interface Registry

sidebarposition: 300

title: Universal Smart Contract Interface Registry

description: "See the ERC1820 standard interface, address support and smart contract implementation"

tags: [smart contracts, rsk, rootstock, developers, interface registry]

---

The ERC1820 standard defines a universal registry smart contract where any address (contract or regular account) can register which interface it supports and which smart contract is responsible for its implementation.

## Description

This standard defines a registry where smart contracts and regular accounts can publish which functionality they implement - either directly or through a proxy contract.

Anyone can query this registry to ask if a specific address implements a given interface and which smart contract handles its implementation.

This registry may be deployed on any chain and shares the same address on all chains.

Interfaces with zeroes (0) as the last 28 bytes are considered ERC165 interfaces, and this registry SHALL forward the call to the contract to see if it implements the interface.

This contract also acts as an ERC165 cache to reduce gas consumption.

## Motivation

### EIP1820

allows contracts to register interface and query the registry, avoiding miscommunications that may result in the loss of funds.

For example, with an ERC20 smart contract, a mistake can result in tokens being burnt.

Though the ERC20 token standard is well documented and well implemented overall, it contains a bug. This bug has result in losses of tokens worth millions of US dollars. With the transfer function, you can only send tokens to an externally account. If you use the transfer function to send tokens to a smart contract (which is not an externally owned account), you will see a successful transaction, but the contract will never receive the tokens. This effectively destroys the tokens forever, as they cannot be retrieved. By using the wrong function, several users have lost their tokens for good!

The ERC777 token standard solves this problem using the EIP1820 interface registry, and it is backward compatible with the ECR20 token standard.

In order to enable implementations based on the ERC777 token standard, as well as any other smart contracts that would benefit from

the use of a universal smart contract interface registry, Rootstock has deployed an implementation of the EIP1820 registry on both its Mainnet, and Testnet.

## Links and Information

Original:

- EIP1820
- Comparing ERC777 to ERC20
- Comparing ERC777 and ERC223 to ERC20

Rootstock:

- Rootstock Mainnet deployed EIP1820 smart contract
- Rootstock Testnet deployed EIP1820 smartcontract
- Corresponding Rootstock Improvement Proposal: RSKIP148

# index.md:

```
---
sidebarlabel: Verify Smart Contracts using the Hardhat Verify Plugin
sidebarposition: 400
title: Verify a Smart Contract using the Hardhat Verification Plugin
description: "Configuring Hardhat Verification plugin for Rootstock"
tags: [hardhat, tutorial, developers, quick starts, rsk, rootstock]
---
```

Smart contracts are the backbone of decentralized applications (dApps). They automate agreements and processes, but their code can be complex and prone to errors. Verifying your smart contracts is crucial to ensure they function as intended.

This tutorial will guide you through verifying your contracts using the Hardhat Verification Plugin on the Rootstock Blockscout Explorer. This plugin simplifies the verification of Solidity smart contracts deployed on the Rootstock network. By verifying the contracts, you allow Blockscout, an open-source block explorer, to link your contract's source code with its deployed bytecode on the blockchain, allowing for more trustless interaction with the code.

In this tutorial, we'll do the following steps:

- Set up your hardhat config environment in your project
- Use the hardhat-verify plugin to verify a contract address.

## Prerequisites

To follow this tutorial, you should have knowledge of the following:

Hardhat

Basic knowledge of smart contracts

:::note[Hardhat Starter dApp]

A Hardhat Starter dApp has been created with preset configuration for the Rootstock network. Clone and follow the instructions in the README to setup the project. Note: To set the .env variables to match the hardhat.config.ts file, if using the starter dApp for this tutorial.



...

What is hardhat-verify?

Hardhat is a full-featured development environment for contract compilation, deployment and verification.

The hardhat-verify plugin supports contract verification on the Rootstock Blockscout Explorer.

> Note: The hardhat-verify plugin will be available soon on the Rootstock Explorer.

## Installation

```
bash
npm install --save-dev @nomicfoundation/hardhat-verify
```

And add the following code to your hardhat.config.ts file:

```
bash
require("@nomicfoundation/hardhat-verify");
```

Or, if you are using TypeScript, add this to your hardhat.config.ts:

```
bash
import "@nomicfoundation/hardhat-verify";
```

## Usage

You need to add the following Etherscan config to your hardhat.config.ts file:

```
bash
// Hardhat configuration
const config: HardhatUserConfig = {
  defaultNetwork: "hardhat",
  networks: {
    hardhat: {
      // If you want to do some forking, uncomment this
      // forking: {
      //   url: MAINNETRPCURL
      // }
    },
    localhost: {
      url: "http://127.0.0.1:8545",
    },
    rskMainnet: {
      url: RSKMAINNETRPCURL,
      chainId: 30,
      gasPrice: 60000000,
    },
    accounts: [PRIVATEKEY],
  },
  rskTestnet: {
    url: RSKTESTNETRPCURL,
```

```

    chainId: 31,
    gasPrice: 60000000,
accounts:[PRIVATEKEY]
  },
},
namedAccounts: {
  deployer: {
    default: 0, // Default is the first account
    mainnet: 0,
  },
  owner: {
    default: 0,
  },
},
solidity: {
  compilers: [
    {
      version: "0.8.24",
    },
  ],
},
sourcify: {
  enabled: false
},
etherscan: {
  apiKey: {
    // Is not required by blockscout. Can be any non-empty string
    rskTestnet: 'RSKTESTNETRPCURL',
    rskMainnet: 'RSKMAINNETRPCURL'
  },
  customChains: [
    {
      network: "rskTestnet",
      chainId: 31,
      urls: {
        apiURL: "https://rootstock-testnet.blockscout.com/api/",
        browserURL: "https://rootstock-testnet.blockscout.com/",
      }
    },
    {
      network: "rskMainnet",
      chainId: 30,
      urls: {
        apiURL: "https://rootstock.blockscout.com/api/",
        browserURL: "https://rootstock.blockscout.com/",
      }
    },
  ],
},
];

export default config;

```

Now, run the verify task, passing the address of the contract, the network where it's deployed, and any other arguments that was used to deploy the contract:

```
bash
```

```
npm run hardhat verify --network rskTestnet DEPLOYEDCONTRACTADDRESS
```

```
or
```

```
bash
```

```
npm run hardhat verify --network rskMainnet DEPLOYEDCONTRACTADDRESS
```

:::tip[Tip]

- Replace DEPLOYEDCONTRACTADDRESS with the contract address.
- This can be gotten from the MockERC721.json file containing the addresses and abi under the deployments/network folder.

:::

The response should look like this:

```
bash
```

```
npm run hardhat verify --network rskTestnet 0x33aC0cc41B11282085ff6db7E1F3C3c757143722
```

```
Successfully submitted source code for contract
```

```
contracts/ERC721.sol:MockERC721 at 0x33aC0cc41B11282085ff6db7E1F3C3c757143722
```

```
for verification on the block explorer. Waiting for verification result...
```

```
Successfully verified contract MockERC721 on the block explorer.
```

```
https://rootstock-testnet.blockscout.com/address/0x33aC0cc41B11282085ff6db7E1F3C3c757143722#code
```

## Resources

- Visit [hardhat-verify](#)
- Visit [blockscout](#)
- [Hardhat Starter Kit for Rootstock](#)

# index.md:

---

sectionposition: 200

sidebarlabel: Getting Started with Foundry

title: Getting Started with Foundry

description: 'How to write, test, and deploy smart contracts with Foundry'

tags: [rsk, foundry, developers, developer tools, rootstock, testing, dApps, smart contracts]

---

In this guide, we will learn about Foundry and its benefits for smart contract development, how to setup your environment, create a Foundry project and execute a deployment script.

## Prerequisites

To get started with Foundry, ensure the following tools are installed:

- The Rust Compiler
- Cargo Package Manager

> For an easy installation of the above tools, use the [rustup](#) installer.

## Installation

To install, use Foundryup. Foundryup is the Foundry toolchain installer. You can find more information in the Foundry README.

```
bash
curl -L https://foundry.paradigm.xyz | bash
```

Running foundryup by itself will install the latest (nightly) precompiled binaries: forge, cast, anvil, and chisel.

> Visit the installation guides for more information.

Create a foundry project

To start a new project with Foundry, use forge init.

```
bash
forge init hellofoundry
```

> See more details on how to create a new project using the Foundry guide.

Write your first contract

Let's view the file structure for a default foundry project:

```
bash
$ cd hellofoundry
$ tree . -d -L 1
```

```
.
├── lib
├── script
├── src
└── test
```

4 directories

The src directory contains counter smart contract with test written in the test directory. Now, let's build the foundry project.

```
bash
forge build
```

And then run tests.

```
bash
forge test
```

Deploy contract on the Rootstock

To deploy the counter contract on Rootstock mainnet or testnet, further configure Foundry by setting up a

Rootstock RPC url and a private key of an account that's funded with tRBTC.

## Environment Configuration

Once you have an account with a private key, create a .env file in the root of the foundry project and add the variables.

Foundry automatically loads a .env file present in the project directory.

The .env file should follow this format:

```
bash
ROOTSTOCKRPCURL=https://rpc.testnet.rootstock.io/{YOURAPIKEY}
PRIVATEKEY=0x...
```

At the root of the project, run:

```
bash
To load the variables in the .env file
source .env
```

## Modify Deployment Script

Modify the deployment counter deploy script in the scripts directory to use the private key by modifying the run method, see below example:

```
solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Script, console} from "forge-std/Script.sol";

import "../src/Counter.sol";

contract CounterScript is Script {
    function setUp() public {}

    function run() public {
        vm.startBroadcast(vm.envUint("PRIVATEKEY"));

        new Counter();

        vm.stopBroadcast();
    }
}
```

By default, scripts are executed by calling the function named run at the entrypoint.

```
solidity
```

```
uint256 deployerPrivateKey = vm.envUint("PRIVATEKEY");
```

- > - CAUTION: Be cautious when exposing private keys in a .env file and loading them into programs.
- > - This is only recommended for use with non-privileged deployers or for local / test setups.

When calling `vm.startBroadcast()`, the contract creation will be recorded by Forge, and we can broadcast the transaction to deploy the contract on-chain.

Execute the deployment script

We will use Forge to run our script and broadcast the transactions - this can take a little while, since Forge also waits for the transaction receipts.

```
bash
forge script script/Counter.s.sol --rpc-url $ROOTSTOCKRPCURL --broadcast --legacy
```

```
:::info[Info]
```

- EIP-1559 is not supported or not activated on the Rootstock RPC url
- The `--legacy` flag is passed to use legacy transactions instead of EIP-1559.

```
:::
```

The result should look like this:

```
bash
[ ] Compiling...
No files changed, compilation skipped
Script ran successfully.
```

```
== Logs ==
Counter:
```

Setting up 1 EVM.

```
=====
```

Chain 31

Estimated gas price: 0.065164 gwei

Estimated total gas used for script: 138734

Estimated amount required: 0.000009040462376 ETH

```
=====
```

```
##
```

Sending transactions [0 - 0].

```
[00:00:00]
```

```
[#####] 1/1 txes (0.0s)##
```

Waiting for receipts.

```
[00:00:25]
```

```
[#####] 1/1 receipts (0.0s)
```

31

[Success]Hash: 0x015de35ffae94f491d4630f2aec84c49ae8170d5ecf3f4c1cdc8718bc4a00052  
Contract Address: 0x64B24E046259042e16a337Be4648CeAAF8Eb72C6  
Block: 5071408  
Gas Used: 106719

=====

ONCHAIN EXECUTION COMPLETE & SUCCESSFUL.

Total Paid: 0. ETH (106719 gas avg 0 gwei)

Transactions saved to: /hellofoundry/broadcast/Counter.s.sol/31/run-latest.json

Sensitive values saved to: /hellofoundry/cache/Counter.s.sol/31/run-latest.json

> The broadcast directory will be updated automatically with the latest output of the deployment.

> See the foundry deployment documentation.

# architecture.md:

---

sidebarlabel: Architecture  
sidebarposition: 980  
title: RIF Relay - Architecture  
tags: [rif, envelope, relay, integrate]  
description: RIF Relay Architecture  
---

The RIF Relay system is designed to achieve transaction sponsorship at a low cost. The cost of the relay service provided by the “sponsors” is agreed upon among the parties off-chain. The low cost of transactions on Rootstock (RSK) contributes to keeping overall service costs low as well.

The RIF Relay system is made up of various components, some of which are essential and others which are auxiliary.

An overview of this is as follows:

On-chain, the system cannot work without its Smart Contracts, which encompass Smart Wallets plus the Relay Hub and Verifiers.

Off-chain, at least one Relay Server is needed to interact with the contracts. Without a Relay Server, envelopes cannot be created and sent to the contracts.

Details for each of these components are expanded down below, as well as an introductory glossary.

Glossary

| Term  | Description |
|-------|-------------|
|       | -----       |
| ----- | -----       |

| Sponsor | A third party that pays the gas consumed by a sponsored transaction (see below) by submitting it to the blockchain. |

| Sponsored Transaction | A transaction sent by the requester (see below) through the Sponsor, this type of transaction aims to separate the gas payer from the sender of the transaction. |

| Requester | It's an EOA (see below). The requester sends a sponsored transaction to the Sponsor. They do not pay the gas with native cryptocurrency but with an accepted token by the Sponsor, if they don't subsidize it. |

| Recipient | An abbreviation for recipient contract. It's the destination of the requester's transaction. |

| Envelope | Using the "envelopes" analogy, it's the transaction, (funded with native cryptocurrency as gas) sent by the Sponsor to the blockchain, that wraps the requester's transaction payload (sponsored transaction). |

| RIF Relay | The entire system which allows the relay of sponsored transactions. |

| DoS | A Denial of Service is an information-security threat whose goal is to become a service unavailable. |

| DeFi | An acronym for Decentralized Finance, it's a novel form for finance based in blockchain technology. |

| EOA | An Externally Owned Account (EOA) is an account managed with a key, which is capable of signing and sending transactions, and paying the cost for it. |

| Fee | Token amount that is being charged for each relayed transaction. |

| Revenue Sharing Model | A way to relay transactions so that fees are shared among multiple partners. |

| Fees Receiver | Is the designated Worker/Collector that will receive the fees |

## On-Chain components

### Relay Hub

The Relay Hub is the main component of the RIF Relay architecture. It acts as an interface with the Relay Server and the whole on-chain architecture. It forwards all the transactions to their respective contracts while checking the validity of the worker that is processing the transaction.

It also forms part of the Relay Workers registration process together with the Relay Managers. Furthermore, the Relay Hub keeps the stake amount for each Relay Manager to guarantee good behavior from their workers.

The account staking for a specific Relay Manager for the first time becomes the owner of the stake, only this account can make subsequent stakes for this specific RelayManager.

When a Relay Manager unauthorized a Relay Hub, it means it is unstaking from it, which also means not being able to relay through that hub any more. Any balance held in the Relay Hub is sent to the original sender of the stake (the owner).

Unstaking has a predefined delay (in blocks). This is intended to prevent the Relay Manager from unstaking before a slashing that was going to occur.

### Smart Wallet

It's the "contract-based account" owned by the Requester's EOA. Before executing any transaction using the smart wallet, the smart wallet contract needs to be deployed.

Smart Wallet are contracts that verify forwarded data and subsequently invoke the recipient contract of the transaction. Creating smart wallets does not have any gas cost providing the advantage that can be deployed only when necessary.

It is the component that calls the Recipient contract (i.e, the msg.sender address the Recipient will see). During the execution, the contract verifies the Relay Request and, if it's valid, it calls the defined Recipient's function, otherwise it reverts the invocation. The verification includes checking that the owner



of the SmartWallet made the request, rejecting any request with an invalid signature, and preventing replay attacks using a nonce.

The smart wallet was designed to only interact with the RIF Relay system, therefore any native currency balance will be transferred back to the owner of smart wallet after each transaction.

### Native Holder Smart Wallet

The native holder smart wallet is a smart wallet that was designed to have interactions outside the RIF Relay system. This means that it can hold native currency as it's name describes.

The behavior of the native holder smart wallet is the same as the smart wallet with the difference that the native currency will not be transferred back to the owner after each transaction and can dispose the usage of the native currency.

### Custom Smart Wallet

The custom smart wallet is a smart wallet that was designed to execute custom logic after each transaction. The custom logic is created at the moment of the contract deployment and can be executed on each transaction.

### Relay Manager

An EOA that has a staked balance. Any penalization done against a Relay Worker impacts the Relay Manager's stake. A Relay Worker can be managed by only one Relay Manager. A Relay Manager can have one or more Relay Workers. The responsibilities of the Relay Manager are: registering the Relay Server and adding Relay Workers, both in the Relay Hub.

### Stake Manager

The Stake Manager supports multiple Relay Hubs, the stakers are the Relay Managers and they can authorize/de-authorize specific Relay Hubs so they can penalize the managers if needed. The staked cryptocurrency is held in the StakeManager contract.

The account staking for a specific Relay Manager for the first time becomes the owner of the stake, only this account can make subsequent stakes for this specific RelayManager.

When a Relay Manager unauthorised a Relay Hub, it means it is unstaking from it, which also means not being able to relay through that hub any more. Any balance held in the Relay Hub is sent to the original sender of the stake (the owner). Also, the workers' balances are transferred to the stake owner, and, if configured, the Relay Manager's balance can also be transferred to the stake owner.

Unstaking has a predefined delay (in blocks). This is intended to prevent the Relay Manager from unstaking before a slashing that was going to occur.

### Relay Worker

An EOA that belongs to only one Relay Manager. Since the Relay Worker is the sender of the request, it is the one that pays for the gas fees for each transaction. It may also collect the fees in ERC20 that are charged for relaying the transactions.

### Relay & Deploy Verifier

Contracts that authorize a specific relay or deploy request (see the Relay & Deploy Requests section).

Two example implementations are provided:

- Relay Verifier: The Relay Verifier has a list of tokens that it accepts. When it receives a relay request, it checks the token's acceptance and the payer's balance for the token.
- Deploy Verifier: An implementation used in the SmartWallet deployment process. It performs the same

relay verifier checks but also makes sure that the SmartWallet to be deployed doesn't already exist. It also checks that a Proxy Factory address is provided in the Relay Request.

## Collector

The Collector is an optional smart contract used for the Revenue Sharing feature. Normally, relay fees are paid to the worker account which relays the transaction.

Collector contracts are designed to hold revenue generated from relayed transactions so that in the future they can be later given out to partners according to the distribution of shares written in the contract. They are initialized with a specific token to hold, a set of partner addresses (each set with their own share of collected revenues) plus an owner which can modify the shares or execute the withdrawal and distribution of funds. Shares for each partner are expressed in integers representing a percentage and must add up to exactly 100. Any number of partners can be specified.

The owner of the contract can be any address, including but not limited to a multisig contract (see Gnosis Safe Contracts). Using a multisig address can be a good idea if ownership of the contract needs to be shared, so that decisions like distributing collected fees to partners or modifying revenue shares can be taken collectively. An ownership transfer function is also available.

Any number of Collector contracts can be deployed and used to share revenue, as long as their addresses are specified in relay requests. The withdrawal of funds from any Collector contract is completely independent of the RIF Relay flow and can be executed at any arbitrary point in time.

For steps on how to deploy a Collector contract (plus other technical details) please see the Collector section of the deployment process.

## Proxies

### Template

It's the logic that would be executed on each transaction. In this specific scenario, it is the Smart Wallet contract.

### Proxy Factory

Factory of Proxies to the SmartWallet. The proxy factory is in charge of deploying the smart wallets contracts using the template, during the deployment it executes the initialization from each smart wallet.

### Proxy

The proxy is only implemented in the Custom Smart Wallet because it delegates every call to a SmartWallet logic address. This proxy is the one instantiated per SmartWallet, and it will receive the SmartWallet address as Master Copy (MC). So every call made to this proxy will end up executing the logic defined in the MC.

For the transaction execution (execute() call), MC logic will do the signature verification and payment. Then it will execute the request, and, if custom logic was defined, it will forward the flow to it before returning.

During the initialization of the Custom Smart Wallet a custom logic can be set (which would impact the proxy's state of course), the initialization process and set of the custom logic can only be done during the deployment of the smart wallet.

## GSNEip712Library

This is an auxiliary library that bridges the Relay Request into a call of a Smart Wallet or Proxy Factory (in such case, the Request is a Deploy Request).

Detailed documentation can be found [here](#).

## Off-chain components

### Relay Server

The Relay Server receives sponsored transactions via HTTP.

The server has only one Relay Manager address and at least one Relay Worker, and points to just one Relay Hub.

When the Relay Server receives an HTTP Relay request, it creates an Envelope, wrapping the sponsored transaction, signs it using its Relay Worker account and then sends it to the Relay Hub contract.

The server is a service daemon, running as an HTTP service. It advertises itself (through the Relay Hub) and waits for client requests.

The Relay Server has mechanisms that try to avoid running out of balance in the workers. The Relay Manager keeps sending native cryptocurrency to the workers based on a specific minimum balance.

### Start Flow

The start flow diagram represents the process that is followed by the Relay Server to start receiving requests, even that the server will be receiving requests doesn't mean that can handle it, since it needs balance to process each request.

!Relay - Start Flow

1. Generates(private keys) the Workers and Manager accounts.
2. Initialise the instance for each contract that will be interacting with the server.
  - The RelayHub contract is the key contract for the start flow since its the contract that have the events of interest.
3. Initialise the Relay Server.
  - The Relay Server has all logic for the interaction between off-chain and on-chain components.
4. Initialise the RegistrationManager.
  - The Registration Manager starts querying for events related to the registration process(StakeAdded, WorkerAdded) to identify if can register the Server on the RelayHub.
5. The Relay Server start querying for changes on the blockchain using the RelayHub.

### Register Flow

The register flow diagram represents the process to provide the necessary stake/balance to the manager/workers for the Relay Server start processing requests and to register the server in the RelayHub.

!Relay - Register Flow

1. Gets the Relay Server data.
2. Validate if the server was already register in the RelayHub.
3. Initialise the instance for each contract that will be interacting with the server.
  - The RelayHub contract is the key contract for the register flow since its the contract that have the events of interest.
4. Query the stakeInfo for the manager and validates if already staked.
5. Funds the manager if necessary.

### Interval Handler

The interval handler diagram represents the process from the Relay Server to interact with the blockchain and process the transactions.

## !Relay - Interval Handler Flow

1. Get the latest mined block by the blockchain.
2. Check if the Relay Server state needs to be refreshed based on the blocks mined.
3. Refresh the gas price.
4. Get the past events from the RelayHub.
5. Add the workers and register the Relay Server if meets the requirements.
6. Keep listening for transactions and new events.

## Relay & Deploy Requests

A relay request is the sponsored transaction, the structure used to relay a transaction. It is formed by Relay Data and Forward Request:

- Relay Data: All information required to relay the defined Forward Request.
- Forward/Deploy Request: It is formed by all the “common” transaction fields in addition to all the token-payment data.

When the Sponsor creates an Envelope (the actual blockchain transaction to submit), it will add this Relay Request (sponsored transaction) as part of the encoded data, along with the contract and method to call (RelayHub and relayCall respectively)

The Relay request is structure that wraps the transaction sent by an end-user. It includes data required for relaying the transaction e.g. address of the payer, address of the original requester, token payment data.

The Deploy request structure that wraps the transaction sent to deploy a Smart wallet.

## Tools

### Relay Client

This is a typescript library to interact with the RIF Relay system. It provides APIs to find a relay, and to send transactions through it. It also exposes methods to interact with the blockchain.

It can work as an access point to the Relay system. It creates, signs, and sends the Sponsored transaction, which is signed by the requester and forwarded to the Relay Server via the HTTP protocol.

It is not strictly needed since any dApp or user could relay transactions using merely a Relay Server and the smart contracts, although this is arguably harder to do manually.

### Relay Provider

It's the access point to the RIF Relay system. It wraps the Relay Client to provide Ethers.js compatibility.

## Execution flow

Relaying (Smart Wallet already deployed)

## !Relay - Execution Flow

1. A Requester creates a request.
2. A Requester sends the request to the Relay Client (through a Relay Provider).
3. The Relay Client wraps the request into a Relay Request and signs it.

4. The Relay Client sends the Relay Request to the Relay Server (via HTTP Client ↔ HTTP Server).
5. The Relay Server create a transaction including the Relay Request and signs it with a Relay Worker account.
6. The Relay Worker account is an EOA registered in the Relay Hub.
7. The Relay Server sends the transaction to the Relay Hub using the same worker account as the previous step, executing the relayCall function of the Relay Hub contract.
  - When the Relay Hub receives a transaction from a Relay Worker, it verifies with the Stake Manager that the Worker's Relay Manager has indeed locked funds for staking. If not, the execution is reverted.
  - The Relay Worker account must have funds to pay for the consumed gas (RBTC).
  - This verification is done in the Relay Client and in the Relay Server as well, by calling the Relay Verifier. The verifier checks that it accepts the token used to pay and that the payer has a sufficient token balance. In addition, it verifies that the used smart wallet is the correct one.
8. The RelayHub instructs the Smart Wallet to execute the Relay Request through the GsnEip712Library library.
9. The Smart Wallet checks the signature and the nonce of the Requester, reverting if it fails the checks.
10. Then, the Smart Wallet performs the token transfer between the Requester and the token recipient, using the data received within the Relay Request.
11. It invokes the recipient contract with the indicated method in the Forward Request.

## Sponsored Smart Wallet deployment

### !Relay - Sponsored Smart Wallet

The gas-less requester has a SmartWallet address where they receive tokens but don't use them. If the requester needs to call a contract, e.g., to send the tokens to another account, they must deploy a Smart Wallet first.

1. A Requester creates a deploy wallet request.
2. A Requester sends the request to the Relay Server through the Relay Client.
3. The Relay Client wraps the transaction sent by the Requester in a Deploy Request to create the Smart Wallet and signs it.
4. The Relay Client sends to the Relay Server a Deploy Request (via HTTP Client ↔ HTTP Server).
5. The Relay Server signs the transaction containing the Deploy Request, with the Relay Worker account.
6. The Relay Server sends the request to the Relay Hub using the Relay Worker account executing the relayCall function of the Relay Hub contract.
  - Here's where the Relay Server will typically call the Deploy Verifier to ensure:
    - The Verifier accepts the offered tokens.
    - The Proxy Factory instance to use is known by the verifier.
    - The Proxy Factory contract isn't creating an existing Smart Wallet.
    - The requester has enough tokens to pay.
7. The Relay Hub calls the Proxy Factory using the method relayedUserSmartWalletCreation.
8. The Proxy Factory performs the following checks:
  - Checks the sender's nonce.
  - Checks the Deploy Request signature.
9. The Proxy Factory creates the Smart Wallet using the create2 opcode.
10. Then it calls the initialize function in the Smart Wallet contract.
  - The Smart Wallet, during its initialization, executes the token transfer.
  - Then it initializes the Smart Wallet state and sets the requester's EOA as the owner of the Smart Wallet.
  - In the case there is a custom logic, its initialization is called as well.

## Deprecated

### Paymaster

V0.2 deprecated the Paymaster contracts in favor of the Verifiers (see versions).

# contracts.md:

---  
sidebarlabel: Contracts  
sidebarposition: 700  
title: RIF Relay - Contracts  
tags: [rif, envelope, relay, integrate]  
description: RIF Relay Contracts  
---

Mainnet

Version 1

Primary contracts

| Contract           | Address                                    |
|--------------------|--|
| Penalizer          | 0x4fd591b8330d352c57CA1CC1dA172dCa516722E3 |
| RelayHub           | 0x438Ce7f1FEC910588Be0fa0fAcD27D82De1DE0bC |
| SmartWallet        | 0x59C40304E8a428BF1D17f03a6aE84B635964DB19 |
| SmartWalletFactory | 0x9EEbEC6C5157bEE13b451b1dfE1eE2cB40846323 |
| DeployVerifier     | 0x2FD633E358bc50Ccf6bf926D621E8612B55264C9 |
| RelayVerifier      | 0x5C9c7d96E6C59E55dA4dCf7F791AE58dAF8DBc86 |

For CustomSmartWallet support

| Contract                        | Address                                    |
|---------------------------------|--|
| CustomSmartWallet               | 0x4b2464E8d062633D18ba0928c064037Da415eD1f |
| CustomSmartWalletFactory        | 0x0002E79883280C1717e41EE5D3705D55960B5bAe |
| CustomSmartWalletDeployVerifier | 0x5910868059431026ACa58a73124DedbEF4cb97db |
| CustomSmartWalletRelayVerifier  | 0xb6eFDB335b4D52705e9973069500fd79410BEC01 |

For Testing purposes

| Contract        | Address                                    |
|-----------------|--|
| SampleRecipient | 0x479eA66AAEfC00EdC1590c9da07152def9452cf9 |
| TestToken       | 0xe49b8906A3ceFd184621A4193e2451b1c3C3dB0B |

Testnet

Version 1

Primary contracts

| Contract    | Address                                    |
|-------------|--|
| Penalizer   | 0x8e67406bD3A926b43Fda158C673230B77f874CDd |
| RelayHub    | 0xAd525463961399793f8716b0D85133ff7503a7C2 |
| SmartWallet | 0x86bD3006B757614D17786428ADf3B442b2722f59 |

|                    |  |  |
|--------------------|--|--|
| SmartWalletFactory | 0xCBc3BC24da96Ef5606d3801E13E1DC6E98C5c877 |  |
| DeployVerifier     | 0xc67f193Bb1D64F13FD49E2da6586a2F417e56b16 |  |
| RelayVerifier      | 0xB86c972Ff212838C4c396199B27a0DBe45560df8 |  |

For CustomSmartWallet support

|                                 |  |  |
|---------------------------------|--|--|
| Contract                        | Address                                    |  |
| -----                           | -----                                      |  |
| CustomSmartWallet               | 0x2c66922D704999Ff9F378838172fe5c5e0Ac2d27 |  |
| CustomSmartWalletFactory        | 0x91C186Dcb11EcBf234c778D7779e8e10f8ADD1a8 |  |
| CustomSmartWalletDeployVerifier | 0x87421afb27FC680D99E82Bce8Df140E81F5c11a3 |  |
| CustomSmartWalletRelayVerifier  | 0x6e23B72723886b066a5C26Baa2b2AfB0b1d51e5c |  |

For Testing purposes

|                 |  |  |
|-----------------|--|--|
| Contract        | Address                                    |  |
| -----           | -----                                      |  |
| SampleRecipient | 0xc4B8B6C02EC34d84630Ba8C684954a0A04C656FC |  |
| TestToken       | 0x3F49BaB0afdC36E9f5784da91b32E3D5156fAa5C |  |

Version 0.2

Primary contracts

|                    |  |  |
|--------------------|--|--|
| Contract           | Address                                    |  |
| -----              | -----                                      |  |
| Penalizer          | 0x5FdeE07Fa5Fed81bd82e3C067e322B44589362d9 |  |
| RelayHub           | 0xe90592939fE8bb6017A8a533264a5894B41aF7d5 |  |
| SmartWallet        | 0x27646c85F9Ad255989797DB0d99bC4a9DF2EdA68 |  |
| SmartWalletFactory | 0xEbb8AA43CA09fD39FC712eb57F47A9534F251996 |  |
| DeployVerifier     | 0x345799D90aF318fd2d8CbA87cAD4894feF2f3518 |  |
| RelayVerifier      | 0xDe988dB9a901C29A9f04050eB7ab08f71868a8fc |  |

For CustomSmartWallet support

|                                 |  |  |
|---------------------------------|--|--|
| Contract                        | Address                                    |  |
| -----                           | -----                                      |  |
| CustomSmartWallet               | 0xB8dB52615B1a94a03C2251fD417cA4d945484530 |  |
| CustomSmartWalletFactory        | 0xA756bD95D8647be254de40B842297c945D8bB9a5 |  |
| CustomSmartWalletDeployVerifier | 0x3c26685CE3ac89F755D68A81175655b4bBE54AE0 |  |
| CustomSmartWalletRelayVerifier  | 0xBcCA9B8faA9cee911849bFF83B869d230f83f945 |  |

For Testing purposes

|                 |  |  |
|-----------------|--|--|
| Contract        | Address                                    |  |
| -----           | -----                                      |  |
| SampleRecipient | 0x4De3eB249409e8E40a99e3264a379BCfa10634F5 |  |
| TestToken       | 0x77740cE4d7897430E74D5E06540A9Eac2C2Dee70 |  |

Version 0.1

|              |  |  |
|--------------|--|--|
| Contract     | Address                                    |  |
| -----        | -----                                      |  |
| StakeManager | 0x4aD91a4315b3C060F60B69Fd0d1eBaf16c14148D |  |
| Penalizer    | 0xd3021763366708d5FD07bD3A7Cd04F94Fc5e1726 |  |

```
| RelayHub      | 0x3f8e67A0aCc07ff2F4f46dcF173C652765a9CA6C |
| TestRecipient | 0xFBE5bF13F7533F00dF301e752b41c96965c10Bfa |
| SmartWallet   | 0xE7552f1FF31670aa36b08c17e3F1F582Af6302d1 |
| ProxyFactory  | 0xb7a5370F126d51138d60e20E3F332c81f1507Ce2 |
| DeployVerifier | 0x3AD4EDEc75570c3B03620f84d37EF7F9021665bC |
| RelayVerifier  | 0x053b4a77e9d5895920cBF505eB8108F99d929395 |
```

# deployment.md:

```
---
sidebarlabel: RIF Relay Deployment
sidebarposition: 500
title: RIF Relay Deployment
tags: [rif, envelope, relay, integration guide]
description: RIF Relay deployment process
---
```

Set Up RIF Relay Contracts and Server

Deploy Contracts

Start by deploying on-chain components. All tools needed are in the RIF Relay Contract repository

Regtest

- 1. Clone the Repository:  
bash  
git clone https://github.com/rsksmart/rif-relay-contracts
- 2. Navigate to the directory and install dependencies:  
bash  
cd rif-relay-contracts  
npm install
- 3. Deploy the contract:  
bash  
npx hardhat deploy --network regtest

> This uses the Regtest configuration from hardhat.config.ts.

After deployment, you'll see a summary of the deployed contracts. This summary includes the on-chain components essential for RIF Relay, and additional contracts for testing and validation purposes.

text

| (index) |             | Values                                       |
|---------|-------------|--|
|         | Penalizer   | '0x77045E71a7A2c50903d88e564cD72fab11e82051' |
|         | RelayHub    | '0xDA7Ce79725418F4F6E13Bf5F520C89Cec5f6A974' |
|         | SmartWallet | '0x83C5541A6c8D2dBAD642f385d8d06Ca9B6C731ee' |



|                                       |  |
|---------------------------------------|--|
| SmartWalletFactory                    | '0xE0825f57Dd05Ef62FF731c27222A86E104CC4Cad' |
| DeployVerifier                        | '0x73ec81da0C72DD112e06c09A6ec03B5544d26F05' |
| RelayVerifier                         | '0x03F23ae1917722d5A27a2Ea0Bcc98725a2a2a49a' |
| CustomSmartWallet                     | '0x1eD614cd3443EFd9c70F04b6d777aed947A4b0c4' |
| CustomSmartWalletFactory              | '0x5159345aaB821172e795d56274D0f5FDFdC6aBD9' |
| CustomSmartWalletDeployVerifier       | '0x7557fcE0BbFAe81a9508FF469D481f2c72a8B5f3' |
| CustomSmartWalletRelayVerifier        | '0x0e19674ebc2c2B6Df3e7a1417c49b50235c61924' |
| NativeHolderSmartWallet               | '0x4aC9422c7720eF71Cb219B006aB363Ab54BB4183' |
| NativeHolderSmartWalletFactory        | '0xBaDb31cAf5B95edd785446B76219b60fB1f07233' |
| NativeHolderSmartWalletDeployVerifier | '0xAe59e767768c6c25d64619Ee1c498Fd7D83e3c24' |
| NativeHolderSmartWalletRelayVerifier  | '0x5897E84216220663F306676458Afc7bf2A6A3C52' |
| UtilToken                             | '0x1Af2844A588759D0DE58abD568ADD96BB8B3B6D8' |
| VersionRegistry                       | '0x8901a2Bbf639bFD21A97004BA4D7aE2BD00B8DA8' |

The deployment summary shows two sets of Smart Wallets, each paired with its verifiers. This is because the verifier is used for both deployment and transaction validation. For testing purposes, the focus will be on using these Smart Wallet Contracts.

#### Testnet

1. Ensure your account is funded. You can get funds from the tRBTC Faucet.
2. Deploy on Testnet:

```
bash
```

```
npx hardhat deploy --network testnet
```

> Remember to configure Testnet in hardhat.config.ts. Existing RIF Relay contracts deployed on Testnet can be found in the contracts section.

#### Mainnet

1. Ensure your account is funded.
2. Deploy on Mainnet:

```
bash
```

```
npx hardhat deploy --network mainnet
```

> Ensure Mainnet is set up in hardhat.config.ts. Existing RIF Relay contracts deployed on Mainnet can be found in the contracts section.

#### Revenue Sharing

Revenue Sharing is an optional feature in RIF Relay that can be implemented using collector contracts. You can deploy multiple Collector contracts, but they are not included in the default Relay contract deployment. For detailed information on Collector contracts, refer to the architecture documentation.

Before deploying a Collector contract ensure the following:

1. Ensure the chosen token for the Collector contract is the same as the one used for transaction fees.
  - > Note: You cannot retrieve any other tokens other than the one set during Collector deployment.
2. Select an appropriate owner for the Collector contract. This owner doesn't have to be the deployer but must have the authority to execute the withdraw function, or else the revenue funds will be locked in the contract.
3. Set up partners and their share percentages, ensuring the total adds up to 100%. Incorrectly sent tokens to an inaccessible address without a private key from the beneficiary will be lost. For an example of a structurally valid revenue shares definition see sample configuration.

## Regtest

To deploy the Collector contract, we'll use the RIF Relay Contract.

1. Create a configuration file named `deploy-collector.input.json` with the required structure:

```
json
{
  "collectorOwner": "0xCD2a3d9F938E13CD947Ec05AbC7FE734Df8DD826",
  "partners": [
    {
      "beneficiary": "0x7986b3DF570230288501EEa3D890bd66948C9B79",
      "share": 20
    },
    {
      "beneficiary": "0x0a3aA774752ec2042c46548456c094A76C7F3a79",
      "share": 35
    },
    {
      "beneficiary": "0xCF7CDBbB5F7BA79d3ffe74A0bBA13FC0295F6036",
      "share": 13
    },
    {
      "beneficiary": "0x39B12C05E8503356E3a7DF0B7B33efA4c054C409",
      "share": 32
    }
  ],
  "tokenAddresses": ["0x1Af2844A588759D0DE58abD568ADD96BB8B3B6D8"],
  "remainderAddress": "0xc354D97642FAa06781b76Ffb6786f72cd7746C97"
}
```

> Note: The `collectorOwner`, `beneficiaries`, and `remainderAddress` are the first five accounts provided by the node in Regtest.

2. Deploy the contract:

```
bash
npx hardhat collector:deploy --network regtest
```

The collector is ready and can start receiving fees.

## Testnet

Using the configuration file you created in the regtest section, run this command to deploy the contract:

```
js
npx hardhat collector:deploy --network testnet
```

## Mainnet

Using the configuration file you created in the regtest section, run this command to deploy the contract:

```
js
npx hardhat collector:deploy --network mainnet
```

## Allow Tokens

RIF Relay only accepts whitelisted tokens, primarily to ensure only tokens of value to the operator are accepted. To whitelist a token:

Execute the `acceptToken(address token)` function on the Relay Verifiers contracts, which include:

- SmartWalletDeployVerifier
- SmartWalletRelayVerifier

:::info[Note]

This action must be performed by the contracts' owner, typically the account that conducted the deployment.

:::

## Regtest

In the RIF Relay Contracts, execute this command:

```
js
npx hardhat allow-tokens --network regtest --token-list <TOKENADDRESSES>
```

> <TOKENADDRESSES> is a comma-separated list of the token addresses to be allowed on the available verifiers. The `allowTokens` uses the first account (referred to as `account[0]`) as the owner of the contracts. This is important because only the account owner can allow tokens.

## Testnet

In the RIF Relay Contracts, execute the command:

```
js
npx hardhat allow-tokens --network testnet --token-list <TOKENADDRESSES>
```

> <TOKENADDRESSES> is a comma-separated list of the token addresses to be allowed on the available verifiers. The `allowTokens` script will use the Testnet network configured in the `hardhat.config.ts`, this network will be required to use the account that deployed the contracts.

> You can also modify the allowed tokens for specific verifiers only by using the `--verifier-list` option as follows:

```
js
npx hardhat allow-tokens --network testnet --token-list <TOKENADDRESSES> --verifier-list
<VERIFIERADDRESSES>
```

> The <TOKENADDRESSES>, <VERIFIERADDRESSES> is a comma-separated list of verifier addresses to allow the tokens for.

## Mainnet

In the RIF Relay Contracts, execute the command:

```
js
npx hardhat allow-tokens --network mainnet --token-list <TOKENADDRESSES>
```

> <TOKENADDRESSES> is a comma-separated list of the token addresses to be allowed on the available verifiers. The allowTokens script will use the Mainnet network configured in hardhat.config.ts, this network will be required to use the account that did the deployment of the contracts.

> You can also modify the allowed tokens for specific verifiers only by using the --verifier-list option as follows:

```
js
npx hardhat allow-tokens --network testnet --token-list <TOKENADDRESSES> --verifier-list
<VERIFIERADDRESSES>
```

> The <TOKENADDRESSES>, <VERIFIERADDRESSES> is a comma-separated list of verifier addresses to allow the tokens for.

...info[Note]

The network name; regtest, testnet, or mainnet, is an optional parameter that is taken from the hardhat.config.ts file. The network name you specify must be the same as the one used to deploy the contract.

...

## Run the RIF Relay Server

After setting up on-chain components, the next step is to set up off-chain components, using the RIF Relay Server.

Configuration of the Relay Server is streamlined using the node-config package. For detailed advantages of this package, visit their wiki.

<b>The TL;DR:</b> In the config directory, create a file named local.json.

For visual insights into how the Relay Server functions, refer to the diagrams available [here](#).

## Regtest

Here's a configuration example for setting up the RSKj node locally with the contracts deployed in Regtest:

```
json
{
  "app": {
    "url": "http://127.0.0.1",
    "port": 8090,
    "devMode": true,
    "logLevel": 1,
    "workdir": "./enveloping/environment/",
  },
  "blockchain": {
    "rskNodeUrl": "http://127.0.0.1:4444",
  },
  "contracts": {
    "relayHubAddress": "0xDA7Ce79725418F4F6E13Bf5F520C89Cec5f6A974",
    "relayVerifierAddress": "0x03F23ae1917722d5A27a2Ea0Bcc98725a2a2a49a",
    "deployVerifierAddress": "0x73ec81da0C72DD112e06c09A6ec03B5544d26F05"
  }
}
```

> Note: Relay and Deploy verifiers use the contracts from the Smart Wallet section in the summary.

The meaning of each key can be found in RIF Relay Server Configuration

To start the server, run the following command:

```
js
  npm run start
```

> By default, the server uses the default.json5 file in the config directory. Depending on the profile in NODEENV, the values in the default.json5 file is overridden.

At this point the server should be running and ready to start processing transactions, however, you still need to register the off-chain components in the Relay Hub.

To enable the server for transaction processing, you must register the off-chain components in the Relay Hub. This step requires the server to be active. To register the components, in a different terminal window, execute the following command:

```
js
  npm run register
```

The register process performs the following actions:

- Stakes the Relay Manager
- Adds the Relay Worker
- Registers the Relay Server

The server is now ready to start processing transactions and a ready message is displayed on the console. For more details on configuration and registration parameters, refer to the RIF Relay Server documentation.

## Testnet

Here's an example configuration file using the off-chain components deployed on the Rootstock Testnet (<https://rpc.testnet.rootstock.io/{YOURAPIKEY}>).

> Important: Due to specific modules enabled in the RSKj nodes, the RIF Relay Server cannot connect to the public nodes.

```
json
{
  "app": {
    "url": "https://backend.dev.relay.rifcomputing.net",
    "port": 8090,
    "devMode": true,
    "logLevel": 1,
    "feePercentage": "0",
    "workdir": "/srv/app/environment"
  },
  "blockchain": {
    "rskNodeUrl": "http://172.17.0.1:4444"
  },
  "contracts": {
    "relayHubAddress": "0xAd525463961399793f8716b0D85133ff7503a7C2",
```

```

    "relayVerifierAddress": "0xB86c972Ff212838C4c396199B27a0DBe45560df8",
    "deployVerifierAddress": "0xc67f193Bb1D64F13FD49E2da6586a2F417e56b16"
  }
}

```

> The contracts used in this setup are the primary contracts available on the Rootstock network. These primary contracts, however, do not include support for the CustomSmartWallet.

For details of each configuration key used in setting up the RIF Relay Server, refer to the RIF Relay Server Configuration documentation.

To start the server, execute the following command:

```

js
npm run start

```

> By default, the server uses the default.json5 file in the config directory. Depending on the profile in NODEENV, the values in the default.json5 file is overridden. Therefore you need to setup the NODEENV environment to testnet.

At this point, the server should be running and ready to start processing transactions; however, you still need to register the off-chain components in the Relay Hub. For the registration process, the Relay Manager and Worker must have funds.

To get the addresses, this requires the server to be active. In a different terminal window, execute the following command:

```

bash
curl http://<SERVERURL>/chain-info

```

```

json
{
  "relayWorkerAddress": "0xabf898bd73b746298462915ca91623f5630f2462",
  "relayManagerAddress": "0xa71d65cbe28689e9358407f87e0b4481161c7e57",
  "relayHubAddress": "0xe90592939fE8bb6017A8a533264a5894B41aF7d5",
  "feesReceiver": "0x52D107bB12d83EbCBFb4A6Ad0ec866Bb69FdB5Db",
  "minGasPrice": "6000000000",
  "chainId": "31",
  "networkId": "31",
  "ready": false,
  "version": "2.0.1"
}

```

1. Send an arbitrary amount of tRBTC, 0.001 tRBTC for example, to the Worker and Manager.
2. Now execute the register command.

```

js
npm run register

```

Here's an example of the register.json5

```

json
{
  "register": {
    "stake": "REGISTERSTAKE",
    "funds": "REGISTERFUNDS",
    "mnemonic": "REGISTERMNEMONIC",
    "privateKey": "REGISTERPRIVATEKEY",
    "hub": "REGISTERHUBADDRESS",
    "gasPrice": "REGISTERGASPRICE",
    "relayUrl": "REGISTERRELAYURL",
    "unstakeDelay": "REGISTERUNSTAKEDELAY"
  }
}

```

The register process performs the following actions:

- Stakes the Relay Manager
- Adds the Relay Worker
- Registers the Relay Server

The server is now ready to start processing transactions and a ready message is displayed on the console. For more details on configuration and registration parameters, refer to the RIF Relay Server documentation.

## Mainnet

- To run RIF Relay Server on the Rootstock Mainnet, the procedure is the same as the one on Testnet, the only difference is the configuration. Configure it to use contracts deployed on Mainnet and an RSKj node connected to Mainnet.

# develop.md:

```

---
sidebarlabel: Develop
sidebarposition: 600
title: RIF Relay Develop
tags: [rif, envelope, relay, user, guide]
description: RIF Relay deployment process
---

```

## Initializing the project

To use RIF Relay, follow these steps to build the project.

## Project structure

The project is divided into multiple modules that interact with each other. Each project has its own documentation in its repository.

1. RIF Relay Contracts
2. RIF Relay Client
3. RIF Relay Server
4. RIF Relay Sample dApp

## Committing changes

To contribute to the project, create a branch with the name of the new feature you are implementing (e.g. gas-optimization). When you commit to git, a hook is executed. The hook executes a linter and all the tests.

# gas-costs.md:

```
---
sidebarlabel: Gas Costs
sidebarposition: 950
title: RIF Relay - Gas Costs
tags: [rif, envelope, relay, integrate]
description: RIF Relay Gas Costs
---
```

The overhead gas cost is the extra amount of gas required to process the relay call requested by the user. Let's call X the gas consumed by the destination contract method call, and Y the total gas consumed by the relay call, then the relay call cost (i.e. overhead gas cost) is:  $Z = Y - X$ .

### SmartWallet templates

RIF Relay V0.1 only has one SmartWallet template, which can be used as-is, or be injected with extra logic during the SmartWallet instance creation.

V0.2 introduces a cheaper template (SmartWallet), to be used when there's no need for extra custom-logic in the smart wallets. The behaviour is the same as the CustomSmartWallet template of V0.2, but without this capability.

Gas cost from the deployment of each template.

| RIF Version | SW Template       | Avg. overhead gas |
|-------------|-------------------|-------------------|
| 0.1         | SmartWallet       | 172400            |
| 0.2         | CustomSmartWallet | 98070             |
| 0.2         | SmartWallet       | 97695             |
| 1           | CustomSmartWallet | TBD               |
| 1           | SmartWallet       | TBD               |

:::tip[Note]

The instance of CustomSmartWallet used didn't point to any extra custom logic.

:::

# installation-requirements.md:

```
---
sidebarlabel: Setup
sidebarposition: 300
title: RIF Relay Installation Requirements
tags: [rif, envelope, relay, user, guide]
description: Requirements for installing RIF Relay
---
```

To set up the RIF Relay system running locally there are some tools that are required. All of these tools are open source and have their own support page. The functionality of RIF Relay does not depend on



these technologies and could be updated or replaced, if necessary.

## Hardware Requirements

- A Computer Running x8664 architecture or Apple Silicon Mac: A Mac or PC with an Intel x64 architecture or Apple M1 chip (or later models) is required.

## Software Requirements

- macOS, Windows or Linux: For macOS, you'll need a recent version that supports Apple Silicon (ARM architecture) and Rosetta 2 translation for running x8664 applications.

Similarly, for Windows or Linux, any recent distribution that suits your preferences or requirements will work.

- Rosetta 2: This translation layer enables x8664 applications to run on Apple Silicon. It's crucial for running software that is yet to be optimized for ARM architecture.

- Homebrew: This is a package manager for macOS used for installing various software, including the x8664 version of Java. Depending on the software requirements, you might need both the ARM and x8664 versions of Homebrew.

- Chocolatey: This is a Windows equivalent of Homebrew that allows you to install various software, including Java JDK.

- Java Development Kit (JDK): An ARM-compatible version of Java JDK (like OpenJDK for ARM).

- x8664 JDK: For compatibility with specific libraries or applications not yet available for ARM, an x8664 version of Java is also needed. This can be installed using Homebrew under Rosetta 2.

- Docker: You need to have docker and docker-compose installed locally. If you don't have these installed, we recommend following the guidelines in the official Docker documentation for installation and updates.

- Node & NPM: We use Node version v18. It's recommended to manage Node versions with nvm. After installing nvm, run these commands to install and switch to Node version 18:

```
bash
nvm install 18
nvm use 18
```

To use Node without nvm, follow the installation instructions on Node's official website. After installation, verify it by executing `node -v` in your command line, which will display the installed Node version. This step ensures Node is correctly installed on your system.

- Ethers: The interaction with the blockchain is done using Ethers v5.

## Getting Started with RIF Relay

For a detailed step-by-step guide on getting started with RIF Relay, refer to the Sample dApp.

## RIF Relay Contract Deployment Requirements

### Hardhat

- We use Hardhat version v2.10.2 for blockchain interactions. For details on how to install Hardhat, follow the instructions on the Hardhat website. Use the `npx` prefix for Hardhat commands to ensure the use of the project-specific version. Verify the installation with `npx hardhat version`. For configuration, refer to `hardhat.config.ts`. Detailed usage and configuration instructions are available in Hardhat's documentation.

### Using Docker

- RIF Relay components can be deployed using Docker or locally using Hardhat. A guide for the RIF Relay Server can be found in the repository.

# integrate.md:

---

sidebarlabel: Integrations  
sidebarposition: 200  
title: RIF Relay Integration  
tags: [rif, envelope, relay, integration guide]  
description: Integrating RIF Relay in a dApp

---

This guide goes over the exposed RIF Relay methods that dApps and wallets can consume to provide relaying as a service, with the purpose of allowing users to pay transaction fees with tokens in a particular system.

## Introduction

There are multiple ways to integrate RIF Relay into a system. These will be detailed down below.

Additionally, it's important to note that not all of the RIF Relay components are needed for a successful integration, as explained in the following section.

## Requirements

### RIF Relay Smart Contracts

These need to be deployed and their addresses known. For steps on how to do this, please refer to the Contract Deployment page of this guide.

### RIF Relay Server

The RIF Relay Server is the off-chain component in charge of receiving transactions and sending them to the on-chain components, chiefly the RIF Relay Hub. The RIF Relay Hub manages information about the RIF Relay Workers and RIF Relay Managers, but also communicates with the rest of the on-chain components in turn: the Smart Wallets, Factory and Verifier contracts.

The RIF Relay Manager owns RIF Relay Worker accounts with funds in native coin. To relay a transaction, a Worker signs it and sends it to the RIF Relay Hub paying for the gas consumed. In the case of a happy flow, transactions will ultimately be relayed through the RIF Relay Hub, using the EIP-712 library.

For more details on this, please refer to the Architecture page.

Users can interact with the RIF Relay Server directly or indirectly. For the latter, a user can communicate with a RIF Relay Server through a RIF Relay Client. A RIF Relay Client knows the addresses of different RIF Relay Servers and it can send on-chain requests to any one of them. The RIF Relay Client then sends the transaction to be sponsored to the RIF Relay Server via HTTP request.

In any case, you'll need to have the server installed and running. To achieve this please refer to the following guides:

1. RIF Relay Installation Requirements
2. RIF Relay Deployment

### RIF Relay Client

The RelayClient class, from the RIF Relay Client library, assists in building a relay request, searching for an available server and sending the request via http protocol.

To create a RelayClient we need to follow these steps:

1. Set the configuration.
2. Set (ethers) provider.
3. Create instance.

```
typescript
import {
  RelayClient,
  setEnvelopingConfig,
  setProvider,
} from '@rsksmart/rif-relay-client';

setEnvelopingConfig({
  chainId: <CHAINID>,
  preferredRelays: <SERVERURLARRAY>,
  relayHubAddress: <RELAYHUBADDRESS>,
  deployVerifierAddress: <DEPLOYVERIFIERADDRESS>,
  relayVerifierAddress: <RELAYVERIFIERADDRESS>,
  smartWalletFactoryAddress: <SMARTWALLETFACTORYADDRESS>
});

setProvider(ethersProvider);

const relayClient = new RelayClient();
```

Where variables are:

CHAINID: Identifies a network to interact with.

SERVERURLARRAY: An array of relay server URL strings that the RelayClient can interact with.

RELAYHUBADDRESS: The relay hub contract address.

DEPLOYVERIFIERADDRESS: The deploy verifier contract address.

RELAYVERIFIERADDRESS: The relay verifier contract address.

SMARTWALLETFACTORYADDRESS: The smart wallet factory contract address.

After setting the configuration and the ethers provider, we can start creating instances from the Relay Client.

## Account Manager

The Account Manager manager is a singleton component from the RIF Relay Client library that helps to sign relay transactions. This component can sign the transactions with an internal account that was previously added or using a wallet provider like metamask. The Account Manager will look first for manually added accounts and, if none is found, will try to use the provider that was previously setup.

The Account Manager accepts Ethers V5 Wallets as internal accounts.

To interact with the Account Manager we need to follow the next steps:

1. Get an instance.
2. Add a new account.

```
typescript
import {
  AccountManager,
```

```

} from '@rsksmart/rif-relay-client';

const accountManager = AccountManager.getInstance();

accountManager.addAccount(<INTERNALACCOUNTOBJECT>);

```

Where variables are:

INTERNALACCOUNTOBJECT: Ethers V5 Wallet object.

## Relay Transaction

To relay transactions we need a smart wallet already deployed, the deployment process and definition of a smart wallet can be found Smart Wallet.

The steps that we must follow are:

1. Deploy the smart wallet.
2. Create the transaction that we would like to relay.
3. Relay the transaction.

typescript

```

const relayTransactionOpts: UserDefinedEnvelopingRequest = {
  request: {
    from: <EOA>,
    data: <DATATOEXECUTE>,
    to: <DESTINATIONADDRESS>,
    tokenContract: <TOKENADDRESS>,
    tokenAmount: <AMOUNTOFTOKENSINWEI>,
  },
  relayData: {
    callForwarder: <SMARTWALLETADDRESS>,
  },
};

const transaction: Transaction = await relayClient.relayTransaction(
  relayTransactionOpts
);

```

Where variables are:

EOA: Externally Owned Account, the owner of the smart wallet.

DATATOEXECUTE: The encoded function that we want to relay.

DESTINATIONADDRESS: The address of the destination contract that we want to execute.

TOKENADDRESS: The token contract address that we want to use to pay for the fee.

AMOUNTOFTOKENSINWEI: The amount that we want to pay for the fee in wei.

SMARTWALLETADDRESS: The smart wallet address that is going to execute the relayed transaction.

## Relay Verifiers

To obtain the verifier addresses we need to execute the command:

```
curl http://<SERVERURL>/verifiers
```

> The command needs to be executed in a different terminal since it needs the server to be running to perform the request.

```
json
{
  "trustedVerifiers": [
    "0x03f23ae1917722d5a27a2ea0bcc98725a2a2a49a",
    "0x73ec81da0c72dd112e06c09a6ec03b5544d26f05"
  ]
}
```

## Build Request

To relay transactions, the Relay Server exposes an HTTP post handler to the following path `http://<SERVERURL>/relay`. The Relay Client provides an abstraction to build and send each transaction to the available servers; although the client can simplify the interaction with the server, it's always possible to send HTTP requests to the server without using the Relay Client.

Each transaction that will be sent, needs to have the following structure:

```
json
{
  "relayRequest": "<DEPLOYREQUEST|RELAYREQUEST>",
  "metadata": "<METADATA>"
}
```

Below we will describe each field that is required in the request.

## Relay Request

[illegible]

```
}  
}
```

Where each key from request is:

relayHub: The relay hub address that will be used to validate the caller from the transaction.  
to: The address of the destination contract that we want to execute.  
data: The encoded function that we want to relay.  
from: Externally Owned Account, the owner of the smart wallet.  
value: The native currency value that wants to be transferred from smart wallet during the execution.  
nonce: Smart Wallet nonce to avoid replay attacks.  
tokenAmount: The amount of token that we want to pay for the fee in wei.  
tokenGas: The gas limit for the token payment transaction.  
tokenContract: The token contract address that we want to use to pay for the fee.  
gas: The gas limit for the execution of the relaying transaction.  
validUntilTime: Transaction expiration time in seconds.

Where each key from relayData is:

gasPrice: The gas price that will be used to relay the transaction.  
callVerifier: The relay verifier address to validate the correctness of the transaction.  
callForwarder: The smart wallet address that is going to execute the transaction.  
feesReceiver: The address of the worker or collector contract that is going to receive fees.

Deploy Request

```
json  
{  
  "request": {  
    "relayHub": "0xDA7Ce79725418F4F6E13Bf5F520C89Cec5f6A974",  
    "to": "0x0000000000000000000000000000000000000000",  
    "data": "0x",  
    "from": "0x553f430066EA56BD4fa9190218AF17bAD23dCdb1",  
    "value": "0",  
    "nonce": "0",  
    "tokenAmount": "0",  
    "tokenGas": "0",  
    "tokenContract": "0x1Af2844A588759D0DE58abD568ADD96BB8B3B6D8",  
    "recoverer": "0x0000000000000000000000000000000000000000",  
    "index": "1",  
    "validUntilTime": 1676747036,  
  },  
  "relayData": {  
    "gasPrice": "60000000",  
    "callVerifier": "0x73ec81da0C72DD112e06c09A6ec03B5544d26F05",  
    "callForwarder": "0xE0825f57Dd05Ef62FF731c27222A86E104CC4Cad",  
    "feesReceiver": "0x9C34f2225987b0725A4201F1C6EC1adB35562126"  
  }  
}
```

Where each key from request is:

relayHub: The relay hub address that will be used to validate the caller from the transaction.  
to: The address of the destination contract that we want to execute

(0x00 for the Smart Wallet deployment).  
data: The encoded function that we want to relay (0x for the Smart Wallet deployment).  
from: Externally Owned Account, the owner of the smart wallet.  
value: The native currency value that wants to be transferred from smart wallet during the execution.  
nonce: The SmartWalletFactory keeps track of the nonces used for each smart wallet owner, to avoid replay attacks. It can be retrieved with IWalletFactory.nonce(from)  
tokenAmount: The amount that we want to pay for the fee in wei.  
tokenGas: The gas limit for the token payment transaction.  
tokenContract: The token contract address that we want to use to pay for the fee.  
recoverer: The recoverer address, to recover funds from the smart wallet. This feature is still pending to implement.  
index: The index from the smart wallet that we want to deploy.  
validUntilTime: Transaction expiration time in seconds.

Where each key from relayData is:

gasPrice: The gas price that will be used to relay the transaction.  
callVerifier: The deploy verifier address to validate the correctness of the transaction.  
callForwarder: The smart wallet factory address that is going to perform the deployment.  
feesReceiver: The address from the worker or collector contract that is going to receive fees.

## Metadata

```
json
{
  "relayHubAddress": "0xDA7Ce79725418F4F6E13Bf5F520C89Cec5f6A974",
  "signature": "0xa9f579cf964c03ac194f577b5fca5271ba13e2965c...",
  "relayMaxNonce": 4
}
```

Where each key is:

relayHubAddress: The relay hub that will be used by the server to relay the transaction.  
signature: The relay transaction signed by the owner. After signing the transaction, it cannot be changed, since there is a on-chain validation that is part of the EIP712.  
relayMaxNonce: Relay worker nonce plus an extra gap.

## Custom worker replenish function

Each relayed transaction is signed by a Relay Worker account. The worker accounts are controlled by the Relay Manager. When a relay worker signs and relays a transaction, the cost for that transaction is paid using the funds in that worker's account. If the transaction is not subsidized, then the worker is compensated with tokens.

