Gnosis Safe Documentation

Gnosis

Feb 14, 2019

Content

The Gnosis Safe aims to provide all users with a convenient, yet secure way to manage their funds and interact with decentralized applications on Ethereum. It comes in two editions: Personal Edition and Team Edition. The Gnosis Safe Personal Edition is targeting individual users using 2 or more factor authentication through native mobile apps for Android and iOS in combination with a browser extension. The Gnosis Safe Team Edition is geared towards teams managing shared crypto funds. It is an improvement of the existing Gnosis MultiSig wallet with redesigned smart contracts, cheaper setup and transaction costs as well as an enhanced user experience.

CHAPTER 1

Learn more about Gnosis Safe

- **Gnosis Safe Website**: https://safe.gnosis.io
- **Gnosis Safe Gitter**: https://gitter.im/gnosis/Safe
- **The State of Storing Funds on Ethereum: Why we need the Gnosis Safe**: https://blog.gnosis.pm/the-state-of-storing-funds-on-Ethereum-fdb4c9a09388
- **Announcement of Gnosis Safe Beta: Personal Edition**: https://blog.gnosis.pm/announcing-the-gnosis-safe-beta-personal-edition-19a69a4453e8

1.1 Smart Contract Overview

GitHub: https://github.com/gnosis/safe-contracts

1.1.1 Architecture

Gnosis Safe Transactions

A Safe transaction has the following parameters: A destination address, an Ether value, a data payload as a bytes array, `operation` and `nonce`.

The operation type specifies if the transaction is executed as a `CALL`, `DELEGATECALL` or `CREATE` operation. While most wallet contracts only support `CALL` operations, adding `DELEGATECALL` operations allows to enhance the functionality of the wallet without updating the wallet code. As a `DELEGATECALL` is executed in the context of the wallet contract, it can potentially mutate the state of the wallet (like changing owners) and therefore should only be used with known, trusted contracts. The `CREATE` operation allows to create new contracts with bytecode sent from the wallet itself.

More information on delegate calls can be found in the solidity docs

The nonce prevents replay attacks. Each transaction should have a different nonce and once a transaction with a specific nonce has been executed it should not be possible to execute this transaction again. The concrete replay protection mechanism depends on the version of the Gnosis Safe and will be explained later.

Contract Creations

As the creation of new contracts is a very gas consuming operation, Safe contracts use a proxy pattern where only one master copy of a contract is deployed once and all its copies are deployed as minimal proxy contracts pointing to the master copy contract. This pattern also allows to update the contract functionality later on by updating the address of the master copy in the proxy contract.

As contract constructors can only be executed once at the time the master copy is deployed, constructor logic has to be moved into an additional persistent setup function, which can be called to setup all copies of the master copy. This setup function has to be implemented in a way it can only be executed once. It is important to note that the master copy contract has to be persistent and there should be no possibility to execute a `selfdestruct` call on the master copy contract.

It is **important** to know that it is possible to "hijack" a contract if the proxy creation and setup method are done in separate transactions. To avoid this it is possible to pass the initialisation data to the ProxyFactory or the DelegatingConstructorProxy.

For more information about Proxy contracts read our blog post about Solidity DelegateProxy Contracts.

Contracts

Base Contracts

SelfAuthorized.sol

The self authorized contract implements the `authorized()` modifier so that only the contract itself is authorized to perform actions.

Multiple contracts use the `authorized()` modifier. This modifier should be overwritten by a contract to implement the desired logic to check access to the protected methods.

Proxy.sol

The proxy contract implements only two functions: The constructor setting the address of the master copy and the fallback function forwarding all transactions sent to the proxy via a `DELEGATECALL` to the master copy and returning all data returned by the `DELEGATECALL`.

DelegateConstructorProxy.sol

This is an extension to the proxy contract that allows further initialization logic to be passed to the constructor.

PayingProxy.sol

This is an extension to the delegate constructor proxy contract that pays a specific amount to a target address after initialization.

ProxyFactory.sol

The proxy factory allows to create new proxy contracts pointing to a master copy and executing a function in the newly deployed proxy in one transaction. This additional transaction can for example execute the setup function to initialize the state of the contract.

MasterCopy.sol

The master copy contract defines the master copy field and has simple logic to change it. The master copy class should always be defined first if inherited.

EtherPaymentFallback.sol

Base contract with a fallback function to receive Ether payments.

Executor.sol

The executor implements logic to execute calls, delegatecalls and create operations.

ModuleManager.sol

The module manager is an executor implementation which allows the management (add, remove) of modules. These modules can execute transactions via the module manager.