Worker accounts must always have some minimum balance to pay gas for the transaction. These balances can be managed by implementing a replenishment strategy. The Relay Manager can use the strategy to top off a relay worker's account when the balance gets too low.

We provide a default implementation for a replenish strategy. RIF Relay solution integrators can implement their own replenish strategy.

To implement and use your own replenish strategy:

1. In the folder src from the RIF Relay Server project, open ReplenishFunction.ts with a text editor.

2. On the function `replenishStrategy` write your new replenish strategy.
3. Re build the project `npm run build`
4. Change the config JSON file to set `customReplenish` on `true`.

# overview.md:

```
---
sidebarlabel: Overview
sidebarposition: 100
title: RIF Relay - Overview
tags: [rif, envelope, relay, integrate, integration guide]
description: RIF Relay Overview
---
```

Most blockchains have native cryptocurrency to pay for transaction fees and gas consumption; this simple design has many benefits. First, to bootstrap an economy, the native cryptocurrency model creates an initial demand for it. Second, it simplifies the interaction between users and miners because it forces them to use the same means of payment. Third, it reduces the complexity of the consensus rules. Finally, it provides Denial of Service (DoS) protection to the network as full nodes can pay what the miners expect to include a received transaction. This way nodes can decide to propagate a transaction or not, preventing the free consumption of network bandwidth, and stop spam transactions.

Cryptocurrencies tend to be associated with volatility and to counter measure this fact, Stablecoins were introduced. Stablecoins bridge the worlds of cryptocurrency and everyday fiat currency because their prices are pegged to a reserve asset like the U.S. dollar or gold.

But with the advent of Decentralized Finance (DeFi), several stable coins have become a preferred means of payment and savings for both users and miners, therefore, separate systems to facilitate alternative payment mechanisms. Transactions that enable paying transactions with any coin other than the native currency are named meta-transactions because in some systems the user transaction is embedded in a higher-level (or meta) transaction created by a third party. A more accessible term for these transactions is “envelopes” or, for the whole system, a relay system. A meta-transaction/relay system can serve at least two different use cases: 1) pay the transaction fees with tokens, where one new party receives the tokens (from the user) and pays the fees on behalf of the user, and 2) enable smart contract developers to subsidize the gas used to interact with their contracts.

With this in mind, the main goal of the RIF Relay Project is to provide the Rootstock (RSK) ecosystem with the means to allow blockchain applications and end-users (wallet-apps) to transact without needing RBTC. The system should allow Rootstock (RSK) users to pay transaction fees with methods of payment (i.e., tokens) other than RBTC while maintaining their accounts as transaction senders.

RIF Relay takes its inspiration from the Gas Station Network (GSN) project. GSN is a decentralized system that improves dApp usability without sacrificing security. In a nutshell, GSN abstracts away gas (used to pay transaction fees) to minimize onboarding and UX friction for dApps. With GSN, "gasless clients" can interact with smart contracts paying for gas with tokens instead of native-currency.

# sample-dapp.md:

```
---
sidebarlabel: RIF Relay Sample dApp
sidebarposition: 400
title: How to use the RIF Relay Sample dApp SDK
tags: [rif, envelope, relay, integration guide]
description: RIF Relay Sample dApp SDK Starter kit
```



## Getting Started

This guide helps to quickly get started with setting up your environment to use RIF Relay and also use the sample dApp to test relay services.

### Step 1: Run the Rootstock node

You need to set up and run a Rootstock node, preferably the latest version from RSKj releases. The node can operate locally or via Docker. In either case, a node.conf file is used.

Refer to the Rootstock Node Installation Guide for a detailed guide on this step.

### Step 2: Add network to Metamask

To interact with the Rootstock network, you need to add it to Metamask. Since we're using the node on --regtest mode, follow the Metamask guide on How to add a custom network RPC and add the Rootstock RegTest Network to Metamask with the following data:

text

- Network name: RSK regtest
- New RPC URL: http://127.0.0.1:4444
- Chain ID: 33
- Currency symbol: tRBTC

To learn more about Metamask and how to add it to Rootstock programmatically, see Metamask and How to add Metamask to Rootstock Programmatically.

### Step 3: Set up RIF Relay contracts

To set up RIF relay contract, clone the RIF Relay Contracts Repository: <https://github.com/rsksmart/rif-relay-contracts>, then follow the RIF Relay Deployment guide to deploy an RIF Relay contract, enable revenue sharing, and whitelist the token by allowing it.

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">Check allowed tokens</Accordion.Header>

<Accordion.Body>

bash

npx hardhat allowed-tokens --network regtest

Response:

bash

```
rif-relay-contracts % npx hardhat allowed-tokens --network regtest
deployVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
relayVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
customDeployVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
customRelayVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
nativeHolderDeployVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
nativeHolderRelayVerifier [ '0x6f217dEd6c86A57f1211F464302e6fA544045B4f' ]
```

```

</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="1">
  <Accordion.Header as="h3">Mint token</Accordion.Header>
  <Accordion.Body>
    - To mint new units of the UtilToken into the Metamask wallet address:
    - Go to the Metamask wallet, and copy the wallet address:
    - Execute the command to mint the token, where:
      - --token-address → this is the address for UtilToken
      - --amount → quantity to be minted
      - --receiver → wallet address
    bash
    npx hardhat mint \
    --token-address 0x6f217dEd6c86A57f1211F464302e6fA544045B4f \
    --amount 10000000000000000000 \
    --receiver <wallet-address> \
    --network regtest

    - Import the minted token into the wallet.
    - To see the token in the wallet, click on “import tokens”, and then paste the token address.
  </Accordion.Body>
</Accordion.Item>
</Accordion>

```

#### Step 4: Set up RIF Relay Server

Clone the RIF Relay Server Repository, then refer to Run the RIF Relay Server for a complete guide on setting up the RIF Relay server.

#### RIF Relay Sample dApp

This sample dApp shows you how to send transactions to the Rootstock blockchain using the RIF Relay Sample dApp SDK. You'll need to connect the dApp with MetaMask for signing transactions with the account managing your Smart Wallets.

#### Clone SDK repository and install dependencies

```

bash
clone repository
git clone https://github.com/rsksmart/relaying-services-sdk-dapp
cd relaying-services-sdk-dapp
install dependencies
npm install --force

```

#### - Configure environment variables

Create a new file named .env in the top directory, and add the following lines in it (with the contract addresses generated when we deployed the contracts) in the Set up RIF Relay Contracts section above:

```

bash
REACTAPPCONTRACTSRELAYHUB=0x463F29B11503e198f6EbeC9903b4e5AaEddf6D29
REACTAPPCONTRACTSDEPLOYVERIFIER=0x14f6504A7ca4e574868cf8b49e85187d3Da9FA70
REACTAPPCONTRACTSRELAYVERIFIER=0xA66939ac57893C2E65425a5D66099Bc20C76D4CD

```

REACTAPPCONTRACTSSMARTWALLETFACTORY=0x79bbC6403708C6578B0896bF1d1a91D2BB2A  
Aa1c

REACTAPPCONTRACTSSMARTWALLET=0x987c1f13d417F7E04d852B44badc883E4E9782e1

REACTAPPRIFRELAYCHAINID=33

REACTAPPRIFRELAYGASPRICEFACTORPERCENT=0

REACTAPPRIFRELAYLOOKUPWINDOWBLOCKS=1e5

REACTAPPRIFRELAYPREFERREDRELAYS=http://localhost:8090

<<<<<<< HEAD

REACTAPPBLOCKEXPLORER=https://explorer.testnet.rootstock.io

=====

REACTAPPBLOCKEXPLORER=https://explorer.testnet.rootstock.io/

>>>>>>> main

Run the dApp

bash

run app in regtest environment

ENVVALUE="regtest" npm run start

!Run the dApp

- Connect metamask wallet for signing

!Connect Metamask Wallet

- Create a new smart wallet

!Create a new Smart Wallet

- Mint tokens to the wallet

- For commands to mint token, See step 6 in the Set up RIF Relay contracts section above.

!Mint Tokens

- Transfer to different addresses, using TKN for transfer fees payment, instead of RBTC

!Transfer using TKN

# smart-wallets.md:

---

sidebarlabel: Smart Wallets

sidebarposition: 700

title: RIF Relay Smart Wallets

tags: [rif, envelope, relay, user, guide]

description: RIF Relay Smart Wallet

---

This guide is intended to explain more about the interaction and deployment of the Smart Wallets. We will be using additional testing contracts that were included in the project, like the UtilToken(ERC20). All the utils scripts are executed from the account[0] from the regtest network.

Prerequisites

Follow the deployment process in Deployment Guide.

The definition of the smart wallet can be found in Architecture

## Ways to create smart wallets

There are two ways to create a Smart Wallet:

1. Regular transaction: The Requester (or another account on behalf of the Requester) calls the Proxy Factory asking to get a new Smart Wallet. Therefore the Proxy Factory creates a proxy to the SmartWallet code, delegating the ownership to the Requester.
2. Sponsored: It needs to go through the RIF Relay process, which is described in detail below. The requester asks a third party to pay for the Smart Wallet deployment, and the requester pays in tokens for that (or free if it is subsidized by the third-party, a.k.a, Sponsor).

### Send funds

In the RIF Relay Contracts there is a script that would help us to mint ERC20 tokens.

We need to execute the following script:

```
shell
npx hardhat mint --token-address <0xabc123> --amount <amountinwei> --receiver <0xabc123> --network regtest
```

> The token contract needs to have a mint function.

### Deploy a Smart Wallet

To deploy a smart wallet we need to follow some steps that will be described below:

1. We need to generate the smart wallet address. As we mentioned before, the Smart Wallet is a contract-based account, therefore, we can generate as many smart wallet addresses as we want without spending gas by calling the `getSmartWalletAddress` from the relay client library.

> A Smart Wallet only needs to be deployed when we need to execute a transaction. The deployment process uses gas so, unless it's subsidized, we need to pay for it.

At this point we should have the Relay Client object created.

```
typescript
import type {
  getSmartWalletAddress,
  UserDefinedDeployRequest,
} from '@rsksmart/rif-relay-client';

const smartWalletAddress = await getSmartWalletAddress(<EOA>, <INDEX>);

const relayTransactionOpts: UserDefinedDeployRequest = {
  request: {
    from: <EOA>,
    tokenContract: <TOKENADDRESS>,
    tokenAmount: <AMOUNTOFTOKENSINWEI>,
    index: <INDEX>,
  },
};

const transaction = await relayClient.relayTransaction(
```

```
    relayTransactionOpts
);
```

> Keep in mind that to pay any amount of token fees during the deployment, the smart wallet must receive funds first.

Where variables are:

EOA: Externally Owned Account, the owner of the smart wallet.

INDEX: The index that we would like to use to generate the smart wallet.

TOKENADDRESS: The token contract address that we want to use to pay for the fee.

AMOUNTOFTOKENSINWEI: The amount that we want to pay for the fee in wei.

# versions.md:

---

sidebarlabel: Versions

sidebarposition: 900

title: RIF Relay Versions

tags: [rif, envelope, rif relay, integration guide]

description: "RIF Relay Versions"

---

The first iteration of RIF Relay was based on the great work done by the Gas Station Network team.

## Version 0.1

RIF Relay V0.1 started as a fork of GSN with two goals in mind:

- Be compatible with existing and future smart contracts without requiring such contracts to be adapted to work with RIF Relay.
- Be as cost effective as possible.

## Version 0.2

### Overview

RIF Relay V0.2 is a redesign of GSN. It reduces gas costs and simplifies the interaction between the different contracts that are part of the system. It achieves this by:

- Securely deploying counterfactual Smart Wallet proxies for each user account: this eliminates the need for relying on msgSender() and msgData() functions, making existing and future contracts compatible with RIF Relay without any modification.
- Allowing relayers to receive tokens in a worker address under their control and decide what to do with funds later on.
- Reducing gas costs by optimizing the GSN architecture.

Our main objective is to provide the Rootstock (RSK) ecosystem with the means to enable blockchain applications and end-users (wallet-apps) to pay for transaction fees using tokens (e.g. RIF tokens), and thereby remove the need to acquire RBTC in advance.

It is important to recall that - as a security measure - V0.1 contracts deployed on Mainnet have limits on the staked amounts to operate; these limits were removed in V0.2.

## Details

RelayHub contract no longer receives payments, the payment for the service (in tokens) is now sent directly to the worker relaying the transaction on behalf of the user.

RelayHub contract now handles relay manager staking.

Gas estimation improvements:

Gas overhead removed from RelayHub; there are no more validations against hardcoded values.

The gas and token gas fields from the request can now be left undefined, and in that case, they will automatically be estimated by the RIF Relay Client.

The maximum gas estimation in the RIF Relay Server is more precise now.

A new utility function is available to estimate the maximum gas a relay transaction would consume, based in a linear fit estimation. This can be used in applications that don't want to sign a payload each time they need an approximation of the cost of relaying the transaction.

Paymaster verifications are done off-chain to optimize gas costs, thus the paymasters are now called Verifiers and they are not part of the on-chain relay flow nor do they handle payments at all.

Gas cost optimization.

Security issues.

## Version 1

### Overview

Including a revenue-sharing mechanism to the RIF Relay service doesn't introduce a price penalty to the RIF Relay users. We modified the address used to receive the payment (in tokens) for a successful transaction relay (or deploy).

In V0.2 implementation, the RelayRequest and the DeployRequest include a relayWorker attribute to identify which account paid for the gas. The RIF Relay SmartWallet pays directly to this account the number of tokens negotiated for the Relay (or Deploy) service. In V1 The relayWorker attribute was removed from the RelayRequest and DeployRequest. A new attribute called feesReceiver was implemented and configured in the RelayServer

This change did not alter the current relay flow, keeping its cost as it is today, and also introduced the flexibility to implement any revenue-sharing strategy needed.

The feesReceiver could be the worker or MultiSig contract.

The SmartWallet will pay to the feesReceiver. The feesReceiver will hold the funds of each user payment.

Upon payment from the SmartWallet, the feesReceiver will not perform any distribution logic to avoid increasing the cost of the relay service for the user.

With this approach, no changes in the relay flow are required, and thus the introduction of a revenue-sharing mechanism will not impact the price of the relay service.

The relevant participants will form part of the MultiSig contract.

The MultiSig contract specify how much of the funds collected go to each participant (e.g., the relayServer operator, the wallet provider, and the liquidity provider could be the participants).

At a later time, an off-chain process will trigger the distribution process from the contract. This process can invoke the distribution function once per week, month, or when the funds in the contract surpass a certain threshold.

The participants can modify the sharing parameters (e.g., the percentages used for the distribution among the participants) if they agree on the particular changes.

Multiple Revenue Sharing Strategies can exist, ideally one per group of participants.

# 01-setup.md:

---

sidebarlabel: Getting Started

sidebarposition: 100

title: Getting Started with the RPC API

tags: [faucet, Rootstock, testnet, address, wallet, tools]

description: "Easily create, interact and deploy EVM compatible smart contracts using a robust set of JSON RPC methods available through the RPC API."

---

The RPC API provides a seamless and intuitive web interface for developers to interact with Rootstock nodes via JSON-RPC methods. It aims to address the challenges faced by developers when trying to access critical information like logs, transactions, and balances through RPC, which can significantly impact the timely development of dApps on the Rootstock blockchain.

In this guide, you will learn:

- How to create an account and make your first API call
- View a list of JSON-RPC methods available.

```
<div class="btn-container">  
  <span></span>  
  <a class="green" href="http://rpc.rootstock.io/">Use the RPC API</a>  
</div>
```

Who is it for?

dApp Developers looking to interact with the Rootstock nodes

Features

Easy Setup:

- Create an API key effortlessly to initiate development.
- Make the First API call in minutes.

API Key Authentication:

- Provides secure authentication for decentralized applications (dApps).
- Limits API requests on a daily or monthly basis.

Getting Started

:::info[Note]

The RPC API is available on TESTNET and MAINNET.

:::

Visit the Rootstock RPC API

```
<div align="center">  
    
</div>
```

Get a FREE account

To create an account, click on Sign up

```
<div align="center">
  
</div>
```

Get an API Key

To get an API key:

Log in to the dashboard, and click on New API key:

```
<div align="center">
  
</div>
```

Choose a name to identify your apikey, and the Network (either Testnet or Mainnet). You can also add a description (optional). Click on Create.

```
<div align="center">
  
</div>
```

Make first API Call

Click on the newly created apikey to get the details:

```
<div align="center">
  
</div>
```

You can make your first api call by using one of the provided examples, or simply by adding a url and apikey to your application.

```
<div align="center">
  
</div>
```

Example Request

```
shell
curl --location --request POST 'https://rpc.testnet.rootstock.io/<your-apikey>' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethblockNumber",
  "params": [],
  "id": 0
}'
```

Response:

```
text
{"jsonrpc":"2.0","id":0,"result":"0x4b7eca"}
```



> The daily limit is 25,000 requests per user, and each user can have up to 4 API keys, which allows an easy differentiation for different applications the user wants to test.

## Get Support

Join the Rootstock Discord to get support or give feedback.

## Useful Links

- Supported JSON RPC Methods
- Quick Start Guide with Hardhat
- RBTC Faucet

# 02-methods.md:

```
---
sidebarlabel: RPC API Methods
sidebarposition: 200
title: RPC API Methods
tags: [faucet, Rootstock, rpc api, testnet, address, wallet, tools]
description: "Easily create, interact and deploy EVM compatible smart contracts using a robust set of JSON RPC methods available through the RPC API."
---
```

Find below a list of methods available on the RPC API. See how to setup the RPC API.

### ethaccounts

- Method: ethaccounts
  - Returns a list of addresses owned by the client. Since Rootstock RPC API does not store keys, this will always return empty.

- Params: None

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethaccounts",
  "params": [],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": []
}
```

### ethblockNumber

- Method: ethblockNumber

- Returns the number of the most recent block.
- Params: None

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethblockNumber",
  "params": [],
  "id": 0
}'
```

- Example Response:

```
text
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x4bdcfb"
}
```

## ethcall

- Executes a new message call immediately without creating a transaction on the blockchain.

- Params:

- transaction: object, the transaction call object which contains the following fields:
  - from: String, the address from which the transaction is sent
  - to: String, required, the address to which the transaction is addressed
  - gas: String, the integer of gas provided for the transaction execution
  - gasPrice: String, the integer of the gasPrice used for each paid gas, encoded as a hexadecimal
  - value: String, the integer of value sent with this transaction encoded as hexadecimal
  - data: string, the hash of the method signature and encoded parameters. For more information, see the Contract ABI description in the Solidity documentation
- blockNumber: String, required. The number of the block (in hex) from which the number of transactions is required, OR one of the following block tags:
  - latest: the most recent block the client has available.
  - earliest: the lowest numbered block the client has available.
  - pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from a local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- Example:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethcall",
  "params": [{"from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
    "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    "gas": "0x76c0",
    "gasPrice": "0x9184e72a000",
    "value": "0x9184e72a",
    "data":
```

```
"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"
},
  "latest"
],
  "id":0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x"
```

## ethchainId

- Method: ethchainId
- Returns the number of the network, in hexadecimal value.
- Params: None

- Responses:

- 0x1f -> Rootstock Testnet
- 0x1e -> Rootstock Mainnet

- Example:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc":"2.0",
  "method":"ethchainId",
  "params":[],
  "id":0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x1f"
```

## ethestimateGas

- Method:

- Generates and returns an estimate of how much gas is necessary to allow the transaction to complete. The transaction will not be added to the blockchain.

- Params:

- transaction: object, the transaction call object which contains the following fields:
  - from: String, the address from which the transaction is sent
  - to: String, required, the address to which the transaction is addressed
  - gas: String, the integer of gas provided for the transaction execution
  - gasPrice: String, the integer of gasPrice used for each paid gas encoded as hexadecimal
  - value: String, the integer of value sent with this transaction encoded as hexadecimal
  - data: string, the hash of the method signature and encoded parameters. For more information, see

the Contract ABI description in the Solidity documentation

- blockNumber: String, optional. The number of the block (in hex) from which the number of transactions is required, OR one of the following block tags:
  - latest: the most recent block the client has available.
  - earliest: the lowest numbered block the client has available.
  - pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- Example:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethestimateGas",
  "params": [{"from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
    "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
    "gas": "0x76c0",
    "gasPrice": "0x9184e72a000",
    "value": "0x9184e72a",
    "data":
      "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],
  "latest"
},
{id": 0
}'
```

- Example Response:

shell

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x5cec"
}
```

ethgasPrice

- Method: ethgasPrice
  - Returns the current price per gas in wei (hexadecimal).
- Params: None
- Example:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgasPrice",
  "params": [],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x3e252e0"
}
```

ethgetBalance

- Method: ethgetBalance

- Returns the balance of the account of a given address (hexadecimal).

- Note: ethgetBalance only returns the balance of the native chain currency (RBTC) and does not include any ERC20 token balances for the given address.

- Params:

- Address: String, required - 20 Bytes (type: account)

- Block: String: optional, either the hexadecimal value of a blockNumber, OR a blockHash, OR one of the following block tags:

- Latest: the most recent block the client has available.

- Earliest: the lowest numbered block the client has available.

- Pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- if not specified, it will return the balance at the latest block available.

- Example request by blockNumber:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBalance",
  "params": [
    "0x1fab9a0e24ffc209b01faa5a61ad4366982d0b7f",
    "latest"],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x2971b6b90ba793f"
}
```

- Example request by blockHash:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBalance",
  "params": [
    "0x1fab9a0e24ffc209b01faa5a61ad4366982d0b7f",
    "latest"],
  "id": 0
}'
```

```

    "params": [
      "0x1fab9a0e24ffc209b01faa5a61ad4366982d0b7f",
      "0x98e7878cc686d5ca61ca2339bda064004c82a6bbf7b6d43d7674897f775edc91"
    ],
    "id": 0
  }

```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x2971b6b90ba793f"
}

```

- Example request by blockTag:

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBalance",
  "params": [
    "0x1fab9a0e24ffc209b01faa5a61ad4366982d0b7f",
    "latest"
  ],
  "id": 0
}'

```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x2971b6b90ba793f"
}

```

ethgetBlockByHash

- Method: ethgetBlockByHash

- Returns information about a block by blockHash.

- Params:

- Block: String: required, the hash of a block.
- Option: Boolean, optional.
- false: returns only the hashes of the transactions (default)
- true: returns the full transactions objects

- Returns:

- object: A block object, or null when no block was found. The returned object has the following properties:

- number: The block number of the requested block encoded as a hexadecimal string. null if pending.

- hash: The block hash of the requested block. null if pending.

- parentHash: Hash of the parent block.

- sha3Uncles: SHA3 of the uncles data in the block.







- latest: the most recent block the client has available.
- earliest: the lowest numbered block the client has available.
- pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from a local mempool. Intuitively, you can think of these as blocks that have not been mined yet.
- Option: Boolean, optional.
  - false: returns only the hashes of the transactions (default)
  - true: returns the full transactions object
- Returns:
  - object - A block object, or null when no block was found. The returned object has the following properties:
    - number - The block number of the requested block encoded as a hexadecimal string. null if pending.
    - hash - The block hash of the requested block. null if pending.
    - parentHash - Hash of the parent block.
    - sha3Uncles - SHA3 of the uncles data in the block.
    - logsBloom - The bloom filter for the logs of the block. null if pending.
    - transactionsRoot - The root of the transaction trie of the block.
    - stateRoot - The root of the final state trie of the block.
    - receiptsRoot - The root of the receipts trie of the block.
    - miner - The address of the beneficiary to whom the mining rewards were given.
    - difficulty - Integer of the difficulty for this block encoded as a hexadecimal string.
    - totalDifficulty - Integer of the total difficulty of the chain until this block encoded as a hexadecimal string.
    - extraData - The “extra data” field of this block.
    - size - The size of this block in bytes as an Integer value encoded as hexadecimal.
    - gasLimit - The maximum gas allowed in this block encoded as a hexadecimal string.
    - gasUsed - The total used gas by all transactions in this block encoded as a hexadecimal string.
    - timestamp - The unix timestamp for when the block was collated.
    - transactions - Array of transaction objects - please see `ethgetTransactionByHash` for exact shape.
    - uncles - Array of uncle hashes.
    - minimumGasPrice: minimum gas price a transaction should have in order to be included in that block.
  - bitcoinMergedMiningHeader: It is the Bitcoin block header of the block that was used for merged mining the RSK block.
  - bitcoinMergedMiningCoinbaseTransaction: It is the coinbase transaction of the Bitcoin block that was used for merged mining the RSK block.
  - bitcoinMergedMiningMerkleProof: It is the Merkle proof that links the Bitcoin block's Merkle root with the coinbase transaction.
  - hashForMergedMining: It is a hash that is calculated from various fields in the RSK block header.
  - paidFees: It represents the total amount of fees paid by all transactions included in the block.
  - cumulativeDifficulty: It represents the total difficulty of the chain up to the current block.

#### - Example Request:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBlockByNumber",
  "params": [
    "0xfcea",
    false
  ],
  "id": 0
}
```



```

56c897a8fa29ce357a909b4933c4ea9f1744e21422550bde9e0c51064f160e7ba0b19646ca7d6d",
    "hashForMergedMining":
"0x9b846df8ecbe1e7b98351144b1672c25f54207e3998ef7d8c8492a320000fcea",
    "paidFees": "0x0",
    "cumulativeDifficulty": "0x47e89477"
  }
}

```

## ethgetCode

- Method: Returns the compiled byte code of a smart contract, if any, at a given address.
- Params:
  - Address: String: required, address
  - Block: String, required, either the hexadecimal value of a blockNumber, OR a blockHash, OR one of the following block tags:
    - latest: the most recent block the client has available.
    - earliest: the lowest numbered block the client has available.
    - pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from a local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

### - Example Request:

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetCode",
  "params": [
    "0xebea27d994371cd0cb9896ae4c926bc5221f6317",
    "latest"
  ],
  "id": 0
}'

```

### - Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x608060405260043610610...."
}

```

## ethgetLogs

- Method: ethgetLogs
  - Returns an array of all the logs matching the given filter object.
- Params:
  - blockHash: String, optional. Using blockHash is:
    - is equivalent to fromBlock = toBlock = the block number with hash blockHash
    - if blockHash is present in the filter criteria, then neither fromBlock nor toBlock are allowed.
  - address: String, optional. Contract address from which logs should originate.
  - fromBlock: String, optional.
    - either the hexadecimal value of a blockNumber, OR one of the following block tags:
      - latest: the most recent block the client has available.

- earliest: the lowest numbered block the client has available.
- pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.
- toBlock: String, optional.
  - either the hexadecimal value of a blockNumber, OR one of the following block tags:
  - latest: the most recent block the client has available.
  - earliest: the lowest numbered block the client has available.
  - pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.
- topics: Array of 32 bytes DATA topics, optional. The required topic to filter.
- Returns:
  - log objects: An array of log objects, or an empty array if nothing has changed since last poll. Log objects contain the following keys and their values:
    - logIndex: Hexadecimal of the log index position in the block. Null when it is a pending log.
    - transactionIndex: Hexadecimal of the transactions index position from which the log created. Null when it is a pending log.
    - transactionHash: 32 bytes. Hash of the transactions from which this log was created. Null when it is a pending log.
    - blockHash: 32 bytes. Hash of the block where this log was in. Null when it is a pending log.
    - blockNumber: Block number where this log was in. Null when it is a pending log.
    - address: 20 bytes. Address from which this log originated.
    - data: Contains one or more 32-bytes non-indexed arguments of the log.
    - topics: An array of 0 to 4 indexed log arguments, each 32 bytes. In solidity the first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except when you declared the event with the anonymous specifier.
  - Constraints:
    - You can make ethgetLogs requests on any block range with a cap of:
      - 10K logs in the response
      - OR a 2K block range with no cap on logs in the response
      - Note that it can be filtered either by blockHash OR (fromBlock and toBlock), but not both.
      - If fromBlock, toBlock, or blockHash are not specified, the query will return the logs corresponding to the latest block
  - Example request by blockHash:
 

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetLogs",
  "params": [
    {"blockHash":
"0xcca8612942582f1a890231a25245174d6947b7e2e990adf74e84c035c52b104f"}],
    "id": 0
  }'
```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": [
```





transactions usually taken from a local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- Example request:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetStorageAt",
  "params": [
    "0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x0"
    "latest"],
  "id": 0
}'
```

- Example Response:

shell

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

ethgetTransactionByHash

- Method: ethgetTransactionByHash

- Returns the information about a transaction requested by transaction hash. In the response object, blockHash, blockNumber, and transactionIndex are null when the transaction is pending.

- Params:

- transactionHash: String, required - A string representing the hash (32 bytes) of a transaction.

- Returns:

- A transaction object, or null when no transaction was found. The transaction object will consist of the following keys and their values: -

- blockHash: 32 bytes. A hash of the block including this transaction. null when it's pending.

- blockNumber: The number of the block including this transaction. null when it's pending.

- from: 20 bytes. The address of the sender.

- to: 20 bytes. The address of the receiver. null when it's a contract creation transaction.

- gas: Gas provided by the sender.

- gasPrice: Gas price provided by the sender in Wei.

- hash: 32 bytes. The hash of the transaction.

- input: The data sent along with the transaction.

- nonce: The number of transactions made by the sender prior to this one.

- v: The ECDSA recovery ID.

- r: 32 bytes. The ECDSA signature r.

- s: 32 bytes. The ECDSA signature s.

- transactionIndex: The transaction's index position in the block, in hexadecimal. null when it's pending.

- type: The transaction type.

- value: The value transferred in Wei.

- Example Request:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
```

```
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc":"2.0",
  "method":"ethgetTransactionByHash",
  "params":["0x359f6010957a25b885387e3201c9262c71f91e47ff487c49e5168a54fc8ea110"],
  "id":0
}'
```

#### - Example Response:

shell

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": [
    {
      "hash": "0x359f6010957a25b885387e3201c9262c71f91e47ff487c49e5168a54fc8ea110",
      "nonce": "0x10",
      "blockHash": "0xf0b093db64e06ff6b94cd3cfc06d85d3664d7b021bef36c4471475b4f1d8b2b9",
      "blockNumber": "0x35aa",
      "transactionIndex": "0x0",
      "from": "0x3843d583b0f087ec7e3476c3495e52dbde5280b3",
      "to": "0x052ef40ccda2d51ca3d49cc3d6007b25965bec5b",
      "gas": "0x20cfb",
      "gasPrice": "0x387ee40",
      "value": "0x0",
      "input": "0xcc6ebc8b000000000000000000000000",
      "v": "0x62",
      "r": "0x1f8bb5859d8194eebf781ed6d12de95d44b66ecf",
      "s": "0x4a98b84d16a534681c5a639318b1ceffe967ce751458f51",
      "type": "0x0"
    }
  ]
}
```

#### ethgetTransactionCount

##### - Method: ethgetTransactionCount

- Returns the number of transactions sent from an address.

##### - Params:

- Address: String, required - 20 Bytes

- Block: String: optional, either the hexadecimal value of a blockNumber, OR a blockHash, OR one of the following block tags:

- latest: the most recent block the client has available.

- earliest: the lowest numbered block the client has available.

- pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- if not specified, it will return the balance at the latest block available.

##### - Returns:

- transaction count: A hexadecimal equivalent of the integer representing the number of transactions sent from the given address.

##### - Example Request:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
```



```
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc":"2.0",
  "method":"ethgetTransactionCount",
  "params":["0x4495768e683423a4299d6a7f02a0689a6ff5a0a4", "latest"],
  "id":0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x9856"
}
```

### ethgetTransactionReceipt

- Method: ethgetTransactionReceipt

- Returns the receipt of a transaction given transaction hash. Note that the receipt is not available for pending transactions.

- Params:

- transactionHash: String, required. A string representing the hash (32 bytes) of a transaction.

- Returns:

- A transaction receipt object, or null when no receipt was found. The transaction receipt object will contain the following keys and their values:

- blockHash: 32 bytes. Hash of the block including this transaction.

- blockNumber: Block number including this transaction.

- contractAddress: 20 bytes. The contract address created if the transaction was a contract creation, otherwise null.

- cumulativeGasUsed: The total amount of gas used when this transaction was executed in the block.

- effectiveGasPrice: The actual value per gas deducted from the sender's account. Before EIP-1559, equal to the gas price.

- from: 20 bytes. The address of the sender.

- gasUsed: The amount of gas used by this specific transaction alone.

- logs: (Array) An array of log objects generated by this transaction.

- logsBloom: 256 bytes. Bloom filter for light clients to quickly retrieve related logs.

- One of the following:

- root: 32 bytes of post-transaction stateroot (pre-Byzantium)

- status: Either 1 (success) or 0 (failure)

- to: 20 bytes. The address of the receiver. null when the transaction is a contract creation transaction.

- transactionHash: 32 bytes. The hash of the transaction.

- transactionIndex: Hexadecimal of the transaction's index position in the block.

- type: the transaction type.

- Example Request:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc":"2.0",
  "method":"ethgetTransactionReceipt",
  "params":[]
}'
```

```
],
  "id":0
}
```

# shell

## ethgetBlockTransactionCountByHash

- Returns the number of transactions for the block matching the given block hash (in hex).

- **blockHash**: String, required. The hash of the block from which the number of transactions is required.

## shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBlockTransactionCountByHash",
  "params": [
    "0xf0b093db64e06ff6b94cd3cfc06d85d3664d7b021bef36c4471475b4f1d8b2b9"],
  "id": 0
}
```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x2"
}

```

#### ethgetBlockTransactionCountByNumber

- Method: ethgetBlockTransactionCountByNumber

- Returns the number of transactions for the block matching the given block number (in hex).

- Params:

- blockNumber: String, required. The number of the block (in hex) from which the number of transactions is required, OR one of the following block tags:

- latest: the most recent block the client has available.

- earliest: the lowest numbered block the client has available.

- pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- Example

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetBlockTransactionCountByNumber",
  "params": ["0xfcea"],
  "id": 0
}'

```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x1"
}

```

#### ethgetTransactionByBlockHashAndIndex

- Method: ethgetTransactionByBlockHashAndIndex

- Returns information about a transaction for a specific block and transaction index position.

- Params:

- blockHash: String, required. The hash of the block in which the transaction is recorded.

- index: String, required. The position number of the transaction (in Hex).

- Example:

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetTransactionByBlockHashAndIndex",
  "params": [

```



```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethgetUncleCountByBlockHash",
```

```

    "params":[
      "0xf0b093db64e06ff6b94cd3cfc06d85d3664d7b021bef36c4471475b4f1d8b2b9"
    ],
    "id":0
  }
}

```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x3"
}

```

ethgetUncleCountByBlockNumber

- Method: ethgetUncleCountByBlockNumber

- Returns the number of uncles for the block matching the given block number (in hex).

- Params:

- blockNumber: String, required. The number of the block (in hex) from which the number of transactions is required, OR one of the following block tags:

- latest: the most recent block the client has available.

- earliest: the lowest numbered block the client has available.

- pending: A sample next block built by the client on top of latest and containing the set of transactions usually taken from local mempool. Intuitively, you can think of these as blocks that have not been mined yet.

- Example:

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc":"2.0",
  "method":"ethgetUncleCountByBlockNumber",
  "params":[
    "0x35aa"
  ],
  "id":0
}'

```

- Example Response:

```

shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x3"
}

```

ethprotocolVersion

- Method: ethprotocolVersion

- Returns the current protocol version.

- Params: None

- Example:

```

shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \

```

```
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethprotocolVersion",
  "params": [],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x3e"
}
```

ethsendRawTransaction

- Method: ethsendRawTransaction

- Creates a new message call transaction or a contract creation for signed transactions.

- Response: The transaction hash, or the zero hash if the transaction is not yet available.

- Params:

- transactionData: Required, the signed transaction data (typically signed with a library, using your private key). Use ethgetTransactionReceipt to get the contract address, after the transaction was mined, when you created a contract.

- Example:

```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "ethestimateGas",
  "params": [
    "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"
```

```
],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x359f6010957a25b885387e3201c9262c71f91e47ff487c49e5168a54fc8ea110"
}
```

netversion

- Method: netversion

- Returns the number of the network, in decimal value.

- Params: None

- Responses:

- 31 -> Rootstock Testnet
- 30 -> Rootstock Mainnet

- Example:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "netversion",
  "params": [],
  "id": 0
}'
```

- Example Response:

shell

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "31"
}
```

web3clientVersion

- Method: web3clientVersion

- Returns the current client version.

- Params: None

- Example:

shell

```
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "web3clientVersion",
  "params": [],
  "id": 0
}'
```

- Example Response:

shell

```
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "RskJ/6.2.0/Linux/Java1.8/ARROWHEAD-45eb751"
}
```

web3sha3

- Method: web3sha3

- Returns Keccak-256 (not the standardized SHA3-256) hash of the given data.

- Params:

- data: Required, string: The data in hexadecimal form to convert into a SHA3 hash

- Example:



```
shell
curl --location 'https://rpc.testnet.rootstock.io/<api-key>' \
--request POST \
--header 'accept: application/json' \
--header 'Content-Type: application/json' \
--data '{
  "jsonrpc": "2.0",
  "method": "web3sha3",
  "params": ["0x68656c6c6f20776f726c64"],
  "id": 0
}'
```

- Example Response:

```
shell
{
  "jsonrpc": "2.0",
  "id": 0,
  "result": "0x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad"
}
```

# rif-wallet-lib.md:

```
---
title: RIF Wallet Libraries
tags: [rif-wallet, rif, rootstock, wallet, libraries]
sidebarlabel: RIF Wallet Libraries
sidebarposition: 200
description: "RIF Wallet is an open source smart contract wallet which enables businesses to create and
deploy fully customizable on-chain wallets."
---
```

RIF Wallet Libraries contain a set of packages that is used by the RIF Wallet. You can install the RIF Wallet Libraries directly from inside the app. For more information, visit the RIF Wallet Lib Repo.

## Packages

### RIF Wallet Core

The RIF Wallet Core is the wallet library that connects the UI with the RIF Relay SDK. This class accepts an Ethers Signer that handles the majority of crypto methods, such as creating a Wallet, sign tx or message, estimate gas, send transactions and deploy the smart wallets.

The onRequest function is where the UX handles the transaction or interaction. A transaction is sent to the RIFWallet then passed to the onRequest method. At this point, the UX can prompt the user to click 'accept' or 'deny'. This means that the wallet can be injected into WalletConnect, and injectedBrowser or used via the UX and when a transaction comes in, it will always prompt the user to accept or deny the action.

See the README for more information.

### RIF Wallet ABI Enhancer

The ABI Enhancer package attempts to decode a transaction into a human readable format. There are different strategies for decoding:

rBTC Transaction - where the data is 0x and the transaction is sending gas from one account to another.  
ERC20 (and variants) Transaction - sending a token from one user to another. In this case the recipient and amount is located in the data field.

Other Transaction - A contract call interaction. In this case, it queries the publicly available list of known method types and attempts to decode it. In this case, the transaction details are not transformed.

See more information in the README.

## Key Management System

The RIF Wallet Key Management System library.

## Bitcoin Library

The RIF Wallet Bitcoin Library is a library to handle receiving and sending bitcoin in React Native.  
See Basic Setup and How to Use in the RIF Wallet Bitcoin README.

## RIF Relay Client SDK

This rif-relay-light-sdk is a light implementation of the client RIF Relay SDK built using ethers and used in the RIF Wallet.

See Basic setup, how to deploy the smart wallet, and how to estimate and relay a transaction in the README.

## RIF Wallet Token

The RIF Wallet Token package contains simple classes for ERC20, ERC677, and rBTC assets/tokens. It includes the ABI for ERC20 and ERC677.

## RIF Wallet EIP681

The RIF Wallet EIP681 is a basic and incomplete implementation of EIP681, URL Format for Transaction Requests.

See the README for more information.

## RIF Wallet Services

This RIF Wallet Services library is responsible for mapping all the endpoints available and making the socket connection in rif-wallet-services (backend).

# index.md:

---

title: Pre-compiled ABIs

tags: [libraries, abi, pre-compiled]

sidebarlabel: Pre-compiled ABIs

sidebarposition: 100

description: "Rootstock Pre-compiled ABIs."

---

Here you will find the ABIs for the existing precompiled contracts in Rootstock. You will also get their addresses and a builder to use it with web3.js.

## Versions

Different versions of the package mentioned are required for different Rootstock releases.

The semantic versioning of this package doesn't correlate to the semantic versioning of Rootstock. For each named release of the RSKj node, there will be a corresponding name version in npm.

This package's support starts with ORCHID.

## Usage

For the installation of these package you must execute in a terminal window:

```
shell
npm install @rsksmart/rsk-precompiled-abis@<version>
```

As an example to define and use it:

1. Include the Web3 package.

```
javascript
const Web3 = require('web3');
```

2. Include the rsk-precompiled-abis package.

```
javascript
const precompiled = require('@rsksmart/rsk-precompiled-abis');
```

3. Create an instance of the contract using package build method and Web3 as a parameter.

(i.e.: using the Bridge)

```
shell
var bridge = precompiled.bridge.build(new Web3('http://localhost:4444'));
```

4. Use a contract's method. For example, here we call `getFederationAddress`, and displays its result in the console.

```
shell
bridge.methods.getFederationAddress().call().then(console.log);
```

:::note[Important]

If the version to be installed is not defined in the command line, the version will correspond to the latest version in rskj releases page.

:::

## Versioning table

| Package Version | Rootstock version |
|-----------------|-------------------|
| -----           | -----             |

|                  |                  |  |
|------------------|------------------|--|
| 1.0.0-ORCHID     | ORCHID-0.6.2     |  |
| 2.0.0-WASABI     | WASABI-1.0.0     |  |
| 2.0.1-WASABI     | WASABI-1.0.0     |  |
| 3.0.0-PAPYRUS    | PAPYRUS-2.0.0    |  |
| 3.0.0-PAPYRUS    | PAPYRUS-2.2.0    |  |
| 3.0.0-IRIS       | IRIS-3.0.0       |  |
| 3.1.0-IRIS       | IRIS-3.1.0       |  |
| 3.2.0-IRIS       | IRIS-3.2.0       |  |
| 3.3.0-IRIS       | IRIS-3.3.0       |  |
| 4.0.0-HOP        | HOP-4.0.0        |  |
| 4.1.0-HOP        | HOP-4.1.0        |  |
| 4.1.1-HOP        | HOP-4.1.1        |  |
| 4.2.0-HOP        | HOP-4.2.0        |  |
| 4.3.0-HOP        | HOP-4.3.0        |  |
| 4.4.0-HOP        | HOP-4.4.0        |  |
| 5.0.0-fingerroot | FINGERROOT-5.0.0 |  |
| 5.1.0-fingerroot | FINGERROOT-5.1.0 |  |
| 5.2.0-fingerroot | FINGERROOT-5.2.0 |  |
| 5.3.0-fingerroot | FINGERROOT-5.3.0 |  |
| 5.4.0-fingerroot | FINGERROOT-5.4.0 |  |
| 6.0.0-ARROWHEAD  | ARROWHEAD-6.0.0  |  |

# index.md:

```

---
sidebarposition: 1
title: Node Operators Overview
sidebarlabel: Overview
tags: [rsk, rootstock, rskj, node, developers, merged mining]
description: "Learn about RSKj, a node implementation for the Rootstock blockchain and merged mining on Rootstock."
---
```

This section guides node operators performing merged mining operations or developers looking to setup and run the Rootstock node (RSKj).

## Navigating Node Operators

| Resource        | Description  |
|-----------------|--|
|                 |  |
| -----           |  |
| Node Setup      | Set up requirements, installation, configuration and run the Rootstock nodes.            |
| JSON RPC        | The JSON-RPC methods supported by Rootstock nodes.                                       |
| Merged Mining   | How to merge mine Rootstock using Bitcoin mining pool software.                          |
| Public Nodes    | Rootstock Nodes: Public nodes (Mainnet, Testnet).  |
| Maintenance     | How to upgrade and update the Rootstock nodes.   |
| Troubleshooting | Learn how to solve some known or frequently encountered issues when setting up the node. |

# index.md:

```

---
sidebarlabel: Implementation Guide
```

title: Implementation Guide

tags: [rsk, mining, bitcoin, pool]

description: "How to merge mine Rootstock using Bitcoin mining pool software."

---

Here are the steps needed to add Rootstock (RSK) merged mining capabilities to mining pool software.

What do you need to do

Add the Rootstock merged mining information in Bitcoin block as a commitment, the complete steps are as follows:

### 1. Get the work from Rootstock node

Use the `mnrgetWork` method from the Rootstock node's JSON-RPC API. This method returns the information of the current block for merged mining, the boundary condition to be met ("target"), and some other data.

### 2. Put the information for merged mining in the Bitcoin block

Format

OPRETURN + Length + RSKBLOCK: + RskBlockInfo

OPRETURN: `is0x6a`

Length is `0x29` and represents the length of the information included after the OPRETURN opcode

RSKBLOCK: is the ASCII string for `0x52534b424c4f434b3a`

RskBlockInfo is the block information in binary format.

For example, if RskBlockInfo is

`e5aad3b6b9dc71a3eb98a069bd29ca32211aee8b03fd462f4ffbbe97cc75a174`

the merged mining information is

`6a2952534b424c4f434b3ae5aad3b6b9dc71a3eb98a069bd29ca32211aee8b03fd462f4ffbbe97cc75a174`

Position

Include as the last output of Bitcoin coinbase transaction.

Restrictions

- The number of bytes immediately after RskBlockInfo, up to the end of the coinbase transaction must be lower than or equal to 128 bytes.
- The trailing raw bytes must not contain the binary string RSKBLOCK:
- The probability of the RSK tag to appear by chance is negligible, but pool servers must not rule out the possibility of a rogue Bitcoin address included in the coinbase transaction having this pattern, and being used as an attack to break the validity of merged mining header.
- The RSKBLOCK: tag may appear by chance or maliciously in the ExtraNonce2 data field that is provided by miners as part of the Stratum protocol. This is not a problem as long as the poolserver adds the RSKBLOCK: tag after the ExtraNonce2 chunk.

### 3. Notify Miners on a faster pace

Rootstock's average block time is 30 seconds, which is faster than Bitcoin's 10 minutes. This fact triggers the following implementation changes:

Retrieve work from Rootstock node every 2 seconds, so as to be always mining on the last Rootstock work.

Sent to miners a mining.notify message, from stratum protocol, every time new Rootstock work is received.

4. Mine until work is enough to meet the target received in the work info

5. Submit Solution to Rootstock node

Use the mnrsSubmitBitcoinBlockPartialMerkle method from Rootstock node's JSON-RPC API. That method has optimum performance, and is preferred among other available methods.

Other submission methods and information about the pros and cons between them can be found in the Mining JSON-RPC API documentation.

## Influence on Bitcoin

As a result of Rootstock's implementation of merged mining, the Bitcoin network does not get filled up with merged mining information. Only a minimal amount of information is stored: An extra output on the coinbase transaction.

Furthermore, no changes are required on Bitcoin node to support merged mining with Rootstock.

# index.md:

---

title: Configure

sidebarlabel: Configure Node for Mining

tags: [rsk, rootstock, rskj, node, node operators, config]

description: "Setting your own config preferences, when using the Java command, Ubuntu, Azure, AWS, or Docker"

---

## Command Line Interface

The Rootstock node can be started with different CLI flags.

### Setting config preferences

See how to set your config:

- Using Ubuntu or Docker
- Using the java command

&hellip; to run the node.

:::tip[Tip]

You need to restart the node if you've changed any configuration option.

:::

### Using Ubuntu or Docker

Your node's config file is in /etc/rsk.

Default configurations are defined there and they are the same as these ones.

You should edit the config related with the network you are using (mainnet.conf, testnet.conf, regtest.conf).

Check here all the configuration options you could change.

## Using Windows

For other operating systems, including Windows, please use the `-Drsk.conf.file` option as specified below.

## Using java command

### 1. Create a .conf file

You can create a file with the configuration options that you want to replace from the default. Default configurations are defined here.

The extension of the file must be `.conf`.

Check `/node-operators/setup/configuration/reference/` for all the configuration option.

As an example, if you want to change the default database directory, your config file should only contain:

```
conf
database {
  dir = /new/path/for/database
  reset = false
}
```

### 2. Specify your config file path

To apply your configuration options, you need to set your own config file's path when you run your node.

This can be done in two ways:

- Running the node with the java command, add `-Drsk.conf.file=path/to/your/file.conf`
- Compiling the node with IntelliJ, add to VM options: `-Drsk.conf.file=path/to/your/file.conf`

## Using RocksDB

:::note[Important Notice]

- Starting from RSKj HOP v4.2.0, RocksDB is no longer experimental. As of the most recent version, RocksDB has now been made the default storage library, replacing LevelDB. This change was made to tackle maintainability and performance issues of LevelDB.
- Previously, RSKj ran using LevelDB by default, with the option to switch to RocksDB. Now, RocksDB is the default storage option, aiming to enable higher performance within the RSKj nodes.

...

## Get Started

RSKj nodes run using RocksDB by default (See important info section). To switch back to LevelDB, modify the relevant RSKj config file (`.conf`) and set the config: `keyvalue.datasource=leveldb`.

The `keyvalue.datasource` property in the config

may only be either rocksdb or leveldb.

> If you wish to switch between the different storage options, for example from leveldb to rocksdb or vice versa, you must restart the node with the import option.

The following sample command shows how to do this when the RSKj node was previously running the default (leveldb), and wants to run with rocksdb next.

> Note the use of the --import flag, which resets and re-imports the database.

```
java
java -Dkeyvalue.datasource=rocksdb -jar ./rskj-core/build/libs/rskj-core--all.jar --testnet --import
```

### Advantages:

RocksDB uses a log structured database engine, written entirely in C++, for maximum performance. Keys and values are just arbitrarily-sized byte streams.

RocksDB is optimized for fast, low latency storage such as flash drives and high-speed disk drives. RocksDB exploits the full potential of high read/write rates offered by flash or RAM.

RocksDB is adaptable to different workloads. From database storage engines such as MyRocks to application data caching to embedded workloads, RocksDB can be used for a variety of data needs.

RocksDB provides basic operations such as opening and closing a database, reading and writing to more advanced operations such as merging and compaction filters.

### Switching between DB Kinds

Switching between different types of databases in your system requires you to modify configuration files, drop the existing database, and restart your node so the node will start syncing from scratch using the new db kind.

:::info[Info]

Nodes that were already running on LevelDB will continue to use LevelDB, and the same applies to RocksDB. However, all nodes setup from scratch will use RocksDB by default.

:::

### Gas Price Setting

The value returned by ethgasPrice can be modified by setting a multiplier to be used while calculating the aforementioned gas price.

This can be done by setting a numeric value on rpc.gasPriceMultiplier in the configuration file. Default value is 1.1.

### Troubleshooting

#### UDP port already in use

If you see the following error message, it means that RSKj is unable to bind to a particular port number, because prior to this, another process has already bound to the same port number.



text

Exception in thread "UDPServer" co.rsk.net.discovery.PeerDiscoveryException: Discovery can't be started.

at co.rsk.net.discovery.UDPServer\$1.run(UDPServer.java:65)

Caused by: java.net.BindException: Address already in use: bind

To rectify this,

change the value of peer.port in the config file,

or add a peer.port flag to the command when you start RSKj.

<Tabs>

<TabItem value="code" label="Linux, Mac OSX" default>

shell

\$ java -Dpeer.port=50505 -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>

<TabItem value="windows" label="Windows">

shell

C:\> java -Dpeer.port=50505 -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>

</Tabs>

# index.md:

---

sidebarlabel: Reference

title: Merged mining reference

tags: [rsk, mining, bitcoin]

description: "How Rootstock leverages the Bitcoin network's consensus mechanism for its own security, and adds additional features to prevent double spending"

---

Satoshi consensus, based on proof-of-work (PoW), is the only consensus system that prevents the rewrite of blockchain history at a low cost. The academic community is advancing the knowledge and study of proof-of-stake (PoS) as an alternative, but currently PoW provides the highest proven security. Merge mining is a technique that allows Bitcoin miners to mine other cryptocurrencies simultaneously with nearly zero marginal cost. The same mining infrastructure and setup they use to mine Bitcoins is reused to mine Rootstock (RSK) simultaneously. This means that as Rootstock rewards the miners with additional transaction fees, the incentive for merged mining becomes high.

We have identified three phases for Rootstock merge-mining growth:

- Bootstrap phase: Merge-mining is below 30% of Bitcoin hashrate.
- Stable phase: Merge-mining is between 30% and 60% of Bitcoin hashrate.
- Mature phase: Merge-mining is higher than 60% of Bitcoin hashrate.

Rootstock has left behind its bootstrapping phase, when rogue merge-miners could theoretically revert Rootstock blockchain at a low cost. As of December 2021, more than 50% of Bitcoin miners are engaged in Rootstock merge-mining. But as Rootstock fees remain low compared to Bitcoin block reward, the cost to attack Rootstock through double-spending is lower than Bitcoin's.

Rootstock has some properties to reduce the risk of double-spend attacks, such as long miner rewards maturity. Still RootstockLabs research team has developed several protections to prevent attacks during

the stable and mature phases of the project:

**Signed notifications:** Rootstock clients can make use of signed notifications by notaries. Nodes can use these notifications to detect Sybil attacks and inform the user.

**Transparent double-spend trails:** this is a method where all Rootstock merge-mining tags are augmented with additional information that can be used to detect selfish Rootstock forks that are public in the Bitcoin blockchain. Selfish-fork proofs are automatically constructed and these proofs are presented to the Rootstock nodes, which spread them over the network. The proofs force nodes to enter a “safe mode” where no transaction is advertised as confirmed. The safe mode prevents merchants and exchanges from accepting payments that could be double-spent. Once the proven selfish-fork is outpaced by the Rootstock mainchain in accumulated PoW, the network reverts to its normal state. This method is a deterrent for any Rootstock double-spend attempt, where the malicious miner still tries to collect Bitcoin rewards when mining the selfish fork.

Once the platform enters the maturity phase, we estimate the security of Rootstock will be enough to support the economy of worldwide financial inclusion.

Main features:

- REMASC consensus protocol
- One-day maturity for mining reward
- No loss of efficiency in Bitcoin mining expected from merge mining (for late mid-state switching)

# index.md:

---

sidebarlabel: Autominer

title: Run with autominer (Ganache-like)

tags: [rsk, rskj, rootstock, node, config]

description: "Learn how to run the Rootstock node with autominer - similar to Ganache default config"

---

Ganache local network runs like what Rootstock (RSK) calls autominer mode, it:

- Creates blocks when new transactions are sent to the node
- Will not create blocks if no transactions are sent
- Allows to mine blocks manually via RPC
- (optionally) Delete the database on restart

To configure the node, we are going to:

1. Run it in --regtest mode
2. Use a custom config to activate the autominer

The configuration we need to use is:

```
miner.client.autoMine = true
```

Create a autominer.conf file in the root of the repo (or other dir., remember to use the correct path afterwards)

This option can be activated when using the node in different modes

Setup Autominer on IntelliJ

On top of the default configuration (Java version and main class), we will need to add

- Program arguments: --regtest and optionally --reset for database reset on restart
- VM options: -Drsk.conf.file=./autominer.conf (or the path you chose)

It should look like this:

```
!autominer:inellijconfig
```

Setup Autominer on CLI

To setup autominer on CLI, use the command below;

> Use this if you are running with JAR.

```
java
java -cp rskj-core-4.1.0-HOP-all.jar -Drsk.conf.file=./autominer.conf co.rsk.Start --regtest --reset
```

Result

Now you have an Rootstock node running locally! It will create blocks only for new transactions, or arbitrarily by using the evmmine RPC call.

See gif image below for example on how to do this;

```
!autominer:demo
```

# index.md:

```
---
sidebarlabel: Remasc
title: REMASC
tags: [rsk, rootstock, mining, bitcoin, remasc]
---
```

Reward Manager Smart Contract (REMASC) is a pre-compiled smart-contract that is executed on every block and has the responsibility to fairly distribute rewards collected from transaction fees into several participants of the network. However the distribution of rewards of a block is only performed once the block reaches a certain maturity. In other words, the rewards are paid only after a fixed number of blocks have confirmed a block. With the exception of the first blocks in the blockchain after genesis, every time a block is added to the blockchain, another previous block reaches maturity and its rewards are paid.

REMASC is an implementation of DECOR+.

How it Works

The REMASC contract maintains different internal accounts. One of these internal accounts is called Reward Balance. The Reward Balance always exists and its value can change when a new block is processed because of any of the following reasons:

The block was accepted on the mainchain and all its transaction fees are added to the Reward balance. Miners and other rewarded parties get paid their reward and the rewarded value is subtracted from the

Reward balance.

As an example, let's assume that a block has 2 transactions: One paying 100000 gas at 2 smart weis and the other paying 25000 gas at 3 smart weis. Let's also assume that prior processing of the block, the Reward Balance was 1000000 smart weis. After processing the block the Reward Balance will be updated to  $1000000 + 200000 + 75000 = 1275000$  smart weis.

From this Reward Balance, the 10% (127500 in the example) will be subtracted to pay the miners having mined at the corresponding height. This creates a synthetic reward, that is equivalent to applying a low-pass filter to the received fees, and so this method has also been called fee smoothing. The 10% amount extracted from the Reward Balance, is called the Full Block Reward and will be referred to as F from now on.

The amount of fees in F will be affected by the following variables:

The number of siblings mined at the same processing height  
The fact that the Selection Rule was respected or broken

Some additional definitions will be introduced before we formalize how the payment is calculated for each miner.

One and only one block is mined at a height N. This block is the main block at height N. Blocks that share a parent with a main block are called siblings. These blocks can be added to the blockchain by publishers, which are always miners mining following blocks.

The payment for the miners of the main block, the siblings and the publishers will occur on the block N + 4000. The payment occurs as specified by the following rules:

$FullBlock\{rwd\}$  is the 100% of the block reward

Rootstock will receive a fee of 20% of the full block reward:

$Rsk\{rwd\} = \frac{FullBlock\{rwd\}}{5}$

Rootstock Federation will receive a fee of 0.8% of the full block reward:

$Fed\{rwd\} = \frac{FullBlock\{rwd\} - Rsk\{rwd\}}{100}$

Miners will receive a payment of 79.2% of the full block reward:

$Miners\{rwd\} = FullBlock\{rwd\} - Rsk\{rwd\} - Fed\{rwd\}$

<br/>

It's important to notice that these are integer divisions where results are rounded down. That's why:

$\frac{4}{5} FullBlock\{rwd\} \neq FullBlock\{rwd\} - \frac{FullBlock\{rwd\}}{5}$

<br/>

Now we present several different scenarios:

1. There are no siblings at height N  
No Rule was broken  
The miner of the block at height N is paid

$$\text{Miners}(\text{rwd})$$

Rule was broken

The miner is paid 90% of the

$$\text{Miners}(\text{rwd})$$

which is defined as

$$\text{Miners}(\text{rwdBroken}) = \text{Miners}(\text{rwd}) - \frac{\text{Miners}(\text{rwd})}{10} \% 22$$

2. There are siblings at height N.

Each sibling will have a respective publisher and miner, so we define:

Publisher Fee (10% of  $\text{Miners}(\text{rwd})$ )

$$\text{PubFee}(\text{rwd}) = \frac{\text{Miners}(\text{rwd})}{10}$$

Miner Fee (90% of  $\text{Miners}(\text{rwd})$ )

$$\text{MinersFee}(\text{rwd}) = \text{Miners}(\text{rwd}) - \text{PubFee}(\text{rwd})$$

If we S is the number of siblings, we define:

Individual Publisher Fee

$$\text{IndPubFee}(\text{rwd}) = \frac{\text{PubFee}(\text{rwd})}{S}$$

Individual Mining Fee

To simplify we define

$$\text{Mining}(\text{rwd}) = \frac{\text{MiningFees}(\text{rwd})}{S+1}$$
,

is given by the Mining Fee over all mined blocks referenced on the blockchain (which is siblings + the main block), then individual mining fee is:

No Rule was broken

$$\text{IndMiningFee}(\text{rwd}) = \text{Mining}(\text{rwd})$$

Rule was broken

$$\text{IndMiningFee}(\text{rwdBroken}) = \text{Mining}(\text{rwd}) - \frac{\text{Mining}(\text{rwd})}{10} - L$$

Finally, with all the previous variables computed, the payments will be performed as follows:

Each publisher receives

$$\text{PubFee}(\text{rwd})$$

The miner of the main block receives

$$\text{IndMiningFee}(\text{rwd})$$

Also, for each sibling, a new amount needs to be calculated. This is, for each late block that the sibling published, it receives a punishment of 5% of the

$$\text{IndMiningFee}(\text{rwd})$$
.

The sibling is added on the block N+D for some positive value of D. A punishment for late publication is calculated for each as

$$L = \frac{(D-1) \text{IndMiningFee}(\text{Rwd})}{20}$$

Then the respective miners are paid

$$\text{IndMiningFeeLate}\{\text{rwd}\} = \text{IndMiningFee}\{\text{Rwd}\} - L$$

The remaining amount of  $\text{Miners}\{\text{rwd}\}$  is added to a balance called Burned Balance. As of this writing, burned money is lost but changes may apply. The Burned Balance is given by rounding errors or punishments.

Example

Suppose the Reward Balance is 90000 smart weis and the payment for this N is 10000 smart weis. Then the reward balance is updated to 100000 smart weis. From this, the 10% will be distributed, which is 10000 smart weis.



A, B and C share the parent P. B is the main block at height N and A and C are siblings. D is publisher of C and E is publisher of A.

This way, we compute:

Rootstock receives

$$\text{Rsk}\{\text{rwd}\} = \frac{\text{FullBlock}\{\text{rwd}\}}{5} \implies \frac{10000}{5} \implies \text{Rsk}\{\text{rwd}\} = 2000$$

RSK Federation receives

$$\text{Fed}\{\text{rwd}\} = \frac{\text{FullBlock}\{\text{rwd}\} - \text{Rsk}\{\text{rwd}\}}{100} \implies \frac{10000 - 2000}{100} \implies \text{Fed}\{\text{rwd}\} = 80$$

Miners receive a total of

$$\text{MinerFee}\{\text{rwd}\} = \text{Miner}\{\text{rwd}\} - \text{PubFee}\{\text{rwd}\} \implies 7920 - 792 \implies \text{MinerFee}\{\text{rwd}\} = 7128$$

B and C blocks receive Individual Mining Fee

$$\text{IndMiningFee}\{\text{rwd}\} = \frac{\text{MinerFee}\{\text{rwd}\}}{S+1} \implies \frac{7128}{3} \implies \text{IndMiningFee}\{\text{rwd}\} = 2376$$

In this case blocks are not published late so L is 0, that is why

$$\text{IndMiningFee}\{\text{rwd}\}$$

is used in the calculation instead of

$$\text{IndMiningFeeLate}\{\text{rwd}\}$$

A receives

$$\text{IndMiningFeeLate}\{\text{rwd}\} = \text{IndMiningFee}\{\text{rwd}\} - L$$

$$\text{IndMiningFeeLate}\{\text{rwd}\} = \text{IndMiningFee}\{\text{rwd}\} - \frac{(D-1)}{20}$$

$$\text{IndMiningFeeLate}\{\text{rwd}\} = 2376 = \text{IndMiningFee}\{\text{rwd}\} - \frac{(2-1)}{20} \cdot 2376$$

$$\text{IndMiningFeeLate}\{\text{rwd}\} = 2257$$

In this case A was published late so L is not 0, that is why

$$\text{IndMiningFeeLate}\{\text{rwd}\}$$

is used in the calculation instead of  
`[(#top "tex-render IndMiningFee{rwd})]`

For this example, an assumption that there wasn't a broken rule for any block was made. Otherwise, fees paid should have been calculated using `[(#top "tex-render IndMiningFeeLate{rwdBroken})]`.

References

- DECOR+
- RSKIP-15

# 01-methods.md:

```
---
title: Supported JSON-RPC Methods
sidebarlabel: RPC Methods
sidebarposition: 100
tags: [rsk, rskj, node, rpc, rpc api, node operators, rootstock]
description: "The JSON-RPC methods supported by Rootstock nodes."
renderfeatures: 'tables-with-borders'
---
```

Here are the supported JSON-RPC Methods.

> For a full description, see the JSON RPC Method details.

| Module | Method                              | Supported | Comments                                     |
|--------|-------------------------------------|-----------|--|
| -----  | -----                               | -----     | -----  |
| web3   | web3clientVersion                   | YES       |  |
| web3   | web3sha3                            | YES       |  |
| eth    | netversion                          | YES       | Mainnet Chain Id = 30, Testnet Chain Id = 31 |
| eth    | netpeerCount                        | YES       |  |
| eth    | netpeerList                         | YES       |  |
| eth    | netlistening                        | YES       |  |
| eth    | ethchainId                          | YES       | Same response as ethprotocolVersion          |
| eth    | ethprotocolVersion                  | YES       |  |
| eth    | ethsyncing                          | YES       |  |
| eth    | ethcoinbase                         | YES       |  |
| eth    | ethmining                           | YES       |  |
| eth    | ethhashrate                         | YES       |  |
| eth    | ethgasPrice                         | YES       |  |
| eth    | ethaccounts                         | YES       |  |
| eth    | ethblockNumber                      | YES       |  |
| eth    | ethgetBalance                       | YES       |  |
| eth    | ethgetStorageAt                     | YES       |  |
| eth    | ethgetTransactionCount              | YES       |  |
| eth    | ethgetBlockTransactionCountByHash   | YES       |  |
| eth    | ethgetBlockTransactionCountByNumber | YES       |  |
| eth    | ethgetUncleCountByBlockHash         | YES       |  |
| eth    | ethgetUncleCountByBlockNumber       | PARTIALLY | Option "pending" not yet supported.          |
| eth    | ethgetCode                          | PARTIALLY | Option "pending" not yet supported.          |
| eth    | ethsign                             | YES       |  |
| eth    | ethsendTransaction                  | YES       |  |
| eth    | ethsendRawTransaction               | YES       |  |

| eth | ethcall | YES ||  
| eth | ethestimateGas | YES ||  
| eth | ethgetBlockByHash | YES ||  
| eth | ethgetBlockByNumber | PARTIALLY | Option "pending" not yet supported. |  
| eth | ethgetTransactionByHash | YES ||  
| eth | ethgetTransactionByBlockHashAndIndex | YES ||  
| eth | ethgetTransactionByBlockNumberAndIndex | PARTIALLY | Option "pending" not yet supported. |  
| eth | ethgetTransactionReceipt | YES ||  
| eth | ethpendingTransactions | YES ||  
| eth | ethgetUncleByBlockHashAndIndex | YES ||  
| eth | ethgetUncleByBlockNumberAndIndex | PARTIALLY | Option "pending" not yet supported. |  
| eth | ethgetCompilers | - | For security reasons, we've decided not to include compilers in node. |  
| eth | ethcompileLLL | - | For security reasons, we've decided not to include compilers in node. |  
| eth | ethcompileSolidity | - | For security reasons, we've decided not to include compilers in node. |  
| eth | ethcompileSerpent | - | For security reasons, we've decided not to include compilers in node. |  
| eth | ethnewFilter | YES ||  
| eth | ethnewBlockFilter | YES ||  
| eth | ethnewPendingTransactionFilter | YES ||  
| eth | ethuninstallFilter | YES ||  
| eth | ethgetFilterChanges | YES ||  
| eth | ethgetFilterLogs | YES ||  
| eth | ethgetLogs | YES ||  
| eth | ethbridgeState | YES ||  
| eth | ethnetHashrate | YES ||  
| db | dbputString | - | Deprecated |  
| db | dbgetString | - | Deprecated |  
| db | dbputHex | - | Deprecated |  
| db | dbgetHex | - | Deprecated |  
| debug | debugtraceTransaction | YES ||  
| debug | debugtraceBlockByHash | YES ||  
| debug | debugwireProtocolQueueSize | YES ||  
| evm | evmincreaseTime | YES ||  
| evm | evmmine | YES ||  
| evm | evmreset | YES ||  
| evm | evmrevert | YES ||  
| evm | evmsnapshot | YES ||  
| evm | evmstartMining | YES ||  
| evm | evmstopMining | YES ||  
| mnr | mnrsSubmitBitcoinBlock | YES ||  
| mnr | mnrsSubmitBitcoinBlockTransactions | YES ||  
| mnr | mnrsSubmitBitcoinBlockPartialMerkle | YES ||  
| mnr | mnrgetWork | YES ||  
| personal | personaldumpRawKey | YES ||  
| personal | personalimportRawKey | YES ||  
| personal | personallistAccounts | YES ||  
| personal | personallockAccount | YES ||  
| personal | personalnewAccountWithSeed | YES ||  
| personal | personalnewAccount | YES ||  
| personal | personalsendTransaction | YES ||  
| personal | personalunlockAccount | YES ||  
| rsk | rskgetRawTransactionReceiptByHash | YES ||  
| rsk | rskgetTransactionReceiptNodesByHash | YES ||  
| rsk | rskgetRawBlockHeaderByHash | YES ||  
| rsk | rskgetRawBlockHeaderByNumber | YES ||  
| rsk | rskprotocolVersion | YES ||



| trace | tracetransaction | YES ||  
| trace | traceblock | YES ||  
| trace | tracefilter | YES ||  
| txpool | txpoolcontent | YES ||  
| txpool | txpoolinspect | YES ||  
| txpool | txpoolstatus | YES ||  
| sco | scobanAddress | YES ||  
| sco | scounbanAddress | YES ||  
| sco | scopeerList | YES ||  
| sco | scobannedAddresses | YES ||  
| sco | scoreputationSummary | YES ||  
| shh | shhpost | - | Whisper protocol not supported. |  
| shh | shhversion | - | Whisper protocol not supported. |  
| shh | shhnewIdentity | - | Whisper protocol not supported. |  
| shh | shhhasIdentity | - | Whisper protocol not supported. |  
| shh | shhnewGroup | - | Whisper protocol not supported. |  
| shh | shhaddToGroup | - | Whisper protocol not supported. |  
| shh | shhnewFilter | - | Whisper protocol not supported. |  
| shh | shhuninstallFilter | - | Whisper protocol not supported. |  
| shh | shhgetFilterChanges | - | Whisper protocol not supported. |  
| shh | shhgetMessages | - | Whisper protocol not supported. |

## JSON RPC method details

These descriptions are taken from  
Ethereum's JSON RPC documentation.

### web3clientVersion

Returns the current client version.

#### Parameters

none

#### Returns

String - The current client version.

#### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"web3clientVersion","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc":"2.0",
  "result": "Mist/v0.9.3/darwin/go1.4.1"
}
```

### web3sha3

Returns Keccak-256 (not the standardized SHA3-256) of the given data.

#### Parameters

1. DATA - the data to convert into a SHA3 hash.

#### Example Parameters

```
js
params: [
  "0x68656c6c6f20776f726c64"
]
```

#### Returns

DATA - The SHA3 result of the given string.

#### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"web3sha3","params":["0x68656c6c6f20776f726c64"],"id":64}'

// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad"
}
```

#### netversion

Returns the current network id.

#### Parameters

none

#### Returns

String - The current network id.

- "30": RSK Mainnet
- "31": Ethercamp test network
- "32": Developer network
- "33": RSK created local network

#### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"netversion","params":[],"id":67}'

// Result
```

```
{  
  "id":67,  
  "jsonrpc": "2.0",  
  "result": "3"  
}
```

## netlistening

Returns true if client is actively listening for network connections.

### Parameters

none

### Returns

Boolean - true when listening, otherwise false.

### Example

```
js  
// Request  
curl -X POST --data '{"jsonrpc":"2.0","method":"netlistening","params":[],"id":67}'  
  
// Result  
{  
  "id":67,  
  "jsonrpc":"2.0",  
  "result":true  
}
```

## netpeerCount

Returns number of peers currently connected to the client.

### Parameters

none

### Returns

QUANTITY - integer of the number of connected peers.

### Example

```
js  
// Request  
curl -X POST --data '{"jsonrpc":"2.0","method":"netpeerCount","params":[],"id":74}'  
  
// Result  
{  
  "id":74,  
  "jsonrpc": "2.0",
```

```
"result": "0x2" // 2
}
```

## netpeerList

Returns list of peers known to the client.

### Parameters

none

### Returns

Array - The list of peers.

### Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"netpeerList","params":[],"id":1}'
```

// Result

```
{
  "jsonrpc":"2.0",
  "id":1,
  "result": [
    "3fd44f66 | ec2-52-15-37-171.us-east-2.compute.amazonaws.com/52.15.37.171:5050",
    "50517861 | bootstrap14.rsk.co/54.169.136.187:5050",
    "434f8932 | bootstrap07.rsk.co/54.169.12.15:5050"
  ]
}
```

## ethchainId

Returns the currently configured chain id, a value used in replay-protected transaction signing as introduced by EIP-155.

### Parameters

none

### Returns

String - The current chainId.

### Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethchainId","params":[],"id":67}'
```

// Result

```
{
```

```
"id":67,  
"jsonrpc": "2.0",  
"result": "0x1e"  
}
```

## ethprotocolVersion

Returns the current ethereum protocol version.

Parameters  
none

Returns

String - The current ethereum protocol version.

## Example

```
js  
// Request  
curl -X POST --data '{"jsonrpc":"2.0","method":"ethprotocolVersion","params":[],"id":67}'  
  
// Result  
{  
  "id":67,  
  "jsonrpc": "2.0",  
  "result": "0x54"  
}
```

## ethsyncing

Returns an object with data about the sync status or false.

Parameters  
none

Returns

Object|Boolean, An object with sync status data or FALSE, when not syncing:

- startingBlock: QUANTITY - The block at which the import started (will only be reset, after the sync reached his head)
- currentBlock: QUANTITY - The current block, same as ethblockNumber
- highestBlock: QUANTITY - The estimated highest block

## Example

```
js  
// Request  
curl -X POST --data '{"jsonrpc":"2.0","method":"ethsyncing","params":[],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": {
    startingBlock: '0x384',
    currentBlock: '0x386',
    highestBlock: '0x454'
  }
}
// Or when not syncing
{
  "id":1,
  "jsonrpc": "2.0",
  "result": false
}
```

ethcoinbase

Returns the client coinbase address.

Parameters

none

Returns

DATA, 20 bytes - the current coinbase address.

Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethcoinbase","params":[],"id":64}'
```

```
// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0xc94770007dda54cF92009BFF0dE90c06F603a09f"
}
```

ethmining

Returns true if client is actively mining new blocks.

Parameters

none

Returns

Boolean - returns true if the client is mining, otherwise false.

Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethmining","params":[],"id":71}'
```

// Result

```
{
  "id":71,
  "jsonrpc": "2.0",
  "result": true
}
```

ethhashrate

Returns the number of hashes per second that the node is mining with.

Parameters

none

Returns

QUANTITY - number of hashes per second.

Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethhashrate","params":[],"id":71}'
```

// Result

```
{
  "id":71,
  "jsonrpc": "2.0",
  "result": "0x38a"
}
```

ethgasPrice

Returns the current price per gas in wei.

Parameters

none

Returns

QUANTITY - integer of the current gas price in wei.

Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgasPrice","params":[],"id":73}'

// Result
{
  "id":73,
  "jsonrpc": "2.0",
  "result": "0x09184e72a000" // 10000000000000
}
```

ethaccounts

Returns a list of addresses owned by client.

Parameters

none

Returns

Array of DATA, 20 Bytes - addresses owned by the client.

Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethaccounts","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["0xc94770007dda54cF92009BFF0dE90c06F603a09f"]
}
```

ethblockNumber

Returns the number of most recent block.

Parameters

none

Returns

QUANTITY - integer of the current block number the client is on.



## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethblockNumber","params":[],"id":1}'

// Result
{
  "id":83,
  "jsonrpc": "2.0",
  "result": "0xc94" // 1207
}
```

## ethgetBalance

Returns the balance of the account of given address.

### Parameters

1. DATA, 20 Bytes - address to check for balance.
2. QUANTITY|TAG|MAP - integer block number, or the string "latest", "earliest" or "pending", see the default block parameter, or a map containing a block hash string, under the key "blockHash" or a string hexadecimal number, under the key "blockNumber".

### Example Parameters

```
js
params: [
  '0xc94770007dda54cF92009BFF0dE90c06F603a09f',
  'latest'
]
```

### Returns

QUANTITY - integer of the current balance in wei.

## Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetBalance","params":["0xc94770007dda54cF92009BFF0dE90c06F603a09f", "latest"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0234c8a3397aab58" // 158972490234375000
}
```

Returns the value from a storage position at a given address.

1. DATA, 20 Bytes - address of the storage.
2. QUANTITY - integer of the position in the storage.
3. QUANTITY|TAG|MAP - integer block number, or the string "latest", "earliest" or "pending", see the default block parameter, or a map containing a block hash string, under the key "blockHash" or a string hexadecimal number, under the key "blockNumber".

DATA - the value at this storage position (A 32-byte zero value is returned for non-existing keys).

Calculating the correct position depends on the storage to retrieve. Consider the following contract deployed at 0x295a70b2de5e3953354a6a8344e616ed314d7251 by address 0x391694e7e0b0cce554cb130d723a9d27458f9298.

```
contract Storage {
    uint pos0;
    mapping(address => uint) pos1;

    function Storage() {
        pos0 = 1234;
        pos1[msg.sender] = 5678;
    }
}
```

```
js
curl -X POST --data '{"jsonrpc":"2.0", "method": "ethgetStorageAt", "params":
["0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x0", "latest"], "id": 1}' localhost:8545
```

[illegible]

```
js
keccak(LeftPad32(key, 0), LeftPad32(map position, 0))
```

[illegible]

[illegible]

```
js
curl -X POST --data '{"jsonrpc":"2.0", "method": "ethgetStorageAt", "params":
["0x295a70b2de5e3953354a6a8344e616ed314d7251",
"0x6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9", "latest"], "id": 1}'
localhost:8545
```

```
js
// Request
curl -X POST --data
 '{"jsonrpc":"2.0","method":"ethgetTransactionCount","params":["0xc94770007dda54cF92009BFF0dE90c0
6F603a09f","latest"],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

## ethgetBlockTransactionCountByHash

Returns the number of transactions in a block from a block matching the given block hash.

### Parameters

1. DATA, 32 Bytes - hash of a block.

### Example Parameters

```
js
params: [
  '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238'
]
```

### Returns

QUANTITY - integer of the number of transactions in this block, or null when no block was found.

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetBlockTransactionCountByHash","params":["0xc94770007dda54cF92009BFF0dE90c06F603a09f"],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc" // 11
}
```

## ethgetBlockTransactionCountByNumber

Returns the number of transactions in a block matching the given block number.

### Parameters

1. QUANTITY|TAG - integer of a block number, or the string "earliest", "latest" or "pending", as in the default block parameter.

#### Example Parameters

```
js
params: [
  '0xe8', // 232
]
```

#### Returns

QUANTITY - integer of the number of transactions in this block.

#### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetBlockTransactionCountByNumber","params":["0xe8"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xa" // 10
}
```

#### ethgetUncleCountByBlockHash

Returns the number of uncles in a block from a block matching the given block hash.

#### Parameters

1. DATA, 32 Bytes - hash of a block.

#### Example Parameters

```
js
params: [
  '0xc94770007dda54cF92009BFF0dE90c06F603a09f'
]
```

#### Returns

QUANTITY - integer of the number of uncles in this block.

#### Example

```
js
// Request
curl -X POST --data
```

```
'{"jsonrpc":"2.0","method":"ethgetUncleCountByBlockHash","params":["0xc94770007dda54cF92009BFF0dE90c06F603a09f"],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc" // 1
}
```

## ethgetUncleCountByBlockNumber

Returns the number of uncles in a block from a block matching the given block number.

### Parameters

1. QUANTITY|TAG - integer of a block number, or the string "latest", "earliest" or "pending", see the default block parameter.

```
js
params: [
  '0xe8', // 232
]
```

### Returns

QUANTITY - integer of the number of uncles in this block.

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetUncleCountByBlockNumber","params":["0xe8"],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

## ethgetCode

Returns code at a given address.

## Parameters

1. DATA, 20 Bytes - address.
2. QUANTITY|TAG|MAP - integer block number, or the string "latest", "earliest" or "pending", see the default block parameter, or a map containing a block hash string, under the key "blockHash" or a string hexadecimal number, under the key "blockNumber".

## Example Parameters

```
js
params: [
  '0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b',
  '0x2' // 2
]
```

## Returns

DATA - the code from the given address.

## Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetCode","params":["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
"0x2"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result":
"0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f25b600060
078202905091905056"
}
```

## ethsign

The sign method calculates an Ethereum specific signature with: `sign(keccak256("\x19Ethereum Signed Message:\n" + len(message) + message)))`.

By adding a prefix to the message makes the calculated signature recognisable as an Ethereum specific signature. This prevents misuse where a malicious DApp can sign arbitrary data (e.g. transaction) and use the signature to impersonate the victim.

Note the address to sign with must be unlocked.

## Parameters

account, message

1. DATA, 20 Bytes - address.
2. DATA, N Bytes - message to sign.

## Returns

DATA: Signature

## Example

js

// Request

curl -X POST --data

```
'{"jsonrpc": "2.0", "method": "ethsign", "params": ["0x9b2055d370f73ec7d8a03e965129118dc8f5bf83", "0xdeadbeaf"], "id": 1}'
```

// Result

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result":
"0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdce3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b"
}
```

## ethsendTransaction

Creates new message call transaction or a contract creation, if the data field contains code.

## Parameters

### 1. Object - The transaction object

- from: DATA, 20 Bytes - The address the transaction is sent from.
- to: DATA, 20 Bytes - (optional when creating new contract) The address the transaction is sent to.
- gas: QUANTITY - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
- gasPrice: QUANTITY - (optional, default: 0) Integer of the gasPrice used for each paid gas
- value: QUANTITY - (optional) Integer of the value sent with this transaction
- data: DATA - The compiled code of a contract OR the hash of the invoked method signature and encoded parameters. For details see Ethereum Contract ABI
- nonce: QUANTITY - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

## Example Parameters

js

params: [{

"from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",

"to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",

"gas": "0x76c0", // 30400

"gasPrice": "0x9184e72a000", // 10000000000000

"value": "0x9184e72a", // 2441406250

"data":

"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"

}]



## Returns

DATA, 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use `eth.getTransactionReceipt` to get the contract address, after the transaction was mined, when you created a contract.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethsendTransaction","params":[{"see above}], "id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

## ethsendRawTransaction

Creates new message call transaction or a contract creation for signed transactions.

## Parameters

1. DATA, The signed transaction data.

## Example Parameters

```
js
params:
["0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"]
```

## Returns

DATA, 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use `eth.getTransactionReceipt` to get the contract address, after the transaction was mined, when you created a contract.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethsendRawTransaction","params":[{"see above}], "id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

## ethcall

Executes a new message call immediately without creating a transaction on the block chain.

### Parameters

#### 1. Object - The transaction call object

- from: DATA, 20 Bytes - (optional) The address the transaction is sent from.
- to: DATA, 20 Bytes - The address the transaction is directed to.
- gas: QUANTITY - (optional) Integer of the gas provided for the transaction execution. ethcall consumes zero gas, but this parameter may be needed by some executions.
- gasPrice: QUANTITY - (optional) Integer of the gasPrice used for each paid gas
- value: QUANTITY - (optional) Integer of the value sent with this transaction
- data|input: DATA - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation

#### 2. QUANTITY|TAG - integer block number, or the string "latest", "earliest" or "pending", see the default block parameter

### Example Parameters

```
js
params: [{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567"
}, "latest"]
```

### Returns

DATA - the return value of executed contract.

### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethcall","params":[{"see above}],"id":1}'
```

#### // Result

```
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x"
}
```

#### // In case of REVERT error

```
{
  "jsonrpc":"2.0",
  "id":1,
  "error":{
    "code":-32015,
    "message":"VM Exception while processing transaction: revert reason",
    "data":"0x08c379a..."
  }
}
```

```
}  
}
```

## ethestimateGas

Generates and returns an estimate of how much gas is necessary to allow the transaction to complete. The transaction will not be added to the blockchain. Note that the estimate may be significantly more than the amount of gas actually used by the transaction, for a variety of reasons including EVM mechanics and node performance.

### Parameters

See ethcall parameters, expect that all properties are optional. If no gas limit is specified geth uses the block gas limit from the pending block as an upper bound. As a result the returned estimate might not be enough to executed the call/transaction when the amount of gas is higher than the pending block gas limit.

### Returns

QUANTITY - the amount of gas used.

### Example

```
js  
// Request  
curl -X POST --data '{"jsonrpc":"2.0","method":"ethestimateGas","params":[{"see above}], "id":1}'  
  
// Result  
{  
  "id":1,  
  "jsonrpc": "2.0",  
  "result": "0x5208" // 21000  
}
```

## ethgetBlockByHash

Returns information about a block by hash.

### Parameters

1. DATA, 32 Bytes - Hash of a block.
2. Boolean - If true it returns the full transaction objects, if false only the hashes of the transactions.

### Example Parameters

```
js  
params: [  
  '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',  
  true  
]
```

## Returns

Object - A block object, or null when no block was found:

- number: QUANTITY - the block number. null when its pending block.
- hash: DATA, 32 Bytes - hash of the block. null when its pending block.
- parentHash: DATA, 32 Bytes - hash of the parent block.
- nonce: DATA, 8 Bytes - hash of the generated proof-of-work. null when its pending block.
- sha3Uncles: DATA, 32 Bytes - SHA3 of the uncles data in the block.
- logsBloom: DATA, 256 Bytes - the bloom filter for the logs of the block. null when its pending block.
- transactionsRoot: DATA, 32 Bytes - the root of the transaction trie of the block.
- stateRoot: DATA, 32 Bytes - the root of the final state trie of the block.
- receiptsRoot: DATA, 32 Bytes - the root of the receipts trie of the block.
- miner: DATA, 20 Bytes - the address of the beneficiary to whom the mining rewards were given.
- difficulty: QUANTITY - integer of the difficulty for this block.
- cumulativeDifficulty: QUANTITY - integer of the difficulty for this block plus its uncles' difficulties.
- totalDifficulty: QUANTITY - integer of the total difficulty of the chain until this block.
- extraData: DATA - the "extra data" field of this block.
- size: QUANTITY - integer the size of this block in bytes.
- gasLimit: QUANTITY - the maximum gas allowed in this block.
- gasUsed: QUANTITY - the total used gas by all transactions in this block.
- timestamp: QUANTITY - the unix timestamp for when the block was collated.
- transactions: Array - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- uncles: Array - Array of uncle hashes.

## Example

js

// Request

curl -X POST --data

```
'{"jsonrpc":"2.0","method":"ethgetBlockByHash","params":["0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331", true],"id":1}'
```

// Result

```
{
  "id":1,
  "jsonrpc":"2.0",
  "result": {
    "number": "0x1b4", // 436
    "hash": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "parentHash": "0x9646252be9520f6e71339a8df9c55e4d7619deeb018d2a3f2d21fc165dde5eb5",
    "nonce": "0xe04d296d2460cfb8472af2c5fd05b5a214109c25688d3704aed5484f9a7792f2",
    "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
    "logsBloom": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "transactionsRoot": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421",
    "stateRoot": "0xd5855eb08b3387c0af375e9cdb6acfc05eb8f519e419b874b6ff2ffda7ed1dff",
    "miner": "0x4e65fda2159562a496f9f3522f89122a3088497a",
    "difficulty": "0x027f07", // 163591
    "cumulativeDifficulty": "0x027f07", // 163591
    "totalDifficulty": "0x027f07", // 163591
    "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "size": "0x027f07", // 163591
  }
}
```

```

    "gasLimit": "0x9f759", // 653145
    "gasUsed": "0x9f759", // 653145
    "timestamp": "0x54e34e8e" // 1424182926
    "transactions": [{...},{ ... }]
    "uncles": ["0x1606e5...", "0xd5145a9..."]
  }
}

```

## ethgetBlockByNumber

Returns information about a block by block number.

### Parameters

1. QUANTITY|TAG - integer of a block number, or the string "earliest", "latest" or "pending", as in the default block parameter.
2. Boolean - If true it returns the full transaction objects, if false only the hashes of the transactions.

### Example Parameters

```

js
params: [
  '0x1b4', // 436
  true
]

```

### Returns

See [ethgetBlockByHash](#)

### Example

```

js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgetBlockByNumber","params":["0x1b4", true],"id":1}'

```

Result see [ethgetBlockByHash](#)

## eth.getTransactionByHash

Returns the information about a transaction requested by transaction hash.

### Parameters

1. DATA, 32 Bytes - hash of a transaction

### Example Parameters

```

js
params: [

```

```
"0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b"
]
```

## Returns

Object - A transaction object, or null when no transaction was found:

- blockHash: DATA, 32 Bytes - hash of the block where this transaction was in. null when its pending.
- blockNumber: QUANTITY - block number where this transaction was in. null when its pending.
- from: DATA, 20 Bytes - address of the sender.
- gas: QUANTITY - gas provided by the sender.
- gasPrice: QUANTITY - gas price provided by the sender in Wei.
- hash: DATA, 32 Bytes - hash of the transaction.
- input: DATA - the data send along with the transaction.
- nonce: QUANTITY - the number of transactions made by the sender prior to this one.
- to: DATA, 20 Bytes - address of the receiver. null when its a contract creation transaction.
- transactionIndex: QUANTITY - integer of the transaction's index position in the block. null when its pending.
- value: QUANTITY - value transferred in Wei.
- v: QUANTITY - ECDSA recovery id
- r: QUANTITY - ECDSA signature r
- s: QUANTITY - ECDSA signature s

## Example

js

// Request

```
curl -X POST --data
```

```
'{"jsonrpc":"2.0","method":"ethgetTransactionByHash","params":["0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b"],"id":1}'
```

// Result

```
{
  "jsonrpc":"2.0",
  "id":1,
  "result":{
    "blockHash":"0x1d59ff54b1eb26b013ce3cb5fc9dab3705b415a67127a003c3e61eb445bb8df2",
    "blockNumber":"0x5daf3b", // 6139707
    "from":"0xa7d9ddbe1f17865597fbd27ec712455208b6b76d",
    "gas":"0xc350", // 50000
    "gasPrice":"0x4a817c800", // 20000000000
    "hash":"0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b",
    "input":"0x68656c6c6f21",
    "nonce":"0x15", // 21
    "to":"0xf02c1c8e6114b1dbe8937a39260b5b0a374432bb",
    "transactionIndex":"0x41", // 65
    "value":"0xf3dbb76162000", // 4290000000000000
    "v":"0x25", // 37
    "r":"0x1b5e176d927f8e9ab405058b2d2457392da3e20f328b16ddabcebc33eaac5fea",
    "s":"0x4ba69724e8f69de52f0125ad8b3c5c2cef33019bac3249e2c0a2192766d1721c"
  }
}
```

## ethgetTransactionByBlockHashAndIndex

Returns information about a transaction by block hash and transaction index position.

### Parameters

1. DATA, 32 Bytes - hash of a block.
2. QUANTITY - integer of the transaction index position.

### Example Parameters

```
js
params: [
  '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
  '0x0' // 0
]
```

### Returns

See [ethgetTransactionByHash](#)

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetTransactionByBlockHashAndIndex","params":["0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b", "0x0"],"id":1}'
```

Result see [ethgetTransactionByHash](#)

## ethgetTransactionByBlockNumberAndIndex

Returns information about a transaction by block number and transaction index position.

### Parameters

1. QUANTITY|TAG - a block number, or the string "earliest", "latest" or "pending", as in the default block parameter.
2. QUANTITY - the transaction index position.

### Example Parameters

```
js
params: [
  '0x29c', // 668
  '0x0' // 0
]
```

### Returns

See `ethgetTransactionByHash`

Example

js

// Request

curl -X POST --data

```
'{"jsonrpc":"2.0","method":"ethgetTransactionByBlockNumberAndIndex","params":["0x29c", "0x0"],"id":1}'
```

Result see `ethgetTransactionByHash`

`ethgetTransactionReceipt`

Returns the receipt of a transaction by transaction hash.

Note That the receipt is not available for pending transactions.

Parameters

1. DATA, 32 Bytes - hash of a transaction

Example Parameters

js

params: [

```
'0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238'
```

]

Returns

Object - A transaction receipt object, or null when no receipt was found:

- `transactionHash` : DATA, 32 Bytes - hash of the transaction.
- `transactionIndex`: QUANTITY - integer of the transaction's index position in the block.
- `blockHash`: DATA, 32 Bytes - hash of the block where this transaction was in.
- `blockNumber`: QUANTITY - block number where this transaction was in.
- `from`: DATA, 20 Bytes - address of the sender.
- `to`: DATA, 20 Bytes - address of the receiver. null when it's a contract creation transaction.
- `cumulativeGasUsed` : QUANTITY - The total amount of gas used when this transaction was executed in the block.
- `gasUsed` : QUANTITY - The amount of gas used by this specific transaction alone.
- `contractAddress` : DATA, 20 Bytes - The contract address created, if the transaction was a contract creation, otherwise null.
- `logs`: Array - Array of log objects, which this transaction generated.
- `logsBloom`: DATA, 256 Bytes - Bloom filter for light clients to quickly retrieve related logs.

It also returns either :

- `root` : DATA 32 bytes of post-transaction stateroot (pre Byzantium)
- `status`: QUANTITY either 1 (success) or 0 (failure)



## Example

js

// Request

curl -X POST --data

```
'{"jsonrpc":"2.0","method":"ethgetTransactionReceipt","params":["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"],"id":1}'
```

// Result

```
{
  "id":1,
  "jsonrpc":"2.0",
  "result": {
    transactionHash: '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238',
    transactionIndex: '0x1', // 1
    blockNumber: '0xb', // 11
    blockHash: '0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b',
    cumulativeGasUsed: '0x33bc', // 13244
    gasUsed: '0x4dc', // 1244
    contractAddress: '0xb60e8dd61c5d32be8058bb8eb970870f07233155', // or null, if none was created
    logs: [{
      // logs as returned by getFilterLogs, etc.
    }, ...],
    logsBloom: "0x00...0", // 256 byte bloom filter
    status: '0x1'
  }
}
```

## ethpendingTransactions

Returns the pending transactions submitted by the node operator.

### Parameters

none

### Returns

Array - A list of pending transactions submitted by the node operator.

## Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethpendingTransactions","params":[],"id":1}'
```

// Result

```
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
    blockHash: '0x0000000000000000000000000000000000000000000000000000000000000000',
    blockNumber: null,
    from: '0x28bdb9c230f4d5e45435e4d006326ee32e46cb31',
  }
]
```

```

gas: '0x204734',
gasPrice: '0x4a817c800',
hash: '0x8dfa6a59307a490d672494a171feee09db511f05e9c097e098edc2881f9ca4f6',
input: '0x6080604052600',
nonce: '0x12',
to: null,
transactionIndex: '0x0',
value: '0x0',
v: '0x3d',
r: '0xaabc9ddaaffb2ae0bac4107697547d22d9383667d9e97f5409dd6881ce08f13f',
s: '0x69e43116be8f842dcd4a0b2f760043737a59534430b762317db21d9ac8c5034',
type: '0x0'
},...,{
blockHash: '0x0000000000000000000000000000000000000000000000000000000000000000',
blockNumber: null,
from: '0x28bdb9c230f4d5e45435e4d006326ee32e487b31',
gas: '0x205940',
gasPrice: '0x4a817c800',
hash: '0x8e4340ea3983d86e4b6c44249362f716ec9e09849ef9b6e3321140581d2e4dac',
input: '0xe4b6c4424936',
nonce: '0x14',
to: null,
transactionIndex: '0x0',
value: '0x0',
v: '0x3d',
r: '0x1ec191ef20b0e9628c4397665977cbe7a53a263c04f6f185132b77fa0fd5ca44',
s: '0x8a58e00c63e05cfeae4f1cf19f05ce82079dc4d5857e2cc281b7797d58b5faf',
type: '0x0'
}]
}

```

## ethgetUncleByBlockHashAndIndex

Returns information about an uncle of a block by hash and the uncle index position.

### Parameters

1. DATA, 32 Bytes - hash a block.
2. QUANTITY - the uncle's index position.

```

js
params: [
  '0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b',
  '0x0' // 0
]

```

### Returns

See [ethgetBlockByHash](#)

Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"ethgetUncleByBlockHashAndIndex","params":["0xc6ef2fc5426d6ad6fd9e2a26
abeab0aa2411b7ab17f30a99d3cb96aed1d1055b", "0x0"],"id":1}'
```

Result see [ethgetBlockByHash](#)

Note: An uncle doesn't contain individual transactions.

[ethgetUncleByBlockNumberAndIndex](#)

Returns information about a uncle of a block by number and uncle index position.

Parameters

1. QUANTITY|TAG - a block number, or the string "earliest", "latest" or "pending", as in the default block parameter.
2. QUANTITY - the uncle's index position.

Example Parameters

```
js
params: [
  '0x29c', // 668
  '0x0' // 0
]
```

Returns

See [ethgetBlockByHash](#)

Note: An uncle doesn't contain individual transactions.

Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgetUncleByBlockNumberAndIndex","params":["0x29c",
"0x0"],"id":1}'
```

Result see [ethgetBlockByHash](#)

[ethnewFilter](#)

Creates a filter object, based on filter options, to notify when the state changes (logs).  
To check if the state has changed, call [ethgetFilterChanges](#).



## ethnewBlockFilter

Creates a filter in the node, to notify when a new block arrives.  
To check if the state has changed, call `ethgetFilterChanges`.

### Parameters

None

### Returns

QUANTITY - A filter id.

### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethnewBlockFilter","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

## ethnewPendingTransactionFilter

Creates a filter in the node, to notify when new pending transactions arrive.  
To check if the state has changed, call `ethgetFilterChanges`.

### Parameters

None

### Returns

QUANTITY - A filter id.

### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethnewPendingTransactionFilter","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

## ethuninstallFilter

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additionally Filters timeout when they aren't requested with ethgetFilterChanges for a period of time.

## Parameters

1. QUANTITY - The filter id.

## Example Parameters

```
js
params: [
  "0xb" // 11
]
```

## Returns

Boolean - true if the filter was successfully uninstalled, otherwise false.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethuninstallFilter","params":["0xb"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

## ethgetFilterChanges

Polling method for a filter, which returns an array of logs which occurred since last poll.

## Parameters

1. QUANTITY - the filter id.

## Example Parameters

```
js
params: [
  "0x16" // 22
]
```

## Returns

Array - Array of log objects, or an empty array if nothing has changed since last poll.

- For filters created with `ethnewBlockFilter` the return are block hashes (DATA, 32 Bytes), e.g. ["0x3454645634534..."].
- For filters created with `ethnewPendingTransactionFilter` the return are transaction hashes (DATA, 32 Bytes), e.g. ["0x6345343454645..."].
- For filters created with `ethnewFilter` logs are objects with following params:
  - removed: TAG - true when the log was removed, due to a chain reorganization. false if its a valid log.
  - logIndex: QUANTITY - integer of the log index position in the block. null when its pending log.
  - transactionIndex: QUANTITY - integer of the transactions index position log was created from. null when its pending log.
  - transactionHash: DATA, 32 Bytes - hash of the transactions this log was created from. null when its pending log.
  - blockHash: DATA, 32 Bytes - hash of the block where this log was in. null when its pending. null when its pending log.
  - blockNumber: QUANTITY - the block number where this log was in. null when its pending. null when its pending log.
  - address: DATA, 20 Bytes - address from which this log originated.
  - data: DATA - contains the non-indexed arguments of the log.
  - topics: Array of DATA - Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. `Deposit(address,bytes32,uint256)`), except you declared the event with the anonymous specifier.)

#### Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgetFilterChanges","params":["0x16"],"id":73}'
```

// Result

```
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
    "logIndex": "0x1", // 1
    "blockNumber": "0x1b4", // 436
    "blockHash": "0x8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcfdf829c5a142f1fcd7d",
    "transactionHash": "0xdf829c5a142f1fcd7d8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcf",
    "transactionIndex": "0x0", // 0
    "address": "0x16c5785ac562ff41e2dcfdf829c5a142f1fcd7d",
    "data": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "topics": ["0x59ebeb90bc63057b6515673c3ecf9438e5058bca0f92585014eced636878c9a5"]
  }, {
    ...
  }]
}
```

#### ethgetFilterLogs

Returns an array of all logs matching filter with given id.

#### Parameters

1. QUANTITY - The filter id.

#### Example Parameters

```
js
params: [
  "0x16" // 22
]
```

Returns

See `ethgetFilterChanges`

#### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgetFilterLogs","params":["0x16"],"id":74}'
```

Result see `ethgetFilterChanges`

#### ethgetLogs

Returns an array of all logs matching a given filter object.

#### Parameters

1. Object - The filter options:

- fromBlock: QUANTITY|TAG - (optional, default: "latest") Integer block number, or "latest" for the last mined block or "pending", "earliest" for not yet mined transactions.
- toBlock: QUANTITY|TAG - (optional, default: "latest") Integer block number, or "latest" for the last mined block or "pending", "earliest" for not yet mined transactions.
- address: DATA|Array, 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
- topics: Array of DATA, - (optional) Array of 32 Bytes DATA topics. Topics are order-dependent. Each topic can also be an array of DATA with "or" options.
- blockhash: DATA, 32 Bytes - (optional) With the addition of EIP-234 (Geth >= v1.8.13 or Parity >= v2.1.0), blockHash is a new filter option which restricts the logs returned to the single block with the 32-byte hash blockHash. Using blockHash is equivalent to fromBlock = toBlock = the block number with hash blockHash. If blockHash is present in the filter criteria, then neither fromBlock nor toBlock are allowed.

#### Example Parameters

```
js
params: [{
  "topics": ["0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]
}]
```

Returns

See `ethgetFilterChanges`



## Example

js

// Request

curl -X POST --data

```
'{"jsonrpc":"2.0","method":"ethgetLogs","params":[{"topics":["0x00000000000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]}],"id":74}'
```

Result see ethgetFilterChanges

## ethgetWork

Returns the hash of the current block, the seedHash, and the boundary condition to be met ("target").

### Parameters

none

### Returns

Array - Array with the following properties:

1. DATA, 32 Bytes - current block header pow-hash
2. DATA, 32 Bytes - the seed hash used for the DAG.
3. DATA, 32 Bytes - the boundary condition ("target"),  $2^{256}$  / difficulty.

## Example

js

// Request

```
curl -X POST --data '{"jsonrpc":"2.0","method":"ethgetWork","params":[],"id":73}'
```

// Result

```
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [
    "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
    "0x5EED000000000000000000000000000000000000000000000000000000000000",
    "0xd1ff1c017100000000000000000000000000000000d1ff1c017100000000000000000000"
  ]
}
```

## ethsubmitWork

Used for submitting a proof-of-work solution.

### Parameters

1. DATA, 8 Bytes - The nonce found (64 bits)
2. DATA, 32 Bytes - The header's pow-hash (256 bits)
3. DATA, 32 Bytes - The mix digest (256 bits)

## Example Parameters

```
js
params: [
  "0x000000000000000001",
  "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
  "0xD1FE5700000000000000000000000000D1FE57000000000000000000000000"
]
```

## Returns

Boolean - returns true if the provided solution is valid, otherwise false.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0", "method":"ethsubmitWork", "params":["0x000000000000000001",
"0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
"0xD1GE5700000000000000000000000000D1GE57000000000000000000000000"],"id":73}'

// Result
{
  "id":73,
  "jsonrpc":"2.0",
  "result": true
}
```

## ethsubmitHashrate

Used for submitting mining hashrate.

## Parameters

1. Hashrate, a hexadecimal string representation (32 bytes) of the hash rate
2. ID, String - A random hexadecimal(32 bytes) ID identifying the client

## Example Parameters

```
js
params: [
  "0x00000000000000000000000000000000000000000000000000000000500000",
  "0x59daa26581d0acd1fce254fb7e85952f4c09d0915afd33d3886cd914bc7d283c"
]
```

## Returns

Boolean - returns true if submitting went through successfully and false otherwise.

### Example

js

```
// Request
```

[illegible]

// Result

```
{
  "id":73,
  "jsonrpc":"2.0",
  "result": true
}
```

## ethgetProof

Returns the account- and storage-values of the specified account including the Merkle-proof.

## Parameters

1. DATA, 20 bytes - address of the account or contract
2. ARRAY, 32 Bytes - array of storage-keys which should be proofed and included. See `ethgetStorageAt`
3. QUANTITY|TAG - integer block number, or the string "latest" or "earliest", see the default block parameter

## Example Parameters

params:

[illegible]

## Returns

## Returns

Object - A account object:

balance: QUANTITY - the balance of the account. See [ethgetBalance](#)

codeHash: DATA, 32 Bytes - hash of the code of the account. For a simple Account without code it will return "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470"

nonce: QUANTITY, - nonce of the account. See `eth.getTransactionCount`

storageHash: DATA, 32 Bytes - SHA3 of the StorageRoot. All storage will deliver a MerkleProof starting with this rootHash.

accountProof: ARRAY - Array of rlp-serialized MerkleTree-Nodes, starting with the stateRoot-Node, following the path of the SHA3 (address) as key.





- using the HTTP verb POST
- specifying a Content-Type header of application/json
- with a request body specified as stringified JSON

For example, a curl command to a localhost Rootstock node would look similar to this:

```
shell
curl http://localhost:4444/ \ \
-X POST \
-H "Content-Type: application/json" \
--data
'{"jsonrpc":"2.0","method":"RPCMETHODNAME","params":[RPCREQUESTPARAMETERS],"id":1}'
```

## WebSockets transport protocol

WebSockets connections should be established:

- to the port number specified in the config for rpc.providers.web.ws.port
- this is 4445 by default
- public nodes do not have the WebSockets transport protocol enabled
- to the WebSockets route (/websocket)

Once connected:

- Send a request body specified as stringified JSON
- No "verb" or "headers" are necessary, as these are specific to the HTTP transport protocol

For example, a wscat command to connect to a localhost Rootstock node would look similar to this:

```
shell
wscat -c ws://localhost:4445/websocket
```

After the connection has been established using wscat, you may send multiple RPC requests within the same session. (Note that > marks requests to be input, and that < marks responses that will be printed.)

```
json
{"jsonrpc":"2.0","method":"RPCMETHODNAME","params":[RPCREQUESTPARAMETERS],"id":1}
{"jsonrpc":"2.0","id":1,"result":"RPCRESPONSE"}
{"jsonrpc":"2.0","method":"RPCMETHODNAME","params":[RPCREQUESTPARAMETERS],"id":2}
{"jsonrpc":"2.0","id":2,"result":"RPCRESPONSE"}
```

# 03-personal-module-methods.md:

```
---
title: Personal Module Methods
sidebarlabel: Personal Module
sidebarposition: 300
tags: [rsk, rskj, node, rpc, rpc api, node operators, rootstock]
```

description: "The JSON-RPC methods supported by Rootstock nodes."

---

## personallockAccount

Locks the given account.

### Parameters

1. DATA, 20 Bytes - address.

### Example Parameters

```
js
params: ['0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b']
```

### Returns

Boolean - true if the account was successfully locked, otherwise false.

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"personallockAccount","params":["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

## personalunlockAccount

Unlocks the given account for a given amount of time.

### Parameters

1. DATA, 20 Bytes - address.
2. String - The passphrase of the account.
3. QUANTITY - (optional, default: 1800000 milliseconds) The duration for the account to remain unlocked.

### Example Parameters

```
js
params: [
  "0x11f4d0A3c12e86B4b5F39B213F7E19D048276DAe",
  "test passphrase!",
```

```
"927C0" // 600000 milliseconds (10 min)
]
```

## Returns

Boolean - true if the account was successfully unlocked, otherwise false.

## Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"personalunlockAccount","params":["0x11f4d0A3c12e86B4b5F39B213F7E19D
048276DAe", "test passphrase!",
"927C0"],"id":73}'
```

## // Result

```
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

## personalsendTransaction

Sends a transaction over the management API.

## Parameters

1. Object - The transaction call object
  - from: DATA, 20 Bytes - (optional) The address the transaction is sent from.
  - to: DATA, 20 Bytes - The address the transaction is directed to.
  - gas: QUANTITY - (optional) Integer of the gas provided for the transaction execution. ethcall consumes zero gas, but this parameter may be needed by some executions.
  - gasPrice: QUANTITY - (optional) Integer of the gasPrice used for each paid gas.
  - value: QUANTITY - (optional) Integer of the value sent with this transaction.
  - data: DATA - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation.
2. String - The passphrase of the current account.

## Example Parameters

```
js
params: [{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
  "gas": "0x76c0", // 30400
  "gasPrice": "0x9184e72a000", // 10000000000000
  "value": "0x9184e72a", // 2441406250
  "data":
"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"
```



```
}, "test passphrase!"]
```

## Returns

DATA - The transaction hash.

### Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"personalsendTransaction","params":[{"see above}], "id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

## personalimportRawKey

Imports the given private key into the key store, encrypting it with the passphrase.

## Parameters

1. DATA - An unencrypted private key (hex string).
2. String - The passphrase of the current account.

## Example Parameters

```
js
params: [
  "0xcd3376bb711cb332ee3fb2ca04c6a8b9f70c316fcdf7a1f44ef4c7999483295e",
  "test passphrase!"
]
```

## Returns

DATA - The address of the new account.

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"personalimportRawKey","params":["0xcd3376bb711cb332ee3fb2ca04c6a8b9f70c316fcdf7a1f44ef4c7999483295e",
  "test passphrase!"], "id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
```

```
"result": "0x8f337bf484b2fc75e4b0436645dcc226ee2ac531"
}
```

## personaldumpRawKey

Returns an hexadecimal representation of the private key of the given address.

### Parameters

1. DATA, 20 Bytes - The address of the account, said account must be unlocked.

### Example Parameters

```
js
params: ["0xcd3376bb711cb332ee3fb2ca04c6a8b9f70c316fcd7a1f44ef4c7999483295e"]
```

### Returns

DATA - A hexadecimal representation of the account's key.

### Example

```
js
// Request
curl -X POST --data
'{"jsonrpc":"2.0","method":"personaldumpRawKey","params":["0xcd3376bb711cb332ee3fb2ca04c6a8b9f70c316fcd7a1f44ef4c7999483295e"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "777ebfc1e2b6930b09647e7a2273b3e53f759c751c0056695af466783db3642f"
}
```

## personalnewAccount

Creates a new account.

### Parameters

1. String - The passphrase to encrypt this account with.

### Example Parameters

```
js
params: ["test passphrase!"]
```

### Returns

DATA - The address of the newly created account.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"personalnewAccount","params":["test
passphrase!"],"id":1}'
```

```
// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x8f337bf484b2fc75e4b0436645dcc226ee2ac531"
}
```

## personalnewAccountWithSeed

Creates a new account using a seed phrase.

### Parameters

1. String - The seed phrase to encrypt this account with.

### Example Parameters

```
js
params: ["seed"]
```

### Returns

DATA - The address of the newly created account.

## Example

```
js
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"personalnewAccountWithSeed","params":["seed"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x8f337bf484b2fc75e4b0436645dcc226ee2ac531"
}
```

# 04-configuration-limits.md:

---

title: Configuration of Limits for JSON-RPC Interface

sidebarlabel: Configuration Limits

sidebarposition: 400

tags: [rsk, rskj, node, rpc, rootstock]

description: "The JSON-RPC methods supported by Rootstock nodes."

---

Below are the configuration limits for the following JSON-RPC methods:

> Note: These limits are available from Fingerroot v5.1.0.

### JSON-RPC method ethgetLogs limits

The added configuration in the RSKj client's configuration files allows the control of two limits related to the ethgetLogs JSON-RPC call, which is used to retrieve event logs from smart contracts on the Rootstock blockchain.

#### Maximum blocks to query

The maxBlocksToQuery refers to the maximum number of blocks to query.

This parameter determines the maximum number of blocks the RSKj client will query on the blockchain when executing an ethgetLogs call. By default, this value is disabled, meaning that if no value is specified, the RSKj client will query event logs from all blocks specified in the parameters of the ethgetLogs call. If a limit is defined and the ethgetLogs call exceeds this limit, the query execution will be terminated, and an error code will be returned.

#### Maximum Logs to Return

The maxLogsToReturn refers to the maximum number of logs to return.

This parameter determines the maximum number of event logs that the RSKj client will return in response to an ethgetLogs call. By default, this value is disabled (i.e, set to 0), indicating that the RSKj client will return all event logs that match the search criteria. If the limit is defined and the call exceeds this limit, the query execution will be terminated returning an error code.

:::warning[Warning]

Disabling the limit (maxLogsToReturn = 0) could lead to the inclusion of a large number of logs in the response. However, enabling the limit helps protect the node's resources and prevents malicious usages.

:::

### JSON-RPC Interface Limit

The RSKj client now introduces a new configuration option to limit the maximum size of responses returned by the JSON-RPC interface.

#### Maximum JSON-RPC response size

The maxResponseSize refers to the maximum JSON-RPC response size.

This parameter allows you to set a limit on the maximum size of responses returned by the JSON-RPC interface. The response size is measured in bytes. By default, this value is disabled with maxResponseSize = 0, meaning that there is no limit imposed on the size of JSON-RPC responses.

:::info[Info]

When `maxResponseSize` is enabled and set to a specific value, the JSON-RPC interface will truncate or reject responses that exceed the specified size limit.

...

## Configuration Usage

By adding these configurations to the RSKj client's configuration files, you can manage the limits according to your specific needs and requirements. With the added functionality of limiting the JSON-RPC response size, you can control the amount of data returned by the interface to avoid excessive resource consumption.

It is recommended to set reasonable values for these limits, considering the network's load and the available resources for the RSKj client.

:::info[Info]

The configuration may vary based on the version of the RSKj client you are using and how it integrates with other components of your system. Always refer to the official RSKj documentation and relevant specifications for more precise details about the configuration.

...

# 05-management-api-methods.md:

```
---
title: Management API Methods
sidebarlabel: Management API Methods
sidebarposition: 500
tags: [rsk, rskj, node, rpc, rpc api, node operators, rootstock]
description: "The JSON-RPC methods supported by Rootstock nodes."
---
```

| Method                   | Supported | Comments |
|--------------------------|-----------|----------|
| adminaddPeer             | NO        |          |
| admindatadir             | NO        |          |
| adminnodeInfo            | NO        |          |
| adminpeers               | NO        |          |
| adminsetSolc             | NO        |          |
| adminstartRPC            | NO        |          |
| adminstartWS             | NO        |          |
| adminstopRPC             | NO        |          |
| adminstopWS              | NO        |          |
| debugbacktraceAt         | NO        |          |
| debugblockProfile        | NO        |          |
| debugcpuProfile          | NO        |          |
| debugdumpBlock           | NO        |          |
| debuggetBlockRlp         | NO        |          |
| debuggoTrace             | NO        |          |
| debugmemStats            | NO        |          |
| debugseedHash            | NO        |          |
| debugsetHead             | NO        |          |
| debugsetBlockProfileRate | NO        |          |

|                         |           |   |
|-------------------------|-----------|---|
| debugstacks             | NO        |   |
| debugstartCPUProfile    | NO        |   |
| debugstartGoTrace       | NO        |   |
| debugstopGoTrace        | NO        |   |
| debugtraceBlock         | NO        |   |
| debugtraceBlockByNumber | NO        |   |
| debugtraceBlockByHash   | NO        |   |
| debugtraceBlockFromFile | NO        |   |
| debugtraceTransaction   | YES       |   |
| debugvmodule            | NO        |   |
| debugwriteBlockProfile  | NO        |   |
| debugwriteMemProfile    | NO        |   |
| minersetExtra           | NO        |   |
| minersetGasPrice        | NO        |   |
| minerstart              | NO        |   |
| minerstop               | NO        |   |
| minersetEtherBase       | NO        |   |
| personalimportRawKey    | YES       |   |
| personallistAccounts    | YES       |   |
| personallockAccount     | YES       |   |
| personalnewAccount      | YES       |   |
| personalunlockAccount   | YES       |   |
| personalsendTransaction | YES       |   |
| personalsign            | NO        |   |
| personalecRecover       | NO        |   |
| traceblock              | PARTIALLY | Option "pending" not supported. It also supports block hash as parameter. |
| tracetransaction        | YES       |   |
| txpoolcontent           | YES       |   |
| txpoolinspect           | YES       |   |
| txpoolstatus            | YES       |   |

## RPC PUB SUB methods

|                |           |   |
|----------------|-----------|---|
| Method         | Supported | Comments  |
| -----          | -----     | -----   |
| ethsubscribe   | PARTIALLY | Only options "newHeads" and "logs" are supported. |
| ethunsubscribe | YES       |   |

## RPC SPV methods

|                                     |           |  |
|-------------------------------------|-----------|--|
| Method                              | Supported | Comments   |
| -----                               | -----     | -----  |
| rskgetRawBlockHeaderByNumber        | YES       | Obtains the RLP encoded block header used for SPV, if this is hashed using Keccak256 it gives the block hash. This function takes the block number (in hexa) or the string "latest", "pending", "genesis". |
| rskgetRawBlockHeaderByHash          | YES       | Obtains the RLP encoded block header used for SPV, if this is hashed using Keccak256 it gives the block hash. This function takes the block hash as parameter.   |
| rskgetRawTransactionReceiptByHash   | YES       | Obtains the RLP encoded Transaction Receipt, if this is hashed using Keccak256 it gives the transaction receipt hash. This function takes the transaction hash as parameter.                               |
| rskgetTransactionReceiptNodesByHash | YES       | Obtains an array of nodes of the transactions receipt Trie. This is used to hash up to the transaction receipt root. This function takes the block hash and transaction hash as parameters.                |

# reproducible-build.md:

---

sidebarlabel: Reproducible Build

sidebarposition: 7

title: Gradle building

tags: [rsk, node, compile, reproducible, checksum, rootstock]

description: "A deterministic build process used to build Rootstock node JAR file. Provides a way to be reasonable sure that the JAR is built from GitHub RSKj repository. Makes sure that the same tested dependencies are used and statically built into the executable."

---

Setup instructions for gradle build in docker container

This is a deterministic build process used to build Rootstock node JAR file. It provides a way to be reasonably sure that the JAR is built from the GitHub RSKj repository. It also makes sure that the same tested dependencies are used and statically built into the executable.

It's highly recommended to follow the steps by yourself to avoid contamination of the process.

Install Docker

Depending on your OS, you can install Docker following the official Docker guide:

- Mac
- Windows
- Ubuntu
- CentOS
- Fedora
- Debian
- Others

:::info[Info]

See how to Setup and Run RSKj using Java.

:::

Build Container

Create a Dockerfile to setup the build environment.

<Tabs>

<TabItem value="linux" label="Linux" default>

```
bash
// FROM ubuntu:16.04
apt-get update -y && \
  apt-get install -y git curl gnupg-curl openjdk-8-jdk && \
  rm -rf /var/lib/apt/lists/ && \
  apt-get autoremove -y && \
  apt-get clean
gpg --keyserver https://secchannel.rsk.co/release.asc --recv-keys
1A92D8942171AFA951A857365DECF4415E3B8FA4
gpg --finger 1A92D8942171AFA951A857365DECF4415E3B8FA4
git clone --single-branch --depth 1 --branch ARROWHEAD-6.3.1 https://github.com/rsksmart/rskj.git
/code/rskj
git clone https://github.com/rsksmart/reproducible-builds
```

```

CP /Users/${USER}/reproducible-builds/rskj/6.3.1-arrowhead/Dockerfile /Users/${USER}/code/rskj
WORKDIR /code/rskj
gpg --verify SHA256SUMS.asc
sha256sum --check SHA256SUMS.asc
./configure.sh
./gradlew clean build -x test
</TabItem>
<TabItem value="mac" label="Mac OSX">
  bash
  brew update && \
  brew install git gnupg openjdk@8 && \
  rm -rf /var/lib/apt/lists/ && \
  brew autoremove && \
  brew cleanup
  gpg --keyserver https://secchannel.rsk.co/release.asc --recv-keys
1A92D8942171AFA951A857365DECF4415E3B8FA4
  gpg --finger 1A92D8942171AFA951A857365DECF4415E3B8FA4
  git clone --single-branch --depth 1 --branch ARROWHEAD-6.3.1 https://github.com/rsksmart/rskj.git
./code/rskj
  git clone https://github.com/rsksmart/reproducible-builds
  CP /Users/${USER}/reproducible-builds/rskj/6.3.1-arrowhead/Dockerfile /Users/${USER}/code/rskj
  cd /code/rskj
  gpg --verify SHA256SUMS.asc
  sha256sum --check SHA256SUMS.asc
  ./configure.sh
  ./gradlew clean build -x test
</TabItem>
</Tabs>

```

Response:

You should get the following as the final response,  
after running the above steps:

```

bash
BUILD SUCCESSFUL in 55s
14 actionable tasks: 13 executed, 1 up-to-date

```

If you get the error: zsh: command not found: sha256sum  
> Run the command `brew install coreutils`

If you are not familiar with Docker or the Dockerfile format: what this does is use the Ubuntu 16.04 base image and install git, curl, gnupg-curl and openjdk-8-jdk, required for building the Rootstock node.

Run build

To create a reproducible build, run the command below in the same directory:

```

bash
docker build -t rskj/6.3.1-arrowhead .

```



:::danger[Error]

if you run into any problems, ensure you're running the commands on the right folder and also ensure docker daemon is running is updated to the recent version.

:::

This may take several minutes to complete. What is done is:

- Place in the RSKj repository root because we need Gradle and the project.
- Runs the secure chain verification process.
- Compile a reproducible RSKj node.
- ./gradlew clean build -x test builds without running tests.

## Verify Build

The last step of the build prints the sha256sum of the files, to obtain SHA-256 checksums, run the following command in the same directory as shown above:

```
bash
docker run --rm rskj/6.3.1-arrowhead sh -c 'sha256sum | grep -v javadoc.jar'
```

## Check Results

After running the build process, a JAR file will be created in /rskj/rskj-core/build/libs/, into the docker container.

You can check the SHA256 sum of the result file and compare it to the one published by Rootstock for that version.

```
bash
27f088e5c7535974203bc77711a1e9bbaa258cd7e5d69cd368d8ca5529b38115
rskj-core-6.3.1-ARROWHEAD-all.jar
101e35cbaef4c3997432e561417f208f93e5121c52be506bcf2bf22357855bb8
rskj-core-6.3.1-ARROWHEAD-sources.jar
9e29606d4bd71dd0458ec3eaa61ab4003d6aecad4bf5a5a40cc32d8d6535de75
rskj-core-6.3.1-ARROWHEAD.jar
bb31e2e2b14f9a15f6c7ca2595a7817860334f38f5d7c6e146306275834b7564
rskj-core-6.3.1-ARROWHEAD.module
ad539153c8bb8d09307c4a77c9a8af062531241f6a5155c2cd21f0892da27b18
rskj-core-6.3.1-ARROWHEAD.pom
```

For SHA256 sum of older versions check the releases page.

If you check inside the JAR file, you will find that the dates of the files are the same as the version commit you are using.

## More Resources

=====

Install Rootstock Node  
See Reproducible builds

Check out the latest rskj releases

# requirements.md:

```
---
sidebarlabel: Hardware Requirements
sidebarposition: 3
title: Minimum Hardware Requirements to Run a Rootstock Node
tags: [hardware, specs, requirements]
description: "Minimum hardware requirements for Rootstock."
---
```

These are the minimum requirements that must be met to run an Rootstock node (Mainnet and Testnet):

- 2 cores
- 8 GB RAM
- 128 GB storage
- OS x64

> RSKj allows you to run a Rootstock node, crucial for local development and testing. It supports connections to Regtest (local), Testnet (testing), and Mainnet (production). Visit the official RSKj GitHub repository to download the latest stable release.

# security-chain.md:

```
---
sidebarlabel: Security Chain
sidebarposition: 8
title: Verify security chain of RSKj source code
tags: [rsk, rskj, node, security, node operators, reproducible builds, verification]
description: "All the different ways that you can verify RSKj: Release signing key, fingerprint of the public key, SHA256SUMS.asc, binary dependencies, secure environment script"
---
```

Verify authenticity of RSKj source code and its binary dependencies

The authenticity of the source code must be verified by checking the signature of the release tags in the official Git repository. See Reproducible builds. The authenticity of the binary dependencies is verified by Gradle after following the steps below to install the necessary plugins.

Download Rootstock Release Signing Key public key

For Linux based OS (Ubuntu for example), it's recommended to install curl and gnupg-curl in order to download the key through HTTPS.

We recommend using GPG v1 to download the public key because GPG v2 encounters problems when connecting to HTTPS key servers. You can also download the key using curl, wget or a web browser but always check the fingerprint before importing it.

```
bash
gpg --keyserver https://secchannel.rsk.co/SUPPORT.asc --recv-keys A6DBEAC640C5A14B
```

You should see the output below:

text  
Output:  
gpg: key A6DBEAC640C5A14B: "IOV Labs Support <support@iovlabs.org>" imported  
gpg: Total number processed: 1  
gpg: imported: 1 (RSA: 1)

Verify the fingerprint of the public key

bash  
gpg --finger A6DBEAC640C5A14B

The output should look like this:

text  
Output:  
pub rsa4096 2022-05-11 [C]  
1DC9 1579 9132 3D23 FD37 BAA7 A6DB EAC6 40C5 A14B  
uid [ unknown] IOV Labs Support <support@iovlabs.org>  
sub rsa4096 2022-05-11 [S]  
sub rsa4096 2022-05-11 [E]

Verify the signature of SHA256SUMS.asc

The fileSHA256SUMS.asc is signed with Rootstock public key and includes SHA256 hashes of the files necessary to start the build process.

Note: Ensure to cd into the rskj directory before executing the commands below.

bash  
gpg --verify SHA256SUMS.asc

The output should look like this:

text  
Output:  
gpg: Signature made Wed May 11 10:50:48 2022 -03  
gpg: using RSA key 1F1AA750373B90D9792DC3217997999EEA3A9079  
gpg: Good signature from "IOV Labs Support <support@iovlabs.org>" [unknown]  
gpg: WARNING: This key is not certified with a trusted signature!  
gpg: There is no indication that the signature belongs to the owner.  
Primary key fingerprint: 1DC9 1579 9132 3D23 FD37 BAA7 A6DB EAC6 40C5 A14B  
Subkey fingerprint: 1F1A A750 373B 90D9 792D C321 7997 999E EA3A 9079

Note: Learn more about key management [here](#).

Verification of binary dependencies

The authenticity of the script configure.sh is checked using the sha256sum command and the signed SHA256SUM.asc file. The script is used to download and check the authenticity of the Gradle Wrapper

and Gradle Witness plugins. After these plugins are installed, the authenticity of the rest of the binary dependencies is checked by Gradle.

Linux - Windows (bash console)

```
<Tabs>
<TabItem value="linux" label="Linux" default>
  bash
  sha256sum --check SHA256SUMS.asc

</TabItem>
<TabItem value="mac" label="Mac OSX">
  bash
  shasum --check SHA256SUMS.asc

</TabItem>
</Tabs>
```

Run configure script to configure secure environment

```
<Tabs>
<TabItem value="linux" label="Linux, Mac OSX" default>
  bash
  ./configure.sh

</TabItem>
</Tabs>
```

# docker.md:

```
---
sidebarlabel: Setup node on Docker
sidebarposition: 200
title: Setup node on Docker
tags: [docker, rootstock, desktop, macOS, rskj, windows, install, rsk, node, how-to, network, requirements,
mainnet, testnet, regtest]
description: "Install RSKj using Docker."
---
```

Before installing Docker, ensure your system meets the minimum requirements before installing the Rootstock node (RSKj).

If you already have docker installed. See how to Install the RSKj node using Docker.

Install Docker Desktop Client

Docker Desktop provides an easy and fast way for running containerized applications on various operating systems.

```
<Tabs>
<TabItem value="mac" label="Mac OSX, Windows" default>
  - Download and install
  - Start the Docker Desktop client
  - Login with a Docker Hub free account
</TabItem>
```

```
<TabItem value="linux" label="Linux">
- Install Docker Engine Community
- Note that you will need to use sudo for all docker commands, by default. To avoid this additional steps
are required.
</TabItem>
</Tabs>
```

...tip[For Mac M1 / M2 (Apple Chips) using x86 based software]

- Ensure you have Rosetta installed. This is typically pre-installed on recent macOS versions.
- Download an x86 JDK build, such as Azul Zulu 11 (x86), to ensure compatibility with x86 based software.

...

Ensure that docker is running by running the following command - it should run without any errors.

```
shell
docker ps
```

You should see the following response:

```
text
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

More information about Docker install [here](#).

## Install RSKj Using Docker

To install a RSKj node using Docker, visit the Docker Hub for installation instructions or use the Reproducible Build.

## Logging in RSKj

By default, logs are exclusively directed to a single file. However, if you wish to enable the logging output to STDOUT, you can specify this system property via the command line using `-Dlogging.stdout=<LOGLEVEL>`. That command should look something like this:

```
java
java -Dlogging.stdout=INFO -cp <classpath> co.rsk.Start --reset --<RSK network>
```

Regarding the RSKj Docker containers, logs are printed to STDOUT by default, making it easy to view the logs while the container is running. In order to modify this, you can run the Docker container with the environment variable set to a different LOGLEVEL (For example, DEBUG log level). That command should follow this structure:

```
bash
docker run -e RSKJLOGPROPS=DEBUG <container-name>
```

# index.md:

---

sidebarlabel: Node Installation

sidebarposition: 4

title: RSKj Node Installation

tags: [rsk, rootstock, rskj, node, developers, merged mining]

description: "Install RSKj on different OS."

---

Rootstock nodes can be installed on Ubuntu OS, Windows, Docker and Java. Here, we provide step-by-step instructions for all supported dev environments. Depending on your network performance, it usually takes 10 to 15 mins to setup a working node on Mainnet.

> Ensure that your system meets the minimum requirements before installing the Rootstock nodes.

> Looking to test your dApps on testnet in minutes before deploying to mainnet? Use the RPC API

Supported Environments

| Environments | How to Install |

| --- | --- |

| Ubuntu Package | Visit Setup Node on Ubuntu for instructions on installing Rootstock Node on ubuntu systems |

| Java - Fat / Uber JAR | Visit Setup Node using JAR for instructions on installing Rootstock Node on any system with Fat JAR or Uber JAR. |

| Docker | Visit the Docker Hub for instructions on installing Rootstock Node as a docker container on any system. |

| Windows | Visit Setup Node on Windows OS |

# java.md:

---

sidebarlabel: Setup node using Java

sidebarposition: 100

title: Setup node using Java

tags: [java, install, rootstock, rskj, node, how-to, network, requirements, mainnet, jar]

description: "Install RSKj using Java."

---

To setup a Rootstock node using Java, you need to:

- Ensure your system meets the minimum requirements for installing the Rootstock node.
- Install Java 8 JDK.

:::tip[For Mac M1 / M2 (Apple Chips) using x86 based software]

- Ensure you have Rosetta installed. This is typically pre-installed on recent macOS versions.
- Download an x86 JDK build, such as Azul Zulu 11 (x86), to ensure compatibility with x86 based software.

...

Video walkthrough

<Video url="https://www.youtube-nocookie.com/embed/TxpS6WhxUiU?ccloadpolicy=1" thumbnail="/img/thumbnails/install-node-java-thumbnail.png" />

Install the node using a JAR file

## Download and Setup

1. Download the JAR: Download the Fat JAR or Uber JAR from RSKj releases, or compile it reproducibly.

2. Create Directory: Create a directory for the node.

```
jsx
mkdir rskj-node-jar
cd /rskj-node-jar
```

3. Move the JAR: Move or copy the just downloaded jar file to your directory.

```
jsx
mv /Downloads/rskj-core-6.3.1-ARROWHEAD-all.jar SHA256SUMS.asc /Users/{user}/rskj-node-jar/
```

## Configuration

1. Create Config Directory: Create another directory inside /rskj-node-jar/config

```
jsx
mkdir config
```

2. Download Config File: Get node.conf from [here](#).

3. Move Config File: Move the node.conf file to the config directory.

## Run the Node

```
<Tabs>
<TabItem value="1" label="Linux, Mac OSX" default>
  shell
  java -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>
<TabItem value="2" label="Windows">
  shell
  java -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>
</Tabs>
```

:::tip[Tip]

Replace <PATH-TO-THE-RSKJ-JAR> with the actual path to your JAR file. For example, C:/RskjCode/rskj-core-6.3.1-ARROWHEAD-all.jar.

...

## Using Import Sync

Instead of the default synchronization, you can use import sync to import a pre-synchronized database from a trusted origin, which is significantly faster.

```
<Tabs>
<TabItem value="3" label="Linux, Mac OSX" default>
  shell
  java -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start --import
```

```

</TabItem>
<TabItem value="4" label="Windows">
  shell
  java -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start --import
</TabItem>
</Tabs>

```

## Resolving memory issues

Memory Issues? If you encounter memory errors and meet the minimum hardware requirements, consider using -Xmx4G flag to allocate more memory as shown below:

```

<Tabs>
<TabItem value="5" label="Linux, Mac OSX" default>
  shell
  java -Xmx4G -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start --import
</TabItem>
<TabItem value="6" label="Windows">
  shell
  C:\> java -Xmx4G -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start --import
</TabItem>
</Tabs>

```

:::tip[Tip]

Replace <PATH-TO-THE-RSKJ-JAR> with your JAR file path. For configuration details, see database.import setting.

...

## Check the RPC

:::info[Info]

After starting the node, if there's no output, this means it's running correctly.

...

1. To confirm, open a new console tab (it is important you do not close this tab or interrupt the process) and test the node's RPC server. A sample cURL request:

```

<Tabs>
<TabItem value="7" label="Linux, Mac OSX" default>
  shell
  curl http://localhost:4444 -s -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3clientVersion","params":[],"id":67}'
</TabItem>
<TabItem value="8" label="Windows">
  shell
  curl http://localhost:4444 -s -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3clientVersion","params":[],"id":67}'
</TabItem>

```



</Tabs>

Output:

```
shell
{"jsonrpc":"2.0","id":67,"result":"RskJ/6.3.1/Mac OS X/Java1.8/ARROWHEAD-202f1c5"}
```

2. To check the block number:

<Tabs>

<TabItem value="9" label="Linux, Mac OSX" default>

shell

```
curl -X POST http://localhost:4444/ -H "Content-Type: application/json" --data '{"jsonrpc":"2.0",
"method":"ethblockNumber","params":[],"id":1}'
```

</TabItem>

<TabItem value="10" label="Windows">

windows-command-prompt

```
curl -X POST http://localhost:4444/ -H "Content-Type: application/json" --data '{"jsonrpc":"2.0",
"method":"ethblockNumber","params":[],"id":1}'
```

</TabItem>

</Tabs>

Output:

```
jsx
{"jsonrpc":"2.0","id":1,"result":"0x0"}
```

:::success[Success]

Now, you have successfully setup a Rootstock node using the jar file.  
The result property represents the latest synced block in hexadecimal.

...

## Switching networks

To change networks on the RSKj node, use the following commands:

- Mainnet

bash

```
java -cp <PATH-TO-THE-RSKJ-FATJAR> co.rsk.Start
```

- Testnet

bash

```
java -cp <PATH-TO-THE-RSKJ-FATJAR> co.rsk.Start --testnet
```

- Regtest

bash

```
java -cp <PATH-TO-THE-RSKJ-FATJAR> co.rsk.Start --regtest
```

:::tip[Tip]

Replace <PATH-TO-THE-RSKJ-FATJAR> with the actual path to your jar file. For example:  
C:/RskjCode/rskj-core-6.3.1-ARROWHEAD-all.jar.

...

# ubuntu.md:

---

sidebarlabel: Setup node on Ubuntu

sidebarposition: 300

title: Setup node on Ubuntu

tags: [ubuntu, install, rsk, rskj, node, how-to, network, requirements, mainnet]

description: "Install RSKj on Ubuntu."

---

Make sure your system meets the minimum requirements before installing the Rootstock nodes.

Video

```
<div class="video-container">
  <iframe width="949" height="534"
src="https://www.youtube-nocookie.com/embed/eW9UF2aJQgs?ccloadpolicy=1" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
</div>
```

### Install via Ubuntu Package Manager

The easiest way to install and run a Rootstock node on Ubuntu is to do it through Ubuntu Package Manager.

Type the commands below to install RSKj on Ubuntu using our PPAs for Ubuntu.

The installed repo public key Fingerprint is 5EED 9995 C84A 49BC 02D4 F507 DF10 691F 518C 7BEA. Also, the public key could be found in document Ubuntu Key Server.

```
shell
$ sudo add-apt-repository ppa:rsksmart/rskj
$ sudo apt-get update
$ sudo apt-get install rskj
```

During the installation, you will be asked to accept the terms and confirm the network.

</img>

Choose Yes and Enter to accept the license to continue

</img>

Choose mainnet and press Enter to continue

### Install via Direct Downloads

You can also download the RSKj Ubuntu Package for the latest RSKj release ARROWHEAD 6.0.0 and install it with the dpkg command. Follow this download link to download the matching package for your ubuntu system.

```
shell
```

first install openjdk-8-jre or oracle-java8-installer  
sudo apt-get install openjdk-8-jre

download the RSKj package and find the file rskj2.0.1yourUbuntuVersionNameamd64.deb

run this command in the same directory as the deb file above  
dpkg -i rskj2.0.1yourUbuntuVersionNameamd64.deb

We recommend that you check that the SHA256 hash of the downloaded package file matches, prior to installation:

```
rskj2.0.1bionicamd64.deb: b2f0f30ac597e56afc3269318bbdc0a5186f7c3f7d23a795cf2305d7c7b12638
rskj2.0.1bionici386.deb: 3ca031ee133691ed86bb078827e8b2d82600d7bbd76194358289bbc02385d971
rskj2.0.1trustyamd64.deb: 4c56d8d0ed0efc277afe341aa7026e87f47047ff69bd6dd99296c5ecab1fa550
rskj2.0.1trustyi386.deb: e5cb7b72e4aff8be4cbcd5d1e757e1fda463f1565154ae05395fcf1796ecf9fb
rskj2.0.1xenialamd64.deb: 70c245388a7f521b96905bf49b93e38f58c54970e4e4effa36d7f2b0a2aa8ef4
rskj2.0.1xeniali386.deb: f067301454eb5976bbf00052ccd6523b1ee61f6aeb33ef4ea6fcb07ff0328668
```

After installation

By default, the node connects to Mainnet. To change the network choice (Mainnet/ Testnet/ Regtest), refer to the instructions in switching networks. To change configurations for the node, refer to the instructions in Rootstock Node Configuration.

The installer will configure your node in the following paths:

```
/etc/rsk: the directory where the config files will be placed.
/usr/share/rsk: the directory where the RSKj JAR will be placed.
/var/lib/rsk/database: the directory where the database will be stored.
/var/log/rsk: the directory where the logs will be stored.
```

</img>

Start/Stop the Node

After installation, you can use the following commands to manage your node.

To start the node:

```
shell
sudo service rsk start
```

To stop the node:

```
shell
sudo service rsk stop
```

To restart the node:

```
shell
sudo service rsk restart
```

To check the status of the node service:

```
shell
sudo service rsk status
```

</img>

# 02-cli.md:

```
---
sidebarlabel: CLI
sidebarposition: 200
title: Command Line Interface
tags: [rsk, rskj, node, cli]
description: "Command Line Interface for Rootstock (RSK)"
---
```

Command line (CLI) arguments are arguments specified after the RSK start class. See Config Options

> New entry points (Java classes with static public main method) have been included from the RSK Hop Release v4.3.0.

The CLI arguments have two forms; the parameter and the flag:

- Parameter
  - Has a name and an associated value, separated by a space
  - Starts with a dash
- Flag
  - It's a single text, without spaces
  - Starts with a double dash

Find below a list of CLI flags and parameters available:

## Parameters and Flags

### Network related

The following CLI flags determine which network the Rootstock (RSK) node will connect to.

- --main:  
This indicates that the configuration for the Rootstock Mainnet (public network) should be used.
- --testnet:  
This indicates that the configuration for the Rootstock Testnet (public network) should be used.
- --regtest:  
This indicates that the configuration for the Rootstock Regtest (localhost network) should be used.
  - Example: java -cp rsk-core-<VERSION>.jar co.rsk.start --regtest

- > - Only one of these three CLI flags should be specified.
- > - When none of these are specified, Rootstock Mainnet is used by default.

### Database related

The Rootstock (RSK) node stores transactions, blocks,

and other blockchain state on disk.  
This is known as the Blockchain Database.

- --reset:

This indicates that the block database should be erased, and should start from scratch, i.e. from genesis block.

This is typically expected to be used when connecting to Rootstock Regtest, or in select debugging scenarios.

It is also used when switching between different databases, e.g. between leveldb and rocksdb.

- --import:

This indicates that the block database should be imported from an external source.

This is typically expected to be used when connecting to Rootstock Testnet or Rootstock Mainnet, and when a reduction in "initial sync time" is desired.

It is also used when switching between different databases, e.g. between leveldb and rocksdb.

## Configuration related

- --verify-config:

This indicates that the configuration file used by this run of the Rootstock node should be validated. By default this step is always performed.

- --print-system-info:

This indicates that the system information of the computer that the Rootstock node is running on should be output. By default, this is always output.

- --skip-java-check:

This indicates that the detection of the version of the Java Virtual Machine that the Rootstock node is running in is supported. By default, this check is always performed, to ensure that the Rootstock node is running in a compatible environment.

- -base-path:

This specifies the value of database.dir, where the blockchain database is stored.

> Example: `java -cp rsk-core-<VERSION>.jar co.rsk.start -base-path home/rsk/data`

- -rpccors

This specifies the value of rpc.providers.web.cors to control cors configuration.

> Example: `java -cp rsk-core-<VERSION>.jar co.rsk.start -rpccors`

## Command Line Tools

### Database related commands

#### ExportState

The ExportState command is a tool for exporting the state at a specific block in the Rootstock blockchain to a file.

#### Usage:

- `java -cp rsk.jar co.rsk.cli.tools.ExportState -b <blocknumber> -f <filepath>`

#### Options:

-b, --block: The block number to export the state from.

-f, --file: The path to a file to export the state to.

Example:

In this example, the state information of block 2000 will be exported to the file “test.txt” on the regtest network.

```
- java -cp rsk.jar co.rsk.cli.tools.ExportState -b 2000 -f test.txt --regtest
```

Output:

shell

```
INFO [clitool] [main] ExportState started
INFO [o.e.d.CacheSnapshotHandler] [main] Loaded 194912 cache entries from 'unitrie/rskcache'
INFO [clitool] [main] ExportState finished
INFO [o.e.d.CacheSnapshotHandler] [main] Saved 194912 cache entries in 'unitrie/rskcache'
```

The “test.txt” should look like this:

text

```
5c0700caa04b0e28aa38d3d4a74a560332c38111cb1ac2292a89512d009658d2a7ed7d2cecc372b5c2579
9434b1cc5e49d795fc371db88e3d0c3635e273fc3c496b897fdc60c
4ce5c4861124fac10fdda43f62df4cf8137136e4c654305a8e9e3572f76b46c9fd9ce59676
...etc
```

ExportBlocks

The ExportBlocks command is a tool for exporting blocks from a specific block range to a file. The tool retrieves the block range and exports each block in the range to a specified file.

Usage:

```
java
java -cp rsk.jar co.rsk.cli.tools.ExportBlocks -fb <fromblocknumber> -tb <toblocknumber> -f <filepath>
```

Options:

- > -fb, --fromBlock: The block number to start exporting from.
- > -tb, --toBlock: The block number to stop exporting.

Example:

In this example, the blocks from block 2000 to block 3000 will be exported to the specified file.

java

```
java -cp rsk.jar co.rsk.cli.tools.ExportBlocks -fb 2000 -tb 3000 -f test-blocks.txt
```

Output:

shell

```
2023-04-24-16:23:25.0897 INFO [clitool] [main] ExportBlocks started
2023-04-24-16:23:26.0373 INFO [clitool] [main] ExportBlocks finished
```

Blocks.txt should show the following:

text

```
50,d6c7a4388337d931d9478e742c34c276cefe0976e12b2f7077bd6664b6ecc163,0629fdd6...
51,d4a9091304e64008f06c395b4f26da1c710bdd83f30f0d15666826b57c9b7a1e,0651fde4b...
...
99,92c333550ada4b2588d957155f1aa130ef0092b9499d575a3e4034e1f3f20926,0f271cc73...
100,b445214290eb98e1e066713aa8a76ff4282c7890d232471d62be5932d21f25b8,0f57a...
```

Configuration related commands

StartBootstrap

The StartBootcamp command starts a bootstrap node with one service, which only participates in the peer discovery protocol.

Example:

```
java -cp rsk.jar co.rsk.cli.tools.StartBootstrap
```

Output:

shell

```
2023-04-24-15:51:14.0793 INFO [fullnoderunner] [main] Starting RSK
2023-04-24-15:51:14.0794 INFO [fullnoderunner] [main] Running orchid-testnet.json,
core version: 5.1.0-SNAPSHOT
....
```

RewindBlocks

The RewindBlocks command is used to rewind the Rootstock blockchain to a specific block. It can also be used to find and print the minimum inconsistent block (a block with missing state in the states database).

Example:

shell

```
java -cp rsk.jar co.rsk.cli.tools.RewindBlocks [-b <BLOCKNUMBER>] [-fmi] (or) [-rbc]
```

Options:

- b, --block=BLOCKNUMBERTOREWINDTO block number to rewind blocks to.
- fmi, --findMinInconsistentBlock flag to find a min inconsistent block. The default value is false.
- rbc, --rewindToBestConsistentBlock flag to rewind to the best consistent block. The default value is false.

> - An inconsistent block is the one with missing state in the states db (this can happen in case of

improper node shutdown).

> - fmi option can be used for finding a minimum inconsistent block number and printing it to stdout. It will print -1, if no such block is found.

> - rbc option does two things: it looks for a minimum inconsistent block and, if there's such, rewinds blocks from top one till the found one.

> - Note that all three options are mutually exclusive, you can specify only one per call.

Example:

```
java -cp rsk.jar co.rsk.cli.tools.RewindBlocks -b 2000 --regtest
```

Output:

shell

```
INFO [clitool] [main] RewindBlocks started
INFO [clitool] [main] Highest block number stored in db: 49703
INFO [clitool] [main] Block number to rewind to: 2000
INFO [clitool] [main] Rewinding...
INFO [clitool] [main] Done
INFO [clitool] [main] New highest block number stored in db: 2000
INFO [clitool] [main] RewindBlocks finished
```

## DbMigrate

The DbMigrate command is a tool for migrating between different databases such as leveldb and rocksdb.

Usage:

```
java -cp rsk.jar co.rsk.cli.tools.DbMigrate -t <targetdatabase>
```

Options:

- -t, --targetDb: The target database to migrate to. ("leveldb" or "rocksdb").

Example:

In this example, the current database will be migrated from leveldb to rocksdb.

```
java -cp rsk.jar co.rsk.cli.tools.DbMigrate -t rocksdb
```

Output:

shell

```
INFO [clitool] [main] DbMigrate started
INFO [clitool] [main] DbMigrate finished
```

:::tip[Tip]

If the target database is the same as the one working on the node, the node will throw an error: Cannot migrate to the same database, current db is XDB and target db is XDB.

:::



## Dev-related commands

### ShowStateInfo

The ShowStateInfo command is a tool for displaying state information of a specific block in the Rootstock blockchain.

#### Usage:

```
- java -cp rsk.jar co.rsk.cli.tools.ShowStateInfo -b <blocknumber>
```

#### Options:

- -b, --block: The block number or "best" to show the state info.

#### Example:

```
- java -cp rsk.jar co.rsk.cli.tools.ShowStateInfo -b 20000
```

> In this example, the state information of block 20000 will be displayed.

#### Output:

shell

```
INFO [clitool] [main] ShowStateInfo started
INFO [clitool] [main] Block number: 20000
INFO [clitool] [main] Block hash:
53fe6e9269d26a38d15f368a3b8b647ae6b66e4fe27bd6bd6ee5f4b675129753
INFO [clitool] [main] Block parent hash:
6f49a755c9d6b74d25e15787232887c9dba8e713a8455d9beed97b08bf900b17
INFO [clitool] [main] Block root hash:
587e0645e09d60af77ee04591ac843af14c99f6c498713aeb565f25f2d419cd0
INFO [clitool] [main] Trie nodes: 53
INFO [clitool] [main] Trie long values: 0
INFO [clitool] [main] Trie MB: 0.002902984619140625
INFO [clitool] [main] ShowStateInfo finished
```

### IndexBlooms

The IndexBlooms is a tool for indexing block blooms for a specific block range.

#### Usage:

shell

```
java -cp rsk.jar co.rsk.cli.tools.IndexBlooms [-fb=<fromBlock>] [-tb=<toBlock>]
```

#### Options:

- -fb, --fromBlock=<fromBlock>: From block number (required)

- -tb, --toBlock=<toBlock>: To block number (required)

Example:

In this example we are indexing block blooms from block number 100 to 200.

```
java -cp rsk.jar co.rsk.cli.tools.IndexBlooms -fb=100 -tb=200
```

Output:

```
INFO [c.r.c.t.IndexBlooms] [main] Processed 28% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 45% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 58% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 71% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 81% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 92% of blocks
INFO [c.r.c.t.IndexBlooms] [main] Processed 100% of blocks
```

> The IndexBlooms CLI tool indexes block blooms for a specific block range. The required arguments are fromBlock and toBlock, which specify the block range to be indexed.

### ImportState

The ImportState command is a tool for importing state data from a file into the Rootstock blockchain.

Usage:

```
- java -cp rsk.jar co.rsk.cli.tools.ImportState -f <filepath>
```

Options:

```
- -f, --file: The path to the file to import state from.
```

Example:

In this example, the state data from the file located at test.txt will be imported into the Rootstock blockchain. Keep in mind that we are using the previously generated state in the ExportState example.

```
- java -cp rsk.jar co.rsk.cli.tools.ImportState -f test.txt
```

Output:

text

```
INFO [clitool] [main] ImportState started
INFO [clitool] [main] ImportState finished
```

### ImportBlocks

The ImportBlocks command is a tool for importing blocks from a file into the Rootstock blockchain. The tool reads a file containing blocks, decodes them and saves them to the Rootstock database.

Usage:

```
java -cp rsk.jar co.rsk.cli.tools.ImportBlocks -f <filepath>
```

Options:

- -f, --file: The path to a file to import blocks from.

Example:

In this example, blocks will be imported from the file /path/to/blocksfile.txt.

- java -cp rsk.jar co.rsk.cli.tools.ImportBlocks -f /path/to/blocksfile.txt

Output:

text

```
INFO [clitool] [main] ImportBlocks started
INFO [clitool] [main] ImportBlocks finished
```

ExecuteBlocks

The ExecuteBlocks command is a tool for executing blocks for a specified block range. This command is useful for testing purposes, debugging or for analyzing the behavior of a blockchain in a given range of blocks.

Usage:

- java -cp rsk.jar co.rsk.cli.tools.ExecuteBlocks -fb <fromblocknumber> -tb <toblocknumber>

Options:

- -fb, --fromBlock: The starting block number.  
- -tb, --toBlock: The ending block number.

Example:

In this example, blocks from 100000 to 200000 will be executed on the regtest network.

shell

```
java -cp rsk.jar co.rsk.cli.tools.ExecuteBlocks -fb 100000 -tb 200000 --regtest
```

Output:

shell

```
2023-04-24-16:27:58.0408 INFO [clitool] [main] ExecuteBlocks started
2023-04-24-16:28:02.0881 INFO [clitool] [main] ExecuteBlocks finished
```

ConnectBlocks

The ConnectBlocks command is a tool for connecting blocks to a chain from an external source file.

Usage:

- java -cp rsk.jar co.rsk.cli.tools.ConnectBlocks -f <filepath>

Options:

- -f, --file: The path to the file containing the blocks to connect.

Example:

In this example, the blocks contained in the file located at /path/to/blocks.txt will be connected to the chain.

```
- java -cp rsk.jar co.rsk.cli.tools.ConnectBlocks -f /path/to/blocks.txt
```

GenerateOpenRpcDoc

The GenerateOpenRpcDoc command is a tool for generating an OpenRPC JSON doc file.

Usage:

```
shell
java -cp rsk.jar co.rsk.cli.tools.GenerateOpenRpcDoc -v <rskjversion> -d <workdirpath> -o
<outputfilepath>
```

Options:

- -v, --version: The RSKj version that will be present in the final docs
- -d, --dir: The work directory where the JSON template and individual JSON files are present.
- -o, --outputFile: The destination file containing the final OpenRPC JSON doc.

Example:

In this example, the tool will generate an OpenRPC JSON doc file located at /path/to/output.json.

```
shell
java -cp rsk.jar co.rsk.cli.tools.GenerateOpenRpcDoc -v 1.0.0 -d /path/to/workdir -o /path/to/output.json
```

Output:

```
text
2023-04-24-16:35:00.0617 INFO [c.r.c.t.GenerateOpenRpcDoc] [main] Loading template...
2023-04-24-16:35:00.0620 INFO [c.r.c.t.GenerateOpenRpcDoc] [main] Loading file
doc/rpc/template.json
...
```

JSON output file:

```
json
{"openrpc" : "1.2.6",
 "info" : {
  "version" : "5.0.0",
  "title" : "RSKj JSON-RPC",
  ... etc
}
```

## Configuration over CLI

Besides the parameters and flags, it's also possible to configure the node over the CLI using the JVM parameters, which starts with the prefix `-D` followed by the full path of the configuration like it is placed inside the configuration file.

Example:

```
shell
java -cp rskj-core-<VERSION>.jar -Ddatabase.dir=/home/rsk/data co.rsk.Start
```

Most of the configurable options or settings for RSKj are available in "config". See config reference for more details.

## Reference implementation

See the definition of the CLI flags in the RSKj codebase:  
NodeCliFlags in NodeCliFlags.java

See the definition of the CLI parameters in the RSKj codebase:  
NodeCliOptions in NodeCliOptions.java

# 03-reference.md:

```
---
sidebarlabel: Reference
sidebarposition: 300
title: Rootstock Node Configuration Reference
tags: [rsk, rskj, node, config, rpc api, node operators, rootstock]
description: "Configuration reference for RSKj"
renderfeatures: 'tables-with-borders'
---
```

See CLI flags for command line flag options.

```
:::info[Important Notice]
From RSKj HOP v4.2.0, RocksDB is no longer experimental. See guide on using RocksDB
:::
```

## Advanced Configuration

For advanced configuration requirements, please refer to this expected configuration file.  
This contains all possible configuration fields parsed by RSKj.

The default values for the config file are defined in this reference config file;  
and are "inherited" and varied based on the selected network.

## Guide

The following detail the most commonly used configuration fields parsed by RSKj.

- peer
- database
- database.import
- vm
- sync
- rpc
- wallet
- scoring
- miner
- blockchain.config.name
- bindaddress
- public.ip
- genesis
- transaction.outdated
- transaction.gasPriceCalculatorType
- play.vm
- hello.phrase
- details.inmemory.storage.limit
- solc.path

## peer

Describes how your node peers with other nodes.

peer.discovery.enabled = [true/false]

enables the possibility of being discovered by new peers.

peer.discovery.ip.list = []

is a list of the peers to start the discovering.

These peers must have the discovery.enabled option set to true.

These are the list of some of RSK bootstrap nodes:

| | ip.list |

| - | -

| Regtest | Not applicable |

| Testnet |

["bootstrap02.testnet.rsk.co:50505","bootstrap03.testnet.rsk.co:50505","bootstrap04.testnet.rsk.co:50505",  
,"bootstrap05.testnet.rsk.co:50505" |

| Mainnet |

"bootstrap01.rsk.co:5050","bootstrap02.rsk.co:5050","bootstrap03.rsk.co:5050","bootstrap04.rsk.co:5050",  
"bootstrap05.rsk.co:5050","bootstrap06.rsk.co:5050","bootstrap07.rsk.co:5050","bootstrap08.rsk.co:5050",  
"bootstrap09.rsk.co:5050","bootstrap10.rsk.co:5050","bootstrap11.rsk.co:5050","bootstrap12.rsk.co:5050",  
"bootstrap13.rsk.co:5050","bootstrap14.rsk.co:5050","bootstrap15.rsk.co:5050","bootstrap16.rsk.co:5050"

|

peer.active = []

is used to connect to specific nodes.

It can be empty.

peer.trusted = []

is a list of peers whose incoming connections are always accepted from.

It can be empty.

peer.port is the port used to listen to incoming connections.

Default port by each RSK network:

| | peer.port |

```
| - | - |  
| Regtest | 30305 |  
| Testnet | 50505 |  
| Mainnet | 5050 |
```

`peer.connection.timeout = number (seconds)`

specifies in seconds the timeout for trying to connect to a peer.

Suggested value: 2.

`channel.read.timeout = number (seconds)`

specifies in seconds how much time you will wait for a message to come before closing the channel.

Suggested value: 30.

- `peer.privateKey = hash`

is a securely generated private key unique for your node that must be set.

`peer.networkId = int`

is the number of the network to connect to.

It's important to maintain these numbers.

It identifies the network you are going to connect to.

For a Regtest private network, you should always use the same  
(not necessarily 34567).

RSK networks IDs:

```
| | peer.networkId |  
| - | - |  
| Regtest | 34567 |  
| Testnet | 779 |  
| Mainnet | 775 |
```

`peer.maxActivePeers = int`

is the max number of active peers that your node will maintain.

Suggested value: 30.

`peer.p2p.eip8 = bool`

forces peer to send handshake message in format defined by  
EIP-8.

## database

Describes where the blockchain database is saved.

`database.dir = path`

is the place to save physical storage files.

`database.reset = [true/false]`

resets the database when the application starts when set to true.

## database.import

Options related to experimental import sync v0.1.

`database.import.url = URL`

is the URL to the S3 bucket that hosts the database.

`database.import.trusted-keys = []`

list of trusted public keys to validate legit source.

`database.import.enabled = [true/false]`

enable the import sync.

## keyvalue.datasource (experimental)

Selects the database that will be used to store the information.

Possible options are:

- leveldb
- rocksdb (default)

If you wish to switch between the different storage options, for example from leveldb to rocksdb or vice versa, you must restart the node with the import option each time you do so.

vm

Enabling the vm.structured will log all the calls to the VM in the local database. This includes all the contract executions (opcodes). When testing, using this module is the only way to see exceptions.

- trace = [true/false]  
enables the vm.structured.
- dir = foldername  
the directory where calls are saved.
- compressed = [true/false]  
compress data when enabled.
- initStorageLimit = int  
the storage limit.

An example:

```
vm.structured {  
  trace = false  
  dir = vmtrace  
  compressed = true  
  initStorageLimit = 10000  
}
```

sync

Options related to syncing the blockchain:

- sync.enabled = [true/false]  
enables the blockchain synchronization.  
Should be set to true to keep the node updated.
- sync.max.hashes.ask = int  
determines the max amount of blocks to sync in a same batch.  
The synchronization is done in batches of blocks.  
Suggested value: 10000.
- sync.peer.count = int  
minimum peers count used in syncing.  
Sync may use more peers than this value but always trying to get at least this number from discovery.  
However, it will continue syncing if it's not reached.
- sync.timeoutWaitingPeers = int (seconds)  
timeout to start to wait for syncing requests.
- sync.timeoutWaitingRequests = int (seconds)  
expiration time in minutes for peer status.
- sync.maxSkeletonChunks = int



maximum amount of chunks included in a skeleton message.

`sync.chunkSize = int`

amount of blocks contained in a chunk,

must be 192 or a divisor of 192:

1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 192

An example:

```
sync {  
  enabled = true  
  max.hashes.ask = 10000  
  peer.count = 10  
  expectedPeers = 5  
  timeoutWaitingPeers = 1  
  timeoutWaitingRequest = 30  
  expirationTimePeerStatus = 10  
  maxSkeletonChunks = 20  
  chunkSize = 192  
}
```

`rpc`

Describes the configuration for the RPC protocol.

`rpc.providers = []`

lists the different providers (up to this moment, just web).

`rpc.providers.web` has a global setting for every web provider, `cors`.

`rpc.providers.web.cors = domain`

restricts the RPC calls from other domains.

Default value is `localhost`.

`rpc.providers.web` options:

`rpc.providers.web.http`

defines HTTP configuration:

`rpc.providers.web.http.enabled = [true/false]`

enables this web provider. Default value is `true`.

`rpc.providers.web.http.port = port`

is the HTTP-RPC server listening port.

By default RSK uses `4444`.

`rpc.providers.web.http.bindaddress = address`

is the HTTP-RPC server listening interface.

By default RSK uses `localhost`.

`rpc.providers.web.http.hosts = []`

is the list of node's domain names or IPs.

Check restrictions on valid host names.

NOTE: For details on how to connect over HTTP, see HTTP Transport Protocol.

`rpc.providers.web.ws`

defines WebSocket configuration:

`rpc.providers.web.ws.enabled = [true/false]`

enables this web provider.

Default value is `true`.

`rpc.providers.web.ws.port = port`

is the WS-RPC server listening port.

By default RSK uses 4445.

`rpc.providers.web.ws.bindaddress = address`

is the WS-RPC server listening interface.

By default RSK uses localhost.

NOTE: For details on how to connect over WebSockets, see Websockets Transport Protocol.

`rpc.modules` lists of different RPC modules.

If a module is not in the list and enabled, its RPC calls are discarded.

Examples:

```
json
modules = [
  { name: "eth", version: "1.0", enabled: "true" },
  { name: "net", version: "1.0", enabled: "true" },
  { name: "rpc", version: "1.0", enabled: "true" },
  { name: "web3", version: "1.0", enabled: "true" },
  { name: "evm", version: "1.0", enabled: "true" },
  { name: "sco", version: "1.0", enabled: "true" },
  { name: "txpool", version: "1.0", enabled: "true" },
  { name: "personal", version: "1.0", enabled: "true" }
]
```

It is possible to enable/ disable particular methods in a module.

```
shell
{
  name: "evm",
  version: "1.0",
  enabled: "true",
  methods: {
    enabled: ["evmsnapshot", "evmrevert"],
    disabled: ["evmreset", "evmincreaseTime"]
  }
}
```

To use the RPC miner module you must include:

```
{ name: "mnr", version: "1.0", enabled: "true" }
```

> Important Info:

- RPC methods for each module can be found in the JSON-RPC documentation.
- See the JSON-RPC Configurable limits

wallet

You can store your accounts on the node to use them to sign transactions. However, it is not secure to use a wallet in a public node.

```

shell
wallet {
  enabled = true
  accounts = [
    {
      "publicKey" : "<PUBLICKEY>"
      "privateKey" : "<PRIVATEKEY>"
    }
  ]
}

```

## scoring

Scoring is the way the node 'punishes' other nodes when bad responses are sent. Punishment can be done by node ID or address.

```

scoring.nodes.number = int
  number of nodes to keep scoring.
scoring.nodes.duration and scoring.addresses.duration
  initial punishment duration (in minutes).
  Default is 10.
scoring.nodes.increment and scoring.addresses.increment
  punishment duration increment (in percentage).
  Default is 10.
scoring.nodes.maximum
  maximum punishment duration (in minutes).
  Default is 0.
scoring.addresses.maximum
  maximum punishment duration (in minutes).
  Default is 1 week.

```

An example:

```

shell
scoring {
  nodes {
    number: 100
    duration: 12
    increment: 10
    maximum: 0
  }
  addresses {
    duration: 12
    increment: 10
    maximum: 6000
  }
}

```

## miner

Check out [Configure RSKj node for mining](#) for detailed information about the miner configuration.

blockchain.config.name

A string that describe the name of the configuration.  
We use:

```
| | blockchain.config.name |  
| - | - |  
| Regtest | regtest |  
| Testnet | testnet |  
| Mainnet | main |
```

bindaddress

The network interface with wire protocol and peer discovery.

This is the last resort for public IP address when it cannot be obtained by other ways.

public.ip

The node's public IP address.

If it is not configured, defaults to the IPv4 returned  
by <http://checkip.amazonaws.com>.

genesis

It is the path to the genesis file.

The folder resources/genesis contains several versions of genesis  
configuration according to the network the peer will run on. Either one of these,  
or an absolute path to a file within the filesystem may be used.

```
| | genesis |  
| - | - |  
| Regtest | rsk-dev.json |  
| Testnet | bamboo-testnet.json |  
| Mainnet | rsk-mainnet.json |  
| Custom | /home/username/rskjconfig/genesis.json |
```

transaction.outdated

It defines when a transaction is outdated:

transaction.outdated.threshold = int

is the number of blocks that should pass before pending transaction is removed.

Suggested value: 10.

transaction.outdated.timeout = int

is the number of seconds that should pass before pending transaction is removed.

Suggested value: 100 (10 blocks 10 seconds per block).

transaction.gasPriceCalculatorType

It defines the type of gas price calculator being used.

Possible Values: WEIGHTEDPERCENTILE, PLAINPERCENTILE (default)

Setting this value to WEIGHTEDPERCENTILE allows for the gas used by the tx to be taken into account.

play.vm

A boolean that invokes VM program on message received.  
If the VM is not invoked, the balance transfer occurs anyway.  
Suggested value: true.

hello.phrase

A string that will be included in the hello message of the peer.

details.inmemory.storage.limit

Specifies when exactly to switch managing storage of the account on autonomous DB.  
Suggested value: 1.

If the in-memory storage of the contract exceeds the limit, only deltas are saved.

solc.path

The path to the Solidity compiler.  
This is set when you may want to use the node to compile your smart contracts.  
If you don't have Solidity installed, you can use /bin/false as value.

# 04-switch-network.md:

```
---
title: Switch network
sidebarposition: 400
sidebarlabel: Switch network
tags: [rsk, rskj, node, config, network, mainnet, testnet, regtest]
description: "How to switch your RSK node between the Mainnet, Testnet, and RegTest networks"
---
```

If you want to switch your node to Mainnet, Testnet, or RegTest networks:

1\. If your node is not in localhost, connect your computer to the node over ssh.

```
shell
ssh user@server
```

2\. Pick a network that you would like to connect to.

Regtest

In order to switch from another network to Regtest:

```
bash
```

```
sudo service rsk stop
cd /etc/rsk
sudo rm -f node.conf
sudo ln -s regtest.conf node.conf
sudo service rsk start
```

Using this network, it allows you to start with some wallets (accounts) on your node. These wallets have funds.

## Testnet

In order to switch from another network to Testnet:

```
bash
sudo service rsk stop
cd /etc/rsk
sudo rm -f node.conf
sudo ln -s testnet.conf node.conf
sudo service rsk start
```

## For Mainnet

In order to switch from another network to Mainnet:

```
bash
sudo service rsk stop
cd /etc/rsk
sudo rm -f node.conf
sudo ln -s mainnet.conf node.conf
sudo service rsk start
```

By running these instructions in your shell, you are:

- Stopping the running RSK service.
- Moving to RSK configuration folder (cd).
- Removing node.conf, that is a symbolic link to the configuration you're using (rm deletes it).
- Linking node.conf with the configuration file you decide (ls with the -s option, it makes symbolic - or soft - links). The node is configured to read directly from the node.conf link.
- Restarting the RSK service.

# 05-verbosity.md:

```
---
title: Configure Verbosity
sidebarposition: 500
sidebarlabel: Configure Verbosity
tags: [rsk, rskj, node, config, node operators, rootstock, logs]
description: "Configure RSKj for desired log verbosity, finding log files, and using logback."
---
```

You can configure the desired log verbosity for your Rootstock node installation according to your needs. More information can be found at: [Logback Project](#).

## Requirements

- RSK Node Installed
- SSH Access
- SuperUser Access (sudo)

## Where to find RSK log files

### Real time log:

```
shell
/var/log/rsk/rsk.log
```

### Compressed logs:

```
shell
/var/log/rsk/rskj-YYYY-MM-DD.N.log.gz
```

## Log level options

When you configure your log level, all log items for that level and below get written to the log file. In the following table, the left column represents the the possible values you can set in your configuration.

### !Log Levels

<!-- TODO fix this image, perhaps convert to a table -->

## Setting your desired verbosity configuration

You need to edit logback.xml file to set your desired level of verbosity.

```
bash
sudo vi /etc/rsk/logback.xml
```

This file allows you to configure different log levels for different parts of the application.

```
xml
<logger name="execute" level="INFO"/>
<logger name="blockvalidator" level="INFO"/>
<logger name="blocksyncservice" level="TRACE"/>
<logger name="blockexecutor" level="INFO"/>
<logger name="general" level="DEBUG"/>
<logger name="gaspricetracker" level="ERROR"/>
<logger name="web3" level="INFO"/>
<logger name="repository" level="ERROR"/>
<logger name="VM" level="ERROR"/>
<logger name="blockqueue" level="ERROR"/>
<logger name="io.netty" level="ERROR"/>
<logger name="block" level="ERROR"/>
```

```

<logger name="minerserver" level="INFO"/>
<logger name="txbuilderex" level="ERROR"/>
<logger name="pendingstate" level="INFO"/>
<logger name="hsqldb.db" level="ERROR"/>
<logger name="TCK-Test" level="ERROR"/>
<logger name="db" level="ERROR"/>
<logger name="net" level="ERROR"/>
<logger name="start" level="ERROR"/>
<logger name="cli" level="ERROR"/>
<logger name="txs" level="ERROR"/>
<logger name="gas" level="ERROR"/>
<logger name="main" level="ERROR"/>
<logger name="trie" level="ERROR"/>
<logger name="org.hibernate" level="ERROR"/>
<logger name="peermonitor" level="ERROR"/>
<logger name="bridge" level="ERROR"/>
<logger name="org.springframework" level="ERROR"/>
<logger name="rlp" level="ERROR"/>
<logger name="messagehandler" level="ERROR"/>
<logger name="syncprocessor" level="TRACE"/>
<logger name="sync" level="ERROR"/>
<logger name="BtcToRskClient" level="ERROR"/>
<logger name="ui" level="ERROR"/>
<logger name="java.nio" level="ERROR"/>
<logger name="org.eclipse.jetty" level="ERROR"/>
<logger name="wire" level="ERROR"/>
<logger name="BridgeSupport" level="ERROR"/>
<logger name="jsonrpc" level="ERROR"/>
<logger name="wallet" level="ERROR"/>
<logger name="blockchain" level="INFO"/>
<logger name="blockprocessor" level="ERROR"/>
<logger name="state" level="INFO"/>
<logger name="messageProcess" level="INFO"/>

<root level="DEBUG">
  <appender-ref ref="stdout"/>
  <appender-ref ref="FILE-AUDIT"/>
</root>

```

Save your changes

RSK logback.xml config will watch and apply changes without restarting RSK Node.

(The watcher can take up to 1 hour to notice the changes and reload the logging configuration)

RSK logs with default installation will rotate on daily basis and/or when the log file reach 100MB

Using this configuration:

- Most areas of the application will only log FATAL and ERROR events for most areas of the application.
- The execute, blockvalidator, blockexecutor, web3, minerserver, pendingstate, blockchain, messageProcess, areas specify INFO, so those will only log FATAL, ERROR, WARN, and INFO events.
- There will be no DEBUG, INFO, and TRACE logs.

Using logback configuration file

A logback configuration example can be downloaded from [here](#).



## Using logback with a installed node

If you're running a node using the release jar file use the following command:

```
bash
java -cp rskj-core-6.0.0-ARROWHEAD-all.jar -Dlogback.configurationFile=/full/path/to/logback.xml
co.rsk.Start
```

## Using logback with a compiled node

If you're running a node by compiling the code on VM Options add:

```
shell
-Dlogback.configurationFile=/full/path/to/logback.xml
```

> Note: path should be written without abbreviations (full path)

# peer-scoring-system.md:

```
---
title: Peer Scoring System
sidebarlabel: Peer Scoring System
sidebarposition: 600
tags: [rsk, rskj, node, config, peer, scoring]
description: " The peer scoring system protects the RSKj node's resources from abusive or malicious peers"
---
```

The main objective of the Peer Scoring System is to protect the Rootstock node's resources from abusive or malicious peers.

For this, the Rootstock's node keeps track of a reputation for each peer that connects to it and disconnects those whose current reputation falls below acceptable levels.

## Actions

There are three possible actions that the Peer Scoring system can perform:

- Record an event
- Automatic penalisation of peer nodes
- Manually ban and unban peer nodes

## Recording an event

All event-firing scenarios will start by the node receiving a message from a peer.

All messages are automatically recorded as events.

However, for the current version, only some of them have a negative impact on the peer's reputation.

These events are:

- INVALIDNETWORK: This occurs when the received network ID is different from the network ID of the node.
- INVALIDBLOCK: This happens when the received block is not valid as per RSK block validation requirements.
- INVALIDMESSAGE: This happens when the message received is not valid as per RSK message validation requirements.

If one of these events is recorded,  
the peer's reputation will be marked accordingly,  
and the penalisation process will start automatically.  
This occurs only in nodes where the peer scoring feature is enabled.

### Automatic Penalisation of Peer Nodes

A peer with a bad reputation will be punished,  
meaning that when the node receives a message from it,  
the peer will be automatically disconnected  
and the received message will be discarded.  
Penalties are applied at two levels:

- The peer's address, and
- the peer's nodeId.

The default duration for the first punishment is 10 minutes.  
This can be changed via the RSK node configuration:

- scoring.addresses.duration for address level
- scoring.nodes.duration for nodeId level

These values are specified in minutes.

It is worth mentioning that penalty duration  
will be incremented by a percentage value  
each time a penalty is applied to a peer.  
The default increase rate is 10%.  
This can be changed via the RSK node configuration:

- scoring.addresses.increment for address level
- scoring.nodes.increment for nodeId level

It is possible to define a maximum time for a node to stay in a penalised state,  
which defaults to 7 days for address level and 0 (unlimited) for nodeId level.  
This can be changed via the RSK node configuration:

- scoring.addresses.maximum for address level
- scoring.nodes.maximum for nodeId level

These values are specified in minutes.

### Manual Banning of Peer Nodes

A banned peer is considered as a peer with a bad reputation.  
Therefore, it will be disconnected the next time a message is received,  
and its messages will be discarded.

However, the action of banning a peer, unlike the Rootstock nodes's automatic reputation tracking, is a manual action.

In order to manually ban or unban a peer, this can be done by address, the following RPC methods should be used:

- scobanAddress(String address):
  - Removes an address or block to the list of banned addresses.
- scounbanAddress(String address):
  - Removes an address or block of addresses from the list of banned addresses.

To check what addresses are banned, use the following method:

- scobannedAddresses():
  - Returns the list of banned addresses and blocks

> Warning: These methods should be used with caution.

## Feature config

This can be changed via the RSK node configuration:

- scoring.punishmentEnabled

This value is specified as a boolean.

> Warning: The recommendation is to keep the defaults, this is a complex and powerful feature that should be used with caution.

## RPC methods

The following related RPC methods are available.

- scobanAddress(String address): see Manually banning of peer nodes.
- scounbanAddress(String address): see Manually banning of peer nodes.
- scobannedAddresses(): see Manually banning of peer nodes.
- scopeerList(): Returns the collected peer scoring information
- scoreputationSummary(): Returns the reputation summary of all the peers connected
- scoclearPeerScoring(String id): Clears scoring for the received id (firstly tried as an address, used as a nodeId otherwise).

Please note the default and recommended config is to NOT expose these methods publicly.

# preferences.md:

---

sidebarlabel: Setting Config Preferences

sidebarposition: 100

title: Set Config Preferences

tags: [rsk, rskj, node, config]

description: "Setting your own config preferences, when using the Java command, Ubuntu, Azure, AWS, or Docker."

---

The Rootstock node can be started with different CLI flags.

## Setting config preferences

See how to set your config:

- Using Ubuntu or Docker
- Using the java command

&hellip; to run the node.

> Remember:

> You need to restart the node if you've changed any configuration option.

## Using Ubuntu or Docker

Your node's config file is in /etc/rsk.

Default configurations are defined there and they are the same as the config.

You should edit the config related with the network you are using (mainnet.conf, testnet.conf, regtest.conf).

Check reference all the configuration options you could change.

## Using Windows

For other operating systems, including Windows, please use the -Drsk.conf.file option as specified below.

## Using java command

### 1. Create a .conf file

You can create a file with the configuration options that you want to replace from the default. Default configurations are defined in the Config.

The extension of the file must be .conf.

Check here for all the configuration option.

As an example, if you want to change the default database directory, your config file should only contain:

```
conf
database {
  dir = /new/path/for/database
  reset = false
}
```

### 2. Specify your config file path

To apply your configuration options, you need to set your own config file's path when you run your node.

This can be done in two ways:

- Running the node with the java command, add -Drsk.conf.file=path/to/your/file.conf

- Compiling the node with IntelliJ, add to VM options: `-Drsk.conf.file=path/to/your/file.conf`

## Using RocksDB

### :::info[Important Notice]

- Starting from RSKj HOP v4.2.0, RocksDB is no longer experimental. As of the most recent version, RocksDB has now been made the default storage library, replacing LevelDB. This change was made to tackle maintainability and performance issues of LevelDB.
- Previously, RSKj ran using LevelDB by default, with the option to switch to RocksDB. Now, RocksDB is the default storage option, aiming to enable higher performance within the RSKj nodes.

...

## Get Started

RSKj nodes run using RocksDB by default (See important info section). To switch back to LevelDB, modify the relevant RSKj config file (.conf) and set the config: `keyvalue.datasource=leveldb`.

The `keyvalue.datasource` property in the config may only be either `rocksdb` or `leveldb`.

> If you wish to switch between the different storage options, for example from `leveldb` to `rocksdb` or vice versa, you must restart the node with the import option.

The following sample command shows how to do this when the RSKj node was previously running the default (`leveldb`), and wants to run with `rocksdb` next.

> Note the use of the `--import` flag, which resets and re-imports the database.

```
java
java -Dkeyvalue.datasource=rocksdb -jar ./rskj-core/build/libs/rskj-core--all.jar --testnet --import
```

## Advantages:

RocksDB uses a log structured database engine, written entirely in C++, for maximum performance. Keys and values are just arbitrarily-sized byte streams.

RocksDB is optimized for fast, low latency storage such as flash drives and high-speed disk drives. RocksDB exploits the full potential of high read/write rates offered by flash or RAM.

RocksDB is adaptable to different workloads. From database storage engines such as MyRocks to application data caching to embedded workloads, RocksDB can be used for a variety of data needs.

RocksDB provides basic operations such as opening and closing a database, reading and writing to more advanced operations such as merging and compaction filters.

## Switching between DB Kinds

Switching between different types of databases in your system requires you to modify configuration files, drop the existing database, and restart your node so the node will start syncing from scratch using the new db kind.

### :::warning[Warning]

Nodes that were already running on LevelDB will continue to use LevelDB, and the same applies to RocksDB. However, all nodes setup from scratch will use RocksDB by default.

...

## Gas Price Setting

The value returned by `ethgasPrice` can be modified by setting a multiplier to be used while calculating the aforementioned gas price.

This can be done by setting a numeric value on `rpc.gasPriceMultiplier` in the configuration file. Default value is 1.1.

## Troubleshooting

### UDP port already in use

If you see the following error message, it means that RSKj is unable to bind to a particular port number, because prior to this, another process has already bound to the same port number.

Exception in thread "UDPServer" co.rsk.net.discovery.PeerDiscoveryException: Discovery can't be started.

at co.rsk.net.discovery.UDPServer\$1.run(UDPServer.java:65)

Caused by: java.net.BindException: Address already in use: bind

To rectify this, change the value of `peer.port` in the config file, or add a `peer.port` flag to the command when you start RSKj.

<Tabs>

<TabItem value="mac" label="Linux, Mac OSX" default>

shell

\$ java -Dpeer.port=50505 -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>

<TabItem value="windows" label=" Windows">

shell

C:\> java -Dpeer.port=50505 -cp <PATH-TO-THE-RSKJ-JAR> co.rsk.Start

</TabItem>

</Tabs>

# rate-limiting.md:

---

sidebarlabel: Transaction Rate Limiter

sidebarposition: 700

title: Transaction rate limiting

description: 'Transaction rate limiting on the Rootstock network. The features, their importance, how to use and configure them.'

tags: [node operators, rocksdb, node, rskj, rate, limit]

---

Let's take a look at what a rate limiter is, the importance of using a rate limiter, its features, and how to

configure the rate limiter in RSKj nodes using config keys.

> Also known as: Virtual Gas limiter

What is a Rate Limiter?

The Rate-Limiter deters accounts from issuing problematic transactions, preventing those transactions from being added to the transaction pool and relayed to other peers. It is designed to not affect normal usage, but limit resource-intensive transactions that exhaust the network capacity. An account can still perform any transaction, even the most expensive transactions (128KB in size) after about 32 minutes of inactivity.

RSKj now prevents such attacks, this is made possible by the Rate-Limiter, by preventing source accounts from broadcasting transactions which consume large amounts of resources. The higher the gas limit, the more the source account gets rate-limited. The rate-limit also takes into account additional factors such as:

- size
- gas price compared with the average
- nonce compared to the expected one
- nonce value
- gas price bump percent

The Rate-Limiter does this by granting a Virtual Gas Quota that will be consumed every time a transaction is received. It then allows this to replenish slowly, while the account is inactive.

If you have interacted with the Rootstock public nodes, you may have already encountered rate limiting in action. Note that the rate limits on the public nodes are implemented using a 3rd party service, and not within the RSKj node itself. Another notable difference is that the rate limits there are IP address based; and not account based.

How it Works

The feature works as follows:

1. A transaction is received by the node
2. The Rate-Limiter calculates the Virtual Gas consumed by the transaction taking into account several transaction factors, as explained above
3. The Rate-Limiter refreshes the sender's Virtual Gas quota taking into account how much time passed since their last transaction (up to a maximum) or it sets a predefined value if this is their first transaction
4. The Rate-Limiter compares consumed and quota values and then:
  1. if quota  $\geq$  consumed :
    1. the consumed value will be subtracted from the sender's quota
    2. the transaction will be added to the pool
    3. the transaction will be relayed to peers
  2. if quota  $<$  consumed:
    1. the transaction will be rejected and will not be added to the pool
    2. the transaction will not be relayed to peers

The implementation of the rate-limiter algorithm can be found within RSKj. See TxQuotaChecker for the implementation.

Why use the rate limiter?

The rate limiter is intended to prevent two variants of Network Denial of Service (DoS) attack against

## Rootstock networks:

- CPU-only attack: This happens when the nodes are forced to consume a high percentage of CPU in verifying signatures, preventing normal transactions from being verified and forwarded in time.
- CPU + bandwidth attack: In addition to the previous one, the network becomes congested with relayed transactions. Furthermore, the CPU is overloaded while compressing, decompressing, and hashing transactions. This prevents normal transactions from being relayed.

Both variants affect congested networks more than blockchains with empty blocks. All variants are highly practical on networks with cheap gas fees, but may be impractical on networks with expensive fees.

This feature has some inherent consequences:

1. Accounts gain trust over time.

When an account sends multiple transactions, and these do not go over the allowed limits, the RSKj node implementation remembers it as being more trustworthy for future transactions.

2. Accounts replenish virtual gas over time.

When an account does not send any transactions for a period of time, it gains the ability to send more frequent and larger future transactions.

## How to configure the rate limiter

The feature can be configured in RSKj nodes via the following configuration keys:

Open the configuration file. See expected Configuration file for more information. Add the following keys below to the config file.

`transaction.accountTxRateLimit.enabled:<boolean>`

- This enables or disables the rate limiting feature.
- Default value is true.
- Note: Disabling this feature is highly discouraged.

`transaction.accountTxRateLimit.cleanerPeriod:<int>`

- This sets how often, in minutes, to clean the collection storing the quotas, with max quota granted.
- The collection is implemented with a maximum size and will automatically discard less relevant entries in favour of more relevant ones.
- Default value is 30 minutes.
- Note: Unless the host has heavy memory constraints, retain the default value.

## # cli.md:

```
---
sidebarlabel: CLI
sidebarposition: 300
title: Running the RSKj Node using a CLI
tags: [rsk, rskj, node, node operators, contribute]
description: "How to compile and run an RSKj node from the command line interface."
---
```

> After fulfilling the steps Pre-requisites, Get the source code, Ensure the security chain and Get external dependencies according to your operating system (Windows, Linux, Mac), you can compile and run a Rootstock node from command line following these steps.



## Compiling and running the node

From the root directory where the code was downloaded, execute:

```
bash
./gradlew build -x test
```

After building, run:

```
bash
java -cp [jar] co.rsk.Start
```

Replace:

- [jar]: the path to the fatjar generated by the gradle shadow command. It can be found on `rskj-core/build/libs/rskj-core-{version}-all.jar`.

It's ready!

You are on the Mainnet by default. If you want to switch the networks, read [this page](#).

# linux.md:

```
---
sidebarlabel: Linux
sidebarposition: 500
title: How to compile and run an RSKj node on Linux
tags: [rsk, rskj, node, contribute, linux]
description: "How to compile and run an RSKj node on Linux. Installing pre-requisites. Get source code.
Ensure security chain. Get external dependencies. Compile and run. Configuring your IDE."
---
```

Here you have the steps to compile and run an Rootstock node on Linux.

### Pre-requisites

First of all, you will need to install:

| Dependency    | Details  |
|---------------|--|
| -----         | -----  |
| Git for Linux | Download this Git command line tool  |
| Java 8 JDK    | Follow the steps to install Java. To check if installation went correctly, check the version with command: java-version. Then, as admin, modify /etc/enviroment adding JAVAHOME="/usr/lib/jvm/java-8-oracle" |

Recommended IDEs:

- IntelliJ IDEA Community
- Visual Studio Code

### Get the source code

Using the installed command-line tool Git, you need to retrieve (or clone) the RSKj Github source code

from here.

Run these commands in the terminal:

```
shell
git clone --recursive https://github.com/rsksmart/rskj.git
cd rskj
git checkout tags/ARROWHEAD-6.0.0 -b ARROWHEAD-6.0.0
```

Note: It is better to download the code into a short path.

Ensure the security chain

Ensure the security chain of the downloaded source code.

Get external dependencies

Before you can launch IntelliJ IDEA, there is an important step.

Browse in your RSKj cloned directory and then launch configure.sh with the following terminal command:

```
shell
./configure.sh
```

This will download and set important components (e.g. Gradle Wrapper).

IntelliJ IDEA setup

Compiling the node

Now, you can launch IntelliJ IDEA:

In a terminal, and from the folder you extracted the tar.gz, go into idea/bin/.  
Then, type the following command to load the script:

```
shell
./idea/sh
```

When IntelliJ IDEA is launched you should have a window with different choices.

- Choose Import project.
  - Browse in the RskJ downloaded code the file rskj\build.gradle and select it. Click NEXT.
  - Within the dialog select Use default gradle wrapper and then click Finish.
- Keep IntelliJ IDEA open.

!img

IDEA Build/Run configuration

We need to create a new configuration profile to run the node from IDEA.

That can be done by clicking on Run -> Edit Configurations or as shown in the image below:

!img

Then set the options as shown below:

!img

- Main Class: co.rsk.Start
- Working directory: /path-to-code/rskJ
- Use classpath of module: rskj-coremain
- JRE need to be set as: Default (1.8 - SDK of 'rsk-coremain' module)

Running the node

We are ready to run the node using IDEA, just press the Start (green arrow) button at the right of the configuration just created.

!img

If everything is OK you should see the debug information like that:

!img

And yes! Congratulations! Now you're running a local RSK node :)

You're joined to Mainnet by default.

If you want to switch the network, add:

- For Testnet: --testnet
- For Regtest: --regtest

Inside the field Program Arguments in your run configuration.

Visual Studio Code setup

Recommended Plugins

- Extension Pack for Java.
- Gradle Plugin

Visual Studio Configuration Files:

In order to setup JDK configuration, we use .vscode/settings.json. Here we can setup the latest JDK for Extension Pack for Java, then use the recommended version for RSKj, for instance:

.vscode/settings.json

text

```
{
  "java.jdt.ls.java.home": "/lib/jvm/java-17-openjdk",
  "java.configuration.runtimes": [
    {
      "name": "JavaSE-1.8",
      "path": "/lib/jvm/java-1.8-openjdk",
      "default": true
    },
    {

```

```

    "name": "JavaSE-17",
    "path": "/lib/jvm/java-17-openjdk",
  },
]
}

```

In this example, we have setup Java 17 for Extension Pack for Java to work as expected and the default java compiler is Java 1.8.

In order to list these paths you can run:

```

bash
sudo update-alternatives --config java

```

```

or
bash
whereis java

```

Be aware that the path may vary depending on how you installed it.

In order to build, run or debug RSKj, we use `.vscode/launch.json`. Here we can setup the commands that will be used to run our application, for instance:

```

.vscode/launch.json
text
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "java",
      "name": "Run RSK Start",
      "request": "launch",
      "mainClass": "co.rsk.Start",
      "args" : "--testnet -Xkeyvalue.datasource=leveldb"
    }
  ]
}

```

In this example, we are going to run the application with the following arguments: `--testnet -Xkeyvalue.datasource=leveldb`.

### Running the project

We are ready to run the node using Visual Studio Code, select your configuration from `launch.json` within Run and Debug.

!img

Click on start (green play icon at the left of your configuration name).

!img

A debug tools menu shows up at the top of the IDE window, where you could run the node step by step!

## Building the project with Gradle in Visual Studio Code

In order to build the project using Gradle, we can simply go to the respective tab. On tab, we should be able to see all available Gradle configurations from the application. Select the project to be built and double-click the desired Gradle Task.

!img

## Testing in Visual Studio Code

In order to run tests, we can simply go to the Testing tab where you can see all the tests. We can also go directly to the test file and right-click the icon at the left of a declaration of a test and then decide to either run or debug the test.

!img

Any problems?

Check out the troubleshooting section, hope it helps!

# macos.md:

```
---
sidebarlabel: MacOS
sidebarposition: 400
title: How to compile and run an RSKj node on Mac OS
tags: [rsk, rskj, node, contribute, mac, osx]
description: "How to compile and run an RSKj node on Mac OSX. Installing pre-requisites. Get source code. Ensure security chain. Get external dependencies. Compile and run. Configuring your IDE."
---
```

Here you have the steps to compile and run an Rootstock node on Mac.

## Pre-requisites

First of all, you will need to install:

| Dependency  | Details   |
|-------------|---|
| Git for Mac | Download this Git command line tool   |
| Java 8 JDK  | Follow the steps to install Java. To check if installation went correctly, check the version with command: java -version. |

## Recommended IDEs:

- IntelliJ IDEA Community
- Visual Studio Code

To complete Java installation you need to configure the JAVAHOME environment variable. You have to run the following commands on terminal:

```
bash
/usr/libexec/javahome

export JAVAHOME=/usr/libexec/javahome

launchctl setenv JAVAHOME /usr/libexec/javahome
```

## Get the source code

Using the installed command-line tool Git, you need to retrieve (or clone) the RSKj Github source code from here.

Run these commands on Git command line:

```
shell
git clone --recursive https://github.com/rsksmart/rskj.git
cd rskj
git checkout tags/ARROWHEAD-6.0.0 -b ARROWHEAD-6.0.0
```

Note: It is better to download the code into a short path.

## Ensure the security chain

Ensure the security chain of the downloaded source code.

## Get external dependencies

Before you can launch IntelliJ IDEA, there is an important step.

Browse in your RSKj cloned directory and then launch configure.sh with the following terminal command:

```
shell
./configure.sh
```

This will download and set important components (e.g. Gradle Wrapper).

## IntelliJ IDEA setup

## Compiling the node

Now, you can launch IntelliJ IDEA.

When IntelliJ IDEA is launched you should have a window with different options.

- Choose Import project.
  - Browse in the RskJ downloaded code the file rskj\build.gradle and select it. Click NEXT.
  - Within the dialog select Use default gradle wrapper and then click Finish.
- Keep IntelliJ IDEA open.

!img

## IDEA Build/Run configuration

We need to create a new configuration profile to run the node from IDEA.

That can be done by clicking on Run -> Edit Configurations or as shown in the following picture:

!img

Then set the options as shown below:

!img

- Main Class: co.rsk.Start
- Working directory: /path-to-code/rskJ
- Use classpath of module: rskj-coremain
- JRE need to be set as: Default (1.8 - SDK of 'rsk-coremain' module)

Running the node

We are ready to run the node using IDEA, just press the Start (green arrow) button at the right of the configuration just created.

!img

If everything is OK you should see the debug information like that:

!img

And yes! Congratulations! Now you're running a local Rootstock node :)

You're joined to Mainnet by default.

If you want to switch the network, add:

- For Testnet: --testnet
- For Regtest: --regtest

Inside the field Program arguments in your run configuration.

Visual Studio Code setup

Recommended Plugins

- Extension Pack for Java.
- Gradle Plugin

Visual Studio Configuration Files:

In order to setup JDK configuration, we use .vscode/settings.json. Here we can setup the latest JDK for Extension Pack for Java, then use the recommended version for RSKj, for instance:

.vscode/settings.json

text

```
{
  "java.jdt.ls.java.home": "/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",
  "java.configuration.runtimes": [
    {
      "name": "JavaSE-1.8",
```

```

    "path": "/Library/Internet Plug-Ins/JavaAppletPlugin.plugin/Contents/Home",
    "default": true
  },
  {
    "name": "JavaSE-17",
    "path": "/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",
  },
]
}

```

In this example, we have setup Java 17 for Extension Pack for Java to work as expected and the default java compiler is Java 1.8.

In order to list these paths you can run:

```

bash
/usr/libexec/javahome -V

```

or

```

bash
whereis java

```

Be aware that the path may vary depending on how you installed it.

In order to build, run or debug RSKj, we use `.vscode/launch.json`. Here we can setup the commands that will be used to run our application, for instance:

```

.vscode/launch.json
text
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "java",
      "name": "Run RSK Start",
      "request": "launch",
      "mainClass": "co.rsk.Start",
      "args": "--testnet -Xkeyvalue.datasource=leveldb"
    }
  ]
}

```

In this example we are going to run the application with the following arguments: `--testnet -Xkeyvalue.datasource=leveldb`.

### Running the project

We are ready to run the node using Visual Studio Code, select your configuration from `launch.json` within Run and Debug.



!img

Click on start (green play icon at the left of your configuration name).

!img

A debug tools menu shows up at the top of the IDE window, where you could run the node step by step!

## Building the project with Gradle in Visual Studio Code

In order to build the project using Gradle, we can simply go to the respective tab. On tab, we should be able to see all available Gradle configurations from the application. Select the project to be built and double-click the desired Gradle Task.

!img

## Testing in Visual Studio Code

In order to run tests, we can simply go to the Testing tab where you can see all the tests. We can also go directly to the test file and right-click the icon at the left of a declaration of a test and then decide to either run or debug the test.

!img

Any problems?

Check out the troubleshooting section, hope it helps!

# windows.md:

---

sidebarlabel: Windows

sidebarposition: 600

title: How to compile and run an RSKj node on Windows

tags: [rsk, rskj, node, contribute, node operators, windows]

description: "How to compile and run an RSKj node on Windows. Installing pre-requisites. Get source code. Ensure security chain. Get external dependencies. Compile and run. Configuring your IDE."

---

Here you have the steps to compile and run an Rootstock node on Windows.

## IntelliJ IDEA setup

### Compiling the node

After opening IDEA, we need to load the RSKj project, this can be done by using the Import project option in IDEA.

To do that follow the next steps:

- Go to File -> New -> Project from Existing Sources...
- Browse in the RSKj downloaded code the file rskj\build.gradle and select it.
- Within the dialog select Use default gradle wrapper and then press Finish.

!img

## IDEA Build/Run configuration

We need to create a new configuration profile to run the node from IDEA.

That can be done by clicking on Run -> Edit Configurations or as shown in the following picture:

!img

Then set the options as shown below:

!img

- Main Class: co.rsk.Start
- Working directory: /path/to/code/rskJ
- Use classpath of module: rskj-coremain
- JRE need to be set as: Default (1.8 - SDK of 'rsk-coremain' module)

:::info[Info]

- If it isn't configured the default JDK, you have to set it in: File -> Project Structure.
- If the IDE doesn't recognize the configuration options, open rskj/rskj-core/build.gradle and sync it from Gradle tab.

:::

## Running the node

We are ready to run the node using IDEA, just press the Start button at the right of the configuration we've just created.

!img

If everything is OK, you should see the debug information like that:

!img

And yes! Congratulations! Now you're running a local Rootstock node :)

You're joined to Mainnet by default.

If you want to switch the network, add:

- For Testnet: --testnet
- For Regtest: --regtest

Inside the field Program Arguments in your run configuration.

## Visual Studio Code setup

### Recommended Plugins

- Extension Pack for Java.
- Gradle Plugin

### Visual Studio Configuration Files:

In order to setup JDK configuration, we use .vscode/settings.json. Here we can setup the latest JDK for

Extension Pack for Java, then use the recommended version for RSKj, for instance:

.vscode/settings.json

```
{
  "java.jdt.ls.java.home": "C:\\jdk-17",
  "java.configuration.runtimes": [
    {
      "name": "JavaSE-1.8",
      "path": "C:\\jdk-1.8",
      "default": true
    },
    {
      "name": "JavaSE-17",
      "path": "C:\\jdk-17",
    },
  ]
}
```

In this example, we have setup Java 17 for Extension Pack for Java to work as expected and the default java compiler is Java 1.8. These paths should point to your java home.

Be aware that the path may vary depending on how you installed it.

In order to build, run or debug RSKj, we use .vscode/launch.json. Here we can setup the commands that will be used to run our application, for instance:

.vscode/launch.json

text

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "java",
      "name": "Run RSK Start",
      "request": "launch",
      "mainClass": "co.rsk.Start",
      "args": "--testnet -Xkeyvalue.datasource=leveldb"
    }
  ]
}
```

In this example we are going to run the application with the following arguments: --testnet -Xkeyvalue.datasource=leveldb.

Running the project

We are ready to run the node using Visual Studio Code, select your configuration from launch.json within Run and Debug.

!img

Click on start (green play icon at the left of your configuration name).

!img

A debug tools menu shows up at the top of the IDE window, where you could run the node step by step!

## Building the project with Gradle in Visual Studio Code

In order to build the project using Gradle, we can simply go to the respective tab. On tab, we should be able to see all available Gradle configurations from the application. Select the project to be built and double-click the desired Gradle Task.

!img

## Testing in Visual Studio Code

In order to run tests, we can simply go to the Testing tab where you can see all the tests. We can also go directly to the test file and right-click the icon at the left of a declaration of a test and then decide to either run or debug the test.

!img

Any problems?

Check out the troubleshooting section, hope it helps!

# network-upgrades.md:

---

title: Network Upgrades

sidebarlabel: Network Upgrades

sidebarposition: 100

tags: [rsk, rskj, node, contribute, upgrade, protocol, consensus, rskip]

description: "How to introduce consensus rules changes using network upgrades on an RSK node. What to consider. Adding a new rule. Running tests with new rules."

---

What's a network upgrade ?

A network upgrade is a change or group of changes to the protocol consensus rules, which are activated at a defined block number.

What to consider when introducing a consensus rule change in the code

Every consensus rule change needs to be associated with a specific RSKIP (RSK improvement proposal) in the RSKIPs github repository.

Consensus rules changes are introduced as part of a group of changes called Network Upgrade. Network upgrades are released to the general public as part of a specific RSK node version (defined by a release code name and version number. e.g.: RSK Wasabi 1.0.0), and the consensus rule changes introduced are selected by the community.

How to add a new consensus rule?

1. Set the release version (if not yet defined in the code). Define the new tag in NetworkUpgrade enum file.

```
java
public enum NetworkUpgrade {
    WASABI100("wasabi100"),
}
```

2. Set the network upgrade block activation height in [main||testnet||regtest||devnet].conf files

conf  
IE for main.conf

```
blockchain.config {
    name = main
    hardforkActivationHeights = {
        wasabi100 = 1591000,
    }
}
```

> For local development you should ONLY need to edit regtest.conf.

>[main||testnet||devnet].conf will only need to be edited before a NetworkUpgrade deploy, when the block activation height is already known.

3. Define the consensus rule RSKIP in ConsensusRule enum file.

```
java
public enum ConsensusRule {
    RSKIP106("rskip106"),
}
```

4. Associate the previous RSKIP with a specific Network Upgrade version in reference.conf file.

```
conf
blockchain = {
    config = {
        consensusRules = {
            rskip106 = wasabi100,
        }
    }
}
```

Coding a consensus rule change for an RSK Network Upgrade

When implementing a network upgrade you'll need to check if that change is active:

```
java
if (activations.isActive(ConsensusRule.RSKIP106) && address.equals(HDWALLETUTILSADDRDW)) {
    return new HDWalletUtils(config.getActivationConfig(), HDWALLETUTILSADDR);
}
```

## Testing

To run tests with specific consensus rules changes, you'll need to combine previously described methods at `ActivationConfigTest.BASECONFIG`

```
java
public class ActivationConfigTest {
    private static final Config BASECONFIG = ConfigFactory
        .parseString(String.join("\n",
            "hardforkActivationHeights: {",
            "    wasabi100: 0",
            "}, ",
            "consensusRules: {",
            "    rskip106: wasabi100,",
            "}"
        )));
}
```

# update.md:

```
---
title: Updating the Node
sidebarlabel: Updating the Node
sidebarposition: 200
tags: [rsk, rskj, node, update, version, rootstock]
description: "How to introduce consensus rules changes using network upgrades on an RSK node. What to consider. Adding a new rule. Running tests with new rules."
---
```

### 1. Download rskj

Download the latest release from the Github repo.

### 2. Update jar file

Note that PREVIOUS and NEW refer to version numbers.

```
bash
cd /usr/share/rsk
sudo service rsk stop
sudo mv rsk.jar rsk-PREVIOUS.jar
sudo mv rskj-core-NEW-all.jar rsk.jar
```

### 3. Clean up log directory

This step is optional.

```
bash
sudo mkdir /var/log/rsk/PREVIOUS/
sudo mv /var/log/rsk/rsk /var/log/rsk/PREVIOUS/
sudo service rsk start
```

#### 4. Validate service is running normally

Check logs:

```
bash
tail -f /var/log/rsk/rsk.log
```

Check that Blockchain is moving forward, and adding blocks:

```
bash
curl -s -X POST -H "Content-Type: application/json" -d '{"jsonrpc":"2.0","method":"ethblockNumber",
"params": {}, "id":123}' http://127.0.0.1:4444 | jq .result | tr -d '"' | awk '{print "printf \"%d\\n\" \"$0}' | sh
```

If you run this command a few times and the block number is increasing, it means it is syncing correctly too.

# index.md:

```
---
sidebarlabel: Public Nodes
sidebarposition: 9
title: Using Rootstock Public Nodes (Mainnet & Testnet)
tags: [rsk, networks, rskj node, json rpc, mainnet, testnet, cURI]
description: "Rootstock Public nodes (Mainnet, Testnet), RPC Methods."
---
```

RootstockLabs currently provides two public nodes that you can use for testing purposes, and you will find that information below.

Alternatively, follow the installation instructions, to run your own Rootstock node or use an alternative node provider. This is highly recommended for production environments, and in accordance with the bitcoiners' maxim: Don't trust. Verify.

:::info[RPC Node Providers]

The Rootstock public nodes do not expose WebSockets, they are HTTP only.

To work around this, you may either run your own Rootstock node, or use the Rootstock RPC API or use a third-party node provider, such as Getblock, NowNodes or dRPC.

...

Testnet

text  
<https://public-node.testnet.rsk.co>

Mainnet

text  
<https://public-node.rsk.co>

## Supported JSON RPC methods

List of supported JSON-RPC methods for each module can be found in the JSON-RPC documentation.

## Using cURL

Here's an example request using cURL to get the Mainnet block number:

```
> "Content-Type: application/json"
```

```
bash
```

```
curl https://public-node.rsk.co \
  -X POST -H "Content-Type: application/json" \
  --data '{"jsonrpc":"2.0","method":"ethblockNumber","params":[],"id":1}'
```

```
# index.md:
```

```
---
```

```
title: Troubleshooting
```

```
sidebarlabel: Troubleshooting
```

```
sidebarposition: 11
```

```
tags: [rsk, rskj, rootstock, node, faq, troubleshoot]
```

```
description: "How to solve some known or frequently encountered issues when working with RSKj"
```

```
---
```

This section explains how to solve some known or frequently encountered issues.

If what you need is not in this section, contact us without hesitation through the Rootstock Community on Discord. We will be happy to help you!

```
<Accordion>
```

```
<Accordion.Item eventKey="0">
```

```
<Accordion.Header as="h3">Discovery can't be started</Accordion.Header>
```

```
<Accordion.Body>
```

- On Windows, if you start the node and it doesn't do anything, there is a high chance you have a problem with the UDP port of the node.
- The UDP port is configured in the node's configuration file, specifically with the value `peer.port`. By default this port is configured to 5050.
- To check if that port is already taken by other application you can follow these steps:
  - Open a cmd console and run `netstat -ano -p UDP | findstr :5050` (or replace 5050 with the port of your preference).
  - You will get a result with the process ID (if any) already using that port for UDP.
  - With the process ID (the value at the far right), run this command `tasklist /FI "PID eq processId-you-got"`.
  - This will let you know which application/service is using this port.
  - Please make sure the port of your preference is not taken by other application. If so, you need to change the node configuration, by overwriting the `peer`.

```
</Accordion.Body>
```

```
</Accordion.Item>
```

```
<Accordion.Item eventKey="1">
```

```
<Accordion.Header as="h3">I don't see the logs</Accordion.Header>
```

```
<Accordion.Body>
```

- You can configure your own log level, following these instructions.

```
</Accordion.Body>
```



```

</Accordion.Item>
<Accordion.Item eventKey="2">
  <Accordion.Header as="h3">Plugin with id witness not found</Accordion.Header>
  <Accordion.Body>
    - If you have this error it's possible that you have missed to run rskj's dependencies.
    - So please, follow the instructions depending on your operation system:
    - On Windows
    - On Linux
    - On Mac
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="3">
  <Accordion.Header as="h3">Truffle doesn't seem to work connected to
Rootstock</Accordion.Header>
  <Accordion.Body>
    - If you can not get truffle migrate complete, you will see something like:
    javascript
      Writing artifacts to ./build/contracts
      Using network 'development'.
      Running migration: 1initialmigration.js
      Deploying Migrations...
      0xc82d661d579e40d22c732b2162734f97aeb13fa095946927cbb8cd896b26a7a3

    - Be sure you are using the right configuration in the truffle.js and truffle-config.js files.
    - Remember that you need: node host, node port, networkid and in some cases the from (by default
Truffle uses the first account in the node). This last one should be an account with positive balance
(because it's the one Truffle uses to deploy contract and run transactions) and it should be present
between the node's accounts (you can know that by executing the web3.eth.accounts command).
    - So, your config file should be like this:
    javascript
      module.exports = {
      networks : {
        rsk: {
          from : "0xcd2a3d9f938e13cd947ec05abc7fe734df8dd826",
          host : "localhost",
          port : 4444,
          networkid : "" // Match any network id
        }
      }
    };

  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="4">
  <Accordion.Header as="h3">Can't get public IP</Accordion.Header>
  <Accordion.Body>
    - If you get the error:
    - Can't get public IP when you're trying to run your rskj node, the reason is that rskj uses Amazon
Check IP service to set the public.ip parameter.
    - To solve it, you need to change the public.ip key in config file with your IP address (if you don't
know your IP, simply search for it).
    - Visit the Config page to change a node's configuration file.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="5">

```

```
<Accordion.Header as="h3">Rewind Blocks</Accordion.Header>
<Accordion.Body>
  - This tool should be used in a scenario where an RSK node processes blocks that are corrupted or
  invalid, for example after a hard fork. It allows one to remove such blocks and start from a previously
  known state. It does so by removing the blocks with block number higher than the block number
  parameter command line argument.
  - Note: The node must be turned off before the rewind, and restarted after.
  - Example:
    java -cp rsk-core-<VERSION>.jar co.rsk.cli.tools.RewindBlocks 1000000
  - The above command removes the blocks with number 1000001 or higher.
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="6">
  <Accordion.Header as="h3">DbMigrate: Migrate between databases</Accordion.Header>
  <Accordion.Body>
    - This tool allows the user to migrate between different supported databases such as rocksdb and
    leveldb.
    - How to use
      - To use the DbMigrate tool to migrate between databases, we will need a tool class and CLI
      arguments.
      - The tool class is: co.rsk.cli.tools.DbMigrate
      - Required CLI arguments:
        - args[0] - database target where we are going to insert the information from the current selected
        database.
        - Note: You cannot migrate to the same database or an error will be thrown. It is highly
        recommended to turn off the node in order to perform the migration since latest data could be lost.> > -
        Example migrating from leveldb to rocksdb:
          - java -cp rsk-core-<VERSION>.jar co.rsk.cli.tools.DbMigrate rocksdb
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="7">
    <Accordion.Header as="h3"> ERROR: failed to solve: failed to read dockerfile</Accordion.Header>
    <Accordion.Body>
      - The first error indicates that Docker couldn't find the Dockerfile in your current directory. Make sure
      you're in the correct directory or specify the path to the Dockerfile
      - If your Dockerfile is in txt, move the Dockerfile.txt to Dockerfile: mv /path/to/Dockerfile.txt
      /path/to/Dockerfile
      - Proceed with Docker Build command: docker build -t regtest /path/to/rskj-node-jar
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="8">
    <Accordion.Header as="h3">WARNING: The requested image's platform (linux/amd64) does not
    match</Accordion.Header>
    <Accordion.Body>
      - This warning indicates that the platform of the image doesn't match the platform of your host
      machine. The image is built for linux/amd64 architecture, but your host machine is linux/arm64/v8
      architecture.
      - Use a compatible image: docker run -d --name rsk-node -p 4444:4444 -p 50505:50505
      rsksmart/rskj:arm64v8-latest node --regtest
    </Accordion.Body>
  </Accordion.Item>
</Accordion>
```

# index.md:

---

sidebarposition: 1

title: Resources Overview

sidebarlabel: Overview

tags: [rsk, rootstock, solutions, courses, guides, tutorials]

description: "Find all the resources you need to get started on rootstock whether you're from a developer background, an open source contributor, a startup, or a just looking to learn via tutorials or courses."

---

The Rootstock Resources section serves as a comprehensive knowledge base comprising of links to grants opportunities, events, hackathons, tutorials, community, ambassador programs, and how to contribute to Rootstock platform.

## Navigating Resources

| Resource   | Description   |
|--|---|
| -----  | -----   |
| <a href="#">Contribute to Rootstock</a>                  | Explore clear and concise contribution guidelines and the various ways to contribute to the Rootstock platform. |
| <a href="#">FAQs</a>                                     | Find answers to frequently asked questions (FAQs) and troubleshoot errors you encounter on your journey.        |
| <a href="#">Join the Discord Community</a>               | Be a part of the Rootstock Community on Discord.  |
| <a href="#">Ambassador Programs</a>                      | Learn about the Rootstock Ambassador Program, and how to become one.  |
| <a href="#">Grants</a>                                   | Get support to build your next dApp on Rootstock through the strategic grants program.                          |
| <a href="#">Tutorials</a>                                | Tutorials on Rootstock.   |
| <a href="#">Guides</a>                                   | User Guides on Rootstock.   |
| <a href="#">Hackathons</a>                               | Hackathon and Workshop Resources.   |
| <a href="#">Rootstock Improvement Proposals (RSKIPs)</a> | Rootstock Improvement Proposals.  |

# index.md:

---

sidebarlabel: Courses on Rootstock

sidebarposition: 2

title: Courses on Rootstock

tags: [rsk, faqs, help, support, course, rootstock, ambassador-program]

description: "Welcome to Rootstock Courses; Explore learning materials and courses to enable you get started on building on Rootstock and RIF Technologies."

---

Explore learning materials and courses to enable you get started on building on Rootstock and RIF Technologies.

<Card

image="/img/courses/welcome.jpg"

title="Rootstock Blockchain Developer Course"

description="Learn how to write, test, secure, deploy and verify smart contracts on the Rootstock blockchain network."

link="https://rsk.thinkific.com/courses/blockchain-developer/"

/>

<br></br>

```
<Card
  image="/img/courses/user-course-modules/welcome.jpg"
  title="Rootstock User Course"
  description="Learn how to use and interact with the Rootstock blockchain network."
  link="https://rsk.thinkific.com/courses/blockchain-user"
/>
```

# 01-bug-bounty.md:

```
---
sidebarlabel: Bug Bounty Program
sidebarposition: 2
title: Contribute to RootstockLabs platforms security
tags: [rsk, rif, bounty, security, rootstock]
---
```

RootstockLabs has created the bug bounty program to reward researchers that submit valid vulnerabilities to improve the RootstockLabs platforms security.

```
<div class="btn-container">
  <span></span>
  <a class="green" href="https://hackerone.com/rootstocklabs">Visit the Bug Bounty Program Page on
Hackerone</a>
</div>
```

## Service Level Agreement (SLA)

RootstockLabs aims to meet the following SLAs for hackers participating in our program:

Time to first response (from report submit) - 5 business days  
Time to triage (from report submit) - 7 business days  
Time to bounty (from triage) - 15 business days

We aim to keep you informed about the progress throughout the process.

## Disclosure Policy

Follow HackerOne's disclosure guidelines.

Public disclosure of a vulnerability makes it ineligible for a bounty. If the user reports the vulnerability to other security teams (e.g. Ethereum or ETC) but reports to RootstockLabs with considerable delay, then RootstockLabs may reduce or cancel the bounty.

## Scope and Rules

Visit the Scope Section on Hackerone to view the scope / out of scope vulnerability, and the program rules.

# 02-writing-contest.md:

---

sidebarlabel: Writing Contest

sidebarposition: 3

title: Bitcoin Writing Contest

description: "The #bitcoin Writing Contest, presented by Rootstock and HackerNoon, is now live with a total prize pool of \$17,500 up for grabs! Whether you're a thought leader, a skilled writer, a talented developer, or simply passionate about blockchain technology, this contest invites you to showcase your expertise in various Bitcoin-related topics."

tags: [rootstock, rsk, workshop, resources, hackernoon, writing, tutorials]

---

The #bitcoin Writing Contest, presented by Rootstock and HackerNoon, is now live with a total prize pool of \$17,500 up for grabs!

Contest entries should showcase thought leadership, dev stack and tooling, and also creating engaging tutorials and guides in the realm of Bitcoin and Rootstock. Submit a story with the #bitcoin tag on HackerNoon. Submit a story with the #bitcoin tag on HackerNoon, read the 3 Steps to Enter The #bitcoin Writing Contest.

:::tip[Tip]

Looking to contribute to open source code or contribute to the security of the Rootstock platform? See the Contribute section for more exciting ways to contribute to Rootstock.

:::

```
<div class="btn-container">
```

```
  <span></span>
```

```
    <a class="green" href="https://www.contests.hackernoon.com/bitcoin-writing-contest">Register for the  
Contest on HackerNoon</a>
```

```
</div>
```

## Contest Themes

The contest welcomes submissions under the following themes:

**Thought Leadership:** Explore various topics about Bitcoin, both new and old, such as Runes, what Satoshi's writings mean to you, ordinals, layer 2 solutions, sidechains, how Bitcoin changed finance forever, and/or quality analysis about its price prediction or impact on inflation.

**Dev Stack and Tooling:** Delve into the development stack and tooling related to building on Bitcoin, including SDKs, APIs, smart contract development frameworks, etc. Developers and makers should showcase the technical capabilities for building with Bitcoin.

**Tutorials and Guides:** Provide step-by-step tutorials and guides on how to build applications, smart contracts, or implement specific features. Bonus points for building with Rootstock. These tutorials and guides will serve as practical guides for developers and enthusiasts looking to start building on Bitcoin.

## Timeline and Rounds

The contest consists of 3 rounds and will run for six months.

Round 1: May 22 - July 21, 2024

Round 2: July 22 - September 21, 2024

Round 3: September 22 - November 22, 2024

> For more information, visit the #bitcoin contest page on hackernoon for timelines and when to submit your application.

## Prizes

Here's a breakdown of the prizes for the best submissions.

1st prize - \$2,000  
2nd prize - \$1,500  
3rd prize - \$1,000

> 2 Bonus Prizes worth \$650 each for bitcoin thought-leadership submissions every round!

## Rules and Guidelines

Read the guidelines for the contest on HackerNoon.

## FAQs

Here are some frequently asked questions about the contest. For more information on guidelines prizes, duration and FAQs, see the [#bitcoin Writing Contest Rules & Guidelines](#).

- Who can participate in the HackerNoon Writing Contest?
  - Anyone passionate about Bitcoin and Rootstock can participate. Whether you're a developer, enthusiast, or industry expert, we welcome your unique perspectives.
- What types of content are eligible for submission?
  - We accept articles showcasing thought leadership, development insights, and tutorials related to Bitcoin and Rootstock. This includes opinion pieces, technical guides, and how-to tutorials.
- What are the contest themes?
  - The contest focuses on three main themes: Thought Leadership, Dev Stack & Tooling, and Tutorials. Each theme encourages contributors to explore specific aspects of Bitcoin and Rootstock.
- How long is the contest duration?
  - The contest runs for 6 months, providing ample time for participants to create and submit their entries.
- What are the prizes for the winners?
  - Details about contest prizes will be announced closer to the submission deadline. Stay tuned for exciting rewards!
- How do I register for the contest?
  - To register, visit the Contest Page on HackerNoon and follow the instructions to sign up as a participant.
- Are there any specific guidelines for submissions?
  - Yes, detailed contest rules and guidelines are provided on the HackerNoon platform. Make sure to review them before submitting your entry.
- Where can I find more information about Rootstock?
  - Explore additional resources such as the Rootstock website, Developer Portal, Blog, and Discord Community for insights and inspiration. See [Useful Guides and Resources](#)
- How will winners be notified?
  - Winners will be notified via email or through official announcements on HackerNoon and Rootstock platforms.
- Can I republish my contest entry elsewhere after submission?
  - Participants retain the rights to their submissions but should review HackerNoon's guidelines regarding republishing after the contest.

## Useful Guides and Resources

[Rootstock Website](#)  
[Rootstock Blog](#)

# index.md:

---

title: Contribute to Rootstock  
sidebarlabel: Contribute  
sidebarposition: 3  
description: "An overview of different ways you can contribute to Rootstock."  
tags: [rootstock, rsk, contribute, open source]

---

Are you passionate about web3, bitcoin and the Blockchain? Do you have a passion for Open Source and Web3? Do you enjoy writing, coding, bounty hunting, solving real-world problems, and eager to contribute to the future of Bitcoin and Decentralized Finance? Join the Rootstock Discord Community and start making contributions!

Who is it for?

Whether you're a seasoned developer, creative writer, researcher, bug bounty hunter, or simply enthusiastic about blockchain, the program welcomes contributors from all backgrounds.

Getting Started

From writing, bug bounties and open-source projects, there are various ways to participate and earn valuable incentives.

```
<Card
  title="Bug Bounty Program"
  description="RootstockLabs has created the bug bounty program to reward researchers that submit
  valid vulnerabilities to improve the RootstockLabs platforms security."
  link="/resources/contribute/bug-bounty/"
/>
```

<br></br>

```
<Card
  title="Writing Contests"
  description="Contribute articles, tutorials and guides about Rootstock, Bitcoin, etc."
  link="/resources/contribute/writing-contest/"
/>
```

<br></br>

```
<Card
  title="Open Source Code"
  description="Contribute to Rootstock's Open Source Code Repos."
  link="https://github.com/rsksmart"
/>
```

# index.md:

---

sidebarlabel: FAQs

sidebarposition: 8  
title: Frequently Asked Questions  
tags: [resources, rsk, faqs, help, support, rootstock]  
description: "Explore frequently asked questions about Rootstock and RIF"  
---

Here are some frequently asked questions about the Rootstock and RIF Platforms.

## About Rootstock

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">What is Rootstock?</Accordion.Header>

<Accordion.Body>

Rootstock is the first and longest-lasting Bitcoin sidechain. It is the only layer 2 solution that combines the security of Bitcoin's proof of work with Ethereum's smart contract capabilities. The platform is open-source, EVM-compatible, and secured by over 60% of Bitcoin's hashing power, making it the gateway to a vibrant ecosystem of dApps that continues to evolve to become fully trustless.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">What is a smart contract?</Accordion.Header>

<Accordion.Body>

Smart contracts are digital agreements stored on a blockchain network such as Rootstock and executed automatically without intermediaries. A smart contract allows digital assets to be controlled, exchanged, and transferred. Smart contracts have numerous use cases, such as lending, voting, decentralized payments and exchanges, asset tokenization, etc.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">What is the purpose of Rootstock?</Accordion.Header>

<Accordion.Body>

Rootstock's primary purpose is to enable Bitcoin users to create and execute smart contracts, thereby extending the functionality and use cases of the Bitcoin network. Rootstock achieves this by using a two-way peg system that allows users to send Bitcoin directly to the Rootstock chain, where they become convertible to Rootstock's native cryptocurrency, RBTC. This RBTC can then be used within the Rootstock network to interact with smart contracts and dApps.

- In addition to smart contract functionality, Rootstock also focuses on providing solutions for faster transactions and higher scalability, two of the main challenges in the Bitcoin network. It also supports merged mining, allowing Bitcoin miners to mine both Bitcoin and RBTC simultaneously without additional computational resources.

- Furthermore, Rootstock is also home to the RIF (Rootstock Infrastructure Framework), which provides a range of open and decentralized infrastructure services, including payments, storage, and communications, that enable faster, easier, and scalable development of distributed applications (dApps).

- Finally, the purpose of Rootstock is to enhance the Bitcoin ecosystem by adding smart contract functionality and more without compromising the features that make Bitcoin unique such as security and decentralization.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="3">

<Accordion.Header as="h3">Is the Rootstock network compatible with the Ethereum network?</Accordion.Header>

<Accordion.Body>

- Rootstock is compatible with the Ethereum blockchain at the following layers:



- EVM compatibility
- Interprocess connectivity in JSON-RPC
- Smart contract programming in Solidity
- JavaScript interface with web3.js

- The Rootstock virtual machine (RVM) is highly compatible with the Ethereum Virtual Machine (EVM). Approximately annually, the Ethereum community performs a hard fork to add new functionalities to the blockchain. When these new functionalities align with Rootstock's vision, the community performs a corresponding hard fork to maintain compatibility with the EVM.

- Additionally, the RVM offers improved features over EVM, such as bridging with Bitcoin and querying the Bitcoin blockchain.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="4">

<Accordion.Header as="h3">Do you plan to add support for smart contract programming languages other than Solidity?</Accordion.Header>

<Accordion.Body>

Rootstock aims to support all Ethereum's contracts; therefore, it can generally support any language that compiles to the EVM. This includes Solidity, Julia, Rust and new or experimental programming languages like Vyper.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="5">

<Accordion.Header as="h3">What is the current state of the Rootstock project?</Accordion.Header>

<Accordion.Body>

- As of March 2024, the latest released version of Rootstock is the Arrowhead v6.0.0, an update that is mainly focused on bringing Ethereum compatibility enhancements to the Rootstock virtual machine, along with notable improvements and performance optimization in the 2-way peg protocol. Read more [Arrowhead 6.0.0: What You Need To Know About Rootstock's Upcoming Network Upgrade](#)

- Live statistics about the entire Rootstock network are available at [Rootstock Stats](#), and all the necessary source codes can be found at the Rootstock GitHub organization: [github.com/rsksmart](https://github.com/rsksmart).

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="6">

<Accordion.Header as="h3">How does Rootstock plan to be a reference in terms of smart contracts?</Accordion.Header>

<Accordion.Body>

- Since its inception security and scalability have been, and will continue to be Rootstock's key competitive advantages. With a deep understanding of scalability as a significant challenge in driving blockchain adoption, the Rootstock community continuously works to enable higher transaction throughput and reduce transactional costs.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="7">

<Accordion.Header as="h3">How is Rootstock approaching node diversity?</Accordion.Header>

<Accordion.Body>

- How many nodes does a healthy protocol need?

- The Rootstock community values node diversity and independence more than node quantity. Even though a few hundred Rootstock nodes can support a global cryptocurrency network now, Rootstock prioritizes more variety and autonomy among node operators. That's what decentralization means: don't trust, verify yourself. Rootstock nodes were designed to be lightweight and run to improve decentralization. There are community proposals for light clients to enable mobile nodes. The goal is to ensure Rootstock remains secure and scalable in the long run with sufficient quality and quantity of nodes.

</Accordion.Body>

</Accordion.Item>

</Accordion>

## Rootstock vs Other Platforms

<Accordion>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">How is Rootstock different from Stacks?</Accordion.Header>

<Accordion.Body>

> - Philosophy: Rootstock is a Bitcoin sidechain highly incentive-aligned with the Bitcoin ecosystem participants. It allows Bitcoin miners to earn additional transaction fees and allows Bitcoiners to transact in Bitcoin cheaply. The fact that Rootstock's native currency is RBTC also reinforces this alliance. Stacks has its token (STX) to pay transaction fees, and as of February 2024, it doesn't have any mechanism for transactions in Bitcoin. Therefore, Stacks is not a Bitcoin sidechain but a separate blockchain (or "altcoin") using Bitcoin to achieve consensus.

> - Consensus Mechanism: Rootstock uses merge-mining with Bitcoin, which means that Rootstock blocks are secured by the same miners that secure Bitcoin. Currently, more than 50% of Bitcoin's hash rate is securing Rootstock. The core idea of merge-mining dates back to 2010 and was proposed by Satoshi Nakamoto. It was first put into production by Namecoin in 2011. Merged mining has been battle-tested in many blockchains, such as Litecoin, for almost a decade. Rootstock uses a variant of merged mining called DECOR, which is specifically designed for Rootstock's needs. Instead of relying on proven consensus protocols, Stacks was launched with a new consensus protocol called proof-of-burn but changed its consensus several times, the last time to PoX. Soon, it will switch again to the upgrade, as the previous consensus protocols were complex and presented flaws. These unexpected incentives allowed some Bitcoin miners to get an unfair share of Stacks subsidy and had hard scalability limitations regarding block time. Making things worse, the soundness of the new protocol is not proven, and it will be tested in production.

> - Smart Contract Capabilities: The Rootstock VM is highly compatible with the EVM (Ethereum virtual machine) and with the Ethereum web3 standard. Most Ethereum applications can be ported to Rootstock with a few configuration changes. Solidity is the main language used to program the EVM. It compiles a high-level language that resembles C++ or Java into EVM opcodes. Over the years, Solidity became a de-facto standard for contract development, providing a rich toolchain that includes compilation, testing, and security analysis tools. There are also thousands of online tutorials, libraries, and examples. Almost on the opposite side of the design decision spectrum, Stacks uses the Clarity programming language to code smart contracts. Clarity is a new LISP-like language that is only used by Stacks. Clarity is interpreted on-chain and not compiled, slows execution and limits scalability while providing more transparency. While interesting in theory, the transparency argument was rendered moot in practice as users of rootstock and Ethereum have gotten used to checking the availability and correctness of source code using automated tools that check its matching with the deployed code. A new ClarityWASM was announced in a planned upgrade to cope with the scalability problem of Stacks, but this upgrade still doesn't provide direct EVM compatibility.

> - Peg mechanism: Rootstock two-way peg is based on a federation of functionaries, each running a hardware security module (the PowHSM) that participates in a multi-sig that protects the locked funds. Hundreds of millions of USD in bitcoins are currently secured by Rootstock peg. Stacks doesn't have a peg to Bitcoin that allows it to transfer bitcoins back and forth. A planned upgrade is supposed to add a collateralized two-way peg to Bitcoin.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="9">

<Accordion.Header as="h3">How is Rootstock different from Liquid?</Accordion.Header>

<Accordion.Body>

- While Rootstock and Liquid are Bitcoin sidechains, they have different goals and features. Rootstock is a smart contract platform highly compatible with Ethereum, while Liquid is a federated sidechain that aims to provide fast and secure inter-exchange settlement.

- Some of the main differences are:

> - Consensus mechanism: Rootstock uses merge-mining with Bitcoin, which means that Rootstock

blocks are secured by the same miners that secure Bitcoin. Currently, more than 50% of Bitcoin hashrate is securing Rootstock. Liquid uses a federation of trusted functionaries that validate and sign blocks. Rootstock's merge-mining is more decentralized than Liquid's federation and provides the thermodynamic security of PoW, which Liquid does not.

- > - Smart contract capabilities: Rootstock supports Turing-complete smart contracts and has a virtual machine almost identical to Ethereum's. This allows developers to use the same tools, libraries, and languages as Ethereum and port existing applications to Rootstock. Liquid has a simpler scripting system that is not Turing-complete (within a single transaction) and only supports a limited set of use cases, such as atomic swaps and multi-sig transactions.

- > - Peg mechanism: Both Rootstock's and Liquid's two-way pegs are based on a federation of functionaries, each running a hardware security module (HSM) that participates in a multisig that protects the locked funds. Both also have emergency recovery systems. However, every Liquid withdrawal is advanced by one functionary to a user, and then the Liquid blockchain reimburses the functionary. This is known as a repayment protocol. Some or all functionaries require KYC checks. Rootstock system performs peg-outs directly to the user's Bitcoin wallet and currently does not impose KYC on peg-outs but forces the same user to be on both sides of the transfer. Rootstock Flyover system provides faster peg-outs, also using a repayment system.

- > - Scalability: Rootstock can achieve a higher transaction throughput than Liquid because Rootstock's transactions are smaller, its blocks can contain more transactions, and the Rootstock blockchain has a lower average block interval. Rootstock also has several scalability proposals close to being finalized, such as parallel transaction processing.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="10">

<Accordion.Header as="h3">How is Rootstock different from drivechains?</Accordion.Header>

<Accordion.Body>

- A drivechain is a special sidechain with a specific type of two-way-peg called "hashrate escrow." This peg mechanism gives most Bitcoin miners control of sidechain withdrawals but incentivizes miners to be honest and not abuse their powers. To achieve this, miners publicly confirm or reject withdrawals during a long period that can last 3 months. During this period, the community can detect cheaters that confirm invalid withdrawals. The peg is secure as long as there are strong long-term incentives for the honest majority of miners not to cheat. Rootstock, on the contrary, does not rely on monetary incentives. It uses a federation of PowHSM devices, and the tamper-proof devices vote on withdrawals. At the same time, each device enforces the same type of "hashrate escrow" but with a much-reduced timeframe of days. Therefore both Drivechains and Rootstock require the miner hashrate to support every withdrawal.

- Drivechains are promising but not currently available as they require a soft-fork in Bitcoin, which has been historically considered controversial and may never be performed. While drivechains may provide greater decentralization, the drivechain peg mechanism has never been tested, so the drivechain peg security is still uncertain.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="10">

<Accordion.Header as="h3">How is Rootstock different from Lightning?</Accordion.Header>

<Accordion.Body>

- Rootstock and Lightning are both layer-2 solutions that aim to improve the scalability and functionality of Bitcoin, but they have different approaches and trade-offs. Some of the main differences are:

- > - Architecture: Rootstock is a sidechain connected to the Bitcoin mainchain through a two-way peg mechanism, allowing users to lock and unlock bitcoins on both chains. Lightning is a network of payment channels built on the Bitcoin mainchain, allowing users to send and receive bitcoins off-chain.

- > - Smart contracts: Rootstock supports Turing-complete smart contracts and is compatible with the Ethereum Virtual Machine, which enables a wide range of decentralized applications and use cases on the Bitcoin network. Lightning only supports simple scripts, and transactions mainly focus on fast and cheap payments.

- > - Security and Liveness: Rootstock is secured by merge-mining with Bitcoin, which means that

Rootstock blocks are validated by the same miners and hash power as Bitcoin. The Bitcoin mainchain, the ultimate arbiter and enforcer of the payment channel, secures Lightning. Rootstock has greater liveness guarantees than Lightning. Lightning requires the cooperation of the parties sharing the payment channels and the existence of channel paths to destination addresses for payments to succeed. Rootstock blocks are always being produced, and as long as the Rootstock gas price specified in a transaction is adequate, transactions always get confirmed. Lightning security relies on parties checking their channels occasionally to avoid malicious closures. Rootstock security does not require the users to be active online or monitor their wallets continuously.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="11">

<Accordion.Header as="h3">How does Rootstock compare with Ethereum?</Accordion.Header>

<Accordion.Body>

How does Rootstock compare with Ethereum?

> - Does the Rootstock node require the same resources as an Ethereum geth node?

- Rootstock requires much fewer resources than Ethereum regarding blockchain size and state size.

This is due to less on-chain activity and the fact that Rootstock uses more efficient data structures to manage the state, such as the Unitrie, to achieve potentially higher transaction throughput.

</Accordion.Body>

</Accordion.Item>

</Accordion>

## Rootstock and RIF Token

<Accordion>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">What is the RIF token, and what is its purpose?</Accordion.Header>

<Accordion.Body>

- The RIF token is a utility token that powers the Rootstock Infrastructure Framework (RIF), a set of open-source, decentralized tools and technologies that make it easy to build accessible DeFi products and services on the blockchain. It is designed to enable and facilitate the wide range of decentralized services available on the RIF platform. The RIF token is the means of access for all RIF protocols, such as RIF Wallet, RNS, and third-party-developed infrastructure services. Also, any other apps that might be deployed on RIF's framework that agree to accept RIF Tokens as a means of accessing/consuming the service or app, such as RIF on Chain, the stablecoin protocol underpinning USDRIF. See the RIF Whitepaper for more details.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">What is the RBTC token, and what is its purpose?</Accordion.Header>

<Accordion.Body>

- The RIF token is a utility token that powers the Rootstock Infrastructure Framework (RIF), a set of open-source, decentralized tools and technologies that make it easy to build accessible DeFi products and services on the blockchain. It is designed to enable and facilitate the wide range of decentralized services available on the RIF platform. The RIF token is the means of access for all RIF protocols, such as RIF Wallet, RNS, and third-party-developed infrastructure services. Also, any other apps that might be deployed on RIF's framework that agree to accept RIF Tokens as a means of accessing/consuming the service or app, such as RIF on Chain, the stablecoin protocol underpinning USDRIF. See the RIF Whitepaper for more details.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="3">

<Accordion.Header as="h3">What is the RIF token, and what is its purpose?</Accordion.Header>

<Accordion.Body>

- The RIF token is a utility token that powers the Rootstock Infrastructure Framework (RIF), a set of open-source, decentralized tools and technologies that make it easy to build accessible DeFi products and services on the blockchain. It is designed to enable and facilitate the wide range of decentralized services available on the RIF platform. The RIF token is the means of access for all RIF protocols, such as RIF Wallet, RNS, and third-party-developed infrastructure services. Also, any other apps that might be deployed on RIF's framework that agree to accept RIF Tokens as a means of accessing/consuming the service or app, such as RIF on Chain, the stablecoin protocol underpinning USDRIF. See the RIF Whitepaper for more details.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="4">

<Accordion.Header as="h3">What is the RBTC token, and what is its purpose?</Accordion.Header>

<Accordion.Body>

- Smart Bitcoin (RBTC) is the native token of the Rootstock network. RBTC is pegged 1:1 to BTC, enabling Bitcoin transactions on the Rootstock and networks. It can be converted to and from BTC through the PowPeg.

- RBTC is used as gas to pay for executing transactions and smart contracts on the Rootstock network, rewarding miners and nodes, enabling interoperability among Bitcoin-based applications, and supporting the development of new solutions such as RIF Products.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="5">

<Accordion.Header as="h3">How is the RIF token different from RBTC?</Accordion.Header>

<Accordion.Body>

- The RIF token is different from RBTC in the following ways:

> - Purpose: RBTC is the native token of the Rootstock network used to maintain a one-to-one relationship with Bitcoin. It is also used as gas to pay for smart contract execution and transaction fees on the network. RIF is a utility token used to access the services of the RIF protocols.

> - Portability: RBTC is pegged 1:1 to BTC and can be converted to and from BTC using the 2-way peg mechanism. RIF is an ERC20-compatible token that can be transferred across smart contract platforms.

> - Supply: RBTC has the same supply as BTC, which is capped at 21 million. RIF has a fixed supply of 1 billion tokens, which were pre-mined and distributed according to a token sale and an allocation plan.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="6">

<Accordion.Header as="h3">How can I obtain RBTC and RIF tokens?</Accordion.Header>

<Accordion.Body>

- You can obtain RBTC and RIF tokens through various exchanges like Money on Chain and Oku Trade.

- For an updated list, see Get RBTC.

- For RIF tokens, you can use exchanges like Sovryn, Binance, Gate.io, Lbank, MEXC, Coinex, and Hotbit. Please note that you should always use the right wallet and connect to the right network. RBTC can only be sent to and from Rootstock addresses on the network. Similarly, RIF tokens can only be sent to and from addresses that support the ERC677 token standard.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="7">

<Accordion.Header as="h3">How does the peg work?</Accordion.Header>

<Accordion.Body>

- The Rootstock peg has several modes to accomplish transfers: version 1, version 2, and flyover. The version 1 protocol is quite simple. When a Bitcoin user wants to use the 2-Way Peg, he sends a peg-in transaction to a multisig wallet whose funds are secured by the PowPeg. The same public key associated with Bitcoin addresses related to the source bitcoins in a peg-in transaction is used on the

Rootstock chain to obtain the destination address where the Smart Bitcoins are received. Although both Bitcoin and Rootstock's public and private keys are similar, each blockchain encodes the address in a different format. This means that the addresses on both blockchains are different but can be proven to belong to the same person.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="8">

<Accordion.Header as="h3">What wallets support Rootstock and RIF tokens?</Accordion.Header>

<Accordion.Body>

- Rootstock is currently supported in several different software and hardware wallets. See Wallets on Rootstock pages for more information.

</Accordion.Body>

</Accordion.Item>

</Accordion>

## Rootstock Features and Functionality

<Accordion>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">What is merged mining, and how does it secure the Rootstock network?</Accordion.Header>

<Accordion.Body>

- Merged mining is a technique that allows miners to mine two or more blockchains simultaneously, using the same hash power and without compromising the security of either chain. The Rootstock network is merge-mined with the Bitcoin network and designed such that merge-mining with Bitcoin does not pose any performance penalty to Bitcoin miners. Therefore, merge miners can earn rewards on both Rootstock and Bitcoin simultaneously.

- The merge-mining process secures the Rootstock network by leveraging the hash power of the Bitcoin network, the largest and most secure blockchain in the world. By doing so, Rootstock achieves high decentralization, reliability, and immutability for its smart contracts and transactions.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">What consensus protocol does Rootstock use, and how does it prevent attacks?</Accordion.Header>

<Accordion.Body>

- Rootstock uses DECOR+, a unique variant of Nakamoto Consensus, with the capability to merge mine with Bitcoin or any other blockchain, sharing the Bitcoin block format and proof-of-work.

- The proof-of-work (PoW) consensus mechanism requires miners to solve a cryptographic puzzle to create new blocks and validate transactions. This prevents attacks by making it costly and difficult for malicious actors to alter the blockchain or create fraudulent transactions. PoW also ensures that the longest and most secure chain is always valid.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="3">

<Accordion.Header as="h3">What is the Rootstock Transactional throughput?</Accordion.Header>

<Accordion.Body>

- The block gas limit and the average block rate determine the number of transactions per second executable on the Rootstock platform. The current average block rate is one block every 30 seconds. The miner can vote to increase the block gas limit at each mined block. Currently, the block gas limit is 6.8M gas units per block. A simple RBTC transaction consumes 21K gas, so the Rootstock platform can execute 11 transactions per second today. This limit could increase by activating one of several improvement proposals, such as the parallel transaction proposal specified in RSKIP-144.

</Accordion.Body>

</Accordion.Item>

```
<Accordion.Item eventKey="4">
  <Accordion.Header as="h3">What is the average transaction confirmation time of
Rootstock?</Accordion.Header>
  <Accordion.Body>
    > - How many confirmations are required?
    - On average, the network currently generates a block every 30 seconds. Miners can reduce the
average block time to 15 seconds by optimizing their merge-mining operations. Systems that receive
payments over Rootstock in exchange for a good or service outside the Rootstock blockchain should wait
a variable number of confirmation blocks, depending on the amount involved in the payments. A minimum
of 12 confirmations is recommended, corresponding to an average delay of 6 minutes.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="5">
  <Accordion.Header as="h3">How many transactions per second will the Rootstock network
withstand?</Accordion.Header>
  <Accordion.Body>
    - Beta releases of improved Rootstock nodes have been tested to accommodate 100 tx/s without
incident. As the technology improves, transactions per second may similarly increase.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="6">
  <Accordion.Header as="h3">How does Rootstock protect its network from resource exhaustion
attacks?</Accordion.Header>
  <Accordion.Body>
    - The Rootstock "gas system" prevents attackers from creating, spreading, and including
resource-intensive transactions in blocks without paying the associated fees. Every resource, including
CPU, bandwidth, and storage, is accounted for by the consumption of an amount of gas. Every block has
a gas limit, so the resources a block can consume are limited, making a resource exhaustion attack
ineffective. Additionally, Rootstock nodes have an intelligent transaction rate limiter that protects the
network from DoS attacks before transactions are included in blocks.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="7">
  <Accordion.Header as="h3">Is Rootstock secure from miners abusing the gas system to acquire
resources cheaply, as in Ethereum?</Accordion.Header>
  <Accordion.Body>
    - On Rootstock, there is a minimum gas price, and therefore, miners cannot include transactions that
pay zero fees. The block's miner only gets 10% of the fees paid, and the rest is distributed to future
miners. Therefore, rogue miners cannot get platform resources at no cost. After Ethereum activated
EIP-1559, Ethereum adopted a similar protection called the base fee.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="8">
  <Accordion.Header as="h3">What is the address format of Rootstock, and how is it different from
Bitcoin?</Accordion.Header>
  <Accordion.Body>
    - A Rootstock address is an identifier of 40 hexadecimal characters, while a Bitcoin address is an
identifier of 26-35 alphanumeric characters. Rootstock addresses use uppercase and lowercase letters as
a checksum mechanism.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="9">
  <Accordion.Header as="h3">Is there a correlation between BTC and Rootstock addresses despite
looking like ETH addresses?</Accordion.Header>
  <Accordion.Body>
```

- Rootstock addresses are similar to Ethereum addresses. To avoid situations where users mistakenly send funds to Ethereum addresses or vice versa, Rootstock uses an address checksum mechanism that distinguishes between chains. This is currently in use in almost all Ethereum-like networks. Although this is not enforced in the node itself, it's important to consider it at the client level (e.g., wallets). The checksum mechanism is described in the RSKIP-60 Rootstock Improvement Proposal.

```
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="10">
  <Accordion.Header as="h3">What is the RIF token, and what is its purpose?</Accordion.Header>
  <Accordion.Body>
    ...
  </Accordion.Body>
</Accordion.Item>
</Accordion>
```

## Rootstock and RIF Services

```
<Accordion>
  <Accordion.Item eventKey="1">
    <Accordion.Header as="h3">What is RIF, and what are its goals?</Accordion.Header>
    <Accordion.Body>
      - Rootstock Infrastructure Framework (RIF) is a suite of open and decentralized infrastructure protocols that enable faster, easier, and more scalable distributed application development (dApps) within a unified environment. RIF is built on the Rootstock smart contract network, the first general-purpose smart contract secured by the Bitcoin network. RIF includes support for decentralized, third-party, off-chain payment networks; and easy-to-use interfaces for developers.
      - RIF aims to bridge the gap between blockchain technologies and their mass-market adoption by providing developers and users access to various services across multiple crypto-economies.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="2">
    <Accordion.Header as="h3">What exactly is the value proposition of RIF?</Accordion.Header>
    <Accordion.Body>
      - RIF is a service layer built on the Rootstock blockchain, offering open, decentralized tools and technologies. With RIF, developers can create scalable DeFi products quickly and easily.
      - RIF token is the native token of the RIF ecosystem. It is a utility asset used to interact with RIF products and services.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="3">
    <Accordion.Header as="h3">What is RIF Name Service?</Accordion.Header>
    <Accordion.Body>
      - RIF Name Service (RNS) is a protocol that enables the identification of blockchain addresses by human-readable names or aliases. It can identify other personal resources, such as payment or communication addresses, smart contracts, and Non-Fungible Tokens (NFTs).
      - RNS makes interacting with blockchain resources easier, more user-friendly and enhances interoperability across different platforms.
      > You can learn more about RNS by visiting the RIF website or reading the RIF White Paper.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="4">
    <Accordion.Header as="h3">Can I register a domain in RNS and then sell it in a secondary market?</Accordion.Header>
    <Accordion.Body>
```



- Anyone registering a domain in RNS can sell the domain directly or using a third-party secondary market.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="5">

<Accordion.Header as="h3">What is RIF Rollup? How does RIF Rollup L2 Solution scale Payments on Rootstock?</Accordion.Header>

<Accordion.Body>

- RIF Rollup is a trustless protocol for fast, scalable, low-cost payments on Rootstock. ZkRollup Technology powers it and is a fork of the Zero Knowledge zkSync Lite (v1) developed by Matter Labs. Its current functionality and scope include low gas transfers of RBTC and ERC20 tokens on the Rootstock network.

- RIF Rollup is a layer 2 scaling solution that operates on top of the Rootstock mainnet (layer 1) to increase transaction processing capacity, reduce latency, and lower transaction costs. It uses zk-SNARKs (Succinct Non-Interactive ARgument of Knowledge) to prove the correctness of batches of transactions. It uses on-chain data availability to keep users' funds safe while maintaining the security of layer-1 (Rootstock).

- Here's how it works:

> A group of layer 2 transactions are included in a Rollup block. State changes associated with all layer 2 transactions are communicated to layer 1 using transaction call data. In case of some irrecoverable failure of the rollup system, data availability permits users to reconstruct the layer 2 state and recover locked assets from the rollup contract. For each Rollup block, a SNARK (a family of cryptographic proof systems) is generated to prove the validity of every single transaction in the Rollup block. Once the proof is generated, it can be verified using the Rollup contract on layer 1.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="6">

<Accordion.Header as="h3">What is RIF Flyover?</Accordion.Header>

<Accordion.Body>

- RIF Flyover is a protocol that provides a fast and secure way for users to transfer Bitcoin (BTC) in and out of the Rootstock Ecosystem. Once the BTC is in the Rootstock Ecosystem, it can interact with various applications to send, save, and spend money.

- RIF Flyover enables users to earn interest, access loans, hedge against inflation, and more, all in a decentralized and censorship-resistant way. It essentially turns Bitcoin from a simple store of value into a fully decentralized financial system. This makes it easier to onboard customers into DeFi on Bitcoin.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="7">

<Accordion.Header as="h3">What is RIF Wallet?</Accordion.Header>

<Accordion.Body>

- RIF Wallet is an open-source wallet framework that allows businesses and developers to quickly build a unique wallet experience, giving users full control of their assets.

- Its smart contract technology enables functionalities that enhance security, usability, and adoption. RIF Wallet is an open-source wallet framework that allows businesses and developers to quickly build a wallet experience, giving users full control of their assets.

</Accordion.Body>

</Accordion.Item>

</Accordion>

## Rootstock Security and Scalability

<Accordion>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">What is the PowPeg Federation, and what is its role in the two-way

peg?</Accordion.Header>

<Accordion.Body>

- The PowPeg Federation is a group of functionaries that run specialized hardware called PowHSMs to facilitate the transfer of bitcoins between the main chain and the side chain and protect the bitcoins locked in the two-way peg between Rootstock and Bitcoin. The PowPeg Federation does not directly control the private keys of the Bitcoin multisig but only signs transactions that are proven valid by enough cumulative work. The PowPeg Federation also provides a watch tower service to inform the Rootstock Bridge smart contract about peg-in transactions. The PowPeg Federation's role is to keep their hardware and nodes connected and alive at all times and to audit the changes in the PowHSM, the Powpeg node, and the communication between them.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">What is the Armadillo monitoring system?</Accordion.Header>

<Accordion.Body>

> - How does it protect the Rootstock network from malicious miners?

- The Armadillo monitoring system is a tool that detects and alerts about potential attacks on the Rootstock network. It uses the Rootstock network's block headers and the Bitcoin network's coinbase information to measure the percentage of honest merge-mining. If the percentage drops below 50%, most miners could be trying to attack the Rootstock network by creating a hidden chain or censoring transactions.

- The Armadillo system protects the Rootstock network from malicious miners by providing timely and accurate information to the nodes and the community. The Rootstock nodes can use the Armadillo data to adjust their security parameters and reject blocks that are not sufficiently visible. The community can use the Armadillo data to monitor the network's health and take actions to mitigate the risk of an attack.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="3">

<Accordion.Header as="h3">What is the Rootstock virtual machine, and how is it compatible with Ethereum?</Accordion.Header>

<Accordion.Body>

- The Rootstock virtual machine (RVM) is the core of the Rootstock smart contract platform. The RVM is a forked version of the Ethereum virtual machine (EVM), meaning it can execute the same bytecode and opcodes as the EVM. The RVM is compatible with Ethereum smart contracts and the tools used to deploy and interact with them, such as Solidity, Hardhat, Foundry, Remix, etc. The RVM also has features such as native support for Bitcoin opcodes, precompiled contracts for elliptic curve cryptography, and a performance improvement pipeline.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="4">

<Accordion.Header as="h3">What is the Rootstock two-way peg, and how does it work?</Accordion.Header>

<Accordion.Body>

- The Rootstock two-way peg is a protocol that allows users to transfer bitcoins from the Bitcoin blockchain to the Rootstock blockchain and back, creating a token called RBTC that is pegged to the value of Bitcoin. The Rootstock two-way peg works by locking bitcoins in a multi-signature address on the Bitcoin side and releasing an equivalent amount of RBTC on the Rootstock side. The reverse process is also possible by burning RBTC on the Rootstock side and unlocking bitcoins on the Bitcoin side. A group of reputable organizations controls the multi-signature address called the PowPeg Federation, which uses special hardware devices called PowHSMs to protect private keys and validate transactions. The PowHSMs only sign transactions approved by both the Rootstock and Bitcoin networks using a proof-of-work mechanism. This way, the Rootstock two-way peg ensures high security and decentralization for the peg-in and peg-out transactions.

</Accordion.Body>

</Accordion.Item>

</Accordion>

# dapp-automation-cucumber.md:

---

sidebarlabel: dApp Automation with Cucumber

sidebarposition: 4

title: dApp Automation with Cucumber & Playwright

description: 'Testing decentralized applications (dApps) is crucial for delivering a smooth user experience and ensuring the reliability of decentralized systems. Cucumber and Playwright form a dynamic duo in automated testing, blending behavior-driven development (BDD) and powerful browser automation capabilities.'

tags: [rsk, rootstock, tutorials, resources, tests, dapp-automation, smart contracts, dapps, cucumber, playwright]

---

Rootstock is a blockchain platform that extends the capabilities of the Bitcoin network by incorporating smart contract functionality. Built to be EVM (Ethereum Virtual Machine) compatible, Rootstock enables developers to deploy and execute smart contracts using the same programming languages and tools as Ethereum.

This guide aims to introduce you to an agile automation framework designed exclusively for decentralized applications (dApps) automation and E2E testing.

This solution seamlessly brings together Cucumber's user-friendly behavior-driven development, Playwright's precise browser automation, and the tailored DApp testing capabilities of Synpress. With Cucumber's Gherkin syntax, teams collaboratively define DApp behaviors. Playwright, customized for Chrome, adds finesse to browser automation. Synpress, in its Playwright version, effortlessly integrates with MetaMask (more software wallets to come) for thorough DApp testing.

This way, developers enjoy expressive scenarios, targeted browser automation, and specialized DApp testing features.

## Prerequisites

- Install Nodejs and NPM
  - See Hackathon Dev Starter
- Cucumber
- Code Editor
  - Visual Studio Code

## Getting Started

Clone the repo and cd into the directory:

shell

git clone <https://github.com/rsksmart/e2edappsautomation/>

cd e2edappsautomation

## Install dependencies

To install dependencies, run the command `npm i` in the terminal or run the `npm:install` script.

Create a `.env` file inside config folder, and add your MetaMask test wallet address for testing purposes (seed & password). See how to create a metamask wallet and configure Metamask for rootstock.

See example:

text

```
secretWordsOrPrivateKey=test test test test test test test test test test test  
testpassword=Tester@1234
```

> To export a private key on Metamask, see [How to export an account private key](#).

> - Please note that this is sensitive information, even if it is stored locally in the .env file. If shared anyhow, you could potentially lose all your funds. Ensure the provided wallet is for testing purposes only.

> - Metamask version can be provided either in the .env file or in the src/hooks/fixtures.js file as follows:

shell

```
const metamaskPath = await prepareMetamask(  
  process.env.METAMASKVERSION || "10.25.0"  
);
```

> - You will find the Rootstock network already configured in the config/config.js file as seen in DApp under Test, you will only need to modify the dAppURL constant, which can point also to your localhost.

## Installing and configuring Cucumber

Cucumber already comes as a built-in dependency in this automation framework, when installing the dependencies, just be certain to add the vscode extensions as well, which will let you handle cucumber features seamlessly.

### - Cuke Step Definition Generator

- Author: Muralidharan Rajendran

- Description: This VSCode extension aids users by generating Cucumber Glue / Step Definition snippets for selected statements, streamlining work with Cucumber JS in VS Code.

- View Cuke Step Definition Generator on Marketplace.

### - Cucumber Auto Complete (Gherkin) Full Support

- Author: Alexander Krechik

- Description: This extension provides comprehensive language support for Cucumber (Gherkin), including syntax highlighting, snippets, autocompletion, and format support across multiple programming languages such as JS, TS, Ruby, and Kotlin.

- View Cucumber (Gherkin) Full Support on Marketplace.

### - Add Extensions on VSCode

text

```
{  
  "recommendations": [  
    "muralidharan92.cuke-step-definition-generator",  
    "alexkrechik.cucumberautocomplete"  
  ]  
}
```

## DApp Configuration

To test your DApp on your preferred blockchain, go to config/config.js and modify the following parameters:

shell

```
const dAppURL = 'https://wallet.testnet.rollup.rif.technology/';
```

```
// Custom network under test
const networkConfiguration = {
  networkName: 'Rootstock',
  rpcUrl: 'https://rpc.testnet.rootstock.io/{YOURAPIKEY}',
  chainId: '31',
  symbol: 'RBTC',
  isTestnet: true
}
```

## Running Tests

Since this is a boilerplate project, just a 'demo.feature' has been implemented. Feel free to build your test suite at `src/test/features/dappLivingDocumentation/`.

Execute `test` or `npm test` script to run the tests using chromium.

## Writing E2E Tests using Cucumber

### - A. Identifying test scenarios for dApps on Rootstock

- Identifying scenarios to automate in a UI framework involves considering various factors related to your application, testing goals, and the nature of the scenarios. Here are some guidelines specific to UI automation:

#### - Frequently Executed and Stable Tests:

Prioritize automating scenarios that are executed frequently, especially as part of your regression testing suite. Stable features with consistent behavior are good candidates.

#### - Critical Path and Core Functionality:

Identify and automate scenarios that cover the critical paths and core functionality of your application. These are the key user journeys that are crucial for the application's success.

#### - Data-Driven Testing:

Automate scenarios that involve testing with different sets of data. This is especially useful for formulating data-driven tests to cover a wide range of inputs.

#### - Integration with External Systems:

Automate scenarios that involve the integration of your application with external systems or APIs. Verify that data is exchanged correctly and that integrations function as expected.

#### - User Onboarding and User Experience:

Automate scenarios related to user onboarding and overall user experience. Verify that new users can easily navigate through the application and perform key actions.

### - B. Creating feature files for different use cases

- Inside the features folder, create a new file with a .feature extension. For example, `sample.feature`.
- Write your feature file using Gherkin syntax.
- For example:

Feature: Demo to test Cucumber + Playwright + Synpress

Scenario: Validate metamask connects to Rootstock DApp

Given I open the DApp website

When I connect metamask

Then I verify my wallet is successfully connected to the DApp

### - Defining step definitions to interact with Rootstock dApps

- An easy way to generate step definitions would be:
  - Select a step in the feature file
  - Right mouse click

- Generate Step Definition: Copy To Clipboard option
  - !Generate step definition
  - Then go to the stepDefinitions folder, create a new file with a .steps.js extension. For example, sample.steps.js and paste the generated step. A code snippet like this will be displayed:
- ```
shell
Then(/^I verify my wallet is successfully connected to the dApp$/, () => {
  return true;
});
```
- Since we are using "snippetInterface": "async-await" in the cucumber configuration cucumber.json, you will need to change the previous snippet manually to:

```
shell
Then(/^I verify my wallet is successfully connected to the dApp$/, async function () {
  return true;
});
```

- Now, you just simply need to add your code into that step, for example calling some of your page's methods, remember this is based on the page object model pattern. Here an example of an entire steps file:

```
shell
import { Given, When, Then } from '@cucumber/cucumber';
import metamask from "@synthetixio/synpress/commands/metamask.js";
import DemoPage from "../pages/demo.page.js"
```

```
Given(/^I open the dApp website$/, {timeout: 20 1000}, async function () {
  await DemoPage.navigateToDapp(global.BASEURL);
});
```

```
When(/^I connect metamask$/, {timeout: 20 1000}, async function () {
  await DemoPage.connectWallet();
  await metamask.acceptAccess();
});
```

```
Then(/^I verify my wallet is successfully connected to the dApp$/, {timeout: 20 1000}, async function () {
  await expect(page.locator(".address")).toHaveText("0xf39...92266");
});
```

- Notice, inside those steps there are references to the DemoPage methods as well as metamask methods. This is how the DemoPage class looks like, just stores some web elements and lets you execute certain actions with them.

```
shell
class DemoPage {
  // Page elements
  get btnConnectWallet() {
    return page.locator('[id="btn-core-connect-wallet"]');
  }
  get btnConnectMetamask() {
    return page.locator('.wallet-button-styling .svelte-1vlog3j').first();
  }
  // Methods
  async navigateToDapp(url) {
    await page.goto(url);
  }
  async connectWallet(){
```

```

    await this.btnConnectWallet.click();
    await this.btnConnectMetamask.click();
  }
}
export default new DemoPage();

```

## Reporting

- Generated reports will be located at reports folder
- Since Cucumber is the chosen runner, reports and other config options can be found at e2edappsautomation/cucumber.json

## Conclusion

Testing decentralized applications (dApps) is crucial for delivering a smooth user experience and ensuring the reliability of decentralized systems. Thorough testing of the frontend identifies and addresses usability issues, creating a user-friendly interface. Cucumber and Playwright form a dynamic duo in automated testing, blending behavior-driven development (BDD) and powerful browser automation capabilities. Cucumber, employing the human-readable Gherkin syntax, enables collaboration between technical and non-technical team members by describing application behavior in plain language.

## Useful Links

- For information on other testing tools, see Quick Start: Testing Smart Contracts
- Cucumber
- Playwright

# defillama.md:

```

---
sidebarposition: 6
title: Add a Protocol To DefiLlama
sidebarlabel: Add a Protocol To DefiLlama
description: "DefiLlama is the largest TVL aggregator for DeFi. Learn how to list a DeFi project and write an SDK adapter to add a Protocol to DefiLlama."
tags: [knowledge-base, defillama, protocol, rootstock, defi]
---

```

<!-- !DefiLlama -->

DefiLlama is the leading aggregator for Total Value Locked (TVL) in the decentralized finance (DeFi) ecosystem. Its open-source data is maintained by a community of contributors from various protocols. DefiLlama prioritizes accuracy and transparency in its methodology.

TVL is calculated by assessing the value of tokens locked in the contracts of DeFi protocols and platforms. While bridge projects are included in the calculation, their TVL does not contribute to the overall TVL of any specific blockchain.

> Check out the DefiLlama website and DefiLlama docs for more details.

## How to list a DeFi project

Most adapters featured on DefiLlama are provided and managed by their individual communities, and any modifications are organized through the DefiLlama/DefiLlama-Adapters GitHub repository.

```
<div class="btn-container">
  <span></span>
  <a class="green" href="https://docs.llama.fi/list-your-project/submit-a-project">How to Submit a
Project</a>
</div>
```

## How to write an SDK adapter

An adapter is a piece of code designed to receive a UNIX timestamp and blockchain block heights as inputs. It then provides the balances of assets held within a protocol, considering the associated decimals (i.e., how they are stored on the blockchain). The SDK handles the conversion of raw asset balances into their equivalent in USD and aggregate them to calculate the total TVL. Consequently, the adapter requires minimal processing, as most of the conversion work is performed by the SDK.

```
<div class="btn-container">
  <span></span>
  <a class="green" href="https://docs.llama.fi/list-your-project/how-to-write-an-sdk-adapter">How to write
an SDK Adapter</a>
</div>
```

## Resources

- Visit DeFiLlama About to learn more.

# index.md:

```
---
sidebarposition: 4
title: Tutorials
sidebarlabel: Tutorials
tags: [rsk, rootstock, beginner, quick starts, advanced, port to rootstock, tutorials]
description: "Tutorials and learning resources"
---
```

```
<Filter
values=[
{label: 'Beginner', value: 'beginner'},
{label: 'Advanced', value: 'advanced'},
{label: 'Port to Rootstock', value: 'port-dapps'}
]>
<FilterItem
value="beginner, port-dapps"
title="Interact with Rootstock using Rust"
subtitle="rust"
color="orange"
linkHref="/resources/tutorials/rootstock-rust/"
description="Rust is extensively getting used on backend side of many defi applications, dApps,
developer tools, indexers and bridges. This guide will help developers to start using Rust on Rootstock
blockchain."
/>
<FilterItem
value="beginner, advanced"
title="Add Rootstock to Metamask Programmatically"
subtitle="metamask"
color="orange"
```



```

linkHref="/resources/tutorials/rootstock-metamask/"
description="Learn how to add and initiate a network switch on Metamask from a website."
/>
<FilterItem
value="beginner, advanced"
title="dApp Automation with Cucumber and Playwright"
subtitle="dapp-automation"
color="orange"
linkHref="/resources/tutorials/dapp-automation-cucumber/"
description="Testing decentralized applications (dApps) is crucial for delivering a smooth user experience
and ensuring the reliability of decentralized systems. Cucumber and Playwright form a dynamic duo in
automated testing, blending behavior-driven development (BDD) and powerful browser automation
capabilities."
/>
<FilterItem
value="advanced, port-dapps"
title="Port an Ethereum dApp to Rootstock"
subtitle="Ethereum"
color="orange"
linkHref="/resources/port-to-rootstock/ethereum-dapp/"
description="Porting an Ethereum decentralized application (dApp) to Rootstock presents an exciting
opportunity to leverage the benefits of the Rootstock network, which is a smart contract platform secured
by the Bitcoin network."
/>
<FilterItem
value="advanced"
title="Virtual Testnets on Rootstock using Tenderly"
subtitle="Tenderly"
color="orange"
linkHref="/resources/tutorials/rootstock-tenderly/"
description="Tenderly's virtual testing environment allows the creation of simulated networks, managing
account balances, and manipulating contract storage – all without needing to interact with the Rootstock
mainnet or testnet."
/>
<FilterItem
value="advanced"
title="Add a Protocol To DefiLlama"
subtitle="defillama"
color="orange"
linkHref="/resources/tutorials/defillama/"
description="DefiLlama is the largest TVL aggregator for DeFi. Learn how to list a DeFi project and write
an SDK adapter to add a Protocol to DefiLlama."
/>
</Filter>

```

# rootstock-metamask.md:

```

---
sidebarposition: 3
title: Adding Rootstock to Metamask Programmatically
sidebarlabel: Add Rootstock to Metamask Programmatically
tags: [rsk, rootstock, metamask, tutorials, resources, wallets]
description: "Learn how to add and initiate a network switch on Metamask from a website."
---

```

Rootstock is a blockchain with smart contract capabilities, it is possible to build decentralised applications (dApps) with it. Most dApps are web applications that you access with a regular Internet browser, such as Chrome. However, the blockchain interactions require some additional software, which comes in the form of browser extensions. These browser extensions insert a web3 provider object, with the Javascript parts of the web application used to interact with the blockchain, forming an integral part of a dApp architecture.

Note that these browser extensions store your private keys, and use them to sign transactions. So keep them secure.

In this tutorial, we will learn how to add and initiate a network switch on Metamask from a website. Subsequently, we will create a frontend application to check if our configuration works by connecting our frontend website to metamask.

Note that this functionality is important as it alerts users when they are on a different network than the one needed by your dApp. It will allow them to switch automatically to the correct network when they are connecting their wallet or when interacting with a smart contract.

Stress here is on the ability to switch automatically. Typically switching to a network for the first time is very involved for the end user, involving reading documentation, and manually updating the configuration options in Metamask. This skips the need for all that, and enables a better user experience.

## Requirements

To follow along in this tutorial, you will need the following;

- Metamask wallet

> If you do not have a Metamask wallet installed, follow the instructions in [How to Download, Install, and Setup a Metamask Wallet](#).

## How to Download, Install, and Setup a Metamask Wallet

```
<div class="video-container">
  <iframe width="949" height="534" src="https://youtube.com/embed/VlyqXD1TjJk" frameborder="0"
  allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
  allowfullscreen></iframe>
</div>
```

## Getting started

In this section, we will do the following;

- Clone the initial state of the repo
- List configuration files
- Configure networks
- Configure index.js
- Try out our frontend application
- See common errors

## Clone the initial state of the repository

To get started, clone the demo-code-snippets repository and navigate to the switch-network-to-rsk

directory.

## List Configuration Files

Let's take a look at the contents of the switch-network-to-rsk folder/directory.

### !config files

The index.html file contains a sample HTML file to test out our application. It includes a Connect to Testnet button and a Connect to Mainnet button, and we will see all these in action at the end of this tutorial.

The index.js file imports the network parameters from networks.js and defines the connectProviderTo() and switchToNetwork() functions.

The networks.js file contains all the configuration for the different networks on Rootstock that will be added to Metamask.

## Configure Networks

Here, we will configure the networks for both Mainnet and Testnet.

Open the networks.js file, or copy the code below, then paste into the network.js file. For more information on the different types of networks on Rootstock, see MetaMask installation.

```
js
export const rskTestnet = {
  chainName: 'Rootstock Testnet',
  chainId: '0x1f',
  rpcUrls: ['https://rpc.testnet.rootstock.io/{YOURAPIKEY}'],
  blockExplorerUrls: ['https://explorer.testnet.rootstock.io/'],
  nativeCurrency: {
    symbol: 'tRBTC',
    decimals: 18,
  },
};

export const rskMainnet = {
  chainName: 'Rootstock Mainnet',
  chainId: '0x1e',
  rpcUrls: ['https://rpc.testnet.rootstock.io/{YOURAPIKEY}'],
  blockExplorerUrls: ['https://explorer.rootstock.io/'],
  nativeCurrency: {
    symbol: 'RBTC',
    decimals: 18,
  },
};
```

> See how to Get an API Key from the RPC API

See full configuration on GitHub.

Configure index.js

In this section, we will configure the index.js file which contains all the functionalities powering our DApp, we will import some necessary files, check if metamask is installed, see how to add and switch networks programmatically from within a smart contract.

> Note that index.js has two files, the first file is a redacted version of the code which contains an initial state of the code and the second file contains the full version which contains the complete state of the code for index.js.

### Step 1: Import necessary files

Here, we will import the networks just configured in the previous section in networks.js file into index.js.

```
import { rskTestnet, rskMainnet } from './networks.js';
```

### Step 2: Check if Metamask is installed

initial state

```
javascript
async function connectProviderTo(network) {
  try {
    // TODO: implement request accounts

    await switchToNetwork(network);
    showPage(network.chainName, address);
  } catch (error) {
    showError(error.message);
  }
}
```

### Code Walkthrough

> The connectProviderTo() function initiates the connection to each of the Rootstock networks in Metamask, it uses the window.ethereum API to check if Metamask is installed then throws an error - Please install Metamask, if false, a popup appears triggered by window.ethereum.request method, this requests that the user provides an Ethereum address to be identified by. Once the request is accepted by the user it returns an Ethereum address and wallet is connected. See <https://docs.metamask.io/guide/rpc-api.html#restricted-methods>

Before adding a network, we need to be sure Metamask has been installed, to do this, add the following code to the try block:

```
javascript
try {
  // make sure Metamask is installed
  if (!window.ethereum) throw new Error('Please install Metamask!');
  // connect wallet
  const [address] = await window.ethereum.request({
    method: 'ethrequestAccounts',
  });
}
```

Copy and paste the full code below or see the code on [GitHub](#);

```
javascript
async function connectProviderTo(network) {
  try {
    // make sure Metamask is installed
    if (!window.ethereum) throw new Error('Please install Metamask!');
    // connect wallet
    const [address] = await window.ethereum.request({
      method: 'ethrequestAccounts',
    });
    await switchToNetwork(network);
    showPage(network.chainName, address);
  } catch (error) {
    showError(error.message);
  }
}
```

### Step 3: Adding and switching a network

initial state

```
javascript
async function switchToNetwork(network) {
  // TODO: implement network switching

  // make sure we switched
  const chainId = await window.ethereum.request({ method: 'ethchainId' });
  if (chainId !== network.chainId)
    throw new Error(Could not connect to ${network.chainName});
}
```

### Code Walkthrough

> The `switchToNetwork()` function adds a new network to Metamask and subsequently switches to that network. This function expects a network argument, and awaits a rootstock address which uses the `walletswitchEthereumChain` method and the `chainID`. It then throws an error if the switch prompt was rejected or the `chainID` passed was not found and tries to add the new chain to Metamask.

To programmatically add a network in Metamask, we need to call the `walletaddEthereumChain` method, exposed by `window.ethereum` and pass the network parameters configured in [Configure Networks](#) section. To do this, add a try block inside the `switchToNetwork()` function.

```
javascript
try {
  // trying to switch to a network already added to Metamask
  await window.ethereum.request({
    method: 'walletswitchEthereumChain',
    params: [{ chainId: network.chainId }],
  });
}
```

See the full code below or see [link to the code on GitHub](#);

```

javascript
async function switchToNetwork(network) {
  try {
    // trying to switch to a network already added to Metamask
    await window.ethereum.request({
      method: 'walletswitchEthereumChain',
      params: [{ chainId: network.chainId }],
    });
    // catching specific error 4902
  } catch (error) {
    // this error code indicates that the chain has not been added to Metamask
    if (error.code === 4902) {
      // trying to add new chain to Metamask
      await window.ethereum.request({
        method: 'walletaddEthereumChain',
        params: [network],
      });
    } else {
      // rethrow all other errors
      throw error;
    }
  }
  // make sure we switched
  const chainId = await window.ethereum.request({ method: 'ethchainId' });
  if (chainId !== network.chainId)
    throw new Error(Could not connect to ${network.chainName});
}

```

Show success if network switch was successful

initial state

```

javascript
async function showPage(chainName, address) {
  // TODO: Implement showPage functionality;
}

```

The showPage() function shows that the switch to another network is successful and should display a connected status, the network and a wallet address. This is done by performing DOM manipulation to add a connected status, the chainName and an address.

Add the following code inside the async function, or see the code on GitHub;

```

javascript
async function showPage(chainName, address) {
  document.getElementById('connect-prompt').classList.add('hidden');
  document.getElementById('connected').classList.remove('hidden');
  document.getElementById('chain-name').innerHTML = chainName;
  document.getElementById('wallet-address').innerHTML = address;
}

```

Throw an error if there was a problem

initial state

javascript

```
function showError(message = '') {  
  // TODO Implement showError;  
}
```

The showError() function is called in the event that something went wrong. This function is meant to throw an error containing a message if something went wrong. It should look like this:

!user-rejected-request-error

See common errors section for more explanation.

Add the following code into the showError() function or see code on GitHub;

javascript

```
function showError(message = '') {  
  document.getElementById('error').innerHTML = message;  
  if (!message) return;  
  // hide error message in 3 seconds  
  setTimeout(() => showError(''), 3000);  
}
```

Enable click event listeners to buttons

To add a click event listener to the connect buttons created in index.html, use the DOM to get connect-testnet and connect-mainnet buttons, then add an on "click" event listener which uses the connectProviderTo() function to handle the connection to rskTestnet or rskMainnet respectively.

See the code below, or see link on GitHub;

javascript

```
// add click event listeners to the Connect buttons  
document  
  .getElementById('connect-testnet')  
  .addEventListener('click', () => connectProviderTo(rskTestnet));  
document  
  .getElementById('connect-mainnet')  
  .addEventListener('click', () => connectProviderTo(rskMainnet));
```

Complete code walkthrough

You can find the full code for index.js below, or in the GitHub repository.

javascript

```
import { rskTestnet, rskMainnet } from './networks.js';  
  
async function connectProviderTo(network) {  
  try {
```

```

// make sure Metamask is installed
if (!window.ethereum) throw new Error('Please install Metamask!');
// connect wallet
const [address] = await window.ethereum.request({
  method: 'ethrequestAccounts',
});
await switchToNetwork(network);
showPage(network.chainName, address);
} catch (error) {
  showError(error.message);
}
}

// see details in Metamask documentation:
// https://docs.metamask.io/guide/rpc-api.html#wallet-addethereumchain
async function switchToNetwork(network) {
  try {
    // trying to switch to a network already added to Metamask
    await window.ethereum.request({
      method: 'walletswitchEthereumChain',
      params: [{ chainId: network.chainId }],
    });
    // catching specific error 4902
  } catch (error) {
    // this error code indicates that the chain has not been added to Metamask
    if (error.code === 4902) {
      // trying to add new chain to Metamask
      await window.ethereum.request({
        method: 'walletaddEthereumChain',
        params: [network],
      });
    } else {
      // rethrow all other errors
      throw error;
    }
  }
  // make sure we switched
  const chainId = await window.ethereum.request({ method: 'ethchainId' });
  if (chainId !== network.chainId)
    throw new Error(Could not connect to ${network.chainName});
}

async function showPage(chainName, address) {
  document.getElementById('connect-prompt').classList.add('hidden');
  document.getElementById('connected').classList.remove('hidden');
  document.getElementById('chain-name').innerHTML = chainName;
  document.getElementById('wallet-address').innerHTML = address;
}

function showError(message = '') {
  document.getElementById('error').innerHTML = message;
  if (!message) return;
  // hide error message in 3 seconds
  setTimeout(() => showError(''), 3000);
}

```



```
// add click event listeners to the Connect buttons
document
  .getElementById('connect-testnet')
  .addEventListener('click', () => connectProviderTo(rskTestnet));
document
  .getElementById('connect-mainnet')
  .addEventListener('click', () => connectProviderTo(rskMainnet));
```

## Frontend

Now let's try out our application, follow the steps below to check out the application on your browser.

(1) Open the switch-network-to-rsk folder in VSCode and within the folder open index.html

!index.html

(2) Run index.html with the Live Server VSCode extension by pressing Go Live button

!vscode-go-live-button

In the bottom-right corner of the VSCode window. VSCode will open the web page from index.html in a new browser window.

The browser should open a page on 127.0.0.1 or localhost.

!connect-mainnet-testnet-live-page

(3) On the web page, click on either Connect to Testnet or Connect to Mainnet button to add and switch to the corresponding network

!connect-to-testnet

(4) The browser then opens up a Metamask password prompt window, enter your Metamask password and press Unlock

!metamask-window

(5) Metamask shows a prompt: Allow this site to add a network?

!metamask-prompt

Inside the prompt you should see the details of the network configuration being added to Metamask.

This gives the user the option to verify if the network configuration options are legitimate by comparing against the official documentation.

> - See configurations for Rootstock Mainnet and Testnet.

(6) Press Approve

!metamask-approve

(7) Metamask subsequently shows another prompt: Allow this site to switch the network?. Press Switch network

!metamask-switch

(8) Now Metamask has added the new Rootstock network and switched to it!

!metamask-successful-switch

## Common errors

You may encounter the following errors when trying out the application:

- Error: Cannot destructure property of intermediate value as it is undefined
  - > !show-error-image
  - > - Problem: This can occur if the user is already connected to Rootstock Mainnet or Rootstock Testnet.
  - > - Possible Fix: If you encounter this error, ensure you're logged in to Metamask or check that you're not already connected to Rootstock Mainnet or Rootstock Testnet.
- Error: User rejected the request
  - > !user-rejected-request-error
  - > - Problem: This occurs when the user closes Metamask unexpectedly, or presses "reject" instead of "approve" in the dialogs.
  - > - Possible Fix: Confirm the request if all information is correct. Your dApp should have code to handle the scenario when the user does indeed decide not to go through with adding the new network configuration.

## Wrap up

Congratulations!!

You have learnt how to create a dApp which can programmatically;

- Add a new Rootstock network configuration, and,
- Switch to a Rootstock network.

# rootstock-rust.md:

---

sidebarposition: 2

title: Interact with Rootstock using Rust

sidebarlabel: Interact with Rootstock using Rust

tags: [rsk, rootstock, rust, tutorials, resources, smart contracts, alloy]

description: "Rust is extensively getting used on backend side of many defi applications, dApps, developer tools, indexers and bridges. This guide will help developers to start using Rust on Rootstock blockchain."

---

Rust is a fast and memory-efficient language, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

> This tutorial helps to get started on Rootstock using Rust by performing basic operations such as sending transactions and calling contracts using Alloy crate, similar to ethers.

## Introduction to Alloy

Alloy connects applications to blockchains, it serves as an entry point to connect with evm compatible blockchains. It is a rewrite of ethers-rs library from the ground up, with new features, high performance, etc. An example is Foundry, a tool written in Rust which uses Alloy as a dependency to connect with blockchains.

For more information, see Alloy Examples to help you get started.

## Prerequisites

### Rust

Install the latest version of Rust. If you already have Rust installed, make sure to use the latest version or update using the rustup toolchain.

## Getting Started

### Create a Rust project

Run the command below using cargo to create a starter project.

```
bash
cargo new rootstock-rs
```

> Next step is to update cargo.toml file with dependencies explained in next section.

## Setup Alloy

To install Alloy run the following command below in the root directory of the project:

```
bash
cd rootstock-rs
cargo add alloy --git https://github.com/alloy-rs/alloy
```

## Find more about Alloy setup using meta crate

> Note: All the dependencies required are mentioned in the .toml file below. Copy and paste into the cargo.toml file.

```
bash
[package]
name = "rootstock-alloy"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
alloy = { git = "https://github.com/alloy-rs/alloy", version = "0.1.3", default-features = true, features = ["providers", "rpc-client", "transport-http", "sol-types", "json", "contract", "rpc-types", "rpc-types-eth", "network", "signers", "signer-local"] }
eyre = "0.6.12"
futures-util = "0.3.30"
tokio = { version = "1", features = ["full"] }
```

> The types and import statements in Alloy dependencies are expected to change. If you face any type related errors while running the given examples in this tutorial, its recommended to check the Alloy repo and documentation.

## Connect to the Rootstock node

To connect to the Rootstock node., we will require a provider setup. A Provider is an abstraction of a connection to the Rootstock network, it provides a concise, and consistent interface to standard Ethereum node functionality.

To run this program, use cargo run in the root of the project:

```
bash
cd rootstock-rs
cargo run
```

The response should look like this:

```
bash
Finished dev profile [unoptimized + debuginfo] target(s) in 29.28s
  Running target/debug/rootstock-alloy
Hello, world!
```

Next, update the rootstock-rs/src/main.rs file with the program below:

```
:::info[Info]
```

Replace APIKEY with your RPC API Key. To get an APIKEY, see the RPC Docs.

```
:::
```

```
rs
use alloy::providers::{ Provider, ProviderBuilder };
use eyre::Result;

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Set up the HTTP transport which is consumed by the RPC client.
    let rpcurl = "https://rpc.testnet.rootstock.io/{YOURAPIKEY}".parse()?;

    // Create a provider with the HTTP transport using the reqwest crate.
    let provider = ProviderBuilder::new().onhttp(rpcurl);

    // Get chain id
    let chainid = provider.getchainid().await?;

    println!("chain id: {chainid}");

    // Get latest block number.
    let latestblock = provider.getblocknumber().await?;
```

```
println!("Latest block number: {latestblock}");

    Ok(())
}
```

The response should look like this:

```
bash
Finished dev profile [unoptimized + debuginfo] target(s) in 1.43s
  Running target/debug/rootstock-alloy
chain id: 31
Latest block number: 5292505
```

## Get RBTC / RIF balance

After setting up the provider, interact with Rootstock node, fetch balance of an address or call a contract. Now, copy and paste the code below.

We will do the following:

- Codegen from ABI file to interact with the contract.

- Create an abi directory in the root of the project and put RIF token abi in rif.json file.

Run the below commands in the root folder:

```
bash
mkdir abi
touch rif.json
```

Replace rootstock-rs/abi/rif.json file with the RIF Abi below:

```
bash
[ { "constant": true, "inputs": [ { "name": "owner", "type": "address" } ], "name": "balanceOf", "outputs": [ {
"name": "balance", "type": "uint256" } ], "payable": false, "stateMutability": "view", "type": "function" }, {
"constant": false, "inputs": [ { "name": "to", "type": "address" }, { "name": "value", "type": "uint256" } ],
"name": "transfer", "outputs": [ { "name": "", "type": "bool" } ], "payable": false, "stateMutability":
"nonpayable", "type": "function" } ]
```

Update the rootstock-rs/src/main.rs file with this program:

```
rs
use alloy::providers::{Provider, ProviderBuilder};
use alloy::sol;
use alloy::primitives::{ address, utils::formatunits };

sol!(
    #[allow(missingdocs)]
    #[sol(rpc)]
    RIF,
    "abi/rif.json"
);
```

```
#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Set up the HTTP transport which is consumed by the RPC client.
    let rpcurl = "https://rpc.testnet.rootstock.io/{YOURAPIKEY}".parse()?;

    // Create a provider with the HTTP transport using the reqwest crate.
    let provider = ProviderBuilder::new().onhttp(rpcurl);

    // Address without 0x prefix
    let alice = address!("8F1C0185bB6276638774B9E94985d69D3CDB444a");

    let rbtc = provider.getbalance(alice).await?;

    let formattedbalance: String = formatunits(rbtc, "ether"?);

    println!("Balance of alice: {formattedbalance} rbtc");

    // Using rif testnet contract address
    let contract = RIF::new("0x19f64674D8a5b4e652319F5e239EFd3bc969a1FE".parse()?, provider);

    let RIF::balanceOfReturn { balance } = contract.balanceOf(alice).call().await?;

    println!("Rif balance: {:?}", balance);

    Ok(())
}
```

```
:::info[Info]
```

Replace APIKEY with your RPC API Key. To get an APIKEY, see the RPC Docs. Also replace RIF Testnet contract addresses with your own address as you would be required to use a private key later.

```
:::
```

Note: Run the cargo command in the root of the project:

```
bash
cd rootstock-rs
cargo run
```

You should get the following response:

```
bash
Finished dev profile [unoptimized + debuginfo] target(s) in 3.01s
Running target/debug/rootstock-alloy
Balance of alice: 0.315632721175825996 rbtc
Rif balance: 183000000000000000000
```

Send a transaction

The following program sends tRBTC from one account to the other using TransactionRequest Builder.

Running this program will require setting your PRIVATEKEY env variable and then run the program using cargo run in root.

```
bash
cd rootstock-rs
PRIVATEKEY=0x12... cargo run
```

Replace PRIVATEKEY with your private key in the command above to run this program.

You should see the following response:

```
bash
% cargo run
Finished dev profile [unoptimized + debuginfo] target(s) in 0.35s
  Running target/debug/rootstock-alloy
Balance of alice: 0.315632721175825996 rbtc
Rif balance: 1830000000000000000000
```

Next, update the rootstock-rs/src/main.rs file with this program:

```
rs
use alloy::{
    network::{EthereumWallet, TransactionBuilder},
    primitives::{address, U256},
    providers::{Provider, ProviderBuilder},
    rpc::types::TransactionRequest,
    signers::local::PrivateKeySigner,
};
use eyre::Result;
use std::env;

#[tokio::main]
async fn main() -> Result<()> {

    // Get private key
    let mut pk = String::new();

    match env::var("PRIVATEKEY") {
        Ok(value) => {
            pk.pushstr(value.asstr());
        },
        Err(e) => {
            panic!("Private key not setup");
        },
    }
}

// Set up the HTTP transport which is consumed by the RPC client.
let rpcurl = "https://rpc.testnet.rootstock.io/{YOURAPIKEY}".parse()?;

let signer: PrivateKeySigner = pk.parse().unwrap();

let wallet = EthereumWallet::from(signer);
```

```

// Create a provider with the HTTP transport using the request crate.
let provider = ProviderBuilder::new()
    .wallet(wallet)
    .onhttp(rpcurl);

// Get chain id
let chainid = provider.getchainid().await?;

// Create two users, Alice and Bob.
// Address without 0x prefix
let alice = address!("8F1C0185bB6276638774B9E94985d69D3CDB444a");
let bob = address!("8F1C0185bB6276638774B9E94985d69D3CDB444a");

let nonce = provider.gettransactioncount(alice).await?;

// Build a transaction to send 100 wei from Alice to Bob.
let tx = TransactionRequest::default()
    .withfrom(alice)
    .withto(bob)
    .withchainid(chainid)
    .withnonce(nonce)
    .withvalue(U256::from(100)) // To see value in rbtc: 100 / 10 18 RBTC
    .withgasprice(65164000) // provider.estimategas(&tx).await? 1.1 as u128 / 100)
    .withgaslimit(21000); // Change this value if you face gas related issues

// Send the transaction and wait for the receipt.
let pendingtx = provider.sendtransaction(tx).await?;

println!("Pending transaction... {}", pendingtx.txhash());

// Wait for the transaction to be included.

let receipt = pendingtx.getreceipt().await?;

println!(
    "Transaction included in block {}",
    receipt.blocknumber.expect("Failed to get block number")
);

// asserteq!(receipt.from, alice);
// asserteq!(receipt.to, Some(bob));

Ok(())
}

```

- ERROR: deserialization error: missing field effectiveGasPrice at line 1 column 959
    - It's expected that you will encounter a missing effectiveGasPrice error.
    - Kindly ignore above error. RSKj team is familiar with this error and fix would be part of new release.
- This error does not block the sending of a transaction. Transaction will be mined successfully.

## Transfer ERC20 Token

This program setups up wallet with provider and sends RIF from one account to the other. Run this



program using: cargo run.

Update the rootstock-rs/src/main.rs file with this program:

```
rs
use alloy::{
    network::{EthereumWallet},
    primitives::{address, U256},
    providers::{Provider, ProviderBuilder},
    signers::local::PrivateKeySigner,
};
use eyre::Result;
use std::env;
use alloy::sol;

// Codegen from ABI file to interact with the contract.
// Make a abi directory in the root of the project and put RIF token abi in rif.json file.

sol!(
    #[allow(missingdocs)]
    #[sol(rpc)]
    RIF,
    "abi/rif.json"
);

// See the contents of rootstock-rs/abi/rif.json file below.
/
[ { "constant": true, "inputs": [ { "name": "owner", "type": "address" } ], "name": "balanceOf", "outputs": [ {
"name": "balance", "type": "uint256" } ], "payable": false, "stateMutability": "view", "type": "function" }, {
"constant": false, "inputs": [ { "name": "to", "type": "address" }, { "name": "value", "type": "uint256" } ],
"name": "transfer", "outputs": [ { "name": "", "type": "bool" } ], "payable": false, "stateMutability":
"nonpayable", "type": "function" } ]
/

#[tokio::main]
async fn main() -> eyre::Result<()> {
    // Get private key
    let mut pk = String::new();

    match env::var("PRIVATEKEY") {
        Ok(value) => {
            pk.pushstr(value.asstr());
        }
        Err(e) => {
            panic!("Private key not setup");
        }
    }

    // Set up the HTTP transport which is consumed by the RPC client.
    let rpcurl = "https://rpc.testnet.rootstock.io/{YOURAPIKEY}".parse()?;

    let signer: PrivateKeySigner = pk.parse().unwrap();

    let wallet = EthereumWallet::from(signer);
```

```

// Create a provider with the HTTP transport using the request crate.
let provider = ProviderBuilder::new()
    .wallet(wallet)
    .onhttp(rpcurl);

// Address without 0x prefix
let alice = address!("8F1C0185bB6276638774B9E94985d69D3CDB444a");

let nonce = provider.gettransactioncount(alice).await?;
let chainid = provider.getchainid().await?;

let contract = RIF::new(
    "0x19f64674D8a5b4e652319F5e239EFd3bc969a1FE".parse()?,
    provider,
);

let RIF::balanceOfReturn { balance } = contract.balanceOf(alice).call().await?;

println!("Rif balance: {:?}", balance);

// Transfer
let amount = U256::from(100);
let receipt = contract
    .transfer(alice, amount)
    .chainid(chainid)
    .nonce(nonce)
    .gasprice(65164000) // gas price: provider.estimategas(&tx).await? 1.1 as u128 / 100)
    .gas(25000) // Change this value according to tx type if you face gas related issues
    .send()
    .await?
    .getreceipt()
    .await?;

println!("Send transaction: {}", receipt.transactionhash);

Ok(())
}

```

Run the below command to transfer an ERC20 Token:

```

bash
cd rootstock-rs
PRIVATEKEY=0x12... cargo run

```

> Note to replace PRIVATEKEY with your private key in the command above to run this program.

For more details, see the complete code example

See foundry codebase for more advanced usage of Rust and Alloy to interact with EVM compatible blockchains including Rootstock.

Useful Resources

- Alloy website
- See Alloy reference documentation
- Code examples using Alloy visit this repo
- Alloy GitHub repo

# rootstock-tenderly.md:

---

sidebarposition: 5

title: "Virtual Testnets: Use Tenderly to Fork the Rootstock Mainnet for Development"

sidebarlabel: Create Virtual Testnets on Rootstock using Tenderly

tags: [rsk, rootstock, tenderly, tutorials, resources, smart contracts, virtual testnet, mainnet]

description: "Tenderly's virtual testing environment allows the creation of simulated networks, managing account balances, and manipulating contract storage – all without needing to interact with the Rootstock mainnet or testnet."

---

Need a safe and efficient way to test your dApp features before deploying them on the Rootstock mainnet? Look no further than virtual testnets! These simulated blockchain environments, offered by platforms like Tenderly, provides a perfect testing ground for developers.

Imagine a clone of the Rootstock mainnet, where you can experiment freely without using real tokens. Virtual testnets mimic the behavior of a real blockchain, allowing you to deploy your dApps, interact with smart contracts, and debug transactions – all within a controlled setting.

:::note[Tenderly Virtual Testnets]

Tenderly's virtual testing environment allows the creation of simulated networks, managing account balances, and manipulating contract storage – all without needing to interact with the Rootstock mainnet or testnet.

:::

In this tutorial, we will do the following:

- Set up a Tenderly Account
- Setup a Virtual Test network
- Fork the Rootstock Mainnet: Create a simulated network that replicates the current state of the Rootstock mainnet.
- Integrating a project
- Easily revert to a previous network state for more controlled testing scenarios by using snapshots.
- Set account balances (native token & ERC20).
- Override contract storage

Prerequisites

- A Tenderly Account: Sign up for a free Tenderly account to access the Virtual Testnet features
- Basic familiarity with Smart Contracts.

Getting Started

Creating a Project:

Sign up or Log in to your Tenderly account and create a new project specifically for Rootstock.

!Rootstock - Tenderly Dashboard

## Setting up a Virtual Testnet

In the left navigation, choose Virtual Testnets and click on the button “Create Virtual Testnet”.

Use the configuration below to setup a virtual testnet:

Parent network: RSK  
Network name: Any name of choice  
Chain ID: Default  
Public Explorer: Off, select use latest block.

Your setup should look like the one in the image below; Click Create.

!Rootstock - Tenderly Dashboard

:::note[RPC Configuration]

On Tenderly, there are two RPC configurations: Public RPC and Admin RPC.

- The Public RPC allows standard RCP interactions with the blockchain, such as deploying contracts and interacting with smart contracts.
- The Admin RPC enables you to modify the Testnet network state, including account balances, block numbers, and storage, to support your development requirements.

...

!Rootstock - Tenderly Dashboard

## Integrate a Project

On the left menu, select Integrate to learn how to add the TestNet you just created to your project. Examples are available for Hardhat, Foundry, and other frameworks. In addition to setup examples, you'll find instructions on how to send transactions and fund accounts.

To interact with Tenderly in your Hardhat project, you will need the hardhat-tenderly package, which you can install by running the following command:

```
bash
npm i @tenderly/hardhat-tenderly
```

Next, import and set up Tenderly in your hardhat.config file:

```
js
import as tdly from "@tenderly/hardhat-tenderly";
require("dotenv").config();
```

> This will enable Tenderly's features in your Hardhat project.

## Using Snapshots for Reliable Testing

Using Tenderly's admin RPC, you can capture snapshots of your testnet's current state and revert to these snapshots as needed. This feature is particularly useful when running multiple tests that modify the testnet state. By taking a snapshot before executing a test, making changes during the test, and then

reverting to the snapshot afterward, you ensure that each test starts with a clean, consistent network state. This approach enhances the reliability and repeatability of your tests.

## Take snapshot

To take a snapshot, add the code below;

```
bash
const TENDERLYRPC = 'https://virtual.rsk.rpc.tenderly.co/{id}'

export async function takeSnapshot() {
  const requestOptions = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      jsonrpc: '2.0',
      method: 'evmsnapshot'
    })
  };
  const response = await fetch(TENDERLYRPC, requestOptions);
  const snapshotId = (await response.json()).result;
  return snapshotId;
}
```

:::info[Info]

The function `takeSnapshot` creates a snapshot on an EVM network using a Tenderly RPC endpoint. It sends a POST request with JSON RPC data and returns the snapshot ID.

...

## Revert to snapshot

To revert to snapshot, add the code below;

```
bash
export async function revertToSnapshot(snapshotId: string) {
  const requestOptions = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      jsonrpc: '2.0',
      method: 'evmrevert',
      params: [snapshotId]
    })
  };
  await fetch(TENDERLYRPC, requestOptions);
}
```

:::info[Info]

This takes a snapshotId as a parameter and uses it to revert the EVM state to the specified snapshot. It constructs a POST request with the necessary JSON-RPC data, sends it to the Tenderly RPC endpoint, and awaits the response. Essentially, this function rolls back the EVM to a previous state captured in the snapshot.

...

## Set account balances (native token & ERC20)

Using tenderly admin rpc, you can set both native token and any ERC20 balances for any account in the TestNet. This allows you to create the test scenarios that you need.

### Set native tokens balance

```
bash
export async function setNativeBalance(walletAddress: string, amount: BigInt) {
  const amountHex = ethers.toQuantity(amount.toString());
  await ethers.provider.send("tenderlysetBalance", [
    walletAddress,
    amountHex,
  ]);
}
```

### Set ERC20 balance

To set an account balance, copy and paste the code below;

```
bash
export async function setErc20Balance(erc20Address: string, walletAddress: string, amount: BigInt) {
  const amountHex = ethers.toQuantity(amount.toString());
  await ethers.provider.send("tenderlysetErc20Balance", [
    erc20Address,
    walletAddress,
    amountHex,
  ]);
}
```

## Override the contract storage

Tenderly allows you to override smart contract storage in the TestNet, but we need to know the memory slot of the storage variable we want to modify. For value type variables like address or integer, we can just count the position of the variable in the contract.

```
bash
pragma solidity ^0.8.0;
contract SimpleStorage {
  uint256 public value1; // Stored at slot 0
  uint256 public value2; // Stored at slot 1

  function setValues(uint256 value1, uint256 value2) public {
```

```

    value1 = value1;
    value2 = value2;
  }
}

```

Once you have identified the slot you need to modify, you can set a new value for that slot. Remember to convert the value to 32 bytes, as this is the memory size of a storage slot.

```

bash
async function overrideContractStorage() {
  // where to override
  let storageSlot = 4;
  const abiCoder = new ethers.AbiCoder();
  const storageSlot32Bytes = abiCoder.encode(["uint256"], [storageSlot]);

  // what to override
  const newValue = 2;
  const newValue32Bytes = abiCoder.encode(["uint256"], [newValue]);

  // override
  await ethers.provider.send("tenderlysetStorageAt", [
    addresses.oldMultisig,
    storageSlot32Bytes,
    newValue32Bytes,
  ]);
}

```

:::info[Info]

For reference types like dynamic arrays, we need to follow the principles outlined in the "Layout of State Variables in Storage" to find the storage slot. Another more empirical approach is to read your contract's storage slot by slot to identify the variables stored in each slot, using for example the ethers method `getStorageAt`.

:::

...and that's it, in this guide, we have successfully created a project, setup a Virtual Testnet, forked the Rootstock Mainnet by creating a simulated network that replicates the current state of the Rootstock mainnet. Learned how to integrate an existing project, leverage Snapshots, take snapshots, revert snapshot, set account balances (native token & ERC20), and override the contract storage.

# ethereum-dapp.md:

```

---
sidebarposition: 2
title: Port an Ethereum dApp to Rootstock
sidebarlabel: Port an Ethereum dApp to Rootstock
tags: [rsk, rootstock, resources, tutorials, port to rootstock, Ethereum, dApps, smart contracts]
description: "Porting an Ethereum decentralized application (dApp) to Rootstock (RSK) presents an exciting opportunity to leverage the benefits of the Rootstock network, a Bitcoin L2 compatible with

```

Ethereum. This guide will walk you through porting an Ethereum dApp to the Rootstock network using the Hardhat Ignition deployment tool and leveraging the compatibility between Solidity (used for Rootstock) and Ethereum."

---

Porting an Ethereum decentralized application (dApp) to Rootstock presents an exciting opportunity to leverage the benefits of the Rootstock network, which is a smart contract platform secured by the Bitcoin network.

Rootstock combines Ethereum's flexibility with Bitcoin's security and scalability, offering a compelling environment for dApp development.

With Rootstock, you can bridge the gap between Ethereum and Bitcoin, bringing your existing Ethereum dApps to the Rootstock platform.

This guide will walk you through porting your Ethereum dApp to the Rootstock network using the Hardhat Ignition deployment tool and leveraging the compatibility between Solidity (used for Rootstock) and Ethereum.

## Advantages of Porting Your dApp to Rootstock

### 1. Faster Transaction Speeds

Rootstock performs transactions by merge-mining with Bitcoin. This means that Rootstock transactions benefit from the security of the Bitcoin network while achieving faster confirmation times compared to Ethereum.

### 2. Lower Gas Fees

Rootstock's gas fees are typically lower than Ethereum, averaging around \$0.052. This cost-effectiveness can be especially appealing for dApps that require frequent interactions with the blockchain.

### 3. Leveraging Bitcoin Security

Rootstock is a layer 2 on Bitcoin, which means it inherits the security of the Bitcoin network. This security model provides confidence to builders and users.

## Similarities Between Ethereum and Rootstock

### 1. Solidity as the Programming Language

Both Ethereum and Rootstock use Solidity as their primary smart contract programming language. If you're already familiar with Solidity, transitioning to Rootstock should be relatively straightforward.

### 2. EVM Compatibility

Rootstock is compatible with the Ethereum Virtual Machine (EVM). This compatibility allows developers to reuse existing Ethereum smart contracts on Rootstock with minimal modifications.

## Key Differences Between Ethereum and Rootstock

### 1. Consensus Mechanisms

Ethereum currently uses a Proof of Stake (PoS) consensus mechanism, while Rootstock employs a hybrid PoW/PoS (Proof of Stake) consensus. Rootstock's PoS component enhances scalability and



energy efficiency.

## 2. Token Standards

While Ethereum introduced popular token standards like ERC20 (fungible tokens) and ERC721 (non-fungible tokens), Rootstock has its own token standard called RRC20. Understanding the differences between these standards is crucial when porting tokens.

## 3. Network Fees

As mentioned earlier, Rootstock generally offers lower gas fees. Developers can take advantage of this cost savings when deploying and interacting with smart contracts.

## Getting Started

### Prerequisites

Before you begin, ensure that you have the following:

- Node.js:
  - Make sure you have Node.js installed. If not, you can follow the installation instructions for Windows or MacOS.
- Hardhat:
  - Install Hardhat globally using npm: `npm i -g hardhat`
- A basic knowledge of smart contracts and Solidity

### Steps to Set Up a Hardhat Project for Rootstock

1. Create a New Project: Create a folder for your project and navigate into it:

```
sh
mkdir rsk-hardhat-example
cd rsk-hardhat-example
```

2. Initialize Hardhat: Initialize your Hardhat project by running this command:

```
sh
npx hardhat init
```

3. Select project framework: Choose Create a TypeScript project when prompted as shown below. Then press enter.

```
888 888          888 888      888
888 888          888 888      888
888 888          888 888      888
88888888888 888b. 888d888 .d88888 88888b. 8888b. 888888
888 888  "88b 888P" d88" 888 888 "88b  "88b 888
888 888 .d888888 888 888 888 888 .d888888 888
888 888 888 888 888 Y88b 888 888 888 888 Y88b.
888 888 "Y888888 888  "Y88888 888 888 "Y888888 "Y888
```

```
Welcome to Hardhat v2.22.5
```

```
? What do you want to do? ...
  Create a TypeScript project
```

4. Select the project root (press enter)

```
sh
```

```
What do you want to do? · Create a TypeScript project
```

```
? Hardhat project root: › /path/to/your/project/rsk-hardhat-example
```

5. Add a .gitignore File: If you need a .gitignore file (which is recommended), create one in your project root. You can skip this step if you don't want to use Git.

```
sh
```

```
? Do you want to add a .gitignore? (Y/n) › y
```

6. Install dependencies with npm:

```
sh
```

```
? Do you want to install this sample project's dependencies with npm (hardhat  
@nomicfoundation/hardhat-toolbox)? (Y/n) › y
```

7. Configure Rootstock Networks: By now, your hardhat project should have four main artifacts besides the basic Node configuration:

- contracts/
- ignition/modules/
- test/
- hardhat.config.js

> This guide uses Hardhat version 2.22.5. For this version, the default tool for managing deployments is Hardhat Ignition.

8. Install Hardhat Ignition and TypeScript

```
sh
```

```
npm install --save-dev @nomicfoundation/hardhat-ignition-ethers typescript
```

At this point, your hardhat.config.ts should look like this:

```
typescript
```

```
import { HardhatUserConfig } from "hardhat/config";
```

```
import "@nomicfoundation/hardhat-toolbox";
```

```
import "@nomicfoundation/hardhat-ignition-ethers";
```

```
const config: HardhatUserConfig = {
```

```
  solidity: "0.8.24",
```

```
};
```

```
export default config;
```

## Configure Rootstock Networks

To configure the Rootstock networks, you'll need an RPC URL for both mainnet and testnet and a Private Key of the account that will deploy the contracts.

To get the RPCs, go to the RPC API dashboard from Rootstock Labs create an account if you don't have one, and get an API key for Rootstock testnet or Rootstock mainnet.

```
<Tabs>
  <TabItem value="contribute" label="Mainnet RPC URL should look similar to this:" default>
```

```
https://rpc.mainnet.rootstock.io/<API-KEY>
```

```
</TabItem>
  <TabItem value="contest" label="Testnet RPC URL should look similar to this:">
```

```
https://rpc.testnet.rootstock.io/<API-KEY>
```

```
</TabItem>
```

```
</Tabs>
```

The next step is to retrieve your private key. If you don't know how to get the private key to your wallet, here's a tutorial on Metamask.

Also, if you haven't added Rootstock mainnet or testnet to your Metamask Wallet, you can do it by clicking the Add Rootstock or Add Rootstock Testnet buttons in the footer of mainnet explorer or testnet explorer.

Store the RPC URLs and the Private Key

To securely store the RPC URLs, you can use a .env file or the Hardhat configuration variables. For this example, you'll use the second option.

To store this, type in the terminal in the project's root folder:

```
sh
npx hardhat vars set TESTNETRPCURL
```

And enter the value after pressing enter.

:::note

Make sure your TESTNETRPCURL value is in this format: https://rpc.testnet.rootstock.io/API-KEY

For example, https://rpc.testnet.rootstock.io/eOQAoxAI7Bt6zZ6blwOdMjQQIzKwSW-W (Where eOQAoxAI7Bt6zZ6blwOdMjQQIzKwSW-W is your API-KEY)

...

```
sh
npx hardhat vars set TESTNETRPCURL
Enter value: ·
```

Now repeat this step for your MAINNETRPCURL.

You'll see output similar to this:

The configuration variable has been stored in  
/Users/wisdomnwokocha/Library/Preferences/hardhat-nodejs/vars.json

For the Private key:

When requested to enter your private key, press enter it.

```
sh
npx hardhat vars set PRIVATEKEY
Enter value: ·
```

You'll see output similar to this:

The configuration variable has been stored in  
/Users/wisdomnwokocha/Library/Preferences/hardhat-nodejs/vars.json

Now, update your hardhat.config.ts file to include Rootstock network configurations. Here's an example of how it should look:

```
typescript
import { HardhatUserConfig } from "hardhat/config";
import "@nomicfoundation/hardhat-toolbox";
import "@nomicfoundation/hardhat-ignition-ethers";

const config: HardhatUserConfig = {
  solidity: "0.8.24", // Set your desired Solidity version

  networks: {
    // Mainnet configuration
    mainnet: {
      url: "https://rpc.mainnet.rootstock.io/<API-KEY>",
      accounts: [process.env.PRIVATEKEY],
    },

    // Testnet configuration
    testnet: {
      url: "https://rpc.testnet.rootstock.io/<API-KEY>",
      accounts: [process.env.PRIVATEKEY],
    },
  },
};

export default config;
```

Replace <API-KEY> with your actual API keys obtained from the Rootstock Labs dashboard. Also, store your private key securely (e.g., in a .env file).

### Copy Ethereum Contract Code and Tests

Copy this Ethereum contract and its tests to your Rootstock Hardhat project. Place it inside the contracts folder so the route would be contracts/SimpleStorage.sol.

#### SimpleStorage.sol

```
solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract SimpleStorage {
  uint256 public favoriteNumber;
```

```

    function store(uint256 favoriteNumber) public {
        favoriteNumber = favoriteNumber;
    }
}

```

Copy this test code and create a new file named SimpleStorage.ts inside the test folder. The route will be test/SimpleStorage.ts.

Update SimpleStorage.ts

```

typescript
import { loadFixture } from "@nomicfoundation/hardhat-toolbox/network-helpers";
import { expect } from "chai";
import hre from "hardhat";

describe("SimpleStorage", function () {
    async function deploySimpleStorageFixture() {
        const [owner] = await hre.ethers.getSigners();

        const SimpleStorage = await hre.ethers.getContractFactory("SimpleStorage");
        const simpleStorage = await SimpleStorage.deploy();

        return { simpleStorage, owner };
    }

    describe("Deployment", function () {
        it("Should deploy and initialize favoriteNumber to 0", async function () {
            const { simpleStorage } = await loadFixture(deploySimpleStorageFixture);

            expect(await simpleStorage.favoriteNumber()).to.equal(0);
        });
    });

    describe("Store", function () {
        it("Should store the value 42 and retrieve it", async function () {
            const { simpleStorage } = await loadFixture(deploySimpleStorageFixture);

            const storeTx = await simpleStorage.store(42);
            await storeTx.wait();

            expect(await simpleStorage.favoriteNumber()).to.equal(42);
        });

        it("Should store a different value and retrieve it", async function () {
            const { simpleStorage } = await loadFixture(deploySimpleStorageFixture);

            const storeTx = await simpleStorage.store(123);
            await storeTx.wait();

            expect(await simpleStorage.favoriteNumber()).to.equal(123);
        });
    });
});

```

## Compile Your Contract

To compile your contract, run this command in your terminal:

```
bash
npx hardhat compile
```

After compilation, you'll see output similar to this:

```
Generating typings for: 1 artifacts in dir: typechain-types for target: ethers-v6
Successfully generated 6 typings!
Compiled 1 Solidity file successfully (evm target: paris).
```

## Test Your Contract

To test your contract functionality, run this command in your terminal:

```
bash
npx hardhat test
```

The test results will show whether your contract behaves as expected. You should see something like this:

```
s
SimpleStorage
  Deployment
    Should deploy and initialize favoriteNumber to 0
  Store
    Should store the value 42 and retrieve it
    Should store a different value and retrieve it

3 passing (286ms)
```

## Deploying Your Contract on Rootstock Testnet

### 1. Ensure Sufficient Balance

Before deploying, ensure you have enough testnet tokens (RBTC) in your wallet. If not, obtain some from the Rootstock faucet.

### 2. Create the Deployment Script

Create a file named SimpleStorage.ts inside the ignition/modules folder. Paste the following TypeScript code into that file:

```
typescript
import { buildModule } from "@nomicfoundation/hardhat-ignition/modules";

const SimpleStorageModule = buildModule("SimpleStorageModule", (m) => {
```

```
const simpleStorage = m.contract("SimpleStorage");

return { simpleStorage };
});

export default SimpleStorageModule;
```

### 3. Deploy the Contract

In your terminal, run the deployment command:

```
bash
npx hardhat ignition deploy ignition/modules/SimpleStorage.ts --network rskTestnet
```

This TypeScript script uses Hardhat Ignition to deploy the SimpleStorage contract in a declarative way.

### 4. Confirmation and Deployment

- Confirm the deployment to the Rootstock testnet by typing “yes.”
- Hardhat Ignition will resume the existing deployment (if any) and execute the deployment process.
- You'll see a success message indicating that the contract was deployed.

```
bash
Confirm deploy to network rskTestnet (31)? ... yes
Hardhat Ignition
```

Resuming existing deployment from ./ignition/deployments/chain-31

Deploying [ SimpleStorageModule ]

```
Batch #1
Executed SimpleStorageModule#SimpleStorage
```

[ SimpleStorageModule ] successfully deployed

Deployed Addresses

SimpleStorageModule#SimpleStorage - 0x3570c42943697702bA582B1ae3093A15D8bc2115

```
:::info[Info]
```

If you get an error like IgnitionError: IGN401, try running the command again.

```
:::
```

> If you want to deploy your contract on mainnet, change rskTestnet to rskMainnet in the last command and make sure you have RBTC available in your wallet.

Verify Deployment

Visit Rootstock Testnet Explorer. Paste your contract address

(0x3570c42943697702bA582B1ae3093A15D8bc2115) into the search bar to verify successful deployment.

> If you deployed your contract on Mainnet Explorer.

# index.md:

```
---
sidebarposition: 5
title: Port a dApp from other Chains to Rootstock
sidebarlabel: Port to Rootstock
tags: [rsk, rootstock, resources, beginner, quick starts, advanced, port to rootstock, tutorials]
description: "Port a dApp from other Chains to Rootstock."
---

<Filter
values=[
{label: 'From Ethereum', value: 'ethereum'},
]>
<FilterItem
value="ethereum"
title="Port an Ethereum dApp to Rootstock"
subtitle="Ethereum"
color="orange"
linkHref="/resources/port-to-rootstock/ethereum-dapp/"
description="Porting an Ethereum decentralized application (dApp) to Rootstock presents an exciting opportunity to leverage the benefits of the Rootstock network, which is a smart contract platform secured by the Bitcoin network."
/>
</Filter>
```

# contractaddresses.md:

```
---
sidebarposition: 302
sidebarlabel: Addresses and Links
title: Token Bridge Mainnet Addresses and Links
tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]
---
```

Here, you can find a list of mainnet, testnet addresses, and ABIs used by the Token Bridge.

#### Token Addresses

- USDT: 0x31974a4970BADA0ca9BcDe2E2eE6fC15922c5334
- LINK: 0x2d850c8E369F26bc02fF4c9fFbaE2d50107395CB
- DAI: 0xdF63373ddb5B37F44d848532BBcA14DBf4e8aa53
- DOC: 0xAC3896da7940c8e4Fe9E7F8cd4475Cd2534F37d7
- USDC: 0xbB739A6e04d07b08E38B66ba137d0c9Cd270c750

#### Token Bridge Mainnet Rootstock Addresses

- Bridge: 0x9d11937e2179dc5270aa86a3f8143232d6da0e69
- Federation: 0x7eCfda6072942577D36F939aD528b366b020004b
- AllowTokens: 0xcB789036894a83a008a2AA5b3c2DDDe41D0605A9A
- MultiSigWallet: 0x040007b1804ad78a97f541bebed377dcb60e4138



## Token Bridge Mainnet Ethereum Addresses

- Bridge: 0x12ed69359919fc775bc2674860e8fe2d2b6a7b5d
- Federation: 0x5e29C223d99648C88610519f96E85E627b3ABe17
- AllowTokens: 0xA3FC98e0a7a979677BC14d541Be770b2cb0A15F3
- MultiSigWallet: 0x040007b1804ad78a97f541bebed377dcb60e4138

## Testnet Addresses and Links

- On Rootstock Testnet
  - Bridge: 0x21df59aef6175467fefb9e44fbb98911978a13f2
  - Federation: 0x73de98b3eb19cae5dfc71fb3e54f3ffd4aa02705
  - AllowTokens: 0xa683146bb93544068737dfca59f098e7844cdfa8
  - MultiSigWallet: 0x8285422f7e58ad7f7b90cb5158098edf548e7b38
- On Sepolia
  - Bridge: 0xd31e66af9d830bfc35e493929a8f6523ca2b01b1
  - Federation: 0x091e26c96e7f4aaef0d85746bb99b733ec28df90
  - AllowTokens: 0x926d302f3b6bc4d0eeea9caf6942fd7e0a9a0422
  - MultiSigWallet: 0xbee2572941ffcb2ab2e61450fecc8db75321e6c9

## List of ABIs

See the list of ABIs used

## Useful Links

- Rootstock Testnet Explorer, Faucet and Stats
  - [explorer.testnet.rootstock.io](https://explorer.testnet.rootstock.io)
  - [stats.testnet.rootstock.io](https://stats.testnet.rootstock.io)
  - [faucet.rootstock.io](https://faucet.rootstock.io)
  - [faucet.rifos.org](https://faucet.rifos.org)
- Sepolia (Ethereum Testnet), Explorer and Faucet
  - [sepolia.etherscan.io](https://sepolia.etherscan.io)
  - [alchemy.com/faucets/ethereum-sepolia](https://alchemy.com/faucets/ethereum-sepolia)

# dappguide.md:

```
---
sidebarposition: 301
sidebarlabel: Token Bridge dApp Guide
title: "Token Bridge dApp Guide - Cross-Chain Transactions"
tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]
---
```

This guide describes the steps to transfer tokens using the Web Interface for the Rootstock Token Bridge dApp. Please refer to the project documentation, if you'd like to know more about how this bridge works. It is possible to test the transfer of tokens between Rootstock Testnet and Sepolia networks, or Rootstock Mainnet and Ethereum networks using the Rootstock Tokenbridge web interface.

## Prerequisites

This will require the use of either Chrome or Chromium web browser, with one of the following wallet browsers extensions:

- Metamask using a custom network to add the Rootstock network.

- Get test tokens from the Rootstock Token Faucet
- Get Test Sepolia ETH

:::tip[Tip]

- See the Tools Section for a list of wallets compatible with Rootstock
- See the Contract addresses section for a list of contract addresses.

:::

## Get Started

Start by connecting your wallet and select the network of your choice, in this case we will use Rootstock Testnet network.



Choose from the list of available wallet types, for this guide, we will connect to a Metamask Wallet:



You should see the following screen:



Then choose the original network token that you want to transfer, enter the amount, and the receiver address.



Click the Continue button.

:::info[Info]

- For example, tRUSDT, RDAI, RUSDC, or RLINK, etc token can be obtained from the Rootstock Token Faucet.

You will need to approve the bridge contract to use the token, this will happen only once.

- Min transfer is 1RUSDT and max transfer is 250,000RUSDT

:::

Confirm transaction, fees, and confirmation time and click on Transfer Tokens from Rootstock Testnet.



:::warning[Important]

Don't use the bridge to send tokens to your exchange address, you won't be able to claim it

:::

As soon as the process starts, you will see a loader and a popup from Metamask asking to approve and confirm the transaction.



Once the tokens have crossed, you need to claim them on the Sepolia network., you will be asked to switch network to Sepolia. Click on switch network to Sepolia and approve in MetaMask.

> Switching to the opposite network is important in order to claim your tokens.



If everything worked correctly, you should see a prompt to Claim Tokens. Click on the claim button.



A confirmation popup will appear to send the claim transaction to the network, submit it. You should see a confirmation screen.



After the transaction get mined, you can see your transaction as Claimed by checking your transaction list of claims.



:::success[Success]

- You can check the token contract on the other network by clicking on the transaction hash (in this case RUSDT).

You can also confirm the funds in your wallet. To do this add a custom token on the network where the token crossed using the address mentioned before.

- You can transfer tokens in the other direction too, using the same method.

:::

# faq.md:

---

sidebarposition: 304

sidebarlabel: Token Bridge FAQs

title: "Token Bridge FAQs"

tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]

---

Find a list of frequently asked questions about the Token Bridge.

<Accordion>

<Accordion.Item eventKey="0">

<Accordion.Header as="h3">What is the Token Bridge?</Accordion.Header>

<Accordion.Body>

- The Token Bridge is an interoperability protocol which allows users to move their own Rootstock or Ethereum ERC20 Tokens between networks in a quick and cost-efficient manner.

- The UI is available at:

- Mainnet: <https://dapp.tokenbridge.rootstock.io/>

- Testnet: <https://dapp.testnet.bridges.rootstock.io/>

- !Rootstock-Ethereum Token Bridge

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="1">

<Accordion.Header as="h3">What is a Side Token (mirror ERC20)?</Accordion.Header>

<Accordion.Body>

- Side Token is an ERC777 representation of a ERC20 compatible tokens which is on another network (could be on Ethereum or Rootstock network). The Side Token displays the exact same properties as the standard ERC20 token and allows it to be used in all the same places as ERC20.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="2">

<Accordion.Header as="h3">What is the purpose of having a Side Token?</Accordion.Header>

<Accordion.Body>

- Side Tokens are minted to prove cross chain bridges can work in a safe and secure manner with 2 standalone blockchains. We believe this kind of interoperability technology offers a lot of possibilities for smart contract owners, as they may prefer to do certain operations in one chain, and others in another one. By connecting blockchains with these bridges you allow for a variety of new use cases that never existed before.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="3">

<Accordion.Header as="h3">Will the supply of the original token will increase as a result of Side Tokens?</Accordion.Header>

<Accordion.Body>

- No! It's important to note that there will be no increase in the original tokens. The existing amount of circulating original tokens will stay the same and simply be distributed across 2 networks (Rootstock Network & Ethereum network) instead of 1.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="4">

<Accordion.Header as="h3">What is the difference between original tokens and Side Tokens?</Accordion.Header>

<Accordion.Body>

- The original token lives on the network that it was deployed for example Ethereum, while the Side Token is a representation of the original token on the other network, for example Rootstock.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="5">

<Accordion.Header as="h3">What is the Side Token Contract Address, Symbol, and of Decimal Places in order to add it as a Custom Coin on MyEtherWallet?</Accordion.Header>

<Accordion.Body>

- The symbol of the Side Token is the original token symbol with an r prefix if it is created in Rootstock or an e prefix if it is created in Ethereum. For example, if we cross the RIF token from Rootstock to Ethereum, the Side Token symbol would be eRIF.

- The number of decimal places will be 18. These are the 'addresses' of the deployed contracts in the different networks.

</Accordion.Body>

</Accordion.Item>

<Accordion.Item eventKey="6">

<Accordion.Header as="h3">How do I transform my original tokens to Side Tokens?</Accordion.Header>

<Accordion.Body>

- The Token Bridge will be a public dApp where users will be able to access by using Liquidity Wallet or Metamask. You will be able to send your original tokens and receive an equivalent amount of Side Tokens on the other network. By toggling the network on Metamask, you're also able to transfer the other way around, by sending Side Tokens and receive original tokens.

</Accordion.Body>

</Accordion.Item>

```
<Accordion.Item eventKey="7">
  <Accordion.Header as="h3">If I sell my Side Tokens, what happens to my original
tokens?</Accordion.Header>
  <Accordion.Body>
    - Upon receiving your Side Tokens, you no longer own your original tokens. The moment you use the
bridge to send them to the other network (Rootstock or Ethereum), they are locked up and stored in the
contract address. Thus you effectively have no original tokens on the original network and now have Side
Tokens on the other network.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="8">
  <Accordion.Header as="h3">Is there a limit on how many tokens can be bridged
over?</Accordion.Header>
  <Accordion.Body>
    - There is no limit on the total. Hypothetically, the entire circulating supply can be bridged, though this
is unlikely happen as there will be strong use cases to use the tokens on either network.
    - However, there is a minimum and maximum amount per transaction and daily quotas. You can see
them on https://dapp.tokenbridge.rootstock.io/
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="9">
  <Accordion.Header as="h3">Can any token be bridged over?</Accordion.Header>
  <Accordion.Body>
    - Only whitelisted tokens can cross the bridge, this curated list is used to avoid malicious contracts
and DDoS attacks.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="10">
  <Accordion.Header as="h3">What are the fees for converting original tokens to Side Tokens and
vice-versa? Who will be paying these fees?</Accordion.Header>
  <Accordion.Body>
    - There is a 0.2% fee charge when crossing the tokens, this fee goes to the validators as payment for
crossing the transactions.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="11">
  <Accordion.Header as="h3">How many confirmations are required to convert the original tokens to
Side tokens and vice-versa?</Accordion.Header>
  <Accordion.Body>
    - Confirmations depends on the amount being crossed:
    - Small amounts needs 60 confirmations on the Rootstock Mainnet, and 120 confirmations on the
Ethereum Mainnet.
    - Medium amounts needs 120 confirmations on the Rootstock Mainnet, and 240 confirmations on the
Ethereum Mainnet.
    - Large amounts needs 2880 confirmations on the Rootstock Mainnet, and 5760 confirmations on
the Ethereum Mainnet.
    - > Note that the values of small, medium, and large amount are defined per token basis, and may
change over time.
    - > You can see these amounts defined in the Token List.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="12">
  <Accordion.Header as="h3">How does the Token Bridge work?</Accordion.Header>
  <Accordion.Body>
    - The Token Bridge functionality is quite unique, yet simple to understand. The ratio of tokens during
```

network transfer always remains 1:1 and behaves in the following manner:

- When original tokens are moved to the other network
  - Original tokens are locked in the Token Bridge smart contract
  - Side Tokens are minted and assigned to the same address that originally called the bridge
- When Side Tokens are moved back from the other network
  - Side Tokens are burned
  - Original tokens are unlocked in the Token Bridge smart contract, and transferred to the same

address that originally called the bridge contract.

</Accordion.Body>

</Accordion.Item>

</Accordion>

# index.md:

---

sidebarposition: 4

sidebarlabel: Token Bridge

title: Rootstock Token Bridge

tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]

---

Safely move your ERC20 tokens between Rootstock and Ethereum with the Tokenbridge dApp. This user-friendly interface lets you interact with the Token Bridge contracts directly. It is available on Mainnet or Testnet.

## Rationale

Cross chain events are very important in the future of cryptocurrencies. Exchanging tokens between networks allows the token holders to use them in their favorite chain without being restricted to the network choice of the contract owner. Moreover, this also allows layer 2 solutions to use the same tokens on different chains. The combination of token bridges and stable coins creates a great way of payment with low volatility across networks.

## Overview

We have a bridge smart contract on each network, the bridge on one chain will receive and lock the ERC20 tokens, then it will emit an event that will be served to the bridge on the other chain. There is a Federation in charge of sending the event from one contract to the other. Once the bridge on the other chain receives the event from the Federation, it mints the tokens on the mirror ERC20 contract. See the FAQs to learn more about how it works!



The bridge contracts are upgradeable, this enables a smoother move to a more decentralized bridge in the future. This is the token bridge repository

## Usage

You can use the 'Token Bridge dApp' together with Metamask with custom network to move tokens between networks.

Follow the dApp guide for more details on how to use the token bridge.

Alternatively, you can use a wallet or web3js with the ABI of the contracts. See 'interaction guide using MyCrypto' for more information on how to use the bridge.

## Developers

### Contracts

Here are the 'addresses' of the deployed contracts in the different networks.

The smart contracts used by the bridge and the deploy instructions are in the token bridge repository in the 'bridge folder'.

The ABI to interact with the contracts are in the 'abis folder'

### Federation

There is a federation in charge of notifying the events that have happened in the bridge between one chain and the other. The federation is composed of the creators of the token contracts who want to enable their token for crossing.

See in the 'token bridge federator repository' for more information.

# troubleshooting.md:

---

sidebarlabel: Token Bridge Troubleshooting

title: 'Rootstock Token Bridge Troubleshooting Guide'

description: 'Having issues crossing your tokens on the token bridge? See the troubleshooting guide for help.'

tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]

---

See the Token Bridge FAQs

Visit the Mainnet Token Bridge or the Testnet Token Bridge

```
<!-- <div class="rsk-token-bridge-support">
  <div class="rsk-token-bridge-support-input-area">
    <div>
      <label>Transaction Hash</label>
      <br />
      <input name="txHash" id="rsk-token-bridge-support-txHash" type="text" />
    </div>
    <div>
      <label>Crossing from</label>
      <br />
      <select name="fromNetwork" id="rsk-token-bridge-support-fromNetwork">
        <option value="ethereum-mainnet">Ethereum to Rootstock</option>
        <option value="rsk-mainnet">Rootstock to Ethereum</option>
      </select>
    </div>
  </div>
</div>
```

```

</div>
<div>
  <label>Wallet</label>
  <br />
  <select name="walletName" id="rsk-token-bridge-support-walletName">
    <option value="metamask">MetaMask</option>
    <option value="liquidity">Liquidity</option>
  </select>
</div>
<div>
  <button id="rsk-token-bridge-support-check-button">Check &hellip;</button>
</div>
</div>
<div class="rsk-token-bridge-support-output-area">
</div>
</div>

```

> Note that what follows below are generic troubleshooting queries.

> To see more specific information, use the form above. -->

Transferred tokens from Ethereum, and after 24 hours have not received tokens on Rootstock

Network: ETH to Rootstock

When: Current Block - Transaction Block Number < 5760

Answer: 24 hours is an approximation, it is not fixed. Wait until 5760 blocks have past since the transaction block number, plus 5 minutes.

Transferred tokens from Ethereum, and after 24 hours have not received tokens on Rootstock

Network: ETH to Rootstock

When: Current Block - Transaction Block Number > 5760

Answer: Look in the Rootstock Explorer at the SAME ADDRESS on Rootstock. If you do not see the correct balance in the tokens tab, please share your Transaction Hash in the #token-bridge channel on Rootstock Discord (go to Discord Community to join).

Transferred tokens from Rootstock, and after 24 hours have not received tokens on Ethereum

Network: Rootstock to ETH

When: Current Block - Transaction Block Number < 2880

Answer: 24 hours is an approximation, it is not fixed. Wait until 5760 blocks have past since the transaction block number, plus 5 minutes.

Transferred tokens from Rootstock, and after 24 hours have not received tokens on Ethereum

Network: Rootstock to ETH

When: Current Block - Transaction Block Number > 2880

Answer: Look in Etherscan at the SAME ADDRESS on Rootstock. If you do not see the correct balance



in the tokens tab, please share your Transaction Hash in the #token-bridge channel on Rootstock Discord (go to [Discord Community](#) to join).

Transferred tokens from Ethereum to Rootstock, but do not see them in my wallet

Network: ETH to Rootstock

When: always

Answer: Rootstock has a different derivation path (m/44'/137'/0'/0) from Ethereum (m/44'/60'/0'/0). Software wallets respects this convention. Copy your mnemonic or private key and use Metamask and add Rootstock as custom network, to get the same address as ethereum.

Transferred tokens from Rootstock to Ethereum, but do not see them in my wallet

Network: Rootstock to ETH

When: always

Answer: Rootstock has a different derivation path (m/44'/137'/0'/0) from Ethereum (m/44'/60'/0'/0). Software wallets respects this convention. Copy your mnemonic or private key and use MyEtherwallet or My Crypto with the Rootstock derivation path m/44'/137'/0'/0 to get the same address as Rootstock.

Why does it take 24 hours? Can it be faster?

Network: Both

When: always

Answer: This is for security measures. 24 hours is an approximation, it is not exact. We are working to reduce this time in the next version.

Why can't I choose the address?

Network: Both

When: always

Answer: Currently, it uses the token bridge always sends tokens to the same address on the other blockchain network, and so the sender and the receiver will always have the same address. You will have the option to send to another address in the next version.

Metamask threw an error

Network: ETH

When: always

Answer: This is usually a timeout as the Transaction was not mined on the time expected by Metamask. This does not mean that transaction has not been mined. Please share your Transaction Hash in the #token-bridge channel on Rootstock Discord (go to [Discord Community](#) to join).

I don't see my transaction on the Token Bridge list

Network: N/A

When: always

Answer: The list is stored in the local cache, so it's not shared across devices, and it's erased if you clear your browser cookies and temporary files. You can be sure that if the transaction is mined the tokens will cross no matter what the list says. If this is not the reason why it is not there please let us know in the #token-bridge channel on Rootstock Discord (go to Discord Community to join).

I used the Sovryn Token Bridge

Network: N/A

When: always

If you have used bridge.sovryn.app, note that this is not the same as the Rootstock Token Bridge. To get support, please ask on the Sovryn discord group.

I sent Rootstock tokens to an Ethereum address

Network: N/A

When: always

Note that if you have tokens on the Rootstock network, such as RIF or USDRIF, including "crossed" tokens such as rUSDT or rDAI, you should not send them to an Ethereum address in a regular transaction. This does not work! Instead, you should use the Rootstock Token Bridge to cross the tokens from one blockchain to the other.

If you have done this already, and sent the tokens to an address that is not under your control - where you do not have the private key or the seed phrase - then you have burnt the tokens, and they are not recoverable. If you have done this already, and sent the tokens to an address that is not under your control - where you do have the private key or the seed phrase - then it may be possible to recover your tokens.

I have multiple wallets installed, but i'm only given one option

Network: N/A

When: always

Decentralised apps on websites, such as the Rootstock Token Bridge, interact with the blockchain network through a standard interface known as a web3 provider. Each browser wallet attempts to "inject" a web3 provider as soon as it is loaded. This means that if you have multiple browser extensions doing the same thing, one of them will override the other(s).

In order to avoid this problem, and if you already have multiple wallets installed,

is to choose which wallet you wish to use, and disable the other ones. To do this in in Chrome, enter `chrome://extensions/` in your address bar, which brings you to a settings screen that lists all of the browser extensions that you have installed. Click on the toggle button to disable all of the browser extensions that inject web3 providers, except for the one that you wish to use. After this go to the Rootstock token bridge again, and refresh.

# usingmycrypto.md:

```
---
sidebarposition: 303
sidebarlabel: Interact with dApp using MyCrypto
title: Interact with dApp using MyCrypto
tags: [resources, tokenbridge, blockchain, bridges, tokens, ethereum, rootstock, rsk]
---
```

This guide describes the necessary steps to perform a token transfer between two blockchain networks, which we will refer to as Mainchain and Sidechain, through interaction with special contracts that make up a subsystem called Token Bridge.

The test performed uses the Rootstock (with a local regtest node) and Ethereum (through the Ganache client) nodes as Mainchain and Sidechain respectively. The demonstration images to interact with both Blockchain were taken from the MyCrypto application. Alternatively, MyEtherWallet or similar can be used which will give the same results.

## Preconditions

The following list summarizes the tools and components that are necessary to go through this guide.

- Mainchain network
- Sidechain network
- Wallet that allows to interact with contracts
- Mainchain account with funds
- Contract addresses and JSON ABIs in Mainchain for IERC20 token (or similar) and Bridge contract
- Contract address and JSON ABI in Sidechain for Bridge contract
- Federator process crossing transactions from Mainchain to Sidechain

In particular for this use case, it was used:

- Rootstock (regtest node)
- Ethereum (through Ganache)
- MyCrypto

## Setup

Start by connecting the blockchain interface client to the Mainchain network. In this case the connection is to a custom local node that was previously started.



---

Then access your account using one of the methods available in your application. Make sure to have funds available before continuing.

!"Wallet Access"

Token transfer

The first step to perform the cross-transfer consists in the interaction with the contract located in the Mainchain that contains the tokens to be sent. In this case we will use an IERC20 contract as an example, but it is not subject to any custom functionality so any other ERC20-based contract can be used.

To continue, enter the address of the contract and its JSON ABI interface

!"Access Contract"

---

Afterward select the approve method and complete the parameters with the information of the recipient and the amount we want to send in unit of wei. The spender address will be the address of the so-called Bridge contract in the Mainchain network that will be used as an intermediary for the transfer.

!"Contract Approve"

---

Then confirm the gas price, write and sign the transaction and finally send it. You might be asked to enter the wallet once again before confirming.

!"Transaction Write"

!"Transaction Send"



---

As a result, the corresponding transaction identifier will be obtained. It is recommended to wait for the transaction to be mined and confirmed. You can go to the TX Status section to verify its status.

!"Transaction Status"

---

Receive tokens

Next, access the Bridge contract in the Mainchain, entering its address (which is where we originally sent the tokens to) and the JSON ABI.

!"Access Contract Bridge"

---

On this occasion, invoke the receiveTokens method placing the IERC20 contract address in the tokenToUse input, and the amount, in wei, that we wish to receive.

!"Contract Receive"

---

Again, write, sign and confirm the transaction and wait for it to be approved.



---

Once this transaction is processed on Bridge contract, the federator service will identify the event and cross the information to the Sidechain. The federator may be configured to wait a few blocks before transferring transactions.

## Switch networks

The next step is to connect the Blockchain interface client to the Sidechain, where the tokens will be received. In this case we use the Ganache client with the corresponding contracts previously deployed.

Then connect to the Bridge's Sidechain contract using the corresponding address and JSON ABI once again. This time call the mappedTokens method, passing as a parameter the address of the IERC20 contract of the Mainchain that was previously used. The result of this operation will be the address of the associated contract on the Sidechain that holds the transferred tokens.

!"Contract Mapped Tokens"

---

## Validating result

Using the address obtained from the previous step (0x1684e1C7bd0225917C48F60FbdC7f47b2982a3C2), and the IERC20 ABI interface, let's connect to the contract.

!"Access Contract Sidetoken"

---

To confirm that the transfer was successful, verify the balance of the account used to send the funds from the Mainchain. Invoke the balanceOf method and note that the value has increased in the Sidechain.

!"Contract Balance"

---

Similarly, verify the contract symbol. Notice that in this case the value is eMAIN where 'e' refers to a transfer to the Ethereum Sidechain.

!"Contract Symbol"

# faqs.md:

---

sidebarposition: 1300  
sidebarlabel: FAQs  
title: "Frequently Asked Questions (FAQs)"  
description: "Frequently asked questions."  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

Here, you can find a list of frequently asked questions (FAQs) about the 2 way peg application.

```
<Accordion>
  <Accordion.Item eventKey="0">
    <Accordion.Header as="h3">1. What are the requirements to use 2wp-app?</Accordion.Header>
    <Accordion.Body>
      - To know more about the requirements, see prerequisites
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="1">
    <Accordion.Header as="h3">2. What are the common errors in peg-out
transactions?</Accordion.Header>
    <Accordion.Body>
      - To know more about the common errors, see common errors.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="2">
    <Accordion.Header as="h3">3. How do I derive a BTC private key after sending RBTC through the
PowPeg?</Accordion.Header>
    <Accordion.Body>
      - See section on deriving electrum for how to export a private key.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="3">
    <Accordion.Header as="h3">4. What is the difference between SegWit and Legacy
addresses?</Accordion.Header>
    <Accordion.Body>
      - Legacy address is the original BTC address while SegWit is the newer address format with lower
fees. SegWit means Segregated Witness, where Segregated is to separate and Witness is the
transaction signatures involved with a specific transaction.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="4">
    <Accordion.Header as="h3">5. What type of addresses do I need to perform a peg
in?</Accordion.Header>
    <Accordion.Body>
      > - For information on the type of addresses to use when performing a peg-in transaction
      > - See the supported addresses page.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="5">
    <Accordion.Header as="h3">6. Why use the 2 way peg instead of the PowPeg directly?
</Accordion.Header>
    <Accordion.Body>
      > - The 2 way peg app has a lot of benefits including enabling easier and simplified peg-in
transactions.
      > - See why use the 2 way peg app? for a list of benefits when you use the application.
    </Accordion.Body>
```

```
</Accordion.Item>
<Accordion.Item eventKey="6">
  <Accordion.Header as="h3">7. What are the supported browsers to use
2wp-app?</Accordion.Header>
  <Accordion.Body>
    > - To know more about the supported browsers
    > - See the supported browsers.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="7">
  <Accordion.Header as="h3">8. What are the supported wallets to use 2wp-app?</Accordion.Header>
  <Accordion.Body>
    > - To know more about the requirements;
    > - See supported wallets.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="8">
  <Accordion.Header as="h3">9. How long does it take for a native pegin transaction to
complete?</Accordion.Header>
  <Accordion.Body>
    > - native pegin needs 17 hours to be completed
    > - !Read popup info
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="9">
  <Accordion.Header as="h3">10. How long does it take for a native pegout transaction to
complete?</Accordion.Header>
  <Accordion.Body>
    > - native pegin needs 34 hours to be completed
    > - !Read popup info
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="10">
  <Accordion.Header as="h3">11. What are the min and max for pegin
transaction?</Accordion.Header>
  <Accordion.Body>
    > - The minimum values allowed when creating a peg-in transaction is 0.005 BTC.
    > - The maximum values allowed when creating a peg-in transaction is 10 BTC.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="11">
  <Accordion.Header as="h3">12. What are the min and max for pegout
transaction?</Accordion.Header>
  <Accordion.Body>
    > - The minimum values allowed when creating a peg-in transaction is 0.004 RBTC.
    > - The maximum values allowed when creating a peg-in transaction is 10 RBTC.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="12">
  <Accordion.Header as="h3">13.After making a native pegout, to which address will I receive my
BTCs?</Accordion.Header>
  <Accordion.Body>
    > - During the pegout process, the destination address of your BTC is derived from your signature,
this enables one to know which address will receive the BTCs.
    > - See the Derivation details page
```

```

</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="13">
  <Accordion.Header as="h3">14.When using Trezor i'm receiving the error Forbidden key path
?</Accordion.Header>
  <Accordion.Body>
    > - The latest versions of Trezor Suite have implemented a security rule to disable its use with
non-standard key paths. Therefore, the user must explicitly set Perform Safety Checks to PROMPT
option in Trezor Suite in order to use the Trezor wallet in the 2wp-app application.
    > - If is not enabled you will receive this error !Trezor Error Key Path
    > - This video explains how to enable Perform Safety Checks to PROMPT on Trezor Suite Enabling
Prompt for Key Path
    <Video url="/img/resources/two-way-peg-app/trezor-error-fixed.mp4"
thumbnail="/img/resources/two-way-peg-app/trezor-error.png" />

  </Accordion.Body>
</Accordion.Item>
</Accordion>

```

----

Next

See Glossary section for explanation of terms.

# glossary.md:

```

---
sidebarposition: 1500
sidebarlabel: Glossary
title: "Glossary"
description: "Welcome to the glossary section for the 2 way peg app documentation."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---

```

See a list of terms about/related to the 2 way peg app and their meanings.

```

<Accordion>
  <Accordion.Item eventKey="0">
    <Accordion.Header as="h3">What is the 2 way peg app?</Accordion.Header>
    <Accordion.Body>
      - The 2 way peg app is a frontend application developed in Vue.js and typescript. The or 2 way peg
app is a web application that fosters the interaction between the bitcoin blockchain and the Rootstock
network for faster exchange of BTC and RBTC. See the github repo.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="1">
    <Accordion.Header as="h3">Amount in BTC</Accordion.Header>
    <Accordion.Body>
      - The amount a user is sending. Not less than 0.005 BTC.
    </Accordion.Body>
  </Accordion.Item>
  <Accordion.Item eventKey="2">
    <Accordion.Header as="h3">Device account address</Accordion.Header>

```



```
<Accordion.Body>
  - The account address the user is sending from in BTC.
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="3">
  <Accordion.Header as="h3">Destination Rootstock address</Accordion.Header>
  <Accordion.Body>
    - The account address to receive the RBTC.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="4">
  <Accordion.Header as="h3">Hardware Wallet</Accordion.Header>
  <Accordion.Body>
    - A hardware wallet is a special-purpose device configured to accept supported cryptocurrencies and tokens. Hardware wallets usually take the form of a physical device. Examples of hardware wallets are Ledger and Trezor.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="5">
  <Accordion.Header as="h3">Legacy address</Accordion.Header>
  <Accordion.Body>
    - Legacy address is the original BTC address. It is the most expensive address type because it uses the most amount of space inside a transaction.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="6">
  <Accordion.Header as="h3">Mainnet</Accordion.Header>
  <Accordion.Body>
    - Assets on mainnet have real value and should be used only in Production.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="7">
  <Accordion.Header as="h3">Native SegWit address</Accordion.Header>
  <Accordion.Body>
    - The SegWit native transaction is Bech32, and crypto wallets that support SegWit generally incur lower fees.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="8">
  <Accordion.Header as="h3">Network Fee</Accordion.Header>
  <Accordion.Body>
    - A Network Fee, as the name implies, is a fee you pay to a blockchain network for transferring a digital asset on that network.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="9">
  <Accordion.Header as="h3">Peg-ins</Accordion.Header>
  <Accordion.Body>
    - A conversion from BTC to RBTC. In the peg-in process, the customer sends some BTC and gets the equivalent amount in RBTC inside the Rootstock Blockchain network. The peg-in process is final and cannot be reverted, it requires 100 Bitcoin block confirmation which is approximately 200 Rootstock Blocks.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="10">
```

```
<Accordion.Header as="h3">Peg-outs</Accordion.Header>
<Accordion.Body>
  - A conversion from RBTC to BTC. This locks RBTC on the Rootstock network and releases BTC on
the Bitcoin network.
</Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="11">
  <Accordion.Header as="h3">Refund Bitcoin address</Accordion.Header>
  <Accordion.Body>
    - The bitcoin address to be refunded.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="12">
  <Accordion.Header as="h3">Powpeg</Accordion.Header>
  <Accordion.Body>
    - The powpeg is a unique 2-way peg system that secures the locked bitcoins with the same Bitcoin
hashrate that establishes consensus. Read more about the Powpeg.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="13">
  <Accordion.Header as="h3">SegWit address</Accordion.Header>
  <Accordion.Body>
    - Segwit, short for Segregated Witness, is an upgrade to the Bitcoin protocol, it separates the digital
signature (also known as "the witness") from the transaction, it is a newer address format with lower fees.
It makes Bitcoin transaction sizes smaller, which allows Bitcoin to handle more transactions at once
(scalability). Watch the video: What is Segwit? Explained.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="14">
  <Accordion.Header as="h3">Software Wallet</Accordion.Header>
  <Accordion.Body>
    - A software wallet is an application that is installed on a computer or smartphone. The private keys
are stored on the computer or smartphone.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="15">
  <Accordion.Header as="h3">Testnet</Accordion.Header>
  <Accordion.Body>
    - This is a testing network used for testing and development purposes, assets on the Testnet have
zero value, funds used in this network are called Test tokens (tRBTC), they can be gotten through a
faucet that dispenses tokens. These tokens are utility tokens that are required to operate certain DApps.
Developers of those DApps also need to test them on the Testnet, and hence these are provided as a
convenience for them. The Rootstock network provides a cryptocurrency faucet. The tRBTC faucet
provides the cryptocurrency required to pay for gas fees on the Rootstock Testnet. See how to get tRBTC
using the Rootstock Faucet.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="16">
  <Accordion.Header as="h3">Transaction fee</Accordion.Header>
  <Accordion.Body>
    - The transaction fee, its equivalent, is specified in BTC and USD.
  </Accordion.Body>
</Accordion.Item>
<Accordion.Item eventKey="17">
  <Accordion.Header as="h3">Transaction total</Accordion.Header>
```

```
<Accordion.Body>
  - This comprises of the BTC amount + transaction fee selected.
</Accordion.Body>
</Accordion.Item>
</Accordion>
```

# overview.md:

```
---
sidebarposition: 200
sidebarlabel: Overview
title: "Overview"
description: "Welcome to the overview section of the 2 way peg app documentation."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

In this section, we are going to learn about the 2 way peg app, how it works, its features, and the benefits of using the application.

> To get started, see the prerequisites section.

The 2 way peg is a protocol that converts BTC to RBTC and vice versa. It is secured by the powpeg, which is a unique 2-way peg system that secures the locked bitcoins with the same Bitcoin hashrate that establishes consensus. See the history of the Powpeg.

The 2 way peg app is a web application that fosters the interaction between the bitcoin blockchain and the Rootstock network for easier exchange of BTC and RBTC. It provides a way to visualize the status of transactions, communicate with a user wallet (both hardware wallets and software wallets), while also providing the highest possible level of security for transactions.

## How it Works

The 2 way peg app uses a REST API and a 2 way peg api as the backend, this API uses a daemon) process, which is responsible for listening on blockchain transactions to update the state of peg-ins and in the future, the state of peg-outs, these state changes (tx hash, date change, last status) are stored in a mongodb database.

:::info[Info]

The 2 way peg app is available on both Mainnet and Testnet.

:::

The source code is available on github, and open source:

- Front end
- Back end

## Features

The 2 way peg app, has two primary features, they are:

- Peg-in: A conversion from BTC to RBTC. See Glossary page for more explanation.
  - Note: The peg-in process is final and cannot be reverted.
  - Native pegin transaction has 17 hours estimated time to completion.

- Peg-out: A conversion from RBTC to BTC. This current version of the 2 way peg app. See pegout for more explanation.
- Native pegin transaction has 34 hours estimated time to completion.

Why use the 2 way peg app?

The 2 way peg application has lots of benefits, these include:

#### 1. Simplified transactions

The two way peg (peg-in and pegout) are its nature is a complex process and this app makes it simpler. Using the 2 way peg app enables you to choose where to receive the converted BTC / RBTC, which is also possible without it, but with an even higher level of complexity than a legacy peg-in and peg-out.

#### 2. Visualization of transactions

Enables the visualization of the status of transactions on the Rootstock network

#### 3. Enables communication with a user wallet (hardware and software)

The 2 way peg app communicates directly with the following services:

- Trezor: Directly via usb
- Ledger: Directly via usb and integrated with the manufacturer's application
- Metamask: Through the rLogin application. Learn more about the rLogin application

#### 4. Secure transactions

All transactions need to be confirmed via the device used by the customer, whether a hardware or software wallet, all transaction information and the appropriate signatures are generated through integration with the wallets.

----

Next

Convert BTC to RBTC using the 2 way peg app.

Convert RBTC to BTC using the 2 way peg app.

[View Advanced Operations](#)

# prerequisites.md:

---

sidebarposition: 300

sidebarlabel: Prerequisites

title: "Prerequisites"

description: "Welcome to the overview section of the 2 way peg app documentation."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

Install Apps

:::note[Using the App]

In this guide, we will use the 2 way peg app on 2-Way Peg App - Testnet for learning purposes.

- Note that for transactions using real tokens, use the 2-Way Peg App - Mainnet application.

...


The Bitcoin testnet app does not show on Ledger live manager by default. To be able to see the BTC Testnet app you need to enable the developer mode in Ledger live.

#### - Enable Developer Mode for Bitcoin Testnet

1. Connect your ledger hardware device and unlock it.
2. Open Ledger live, click on Manager and open settings.
3. Navigate to the experimental features menu and enable developer mode. This will show developer and testnet apps in the manager.

 `<div align="left"></div>`

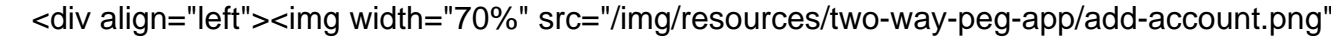
4. Go to Ledger live manager and search for Bitcoin testnet app
5. Click on install to install the Bitcoin Testnet application. To use the testnet app you also need the main Bitcoin app. So install both the apps to your device.

 `<div align="left"></div>`


#### - Get Testnet address

- On your ledger device, you'll find all the apps installed on your device. The Bitcoin app to be used on Mainnet, and Bitcoin Test app to be used on Testnet. To start using testnet, we need the testnet address, to get this address:


1. Open the Bitcoin test app on your ledger device. You will see a "Bitcoin Testnet is ready" screen
2. In the ledger live app, go to accounts tab, click on add account.

 `<div align="left"></div>`


3. Search testnet and select Bitcoin Testnet (BTC). Click on Continue

 `<div align="left"></div>`

4. Approve the Bitcoin Test app on your hardware wallet device
5. On the next screen choose the address format (Native SegWit / SegWit).
6. Click on Add Account:



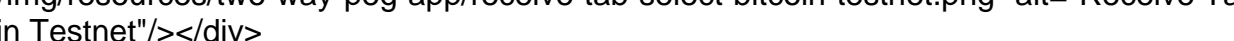
 `<div align="left"></div>`

- > Note: See supported addresses for the types of addresses supported by the 2 way peg app.
- You have successfully added the Bitcoin testnet app to your account.

 `<div align="left"></div>`

#### Get Funds

##### - Get Testnet Tokens

1. Go to the receive tab on Ledger live.  
 `<div align="left"></div>`
2. Select Bitcoin testnet and click on continue.  
 `<div align="left"></div>`
3. Copy BTC Testnet address  
 `<div align="left"></div>`
4. Use the following faucet to receive testnet tokens:
  - Open Coinfaucet

- Paste the address into the field and click on Get Bitcoins.
- > Note: You need at least 0.005 BTC to perform a peg-in on Mainnet and Testnet. Likewise, you need at least 0.004 RBTC to perform a peg-out on Mainnet and Testnet.
- Get Mainnet Tokens
  - See Get Crypto on Rootstock.

:::info[Info]

The 2 way peg app is available on both Mainnet and Testnet. Both applications follow the same process. For production purposes, use Mainnet, for testing and development purposes, use the Testnet.

- See glossary for explanation of these terms.

:::

## Setup Requirements

To get started with the 2-way peg app, ensure you have the following:

- System Requirements
  - A computer with at least Windows 8.1 (64-bit), macOS 10.10, or a Linux distribution, and an internet connection.
- Hardware Wallets
  - Ledger Nano S / Ledger Nano X:
    - Install Ledger Live to manage your device and install the Bitcoin and Bitcoin testnet apps. If you haven't, download it from [here](#).
    - For Ledger Nano S setup, see [Set up your Ledger Nano S](#).
    - For Ledger Nano X setup, see [Set up your Ledger Nano X](#).
  - Trezor Wallet:
    - Follow the setup guide for Trezor hardware wallet.
  - Liquidity Software Wallet (Peg-out Requirements only):
    - Setup the Liquidity software wallet by visiting Liquidity's website.
  - Metamask Wallet (Peg-out Requirements only):
    - For more details, see [Metamask Wallet](#)
- > For Peg-out requirements, ensure you have either the Liquidity or Metamask wallet installed in your browser. For more information, see [Supported Browsers and Supported Wallets](#).

:::info[Funds]

- A minimum balance of 0.005 BTC for peg-in and 0.004 RBTC for peg-out processes.

:::

:::note[Note]

This guide primarily uses the Ledger Nano S hardware wallet for illustration, but all models of Ledger and Trezor wallets are compatible with the 2-way peg app. If you do not have any of the listed hardware wallets, consider purchasing one from the official Ledger or Trezor websites.

:::

## Resources

See the overview section to learn about the 2 way peg app  
[Convert BTC to RBTC using the 2 way peg app](#)

2 way peg app frontend repo  
2 way peg app backend repo  
Rootstock Testnet Faucet  
Design architecture

# design-architecture.md:

---  
sidebarposition: 1300  
sidebarlabel: Design and Architecture  
title: "Design and Architecture"  
description: "Two way peg design and architecture."  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

The 2 way peg app is a protocol to convert BTC to RBTC and vice versa. It is secured by the powpeg, which is a unique 2-way peg system that secures the locked bitcoins with the same Bitcoin hashrate that establishes consensus. See the history of the Powpeg.

In this section, we will cover the design and architecture used in building the 2 way peg application. It comprises of a web interface built with Vue.js, a backend application built with Node.js, and made to run via containers.

## High level

The solution is a web interface, which integrates with a REST API, which in turn communicates with internal services such as the blockchain node and databases. In addition, a daemon/worker will be created that will be responsible for obtaining data from the blockchain and changing the status of the transaction.

This diagram shows the architecture of the 2 way peg app, a Customer (Person) refers to someone who owns BTC or RBTC who wishes to use the 2 way peg app to send a transaction.

!High level diagram - Customer

## Components

The front-end application of the 2 way peg app is developed using Vue.js. The backend application is developed using Nodejs containing a restful API Service and a Daemon service. The API is responsible to serve the data to the front-end, and the Daemon service is responsible for listening for transactions on-chain and updates the database.

!Frontend Application - 2 way peg

## Containers

All applications are available to run using Docker and are built using a Dockerfile. The front-end application will start a node environment with nginx to serve the Vuejs application, and the back-end will start nodejs and start the daemon and api listening by default on port 3000.

!Containers Diagram

# index.md:

---

sidebarposition: 1200

sidebarlabel: Advanced Operations

title: "Overview | Advanced Operations"

description: "See how to perform advanced operations on the two way peg app"

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

This section contains detailed instructions on how to perform advanced operations on the 2 way peg app.

These operations include;

How to review funds in Bitcoin after a pegout by viewing a derived address,

Convert RBTC - BTC and import a key in Electrum, import in Electrum if you are using hardware wallets

Selecting different accounts

Viewing advanced details

Adjusting network fees

Viewing Advanced data

> For more information, see design and architecture, supported addresses, supported wallets, supported browsers

---

Account selection

Pegin:

There are three types of accounts on the 2 way peg app. See supported addresses section for examples of these types of addresses.

To select an account to send BTC from, click on Select the account as shown in the image below. This loads the balance for all the addresses in your hardware wallet.

> Note: Your hardware wallet must be connected to view this section of the 2 way peg app.

!Bitcoin account to send from

Choose the address you want to send TBTC from. See the getting funds section for how to get BTC or TBTC.

Advanced data

Unsigned raw tx

A Bitcoin raw transaction is a chunk of bytes that contains the info about a Bitcoin transaction. That raw transaction will become part of the blockchain when a miner adds it to a block. The pegin transaction has at least one input and two outputs, all information is encoded and displayed for the users' verification.

Adjusting network fees

There are three options to choose from when deciding on which fee rate to use when sending a transaction.

Slow



A slow transaction is less expensive and will take longer to confirm.

!Slow transaction

Average

This is also known as normal, here, the transaction is priced at an average rate and will take an average time to confirm.

!Average transaction

Fast

A fast transaction is the most expensive but the transaction confirms at the quickest time possible.

!Fast transaction

> The type of fee to be selected depends on several variables, like network speed, time, and amount the user is willing to spend on a single transaction.

----

Resources

2 way peg app frontend repo

2 way peg app backend repo

Rootstock Testnet Faucet

Design architecture

# supported-addresses.md:

---

sidebarposition: 1400

sidebarlabel: Supported Addresses

title: "Supported Addresses"

description: "See supported addresses on the two way peg app"

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

The following address types are supported by the 2 way peg app.

- Segwit:

A SegWit address starts with the number 3 and has more elaborated functionality than a legacy address. Types include P2SH-P2WPKH and P2SH-P2WSH.

- Native segwit:

Also known as Bech32 address, native SegWit looks different from the P2-styles as it starts with bc1.

- Legacy address:

These addresses are the original BTC addresses. It uses a special script hash function called P2PKH (Pay-to-Pubkey Hash) address and starts with the number 1.

# supported-browsers.md:

---

sidebarposition: 1500  
sidebarlabel: Supported Browsers  
title: "Supported Browsers"  
description: "See browsers which support the two way peg app"  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

- Google Chrome
- Brave Browser

> Note: The current v1.4.0 of the 2 way peg app is currently not operational on mobile devices.

!Supported Browser

!Mobile browser not supported

# supported-wallets.md:

---

sidebarposition: 1600  
sidebarlabel: Supported Wallets  
title: "Supported Wallets"  
description: "See wallets which supports the two way peg app"  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

Here is a list of supported wallets for the 2 Way Peg App.

- Hardware wallets
  - Ledger
  - Trezor
- Software wallets
  - Leather

:::tip[Tip]

See alternative software wallets supported by the 2 Way Peg App.

:::

# index.md:

---

sidebarposition: 1000  
sidebarlabel: Peg-in  
title: "Overview | Peg-in"  
description: "Here, we will learn how to perform a peg-in transaction using the 2 way peg app."  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

!2 way peg app (peg-out)

In this section, we will learn how to perform a peg-in transaction using the 2 way peg app, to convert BTC to RBTC (peg-in).

To start using 2wp-app, you will need to connect your RSK wallet:

1. Connect your wallet:  
!2 way peg app (connect-your-wallet)

2. Once connected you will see your address and your balance:  
!2 way peg app (wallet-connected)

To get started, see the Requirements for Pegin section.

We will do the following:

1. Set up environment
2. Perform a peg-in (BTC - RBTC) transaction using Ledger Hardware Wallet
3. Perform a peg-in (BTC - RBTC) transaction using Trezor Hardware Wallet
4. View a transaction status.

----

Next

Convert RBTC - BTC using the 2 way peg app.

----

Resources

2 way peg app frontend repo

2 way peg app backend repo

Rootstock Testnet Faucet

Design architecture

# leather.md:

---

sidebarposition: 1200

sidebarlabel: Peg-in using Leather Software Wallet

title: "Performing a peg-in using Leather Software Wallet"

description: "Here, we will learn how to perform a peg-in transaction using the Leather Software Wallet."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

Using Leather Software wallet to perform a peg-in

In this guide, we will be using the 2 way peg app application for unlockcing Leather, access peg-in to use Leather, and verify if peg-in is active.

Acessing pegin to use Leather

!leather-pegin

After click on Leather button, you will see the pegin Leather page, then the 2wp-app will connect with your Leather wallet.

> NOTE: To know how to install the Leather chrome plugin, go to Leather Page.

Unlocking Leather

Unlock Leather using the same password that you created in the installation.

!!leather-error

!!leather-unlocking

Using Correct Network on Leather

If you see the message below, it means that your Leather wallet is in the incorrect environment, change it to the correct environment and try again.

!!leather-error

Select the correct environment on Leather:

!!leather-environment

!!leather-environment-2

Verifying if the plugin is active

If you see the message below, it means that your Leather wallet is not enabled, change it to enable and use the software wallet.

!!leather-not-activated

Go to chrome (manage extensions)[chrome://extensions/] and active your Leather wallet

!!leather-not-activated

!!leather-not-activated

Step 1: CREATING A TRANSACTION

Enter Amount

The next step is to enter an amount you would like to send. The amount entered appears in the BTC field, and you can see the corresponding amount in USD under transaction summary.

!Enter Amount

> - The minimum amount to send to perform a pegin operation is 0.005 BTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The minimum amount to send to perform a pegout operation is 0.004 RBTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The maximum amount to send to perform a pegout or pegout operation is 10 RBTC / BTC, any amount greater than this throws an error message: "The maximum accepted value is 10".

> - Note that the amount sent in BTC is the same amount to be received in RBTC on the Rootstock network.

Step 2: Enter address

To enter an address, we are provided with two options:

- (1) Your connected Rootstock address. See Account based addresses
- (2) Connect to a software wallet. E.g, Metamask. Here, the address is automatically filled in by the account that is connected to your metamask wallet.

!Enter address

Step 3: Add a custom address

Also you can input a custom Rootstock address, different than the connected address.

Step 4: Select Transaction Fee

Here, we can select the fee that will be used for this transaction, this is set on default to average.

Step 5: View transaction summary

In this section, we can confirm the selected values:

- Destination Address
- Estimated time
- BTC fee (Network fee)
- Provider fee (always zero for this option)
- Amount to send
- Value to receive

!Review Transaction

- > - In the instance of an error on this transaction, the amount will be sent to the address indicated in the refund Bitcoin address located in your hardware wallet.
- > - See the glossary section for the meaning of these values.

Step 6: Confirm the Transaction

!leather-verify

Step 7: View transaction status

This shows the status of your transaction, with a transaction ID and a link to check the transaction on the explorer.

!view transaction status

By clicking on the See Transaction button, the user can check the status directly in the transaction status page, by clicking in Start Again button the user can perform another transaction.

See the Viewing Peg-in Transaction Status section for more information.

Now you have successfully performed a peg-in transaction using the 2 way peg application.

# ledger.md:

---

sidebarposition: 1100

sidebarlabel: Peg-in using Ledger Hardware Wallet

title: "Performing a peg-in using Ledger Hardware Wallet"

description: "Here, we will learn how to perform a peg-in transaction using the 2 way peg app."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

## Using hardware wallets

In this guide, we will be using the 2 way peg app on 2 way peg app - Testnet for learning purposes, for transactions using real tokens, please use the 2 way peg app - Mainnet application.

:::warning[Warning]

Ensure to complete the steps in prerequisites before proceeding with this section.

:::

## Ledger Hardware Wallet

!Choose Pegin Option

!Connect Ledger

Step 1: Connect to a ledger wallet

- Plug your Ledger wallet by connecting the USB cable that comes with the Ledger.
- Enter your pin that has already been configured in requirements, to unlock the Ledger.

Step 2: Enter Pin

!Enter Pin

Step 3: Choose Wallet

Here, we will use the Bitcoin Test wallet. For Mainnet, use the Bitcoin wallet.

!Choose Wallet

Note: On the Nano S ledger, whenever you want to confirm an option, click on the 2 upper buttons at the same time.

Step 4: Confirm connection to Bitcoin Testnet

Once the above steps have been completed, a confirmation appears - "Bitcoin Testnet is ready".

!Confirm Connection Testnet

Now, you have successfully connected your Ledger device to the Bitcoin network.

## Performing a peg-in transaction with Ledger

A peg-in is the process of exchanging BTC for RBTC. See the glossary section for more information.

:::tip[Tip]

The minimum values allowed when creating a peg-in transaction is 0.005 BTC.

:::

Open 2 way peg application on Testnet.

### Step 1: Select Conversion Type

Since we are performing a peg-in, choose the BTC - RBTC conversion type, as shown in the image below;

!Select Conversion Type

### Step 2: Choose Hardware Wallet

Here, we are using the ledger hardware wallet to interact with the 2 way peg app. Select your hardware wallet, ensure your device is already connected by inserting your pin into the Ledger device before clicking the Ledger option in the 2 way peg app. See the connect to a ledger wallet section.

!Choose Hardware wallet

### Step 3: Read pop up information

The pop up shown in the image below describes the duration of the peg-in process which requires at least 100 confirmations on the Bitcoin network, this gives an estimate of around 17 hours in total. It also describes the three main steps involved which is; connecting to the hardware wallet, sending a signed transaction to the BTC network until the corresponding RBTC value is made available in the destination wallet and a receipt for this transaction.

!Read popup info

> Click the checkbox - "Don't show again" to turn off this pop-up in the future or close temporarily.

### Step 4: Connect to the app

Click Continue to connect to the 2 way peg application.

!Connect to the app

!Confirm your Ledger wallet

The 2 way peg app shows the pop-up with the connected usb ledger devices, if your device is not visible, unplug the usb device and plug in again, unlock with a pin and click Retry or go back to the connect ledger wallet section.

!Connect device error

To confirm successful connection to the 2 way peg app, you will be directed to the screen below, where we will perform a Peg-in transaction.

!Peg-in screen

> - The balance of the accounts in your hardware wallet will be loaded, and this shows the balance of 3 different types of accounts: segwit, legacy, native segwit. See the supported addresses for the meaning of these types of accounts.

### Step 5: Sending a transaction

## Choose Account

Select the account you would like to send BTC from, by clicking on the dropdown as shown in the image below.

!Select Testnet Bitcoin Account

:::info[Info]

For each selected account type, we will see a corresponding balance.ente

:::

!Bitcoin Account Selected

Enter Amount

After selecting the account you will like to send BTC from, the next step is to enter an amount you would like to send. The amount entered appears in the BTC field, and you can see the corresponding amount in USD under transaction summary.

!Enter Amount

> - The minimum amount to send to perform a pegin operation is 0.005 BTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The minimum amount to send to perform a pegout operation is 0.004 RBTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The maximum amount to send to perform a pegout or pegout operation is 10 RBTC / BTC, any amount greater than this throws an error message: "The maximum accepted value is 10".

> - Note that the amount sent in BTC is the same amount to be received in RBTC on the Rootstock network.

Step 6: Enter address

To enter an address, we are provided with two options:

- (1) Your connected Rootstock address. See Account based addresses
- (2) Connect to a software wallet. E.g, Metamask. Here, the address is automatically filled in by the account that is connected to your metamask wallet.

!Enter address

Step 6a: Add a custom address

Also you can input a custom Rootstock address, different than the connected address.

Step 7: Select Transaction Fee

Here, we can select the fee that will be used for this transaction, this is set on default to average.

!Select Transaction Fee

> - The transaction fee is not part of the amount you're sending via the 2 way peg app, it will only be used for the correct processing of the transaction on the Bitcoin network. Also see the different types of fees



(slow, average, fast) and their corresponding cost in TBTC and USD, depending on preference, you can choose any of those three. See the adjusting network fees section for more information.

> - The time for each type of fee per transaction may vary depending on the number of transactions on the network and the fees charged at the time.

## Step 8: View transaction summary

In this section, we can confirm the selected values:

- Destination Address
- Estimated time
- BTC fee (Network fee)
- Provider fee (always zero for this option)
- Amount to send
- Value to receive

### !Review Transaction

> - In the instance of an error on this transaction, the amount will be sent to the address indicated in the refund Bitcoin address located in your hardware wallet.

> - See the glossary section for the meaning of these values.

### !Review Summary

## Step 9: Confirm and sign transaction

By clicking on the Confirm button, we can see all the transactions that will be made, their corresponding inputs and outputs, and the network fees that will be charged, all this information must be confirmed on your hardware wallet screen.

### !View transaction summary

## Step 10: Send transaction in Ledger Device

After click on the send button, you can confirm or reject the transaction in your hardware wallet. Unlock ledger device to confirm the transaction.

The user needs to review and approve all outputs, the value of the transaction and the fee of the transaction. This test transaction generates 3 outputs.

> To approve or confirm any action on the screen, press on the two buttons beside the ledger hardware device at the same time.

### Review and accept output 1

!review output one

!accept output one

### Review and accept the output 2

!review output two

!accept output two

Review and accept the output 3

!review output three

!accept output three

Confirm amount of test transactions

!confirm amount of test tx

Confirm if the fee value is the same present in the transaction summary screen.

!confirm fee value

Now, confirm all transactions

!confirm transactions

Accept and send the transaction to be broadcasted to the network.

!accept and send

> After signing, the transaction is sent to the network to be processed, taking into account the fee value selected previously.

Step 11: View transaction status

This shows the status of your transaction, with a transaction ID and a link to check the transaction on the explorer.

!view transaction status

By clicking on the See Transaction button, the user can check the status directly in the transaction status page, by clicking in Start Again button the user can perform another transaction.

See the Viewing Peg-in Transaction Status section for more information.

Now you have successfully performed a peg-in transaction using the 2 way peg application.

# status.md:

```
---
sidebarposition: 1300
sidebarlabel: Viewing Peg-in Transaction Status
title: "Viewing Transaction Status"
description: "Here, we will learn how to view a transaction status on the 2 way peg app."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

The transaction status shows the status of transactions performed using the 2 way peg application.

There are two ways to view the transaction status.

1. Using the Transaction status page on the 2 way peg application.
2. View a transaction using Blockcypher Explorer

## Using the transaction status page

To view a transaction status using the 2 way peg application, we will do the following steps;

### Step 1: Go to the homepage

Visit: 2 way peg.

Click on transaction status.

!Transaction status

### Step 2: Enter Transaction ID

Copy the transaction ID derived in Step 12: Performing a Pegin transaction with Ledger, paste into the field as shown below, click on enter or click on the search icon or click on Enter.

!Transaction status field

### Step 3: View transaction status

This shows what stage the transaction is in, the transaction performed was a peg-in transaction (BTC to RBTC), in the image below, you will see whether funds have moved from the Bitcoin network to the Rootstock network, and also when the funds have been successfully delivered to an Rootstock address.

Click on the refresh button by scrolling down on the page below to view the updated status.

!Transaction status update

### Step 4: View transaction summary

Here, you can see the following information:

#### Bitcoin Side:

- Sender Address: The address used to send BTC
- Transactino ID: Bitcoin Transaction hash
- Fee: Bitcoin Network fee
- You will send: The amount send + fee to the Bitcoin Network

#### Rootstock Side

- Recipient Address: The address when you will receive the funds
- Transaction ID: Rootstock Transaction hassh
- Fee: Rootstock Network fee
- You will receive: The amount that you will receive in Recipient Address

:::danger[Error]

- In case an error occurs with this transaction, the amount will be sent back to the refund Bitcoin address.
- See the glossary section for in-depth definition and explanation of these terms.

...

## Using Block Explorer

To view transactions status using Block Explorer, you can click on the open window icon, or copy transaction ID and paste in your preferred block explorer.

# trezor.md:

```
---
sidebarposition: 1400
sidebarlabel: Peg-in using Trezor Hardware Wallet
title: "Performing a peg-in in using Trezor Hardware Wallet"
description: "Here, we will learn how to perform a peg-in transaction using the Trezor Hardware Wallet."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

In this guide, we will be using performing a peg in transaction using the 2 way peg app application.

Access Pegin Page

!Connect Trezor

Verify if you have enabled Perform Safety Checks to PROMPT

- > - If is not enabled you will receive this error !Trezor Error Key Path
- > - This video explains how to enable Perform Safety Checks to PROMPT on Trezor Suite Enabling Prompt for Key Path

<Video url="/img/resources/two-way-peg-app/trezor-error-fixed.mp4"  
thumbnail="/img/resources/two-way-peg-app/trezor-error.png" />

Select Trezor Wallet

!Connect Trezor

Trezor Hardware Wallet

!Connect Trezor

Read pop up information

The pop up shown in the image below describes the duration of the peg-in process which requires at least 100 confirmations on the Bitcoin network, this gives an estimate of around 17 hours in total. It also describes the three main steps involved which is; connecting to the hardware wallet, sending a signed transaction to the BTC network until the corresponding RBTC value is made available in the destination wallet and a receipt for this transaction.

!Connect Trezor

Step 1: Connecting to a trezor wallet

Plug your Trezor wallet by connecting the USB cable that comes with Trezor.

Step 2: Export multiple addresses

In this step, the user is redirected to Trezor's site and needs to click on export to export the addresses.

!Export Testnet Addresses

Step 3: Enter Pin and confirm

Enter a pin for your Trezor, displayed on your hardware wallet. Click confirm.

!Insert Trezor Wallet Pin

!Insert Wallet Pin - Trezor

Step 4: Unlock Trezor with passphrase

!Enter passphrase

Step 5:

- Type Trezor passphrase

- Trezor will display the message: 'Please enter your passphrase using the computer's keyboard'.

!Enter Passphrase using Keyboard

The user fills the passphrase, and confirms passphrase fields, using the Trezor Connect application. The user will see this screen on Trezor: "Access Hidden Wallet?".

!Access hidden wallet notification

!Use passphrase

Now, you have successfully connected your Trezor to the Bitcoin network.

Performing a peg-in transaction with Trezor

Step 1: Connect to the app

Click Continue to connect to the 2 way peg application.

!Connect to the app

The 2 way peg app shows the pop-up with the connected usb ledger devices, if your device is not visible, unplug the usb device and plug in again, unlock with a pin and click Retry or go back to the connect ledger wallet section.

!Connect device error

To confirm successful connection to the 2 way peg app, you will be directed to the screen below, where we will perform a Peg-in transaction.

!Peg-in screen

> - The balance of the accounts in your hardware wallet will be loaded, and this shows the balance of 3 different types of accounts: segwit, legacy, native segwit. See the supported addresses for the meaning of these types of accounts.

Step 5: Sending a transaction

Choose Account

Select the account you would like to send BTC from, by clicking on the dropdown as shown in the image below.

!Select Testnet Bitcoin Account

> Note that for each selected account type, we will see a corresponding balance.

Enter Amount

After selecting the account you will like to send BTC from, the next step is to enter an amount you would like to send. The amount entered appears in the BTC field, and you can see the corresponding amount in USD under transaction summary.

!Enter Amount

> - The minimum amount to send to perform a pegin operation is 0.005 BTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The minimum amount to send to perform a pegout operation is 0.004 RBTC, any amount less than this throws an error message: "You cannot send that amount of BTC, you can only send a minimum of 0.005 BTC".

> - The maximum amount to send to perform a pegout or pegout operation is 10 RBTC / BTC, any amount greater than this throws an error message: "The maximum accepted value is 10".

> - Note that the amount sent in BTC is the same amount to be received in RBTC on the Rootstock network.

Step 6: Enter address

To enter an address, we are provided with two options:

- (1) Your connected Rootstock address. See Account based addresses
- (2) Connect to a software wallet. E.g, Metamask. Here, the address is automatically filled in by the account that is connected to your metamask wallet.

!Enter address

Tips:

> - Use the address verifier to verify if an address is Rootstock-compatible and can be used to perform a peg in a transaction.

> - Use the Metamask-Rootstock tool to automatically connect to Rootstock mainnet or manually connect metamask to the Rootstock mainnet or testnet.

Step 6a: Add a custom address

Also you can input a custom Rootstock address, different than the connected address.

Step 7: Select Transaction Fee

Here, we can select the fee that will be used for this transaction, this is set on default to average.

!Select Transaction Fee

> - The transaction fee is not part of the amount you're sending via the 2 way peg app, it will only be used

for the correct processing of the transaction on the Bitcoin network. Also see the different types of fees (slow, average, fast) and their corresponding cost in TBTC and USD, depending on preference, you can choose any of those three. See the adjusting network fees section for more information.

> - The time for each type of fee per transaction may vary depending on the number of transactions on the network and the fees charged at the time.

## Step 8: View transaction summary

In this section, we can confirm the selected values:

- Destination Address
- Estimated time
- BTC fee (Network fee)
- Provider fee (always zero for this option)
- Amount to send
- Value to receive

### !Review Transaction

> - In the instance of an error on this transaction, the amount will be sent to the address indicated in the refund Bitcoin address located in your hardware wallet.

> - See the glossary section for the meaning of these values.

### !Review Summary

## Step 9: Accept and send the transaction to be broadcasted to the network.

> After signing, the transaction is sent to the network to be processed, taking into account the fee value selected previously.

## Step 10: View transaction status

This shows the status of your transaction, with a transaction ID and a link to check the transaction on the explorer.

### !view transaction status

By clicking on the See Transaction button, the user can check the status directly in the transaction status page, by clicking in Start Again button the user can perform another transaction.

See the Viewing Peg-in Transaction Status section for more information.

Now you have successfully performed a peg-in transaction using the 2 way peg application.

# deriving-electrum.md:

```
---
sidebarposition: 1200
sidebarlabel: Viewing a derived bitcoin address
title: "Viewing a derived bitcoin address"
description: "Here, we will learn how to view a derived address using Electrum."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

## !2 way peg app (peg-out)

This section contains detailed instructions on how to review funds in Bitcoin after a pegout by:

- Convert RBTC - BTC
- Import a key in Electrum
- Import in Electrum if you are using hardware wallets

### Why derive address using Electrum?

During the pegout process, the destination address of your BTC is derived from your signature, this enables one to know which address will receive the BTCs, to view the destination address, follow this step by step guide.

### Prerequisites:

- Wallet private key
- Electrum
- Rootstock Utils

### How to view a derived address

A derived address is the BTC address derived from the RBTC account. When using the 2 way peg app, it is important to know which address you will receive your BTCs. See Viewing advanced details.

We will learn how to view a derived address using Metamask to get a private key. We will also learn how to convert RBTC - BTC and Import a Key in Electrum.

### Getting a wallet private key

#### Using Metamask

Step 1: Open the Metamask wallet on your browser, you can find this in the extensions tab in your browser.

Step 2: Click on the menu icon by the right

Step 3: Choose "Account details"

!metamask - privacy

Step 4: Then click on the "Export private key" button

!metamask - exportprivatekey

Step 5: Fill out wallet password and click on "Confirm"

!metamask - fillwalletpassword

Step 6: Copy the private key and click on "Done"

!metamask - copyassets

### Converting RBTC to BTC



Before converting the funds, we need to convert the private key into a Wallet Import Format (WIF). A WIF private key is just another way of representing your original private key. If you have a WIF private key, you can always convert it back in to its original format.

For more info on WIF, see the Bitcoin Wiki

Using Rootstock Utils (Recommended)

Rootstock Utils is used to convert keys from BTC to Rootstock.

Step 1: Clone the Rootstock utils project.

Step 2: Follow the steps explained in the README.

Step 3: Install webpack using the code below;

```
javascript    npm install webpack@4.46.0 -g    npm i webpack-cli@3.3.12 -g    npm install webpack
```

[Optional] you will need npm to install webpack:

```
npm install --save-dev webpack
```

Step 4: Run webpack

```
webpack
```

Step 5: Open the file in your browser

```
./build/index.html
```

Step 6: Open the generated application and add your private key and convert to WIF, as shown in the image below:

```
!browser - openbrowser
```

Using LearnMeABitcoin

> - IMPORTANT: We discourage users from using websites on the internet, note that if your private key is exposed, your funds will also be exposed, therefore it's recommended that you use the offline option, like Rootstock utils.

Follow the steps below to get started;

Step 1: Visit the url: <https://learnmeabitcoin.com/technical/wif>

```
!metamask - WIF
```

> You will find the Ruby code and a tool to convert the private key into a WIF.

Step 2: Paste the private key gotten in Getting a wallet private key in the "Private Key" field

Step 3: Choose the network: Mainnet or Testnet

Step 4: Choose compressed option true

Step 5: Copy WIF value

> - IMPORTANT: Using the Ruby code is highly recommended

> - This code requires the checksum.rb and base58encode.rb functions as shown in the code below.

Download the 'checksum' file [here](#).

Download the 'base58encode' file [here](#).

```
shell
require_relative 'checksum'
require_relative 'base58encode'
```

Convert Private Key to WIF

```
privatekey = "4fd050a8e4fd767f759d75492b9894bc97875e8201873e38443e3f5eae9c8db2f"
extended = "80" + privatekey + "01"
extendedchecksum = extended + checksum(extended)
wif = base58encode(extendedchecksum)
```

```
puts wif
```

Import key in Electrum

---

Electrum is used to verify a derived address, this address will then be used to receive and verify the converted funds (RBTC - BTC) when the pegout process is finished.

Step 1: Download Electrum for your OS from the [website](#).

Follow the steps below to create a new wallet in Electrum and import the private key:

> NOTE: If you need to run Electrum in Testnet, execute the following commands:

```
cd /Applications/Electrum.app/Contents/MacOS
./runelectrum --testnet
```

Step 2: Start with the "Create New Wallet" option

Step 3: Fill out a new wallet name and click on the "Next" button

Step 4: Choose "Import Bitcoin addresses or private keys" option and click on "Next" button

Step 5: Fill out the WIF value of the private key and click on "Next" button

Step 6: Create a new wallet password and click on the "Next" button

```
!electrum - new
```

> In this screen, you will see the address to receive the BTC funds.

## Import key in Electrum using Hardware Wallets

Electrum is used to verify a derived address, this address will then be used to receive and verify the converted funds (RBTC - BTC) when the pegout process is finished.

Step 1: Download Electrum for your OS from the website.

Follow the steps below to create a new wallet in Electrum and connect to the hardware wallets:

> NOTE: If you need to run Electrum in Testnet, execute the following commands:

```
cd /Applications/Electrum.app/Contents/MacOS  
./runelectrum --testnet
```

Step 2: Start with the “Create New Wallet” option

Step 3: Fill out the name in “Wallet” field and click on “Next” button

Step 4: Select “Standard wallet” option and click on “Next” button

```
!wallet - electrum
```

Step 4: Select “Use a hardware device” option and click on “Next” button

```
!wallet - electrum-hardware-device
```

Step 5: Select the hardware wallet and click on “Next” button

> NOTE: The follow screen is an example of usage the Trezor Hardware Wallet

```
!wallet - electrum-hardware-device-trezor
```

> NOTE: The follow screen is an example of usage the Ledger Hardware Wallet

```
!wallet - electrum-hardware-device-ledger
```

Step 6: Select “legacy (p2pkh)” option, fill out a custom derivation path field and click on “Next” button

Custom derivation path:

Mainnet: m/44'/137'/0'

```
!wallet - electrum-derivation-pathx
```

> NOTE: Testnet: m/44'/37310'/0'

```
!wallet - electrum-derivation-legacy
```

> IMPORTANT: For Ledger it is necessary to approve the custom derivation path in the device

!wallet - electrum-ledger-confirmation

Step 7: Check "Encrypt wallet file" option and click on "Next" button

!wallet - electrum-encrypt

Step 8: Finally in Electrum go to "Addresses" tab and you can see your funds

!wallet - electrum-show-funds

# index.md:

```
---
sidebarposition: 1100
sidebarlabel: Peg-out
title: "Overview | Pegout"
description: "Here, we will learn how to perform a peg-out transaction using the 2 way peg app."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

!2 way peg app (peg-out)

In this section, we will learn how to perform a peg-out transaction on the 2 way peg app, to convert RBTC to BTC.

To start using 2wp-app, you will need to connect your RSK wallet:

1. Connect your wallet:

!2 way peg app (connect-your-wallet)

2. Once connected you will see your address and your balance:

!2 way peg app (wallet-connected)

> To get started, see the Requirements for Pegout section.

We will do the following:

1. Perform a peg-out using MetaMask wallet
2. Perform a peg-out using Ledger+Liquality
3. How to get a Bitcoin derived address in hardware wallet using Electrum
4. Viewing a Transaction Status
5. Troubleshooting and common errors

----

Next

Convert BTC - RBTC using the 2 way peg app.

View Advanced Operations.

----

Resources

2 way peg app frontend repo

2 way peg app backend repo

# ledger-liquidity.md:

---

sidebarposition: 1300

sidebarlabel: Peg-out using Metamask and any hardware wallet

title: "Performing a peg-out using Metamask and hardware wallet"

description: "Here, we will learn how to perform a peg-out using Metamask and hardware wallet."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

!2 way peg app (peg-out)

Performing a peg-out transaction using Ledger and Liquidity

The Liquidity Wallet is a browser extension for accessing Bitcoin, Rootstock, and Ethereum applications.

:::warning[Warning]

Liquidity Wallet has been discontinued. See section on Supported wallets by the 2 Way Peg App.

:::

We will perform a peg-out transaction using the Ledger Hardware Wallet and Liquidity.

> See How to perform a peg-in transaction using Ledger

Get started

To perform a peg-out transaction using the Ledger device with Metamask, follow the steps below:

Step 1: Open Metamask click on select account, and then in the Add account or hardware wallet button:

!Add account or hardware wallet

Step 2: Select Add hardware wallet option:

!Add account or hardware wallet

Step 3: The user will be redirected to wallet selection page:

!Connect hd

To connect with a Trezor Wallet:

!Choose hd wallet trezor

To connect with a Ledger Wallet:

!Choose hd wallet ledger

Now you can see "Ledger" or "Trezpr" label in your Metamask accounts

!Final screen

----

Next

See Performing a peg-out transaction using Trezor

See Performing a peg-out transaction using Ledger

# ledger.md:

---

sidebarposition: 1400

sidebarlabel: Peg-out using Ledger

title: "Performing a peg-out using Ledger Hardware Wallet"

description: "Here, we will learn how to perform a peg-out using Ledger Hardware Wallet."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

!2 way peg app (peg-out)

Performing a peg-out transaction using rLogin(Trezor and Ledger)

> - Note that we will be using the 2 way peg app on 2 way peg app - Testnet for learning purposes.

> - For transactions using real tokens, use the 2 way peg app - Mainnet application.

> - We're using Ledger Nano and Trezor One hardware wallets on this tutorial.

> - To use Ledger hardware wallet to create a peg-in see How to perform a peg-in transaction using Ledger

> - To use Trezor hardware wallet to create a peg-in see How to perform a peg-in transaction using Trezor

Get started with Ledger

To perform a peg-out transaction using the Ledger device directly, follow the steps below:

Step 1: Plug the Ledger device into the computer

Step 2: Enter your pin to unlock it

Step 3: On the device, navigate to the TRSK or RSK Test app on your Ledger device

Step 4: Access peg-out screen:

!pegout screen

Step 5: Choose Ledger option:

!connect-wallet

Step 6: Click on Ledger button <br/>

!connect-wallet

Step 8: The application will show what network you are connecting on. For this tutorial we are using Testnet

!network

Step 9: The application will show a simple tutorial:

!1-plug

!2-install

!3-close

!4-open

!5-confirm

Step 10: Click on the Finish tutorial and connect button: <br/>

!6-finish

Step 11: Select an account <br/>

!7-select

Step 12: Ledger Connected <br/>

!8-connected

<br/>

Step 13: Continue filling in the other fields as amount and click on the Send button

Step 14: After finish the pegout transaction creation, click here to see how to see the steps to access to Bitcoin derived address in hardware wallet using Electrum

----

Next

See Performing a peg-out transaction using Ledger and Liquality

----

Resources

2 way peg app frontend repo

2 way peg app backend repo

Rootstock Testnet Faucet

Design architecture

# metamask.md:

---

sidebarposition: 1600

sidebarlabel: Peg-out using Metamask

title: "Performing a peg-out using Metamask Wallet"

description: "Here, we will learn how to perform a peg-out using Metamask Wallet."

tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]

---

!2 way peg app (peg-out)

Performing a peg-out transaction using MetaMask

Step 1: Select conversion type

To perform a peg-out, open the 2 way peg app - Testnet in your browser.

Step 2: Choose the RBTC - BTC conversion type

!pegout screen

Step 3: Connect your MetaMask wallet

Click on 'Connect wallet' and then select 'MetaMask'.

!connect-wallet

> If your wallet is locked, see images below for steps on how to unlock it.

!Unlock metamask wallet

And then click 'Confirm' to complete the first step.

!Confirm metamask wallet connection

Step 4: Enter an amount

Enter the amount you want to send . You can either enter it manually,  
or click 'Use max available balance' if you want to send all the RBTC you have.

!RBTC amount to send input

Step 5 (Optional) : Verify your Bitcoin destination address

Click 'Get Bitcoin destination address'. Click 'Generate' first in 2-Way Peg App and then in MetaMask.

!Click get bitcoin destination address

!Click sign button

!Metamask signature request

After signing, you will be able to know the derived Bitcoin address where you will receive funds.

!Derived address

> For more details on derived addresses. See the advanced operations section.

Step 6: Send transaction



Confirm the information, click 'Send' in 2-Way Peg App and then click 'Confirm' in MetaMask.

!Click to send pegout transaction

!Sign to confirm and send pegout transaction

!Confirm send on metamask

See final screen as shown in the image below;

!BTC on its way

By clicking on the See Transaction button, the user can check the status directly in the transaction status page, by clicking in Start Again button the user can perform another transaction.

See the Viewing Peg-out Transaction Status section for more information.

# pegout-common-errors.md:

```
---
sidebarposition: 1700
sidebarlabel: Troubleshooting and Common Errors
title: "Common errors when using the app"
description: "Here, we will learn how to perform a peg-out using Metamask Wallet."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

!2 way peg app (peg-out)

You may encounter the following errors when trying out the application:

#### Resources

- 2 way peg app frontend repo
- 2 way peg app backend repo
- Rootstock Testnet Faucet
- Design architecture

# status.md:

```
---
sidebarposition: 1800
sidebarlabel: Viewing Peg-out Transaction Status
title: "Viewing the status of a transaction after peg-out"
description: "Here, we will learn how to view a transaction status after a peg-out."
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]
---
```

Using the transaction status page

To view a transaction status using the 2 way peg application, we will do the following steps;

Step 1: Go to the homepage

Visit: 2 way peg.

Click on transaction status.

!Transaction status

The processing of a pegout transaction is made up of several dependencies, and for each dependency a processing step is added, and at each step in the process, the pegout is shown in a form on the transaction status query screen.

After finish a pegout you can search for the current status in the status page

Step 2: Enter Transaction ID

Copy the transaction ID derived in Step 12: Performing a Pegout transaction with Ledger, paste into the field as shown below, click on enter or click on the search icon or click on Enter.

!Transaction status field

Step 3: View transaction status

This shows what stage the transaction is in, the transaction performed was a peg-out transaction (RBTC to BTC), in the image below, you will see whether funds have moved from the Bitcoin network to the Rootstock network, and also when the funds have been successfully delivered to an Rootstock address.

Click on the refresh button by scrolling down on the page below to view the updated status.

!Transaction status update

Rootstock Side

- Recipient Address: The address when you will receive the funds
- Transaction ID: Rootstock Transaction hash
- Fee: Rootstock Network fee
- You will receive: The amount that you will receive in Recipient Address

Bitcoin Side:

- Sender Address: The address used to send BTC
- Transaction ID: Bitcoin Transaction hash
- Fee: Bitcoin Network fee
- You will send: The amount send + fee to the Bitcoin Network

:::danger[Error]

- In case an error occurs with this transaction, the amount will be sent back to the refund Bitcoin address.
- See the glossary section for in-depth definition and explanation of these terms.

:::

Using Block Explorer

To view transactions status using Block Explorer, you can click on the open window icon, or copy transaction ID and paste in your preferred block explorer.

# trezor.md:

---  
sidebarposition: 1900  
sidebarlabel: Peg-out using Trezor  
title: "Performing a peg-out transaction using Trezor"  
description: "Here, we will learn how to perform a peg-out using Trezor."  
tags: [2 way peg app, powpeg, peg-in, peg-out, bridge, rsk, rootstock]  
---

!2 way peg app (peg-out)

Get started with Trezor

To perform a peg-out transaction using the Ledger device directly, follow the steps below:

Step 1: Plug the Ledger device into the computer

Step 2: Verify if you have enabled Perform Safety Checks to PROMPT

> - If is not enabled you will receive this error !Trezor Error Key Path  
> - This video explains how to enable Perform Safety Checks to PROMPT on Trezor Suite Enabling Prompt for Key Path  
<Video url="/img/resources/two-way-peg-app/trezor-error-fixed.mp4"  
thumbnail="/img/resources/two-way-peg-app/trezor-error.png" />

Step 3: Access peg-out screen:  
!pegout screen

Step 4: Click on Connect wallet button:

!connect-wallet

Step 5: Click on Trezor button<br/>  
!connect-wallet

Step 6: The application will show what network you are connecting on. For this tutorial we are using Testnet

<br/>

!network

Step 6: Plugin your Trezor device:<br/>  
!plugin

Step 7: The trezor window will open to insert the pin and export the addresses  
!pin-and-address

Step 8: Insert the pin and click on confirm button  
!insert-confirm

Step 9: Insert the passphrase  
!insert-passphrase

Step 10: Follow instructions on your device <br/>  
!verify-device

> - Note the trezor app screen will be opened some times, because the system will ask for addresses, each ask will open again the trezor screen, and the user will need to inform the trezor-pin.

Step 11: Select account

<br/>

!select-account

Step 12: Success

!success

Step 13: Continue filling in the other fields as amount and click on the Send button

Step 14: After finish the pegout transaction creation, click here to see how to see the steps to access to Bitcoin derived address in hardware wallet using Electrum

# index.md:

```
---
sidebarlabel: Hackathons
sidebarposition: 7
title: Hackathon and Workshop Resources
tags: [hackathons, rsk, workshop, resources, rootstock]
description: "Hackathon resources and tools"
---
```

This guide details the necessary hardware and software requirements for developing on the Rootstock blockchain. It includes setup instructions for essential tools such as Java, Node.js, Hardhat, and RSKj, ensuring developers have a clear path to prepare their environment for Rootstock projects, whether for local development, testing, or deployment.

```
<Card
  title="Prerequisites"
  description="Prerequisites for developing on Rootstock."
  link="/developers/requirements/"
/>
```

<br></br>

```
<Card
  title="Starter Kits"
  description="Starter Kits for easy Rootstock Development."
  link="/developers/quickstart/"
/>
```

# foundry.md:

```
---
sectionposition: 2
```

sidebarlabel: Foundry  
title: Foundry on Rootstock  
description: 'How to write, test, and deploy smart contracts with Foundry'  
tags: [foundry, quick start, developer tools, rsk, rootstock, ethereum, dApps, smart contracts]  
---

Foundry is a smart contract development toolchain, and user-friendly development environment for writing and testing smart contracts in Solidity. It manages dependencies, compiles, run tests, deploy contracts and allows for interaction with EVM-compatible chains using a command-line tool called Forge.

Why use Foundry?

Forge is ideal for advanced smart contract analysis, auditing, and for fast execution of smart contract tests.

:::note[hardhat-foundry plugin]

Use the hardhat-foundry plugin to have your Foundry project work alongside Hardhat.

:::

Here are some reason why you may prefer Foundry:

#### 1. Local Networks:

It provides a local blockchain environment using the anvil tool, allowing developers to deploy contracts, run tests, and debug code. It can also be used to fork other EVM compatible networks.

#### 2. Advanced Testing:

Forge comes with a number of advanced testing methods including:

- Fuzz testing
- Invariant testing
- Differential testing
- Symbolic Execution
- Mutation Testing

#### 3. Advanced Debugging:

Forge allows for advanced debugging using an interactive debugger.

The debugger terminal is divided into four quadrants:

- Quadrant 1
  - The opcodes in the debugging session, with the current opcode highlighted. Additionally, the address of the current account, the program counter and the accumulated gas usage is also displayed.
- Quadrant 2
  - The current stack, as well as the size of the stack
- Quadrant 3
  - The source view.
- Quadrant 4
  - The current memory of the EVM.

#### Resources

- Getting Started with Foundry
- Getting Started with Hardhat

- Foundry Project Layout
- RPC Calls Using Cast
- Foundry Configurations
- Solidity Scripting
- Deploy Smart Contracts To Deterministic Addresses

# hardhat.md:

```
---
sidebarposition: 3
sidebarlabel: Hardhat
title: Hardhat on Rootstock
description: "How to get started with writing, deploying and testing smart contracts on Rootstock using Hardhat."
tags: [hardhat, quick start, developer tools, rsk, rootstock, ethereum, dApps, smart contracts]
---
```

Hardhat is an Ethereum development environment for developers. It's primarily used in the development of smart contracts for the Rootstock and EVM-compatible chains.

### Key features of Hardhat

1. **Local Ethereum Network:** It provides a local blockchain environment, allowing developers to deploy contracts, run tests, and debug their code.
2. **Automated Testing:** Hardhat facilitates automated testing of smart contracts, which is crucial for ensuring their reliability and security.
3. **Debugging:** It includes a robust debugging tool that helps developers identify and fix issues in their smart contracts.
4. **Hardhat Runtime Environment (HRE):** This is injected into the project's scripts and provides access to Hardhat's functionality and plugins.
5. **Extensible Through Plugins:** Developers can extend Hardhat's capabilities through a wide range of plugins.
6. **Network Management:** It allows for seamless interaction with public and private networks, making deployment processes efficient.
7. **Ethers.js and Waffle Integration:** These integrations provide a set of utilities for writing and testing smart contracts.

### Installation

To install Hardhat, run the following command:

```
bash
npm install --save-dev hardhat
```

### Related Links

- Hardhat Starter dApp
- Getting Started with Hardhat
- Getting Started with Foundry

## - Developer Tools

# index.md:

---

sidebarposition: 1

title: Developer Tools

sidebarlabel: All Tools

tags: [rsk, rootstock, tools, developer tools]

description: "Explore a curated selection of smart contract development tools and languages. From the familiar Solidity to Rust or Developer Environments like Hardhat, you'll find everything you need to interact and deploy your smart contracts on Rootstock."

---

<Filter

values=

{label: 'Bridges', value: 'bridge'},  
{label: 'Starter Kits', value: 'demos'},  
{label: 'Exchanges', value: 'exchange'},  
{label: 'Wallets', value: 'wallet'},  
{label: 'Explorers', value: 'explorer'},  
{label: 'Platforms and Infra', value: 'platform-infra'},  
{label: 'Cross Chain', value: 'cc'},  
{label: 'Data', value: 'data'},  
{label: 'SDKs', value: 'sdk'},  
{label: 'Faucets', value: 'faucet'},  
{label: 'Gas', value: 'gas'},  
{label: 'Dev Environments', value: 'dev-environment'},  
{label: 'Account Abstraction', value: 'aa'},  
{label: 'Code Quality', value: 'code-quality'},  
{label: 'JSON-RPC', value: 'rpc'},  
{label: 'Libraries', value: 'library'},  
{label: 'No Code', value: 'no-code'},  
{label: 'RIF Tools', value: 'rifp'},  
{label: 'Smart Contracts', value: 'sc'},  
{label: 'Mining', value: 'mine'},  
{label: 'Oracles', value: 'oracles'},

]]>

<FilterItem

value="bridge, exchange"

title="2 Way Peg App"

subtitle="bridges"

color="orange"

linkHref="/resources/guides/two-way-peg-app/"

target="blank"

linkTitle="Documentation"

description="Bridge Bitcoin and Rootstock using the 2 Way Peg App."

/>

<FilterItem

value="bridge, cc"

title="Token Bridge"

subtitle="bridges"

color="orange"

linkHref="/resources/guides/tokenbridge/"

target="blank"

```
    linkTitle="Documentation"
    description="Use the Token Bridge to safely and securely move ERC20 tokens from Ethereum to
Rootstock and vice-versa."
  />
<FilterItem
  value="dev-environment, sc, platform-infra"
  title="Foundry"
  subtitle="Dev Environments"
  color="orange"
  linkHref="https://dev.rootstock.io/dev-tools/foundry/"
  linkTitle="Deploy Smart Contracts"
  description="Foundry is a smart contract development toolchain, and user-friendly development
environment for writing and testing smart contracts in Solidity."
  />
<FilterItem
  value="dev-environment, sc"
  title="Hardhat"
  subtitle="Dev Environments"
  color="orange"
  linkHref="/dev-tools/hardhat/"
  linkTitle="Deploy Smart Contracts"
  description="Hardhat is an Ethereum development environment for developers. It's primarily used in
the development of smart contracts for the Rootstock and EVM-compatible chains."
  />
<FilterItem
  value="explorer, sc"
  title="Blockscout Explorer"
  subtitle="Explorers"
  color="orange"
  linkHref="/dev-tools/explorers/blockscout/"
  linkTitle="Use the Explorer"
  description="Blockscout is an open-source tool for exploring transactions on any EVM chain, including
Rootstock."
  />
<FilterItem
  value="explorer, sc"
  title="Rootstock Explorer"
  subtitle="Explorers"
  color="orange"
  linkHref="https://explorer.rootstock.io/"
  linkTitle="Use the Explorer"
  description="Explore transactions, blocks, addresses, tokens, stats and interact with smart contracts on
the Rootstock Explorer."
  />
<FilterItem
  value="rpc"
  title="RPC API"
  subtitle="json rpc"
  color="orange"
  linkHref="/developers/rpc-api/"
  linkTitle="Make First API Call"
  description="The Rootstock RPC API provides a seamless and intuitive web interface for developers to
interact with Rootstock nodes via JSON-RPC methods."
  />
<FilterItem
```



```

    value="rpc, smart contracts"
    title="GetBlock"
    subtitle="json rpc"
    color="orange"
    linkHref="https://getblock.io/nodes/rsk/"
    linkTitle="Make First API Call"
    description="GetBlock provides instant connection to blockchain nodes including Rootstock, Bitcoin
(BTC), Ethereum (ETH), among others."
  />
<FilterItem
  value="rpc, smart contracts"
  title="NOWNodes"
  subtitle="json rpc"
  color="orange"
  linkHref="https://nownodes.io/nodes/rsk"
  linkTitle="Make First API Call"
  description="NOWNodes is a blockchain-as-a-service enterprise solution that lets users get access to
full Nodes and blockbook Explorers via an API."
  />
<FilterItem
  value="rpc, smart contracts"
  title="dRPC"
  subtitle="json rpc"
  color="orange"
  linkHref="https://drpc.org/chainlist/rootstock?utmsource=docs&utmmedium=rootstock"
  linkTitle="Make First API Call"
  description="dRPC provides access to a distributed network of node providers."
  />
<FilterItem
  value="wallet, sc"
  title="MetaMask"
  subtitle="wallets"
  color="orange"
  linkHref="/dev-tools/wallets/metamask/"
  linkTitle="Use MetaMask"
  description="Learn how to create, and add Rootstock tokens to MetaMask."
  />
<FilterItem
  value="wallet, sc"
  title="Rootstock Wallets"
  subtitle="wallets"
  color="orange"
  linkHref="/dev-tools/wallets/"
  linkTitle="Use Wallets"
  description="View all Rootstock Wallets."
  />
<FilterItem
  value="bridge, exchange"
  title="Sovryn Fast BTC"
  subtitle="bridges"
  color="orange"
  linkHref="https://wiki.sovryn.com/en/sovryn-dapp/bridge"
  linkTitle="Get RBTC"
  description="Sovryn is a non-custodial and permissionless smart contract based system for Bitcoin
lending, borrowing and margin trading."

```

```
/>
<FilterItem
  value="bridge, exchange"
  title="RBTC Exchanges"
  subtitle="Exchanges"
  color="orange"
  linkHref="https://rootstock.io/rbtc/"
  linkTitle="Get RBTC"
  description="Exchanges and Bridges to get RBTC."
/>
<FilterItem
  value="bridge, exchange, rifp"
  title="RIF Exchanges"
  subtitle="Exchanges"
  color="orange"
  linkHref="https://rif.technology/rif-token/"
  linkTitle="Get RIF Tokens"
  description="Exchanges and Bridges to get the RIF Token."
/>
<FilterItem
  value="exchange"
  title="RIF on Chain"
  subtitle="Exchanges"
  color="orange"
```

linkHref="https://dapp.rifonchain.com/ipfs/QmWpKDzJ9fUECiiYkGHxqEXKh3CRUEzfvTxYoQonxFBK61/"

linkTitle="Get Started"  
description="Access crypto collateralized digital dollars to save, spend & send. Get RIF, USDRIF, MOC, RIF Pro, etc."

```
/>
<FilterItem
  value="bridge, exchange, rifp"
  title="Flyover"
  subtitle="bridges"
  color="orange"
  linkHref="https://dapp.flyover.rif.technology/"
  linkTitle="Get RBTC"
  description="The Flyover protocol performs fast peg-ins and peg-outs between Bitcoin and Rootstock networks."
/>
```

```
<FilterItem
  value="data"
  title="The Graph"
  subtitle="data & analytics"
  color="orange"
  linkHref="/dev-tools/thegraph/"
  linkTitle="Access on-chain data"
  description="Get historical data on smart contracts when building dApps."
/>
```

```
<FilterItem
  value="data"
  title="Covalent"
  subtitle="data & analytics"
  color="orange"
```

linkHref="https://www.covalenthq.com/docs/networks/rootstock/?utmsource=rootstock&utmmedium=partner-docs"

linkTitle="Access on-chain data"

description="Covalent is a hosted blockchain data solution providing access to historical and current on-chain data for 100+ supported blockchains, including Rootstock."

/>

<FilterItem

value="data"

title="DefiLlama"

subtitle="data & analytics"

color="orange"

linkHref="https://defillama.com/chain/Rootstock"

linkTitle="Access on-chain data"

description="DefiLlama is the largest Total Value Locked (TVL) aggregator in the DeFi space. It assesses the TVL by taking into account the worth of tokens locked within the contracts of a protocol or platform."

/>

<FilterItem

value="data"

title="Tenderly"

subtitle="data & analytics"

color="orange"

linkHref="https://tenderly.co/"

linkTitle="Access on-chain data"

description="Tenderly helps developers build, monitor, and improve smart contracts by providing a set of tools to boost productivity, save time, and ensure efficient smart contracts functionality."

/>

<FilterItem

value="platform-infra, sc, sdk"

title="Thirdweb"

subtitle="platforms"

color="orange"

linkHref="https://thirdweb.com/"

linkTitle="Use Thirdweb"

description="Thirdweb is a Full-stack web3 development tools, production-grade infrastructure platform for developers to build on Rootstockk."

/>

<FilterItem

value="platform-infra, sc"

title="useDApp"

subtitle="platforms"

color="orange"

linkHref="https://usedapp.io/"

linkTitle="Build with useDApp"

description="Build a dApp on Rootstock using useDApp React library."

/>

<FilterItem

value="no-code, platform-infra, sc"

title="Forward Protocol"

subtitle="no-code"

color="orange"

linkHref="https://forwardprotocol.io/"

linkTitle="Build a NoCode dApp"

description="Build a dApp on Rootstock using Forward Protocol's NoCode Tools."

```
/>
<FilterItem
  value="library, sdk, rifp, abs"
  title="RIF Relay"
  subtitle="sdks"
  color="orange"
  linkHref="/developers/integrate/rif-relay/"
  linkTitle="Integrate RIF Relay"
  description="RIF Relay is a secure sponsored transaction system that enables users to pay transaction
fees using ERC-20 tokens."
/>
<FilterItem
  value="dev-environment, sc"
  title="Remix"
  subtitle="Dev Environments"
  color="orange"
  linkHref="https://remix.ethereum.org/"
  linkTitle="Deploy Smart Contracts"
  description="Compile, Interact and Deploy Smart Contracts using Remix."
/>
<FilterItem
  value="library, sdk, wallet, rifp"
  title="RIF Wallet"
  subtitle="sdks"
  color="orange"
  linkHref="/developers/libraries/rif-wallet-lib/"
  linkTitle="Integrate RIF Wallet"
  description="RIF wallet is a fully programmable and extensible DeFi wallet enabling developers and
businesses to build intuitive and secure mobile-first Web3 experiences for their end-users."
/>
<FilterItem
  value="gas"
  title="Gas Station"
  subtitle="gas"
  color="orange"
  linkHref="https://rskgasstation.info/?AspxAutoDetectCookieSupport=1"
  linkTitle="View Gas Price"
  description="Rootstock Gas Station."
/>
<FilterItem
  value="data"
  title="Rootstock Stats"
  subtitle="data & analytics"
  color="orange"
  linkHref="https://stats.rootstock.io/"
  linkTitle="View Stats"
  description="Rootstock Stats."
/>
<FilterItem
  value="faucet"
  title="Rootstock Faucet"
  subtitle="faucets"
  color="orange"
  linkHref="https://faucet.rootstock.io/"
  linkTitle="Get tRBTC"
```

```

    description="Get tRBTC on the Rootstock Testnet Faucet."
  />
<FilterItem
  value="faucet, rifp"
  title="RIF Testnet Faucet"
  subtitle="faucets"
  color="orange"
  linkHref="https://faucet.rifos.org/"
  linkTitle="Get tRIF"
  description="Get tRIF on the RIF Testnet Faucet"
/>
<FilterItem
  value="library, sc"
  title="Ethers.js"
  subtitle="library"
  color="orange"
  linkHref="https://web3js.readthedocs.io/en/v1.10.0/"
  linkTitle="Use Ethers.js Library"
  description="A library for Interacting with the Rootstock Virtual Machine."
/>
<FilterItem
  value="library, sc"
  title="Web3.js"
  subtitle="library"
  color="orange"
  linkHref="https://docs.ethers.org/v5/"
  linkTitle="Use Web3.js Library"
  description="A library for Interacting with the Rootstock Virtual Machine."
/>
<FilterItem
  value="library, sdk, rifp"
  title="RNS"
  subtitle="name service"
  color="orange"
  linkHref="https://rif.technology/products/#rns"
  linkTitle="Register a Domain Name"
  description="RNS provides an architecture which enables the identification of blockchain addresses by
human-readable names."
/>
<FilterItem
  value="code-quality, testing, sc"
  title="SolidityScan"
  subtitle="code quality"
  color="orange"
  linkHref="https://solidityscan.com/"
  linkTitle="Secure Smart Contracts"
  description="Secure your smart contracts on Rootstock, and get accurate security audit results and
detailed reports."
/>
<FilterItem
  value="code-quality, testing, sc"
  title="Slither"
  subtitle="code quality"
  color="orange"
  linkHref="https://github.com/crytic/slither"

```

```

    linkTitle="Analyse Smart Contracts"
    description="Slither built with Solidity & Vyper static analysis framework written in Python3, enables
developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom
analyses."
/>
<FilterItem
    value="code-quality, testing, sc"
    title="Sourcify"
    subtitle="code quality"
    color="orange"
    linkHref="https://sourcify.dev"
    linkTitle="Verify Smart Contracts"
    description="Verify smart contracts on Rootstock, Sourcify enables transparent and human-readable
smart contract interactions through automated Solidity contract verification, contract metadata."
/>
<FilterItem
    value="sc, rollups, aa, platform-infra"
    title="Gelato"
    subtitle="infra"
    color="orange"
    linkHref="https://gelato.network"
    linkTitle="Deploy Rollups"
    description="Deploy production-grade & fully-serviced L2 rollups on Rootstock, natively integrated with
tools like oracles, bridges, data indexers and Account Abstraction."
/>
<FilterItem
    value="mine, platform-infra"
    title="Antpool"
    subtitle="mining"
    color="orange"
    linkHref="https://www.antpool.com/home"
    linkTitle="Start Mining"
    description="Start mining with Antpool."
/>
<FilterItem
    value="platform-infra"
    title="Vottun"
    subtitle="infra"
    color="orange"
    linkHref="https://vottun.com"
    linkTitle="Get Started"
    description="Vottun interoperable multi-blockchain architecture is built to make it easy to develop Web3
applications without the need to understand much of the underlying blockchain technology."
/>
<FilterItem
    value="platform-infra"
    title="WakeUp Labs"
    subtitle="infra"
    color="orange"
    linkHref="https://platform.wakeuplabs.io"
    linkTitle="Get Started"
    description="WakeUp Labs is a software development studio that assists EVM-Compatible
Blockchains, DAOs and traditional organizations in overcoming technical challenges and expediting
product development."
/>

```

```
<FilterItem
  value="bridge, sc"
  title="Wormhole"
  subtitle="Cross-chain Bridges"
  color="orange"
  linkHref="https://docs.wormhole.com/wormhole"
  linkTitle="Start Building"
  description="Build and Deploy a Multi-chain dApp on Rootstock."
/>
<FilterItem
  value="data, sc"
  title="Envio"
  subtitle="data"
  color="orange"
  linkHref="https://envio.dev/"
  linkTitle="Access on-chain data"
  description="Get on-chain data when building dApps on Rootstock."
/>
<FilterItem
  value="mine"
  title="F2Pool"
  subtitle="mining"
  color="orange"
  linkHref="https://www.f2pool.com/"
  linkTitle="Start Mining"
  description="Mining Pool on Rootstock."
/>
<FilterItem
  value="mine"
  title="ViaBTC"
  subtitle="mining"
  color="orange"
  linkHref="https://www.viabtc.com/"
  linkTitle="Start Mining"
  description="Mining Pool on Rootstock."
/>
<FilterItem
  value="mine"
  title="Luxor"
  subtitle="mining"
  color="orange"
  linkHref="https://luxor.tech/mining"
  linkTitle="Start Mining"
  description="Mining Pool on Rootstock."
/>
<FilterItem
  value="mine"
  title="BraaiinsPool"
  subtitle="mining"
  color="orange"
  linkHref="https://braiins.com/pool"
  linkTitle="Start Mining"
  description="Mining Pool on Rootstock."
/>
<FilterItem
```

```
value="bridge"
title="Chainport"
subtitle="Cross-Chain Bridge"
color="orange"
linkHref="https://www.chainport.io/"
linkTitle="Get Started"
description="Cross-chain bridge integrated with Rootstock."
/>
<FilterItem
  value="data"
  title="Tres Finance"
  subtitle="Accounting"
  color="orange"
  linkHref="https://tres.finance/"
  linkTitle="Get Started"
  description="Web3 Accounting, Auditing, and Reporting on Rootstock."
/>
<FilterItem
  value="demos, sc"
  title="Wagmi Starter Kit"
  subtitle="Demos"
  color="orange"
  linkHref="https://github.com/rsksmart/rsk-wagmi-starter-kit"
  linkTitle="Use the Kit"
  description="This starter kit provides a foundation for building decentralized applications (dApps) on the
Rootstock blockchain using React, Wagmi and Shadcn libraries."
/>
<FilterItem
  value="demos, sc"
  title="Hardhat Starter Kit"
  subtitle="Demos"
  color="orange"
  linkHref="https://github.com/rsksmart/rootstock-hardhat-starterkit"
  linkTitle="Use the Kit"
  description="Rootstock Hardhat starter."
/>
<FilterItem
  value="demos, sdk, sc, aa"
  title="Account Abstraction Kit"
  subtitle="Demos"
  color="orange"
  linkHref="https://github.com/rsksmart/rsk-wagmi-starter-kit/tree/aa-sdk"
  linkTitle="Use the Kit"
  description="Account Abstraction Starter dApp using Etherspot."
/>
<FilterItem
  value="sdk, sc, aa, platform-infra"
  title="Etherspot"
  subtitle="Account Abstraction"
  color="orange"
  linkHref="https://etherspot.io/"
  linkTitle="Use Etherspot"
  description="Account Abstraction Development on Rootstock."
/>
<FilterItem
```



```

    value="demos, sc"
    title="dApp Automation"
    subtitle="Demos"
    color="orange"
    linkHref="/resources/tutorials/dapp-automation-cucumber/"
    linkTitle="Automate dApps"
    description="Learn how to automate dApp using Cucumber and Playwright."
  />
<FilterItem
  value="sc, oracles, data"
  title="Umbrella Network"
  subtitle="Oracles"
  color="orange"
  linkHref="https://umb.network/"
  linkTitle="Access On-chain Data"
  description="Access On-Chain data for your smart contracts on Rootstock."
/>
<FilterItem
  value="sc, oracles, data"
  title="Redstone Finance"
  subtitle="Oracles"
  color="orange"
  linkHref="https://redstone.finance/"
  linkTitle="Access On-chain Data"
  description="Access On-Chain data for your smart contracts on Rootstock."
/>
<FilterItem
  value="cc, data"
  title="Router Protocol"
  subtitle="Cross Chain Bridges"
  color="orange"
  linkHref="https://routerprotocol.com/"
  linkTitle="Build Cross Chain dApps"
  description="Router Protocol is a layer-1 blockchain enabling chain abstraction."
/>
</Filter>

```

# thegraph.md:

```

---
sidebarlabel: The Graph
sidebarposition: 2
title: Get Started with The Graph
description: "Easily query on-chain data through a decentralized network of indexers"
tags: [TheGraph, indexers, data, subgraphs, dApps, smart contracts, developers, developer tools, get-started, how-to]
---

```

Getting historical data on smart contracts can be challenging when building dApps. The Graph provides an easy way to query smart contracts data through APIs known as subgraphs. Its infrastructure relies on a decentralized network of indexers, enabling dApps to achieve true decentralization.

These subgraphs only take a few minutes to set up and get running.

To get started, follow these steps below:

1. Initialize a subgraph project
2. Deploy & Publish
3. Query from a dApp
  - Querying is a way to access
4. Publish

> Pricing: All developers receive 100K free queries per month on the decentralized network. After these free queries, you only pay based on usage at \$4 for every 100K queries.

## Getting Started

### Initialize a subgraph project

To initialize a Subgraph project, we need to create a subgraph on Subgraph Studio .

Go to the Subgraph Studio and connect your wallet. Once wallet is connected, begin by clicking "Create a Subgraph".

:::info[Info]

Remember to choose a clear and descriptive name for the subgraph since it can't be edited later.

It is recommended to use a Title Case: "Subgraph Name Chain Name."

...

<center>

  
</center>

On the subgraph's page, all the CLI commands needed will be visible on the right side of the page:

<center>

  
</center>

### Install the Graph CLI

Run the following command locally:

```
bash
npm install -g @graphprotocol/graph-cli
```

> You must have at least v0.76.0 to deploy subgraphs on Rootstock mainnet.

## Initialize a Subgraph

This can be copied directly from your subgraph page to include a specific subgraph slug:

```
bash
graph init --studio <SUBGRAPHSLUG>
```

You'll be prompted to provide some info on your subgraph like this:

```
<center>
  
</center>
```

Once contract is verified on the block explorer, the CLI will automatically obtain the ABI and set up the subgraph. The default settings will generate an entity for each event.

## 2. Deploy & Publish

### Deploy to Subgraph Studio

Run the commands below:

```
bash
$ graph codegen
$ graph build
```

To authenticate and deploy your subgraph, run the commands below. You can copy these commands directly from your subgraph's page in Studio to include a specific deploy key and subgraph slug:

```
bash
$ graph auth --studio <DEPLOYKEY>
$ graph deploy --studio <SUBGRAPHSLUG>
```

You will be asked to provide a version label (e.g., v0.0.1). You can use any format that works for you.

### Test Subgraph

You can test your subgraph by making a sample query in the playground section. The Details tab will show you an API endpoint. You can use that endpoint to test from your dApp.

```
<center>
  
```

</center>

## Publish Subgraph to The Graph's Decentralized Network

Once your subgraph is ready to be put into production, you can publish it to the decentralized network. On your subgraph's page in Subgraph Studio, click on the Publish button:

<center>

  
</center>

Before you can query your subgraph, Indexers need to begin serving queries on it. In order to streamline this process, you can curate your own subgraph using GRT.

> When publishing, you'll see the option to curate your subgraph. As of May 2024, it is recommended that you curate your own subgraph with at least 3,000 GRT to ensure that it is indexed and available for querying as soon as possible.

<center>

  
</center>

:::info[Info]

The Graph's smart contracts are all on Arbitrum One, even though your subgraph is indexing data from Rootstock or any other supported chain.

:::

### 3. Query Subgraph

Congratulations! You can now query your subgraph on the decentralized network!

For any subgraph on the decentralized network, you can start querying it by passing a GraphQL query into the subgraph's query URL which can be found at the top of its Explorer page.

Here's an example from the CryptoPunks Ethereum subgraph by Messari:

<center>

  
</center>

The query URL for this subgraph is:

<https://gateway-arbitrum.network.thegraph.com/api/>

[api-key]: /subgraphs/id/HdVdERFUe8h61vm2fDyycHgxjsde5PbB832NHgJfZNqK

Now, you need to fill in your own API Key to start sending GraphQL queries to this endpoint.

Getting your own API Key

```
<center>
  
</center>
```

In the Subgraph Studio, the “API Keys” menu can be accessed from the top of the page. Here, you can create API Keys.

Appendix

Sample Query

This query below shows the most expensive CryptoPunks sold.

```
graphql
{
  trades(orderBy: priceETH, orderDirection: desc) {
    priceETH
    tokenId
  }
}
```

Passing this into the query URL returns the result below:

```
bash
{
  "data": {
    "trades":
    {
      "priceETH": "124457.067524886018255505",
      "tokenId": "9998"
    },
    {
      "priceETH": "8000",
      "tokenId": "5822"
    },
    ...
  }
}
```

> Trivia: Looking at the top sales on [CryptoPunks website it looks like the top sale is Punk #5822, not #9998. Why? Because they censor the flash-loan sale that happened.

## Sample code

Here's a sample code to use within your subgraph:

```
jsx
const axios = require('axios');

const graphqlQuery = {
  trades(orderBy: priceETH, orderDirection: desc) {
    priceETH
    tokenId
  }
};

const queryUrl =
'https://gateway-arbitrum.network.thegraph.com/api/[api-key]/subgraphs/id/HdVdERFUe8h61vm2fDyyCHg
xjsde5PbB832NHgJfZNqK'

const graphQLRequest = {
  method: 'post',
  url: queryUrl,
  data: {
    query: graphqlQuery,
  },
};

// Send the GraphQL query
axios(graphQLRequest)
  .then((response) => {
    // Handle the response here
    const data = response.data.data
    console.log(data)

  })
  .catch((error) => {
    // Handle any errors
    console.error(error);
  });
```

## Additional resources

- To explore all the ways you can optimize & customize your subgraph for better performance, read more about creating a subgraph.
- Learn more about querying data from your subgraph.
- Subgraph Studio
- Getting GRT

# blockscout.md:

---

sidebarposition: 2

sidebarlabel: Blockscout Explorer

title: Get Started with Rootstock Blockscout Explorer

tags: [rsk, explorer, rootstock, Blockscout, smart contracts, transactions, tools]

---

Blockscout is a robust open-source tool for exploring transactions on any EVM blockchain, including Rootstock, the leading Bitcoin sidechain<sup>1</sup>. With Blockscout, you can access in-depth information, verify and interact with smart contracts, create and manage your account, view advanced statistics, and more.

<div align="center"></div>

To use Blockscout with Rootstock, visit the Rootstock Block Explorer, where you can see the latest blocks, transactions, and addresses on the Rootstock network.

### Search for specific information

You can look up a specific transaction by entering the wallet address, transaction hash block number, or token in the search bar at the top pane of the Rootstock Blockscout page.

<div align="center"></div>

### Overviews

The grids below the search bar show an overview of various components on the Rootstock platform. You can see the total blocks on the network and the total transactions. You can click the respective grid to view the block list or transactions. Likewise, the grids present other details like the average block time, number of wallet addresses, current gas, and total BTC locked in the Rootstock 2-way peg.

<div align="center"></div>

### Chart view

By default, the chart below the grids shows a line graph of the Daily transactions. You can switch to see a line graph of RBTC price and market cap by clicking on the buttons, respectively.

<div align="center"></div>

### Latest transactions/blocks

The next section on the page shows a list of the latest blocks on the network on the left side and a list of the latest transactions on the right, as shown below. You can view the full list of blocks or transactions by clicking "view all blocks" or "view all transactions" at the end of the list.

<div align="center"></div>

### Exploring the blockchain

Aside from the overview on the Roostock Blockscout main page, you can view other details on the Blockchain by clicking "Blockchain" on the left menu pane and selecting the appropriate options. This includes supported smart contracts on Rootstock, which you can view by selecting the verified contracts option.

<div align="center"></div>

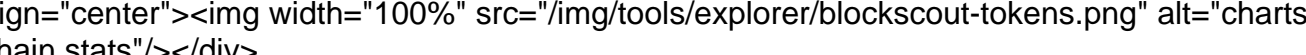
### Tokens on Rootstock

You can view a list of ERC-20 and ERC-721 tokens on the Rootstock network by clicking on the Tokens from the left menu. You can check the details of each token by clicking on the token name.

<div align="center"></div>

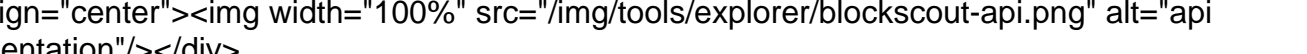
### Charts and stats

Select the Charts & stats option on the left menu to view various advanced statistics and visual representations of data on the Rootstock platform. These include general blockchain stats, Accounts, Transactions, Blocks, Contracts, Gas, and so on.



### Blockscout API

Blockscout allows you to programmatically query various details from the Rootstock network via the API. You can access the Blockscout API by clicking on the "API" button at the bottom of the page, where you can find documentation and examples of using the API for various purposes.



Here's a basic example of how to use Blockscout with Rootstock in Python to get the latest block number:

```
python
import requests

response = requests.get("https://rootstock.blockscout.com/api?module=block&action=ethblocknumber")
latestblock = int(response.json()['result'], 16)  Converts hex to integer
print(f"Latest Rootstock Block Number: {latestblock}")
```

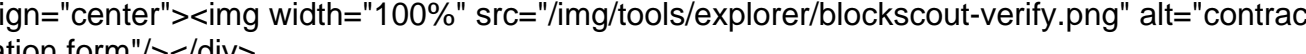
This Python script uses the Blockscout API to fetch the latest block number on the Rootstock mainnet. Response:

```
bash
Latest Rootstock Block Number: 6019497
```

You must have Python and the requests library installed to run this script.

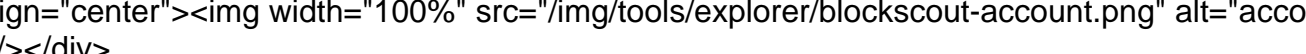
### Verify and publish smart contracts on Rootstock

You can verify and publish a smart contract on the Rootstock network by clicking other>>Verify Contract from the left menu. Then, you supply the contract address and your preferred verification method from the supported contract verification methods list.



### Blockscout account

To create an account on Blockscout, click on the user icon beside the "Connect Wallet" button at the top right corner and sign in/sign up with your email or GitHub account. After creating or signing in to your Blockscout account, you can access other menu options that allow you to add custom tags to addresses or transactions, create a watch list, create custom ABI, or submit public tags requests to the Rootstock team. You can also access your Blockscout API keys.



For additional information see:

- Hardhat Documentation
- Github Repo

# index.md:

---

sidebarposition: 3



sidebarlabel: Rootstock Explorers

title: Overview of the Rootstock Explorer

tags: [overview, explorer, rootstock, smart contracts, transactions, developer tools]

description: Explore transaction

---

You can browse the overview of Rootstock on the explorer page as shown below.

<div align="center"></div>

Switch between Tabs

If you want to view some specific information, you can jump through the tab bar below.

<div align="center"></div>

Search for some information you want

You can search for the information you want by entering the address, block number or tx hash in the search box.

<div align="center"></div>

Last Block Information

The next section is a panel about the last block.

You can see the block number, address, the total number of transactions, and the duration in turn.

Click on the number link and you can jump to the detailed page of this block.

Especially, you can copy the address code by clicking the icon over the string.

<div align="center"></div>

Last Block Transactions

The line graph below shows the transaction density.

<div align="center"></div>

List of Blocks

All the items in this list are the basic information of a block.

Click the link in the title location to update the list.

Click the see all blocks to display the full list.

<div align="center"></div>

## List of Transactions

All the items in this list are the basic information of a transaction.

Click the see all transactions to display the full list.

<div align="center"></div>

# index.md:

---  
sidebarlabel: Rootstock Wallets  
sidebarposition: 2  
title: Wallets compatible with Rootstock  
tags: [rootstock, tools, rsk, wallets]  
description: "Learn how to connect to Rootstock with a compatible Wallet"  
---

The following wallets support RBTC and RIF tokens.

<Carousel width="370" height="260" images=[['/img/rsk/wallets/my-crypto.png',  
'/img/rsk/wallets/metamask-logo.png', '/img/rsk/wallets/edge-wallet-logo.png',  
'/img/rsk/wallets/ledger-logo.png', '/img/rsk/wallets/trezor-wallet.png', '/img/rsk/wallets/dcent-wallet.png',  
'/img/rsk/wallets/math-wallet.png', '/img/rsk/wallets/Exoduslogowhite.png',  
'https://www.myetherwallet.com/img/logo-dark.2fa0f670.png', '/img/rsk/wallets/enkrypt-logo.png',  
'/img/rsk/wallets/block-wallet.png', '/img/rsk/wallets/taho.png', '/img/rsk/wallets/testtwo.png' ]] />

## Compatibility Matrix

In the following matrix you can see the different features by wallet.

| Wallet           | Rootstock | Checksum | Rootstock dPath | Customizable dPath | Platforms               | Networks                  |
|------------------|-----------|----------|-----------------|--------------------|-------------------------|---------------------------|
| Beexo            |           |          |                 |                    | Desktop, Mobile         | Rootstock (RBTC), Bitcoin |
| Edge             |           |          |                 |                    | Desktop, Mobile         | Rootstock (RBTC), Bitcoin |
| Ledger           |           |          |                 |                    |                         |                           |
| MyEtherWallet    |           |          |                 |                    | Desktop, Android, IOS   | RBTC                      |
| Trezor           |           |          |                 |                    |                         |                           |
| Metamask         |           |          |                 |                    | Browser, Mobile         |                           |
| Portis           |           |          |                 |                    | Desktop                 | Rootstock (RBTC), Bitcoin |
| Rabby Wallet     |           | -        | -               | -                  | Chrome, Desktop, Mobile |                           |
| Enkrypt          |           | -        | -               | -                  | Chrome                  | Rootstock (RBTC), Bitcoin |
| MyCrypto         |           | -        | -               | -                  | Desktop                 | RBTC                      |
| TaHo             |           | -        | -               | -                  | Chrome                  | Rootstock (RBTC)          |
| Frontier         |           | -        | -               | -                  | Desktop, Mobile, Chrome | Rootstock (RBTC)          |
| Bitget           |           | -        | -               | -                  | Chrome, Mobile          | RBTC                      |
| SafePal          |           | -        | -               | -                  | Chrome, Mobile          | Rootstock (RBTC), Bitcoin |
| Wallby           |           | -        | -               | -                  | Mobile                  | Rootstock (RBTC), Bitcoin |
| MathWallet       |           |          |                 |                    | Chrome, Desktop, Mobile | Rootstock (RBTC), Bitcoin |
| Block Wallet     |           | -        | -               | -                  | Chrome                  | Rootstock (RBTC)          |
| MtPelerin Bridge |           | -        | -               | -                  | Desktop, Mobile         | Rootstock (RBTC), Bitcoin |
| Gnosis Safe      |           | -        | -               | -                  |                         |                           |

# metamask.md:

```
---
sidebarlabel: MetaMask
sidebarposition: 3
title: Configure MetaMask Wallet for Rootstock
tags: [metamask, rootstock, tools, rsk, wallets, guides]
description: "Learn how to connect to Rootstock with a MetaMask Wallet"
---
```

In this guide, you will learn how to download, install MetaMask, and set up custom networks.

:::note[Download and Install MetaMask]

Visit the [metamask-landing.rifos.org](https://metamask-landing.rifos.org) tool to download/install Metamask, and add Rootstock custom networks or follow the steps in the video below.

:::

```
<div class="video-container">
  <iframe width="949" height="534" src="https://youtube.com/embed/VyPewQoWhn0" frameborder="0"
  allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
  allowfullscreen></iframe>
</div>
```

## Connect with MetaMask

1. Open the MetaMask extension.
2. In the network selector (top right corner), choose Custom RPC.

```
<div style="text-align: center">
  </img>
</div>
```

3. Fill with these values to connect to Rootstock Mainnet or Testnet

```
<table class="table">
  <thead>
    <tr>
      <th scope="col">Field</th>
      <th scope="col">Rootstock Mainnet</th>
      <th scope="col">Rootstock Testnet</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Network Name</td>
      <td>Rootstock Mainnet</td>
      <td>Rootstock Testnet</td>
    </tr>
    <tr>
      <td>RPC URL</td>
      <td>https://rpc.mainnet.rootstock.io/{YOURAPIKEY}</td>
```

|                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------|
| <td>https://rpc.testnet.rootstock.io/{YOURAPIKEY}</td>                                                              |
| </tr>                                                                                                               |
| <tr>                                                                                                                |
| <td>ChainID</td>                                                                                                    |
| <td>30</td>                                                                                                         |
| <td>31</td>                                                                                                         |
| </tr>                                                                                                               |
| <tr>                                                                                                                |
| <td>Symbol</td>                                                                                                     |
| <td>RBTC</td>                                                                                                       |
| <td>tRBTC</td>                                                                                                      |
| </tr>                                                                                                               |
| <tr>                                                                                                                |
| <td>Block explorer URL</td>                                                                                         |
| <td><a href="https://explorer.rootstock.io/" target="blank">https://explorer.rootstock.io/</a></td>                 |
| <td><a href="https://explorer.testnet.rootstock.io/" target="blank">https://explorer.testnet.rootstock.io/</a></td> |
| </tr>                                                                                                               |
| </tbody>                                                                                                            |
| </table>                                                                                                            |

:::tip[Get RPC API Key]

Visit the RPC API Docs to sign up and get an API Key.

:::

Now MetaMask is ready to use with Rootstock!

## Next Steps

Try out the Rootstock Testnet:

- Get test RBTC
- Get test RIF tokens

If you would like to know more about the values used in the custom network configuration above, check out account based addresses on Rootstock.

## Limitations

MetaMask does not yet fully comply with the technical specifications of account based addresses on Rootstock.

Note that there are workarounds available, which allow most users to use MetaMask on Rootstock successfully.

MetaMask uses the Ethereum value for derivation path, and presently does not allow it to be configured. This means that if you use the same seed phrase in MetaMask and other wallets, you will get a different set of addresses. A workaround for this is to use custom derivation paths when using other wallets that support this feature.

MetaMask does not presently support EIP-1191 checksums. This means that if you use the addresses copied from MetaMask, you may encounter checksum validation errors.

A workaround for this is to lowercase the addresses after copying them.

:::warning[Disclaimer]

- Currency may be mistakenly displayed as ETH within some screens of MetaMask.  
The Rootstock network uses RBTC as its cryptocurrency.
- This tutorial uses Rootstock RPC API.  
You can connect to other nodes or use the Public Node by changing the RPC URL.
- The node must enable CORS for browser-based dApps to work.
  - Please review the configuration file reference for CORS settings.

:::

#### Useful Resources

- Connect Rootstock to Metamask Programmatically