A linked list is used to store the enabled modules in the smart contract. To modify the list with minimal gas usage it is required to specify the module that should be modified and the module that points to this module. This is important when disabling a module. If multiple transactions disabling modules are submitted at once it is important to note that the module that points to the module that should be disabled might have changed. The linked list requires a sentinel (start and end pointer). This sentinel is the `0x1` address. Therefore this address cannot be used as a module.

OwnerManager.sol

The owner manager allows the management (addition, removal, replacement) of owners. It also specifies a threshold that can be used for all actions that require the confirmation of a specific amount of owners.

For managing the owners a linked list is used as well (see ModuleManager.sol). Modifying transactions that require to specify the owner pointing to the owner that should be modified include `removeOwner` and `swapOwner`. Also here the sentinel is the `0x1` and therefore it is not possible that this address becomes an owner.

BaseSafe.sol

The Gnosis Safe contract implements all basic multisignature functionality. It allows to execute Safe transactions and interact with Safe modules from internal methods. The contract provides no methods to interact with the Safe contract and also has no functionality to check if any interaction was approved by the required amount of owners. This logic and the methods to interact with the Gnosis Safe need to be implemented by the sub-contracts.

Safe transactions can be used to configure the wallet like managing owners, updating the master copy address or whitelisting of modules. All configuration functions can only be called via transactions sent from the Safe itself. This assures that configuration changes require owner confirmations.

Before a Safe transaction can be executed, the transaction has to be confirmed by the required number of owners.

There are multiple implementations of the Gnosis Safe contract with different methods to check if a transaction has been confirmed by the required owners.

Gnosis Safe

GnosisSafe.sol

This contract implements verification of approvals when execution transactions via the contract.

To execute a transaction the method `execTransaction` can be used. To approve a transaction it is necessary to generate and encode the required signatures.

There are different types of signatures:

1. ECDSA signatures generated by Externally Owned Accounts
2. EIP-1271 based contract signatures
3. Pre-validated transaction hash signatures

For more information on signature types including how to generate and encode them, see Signatures.

Modules

Modules allow to execute transactions from the Safe without the requirement of multiple signatures. For this Modules that have been added to a Safe can use the `execTransactionFromModule` function. Modules define their own requirements for execution. Modules need to implement their own replay protection.

Modules are smart contracts which implement a concrete Safe's functionality separating its logic from the Safe's contract. Keep in mind that modules allow the execution of transactions without needing confirmations, while this allows the implementation of many advanced use cases it also introduces additional attack vectors.

Modules can be included in the Safe according to owners' requirements, making the process of creating Safes more gas efficient (not all Safes should include all modules). They also enable developers to include their own features without compromising a Safe's core functionality, having all benefits of developing an isolated smart contract. Modules that are used on a Safe should always be reviewd and audited in a similar manner as the core fuctionality of the Safe, to make sure that no exploits are introduced.

StateChannelModule.sol

This module is meant to be used with state channels. It is a module similar to the base contract, but without the payment option (it has the same method for execution named `execTransaction`, but with different parameters). Furthermore this version doesn't store the nonce in the contract but for each transaction a nonce needs to be specified.

DailyLimitModule.sol

The Daily Limit Modules allows an owner to withdraw specified amounts of specified ERC20 tokens on a daily basis without confirmation by other owners. The daily limit is reset at midnight UTC. Ether is represented with the token address 0. Daily limits can be set via Safe transactions.

SocialRecoveryModule.sol

The Social Recovery Modules allows to recover a Safe in case access to owner accounts was lost. This is done by defining a minimum of 3 friends' addresses as trusted parties. If all required friends confirm that a Safe owner should be replaced with another address, the Safe owner is replaced and access to the Safe can be restored. Every owner address can be replaced only once.

WhitelistModule.sol

The Whitelist Modules allows an owner to execute arbitrary transactions to specific addresses without confirmation by other owners. The whitelist can be maintained via Safe transactions.

Libraries

Libraries can be called from the Safe via a `DELEGATECALL`. They should not implement their own storage as this storage won't be accessible via a `DELEGATECALL`.

MultiSend.sol

This library allows to batch transactions and execute them at once. This is useful if user interactions require more than one transaction for one UI interaction like approving an amount of ERC20 tokens and calling a contract consuming those tokens. Each sub-transaction of the multi-send contract has an operation. With this it is possible to perform `CALL`s and `DELEGATECALL`s.

CreateAndAddModules.sol

This library allows to create new Safe modules and whitelist these modules for the Safe in one single transaction.

1.1.2 Deployment

The Gnosis Safe smart contract was written with the usage of a proxy contract in mind. Because of that there is no constructor and it is required to call an initialize function on the contract before it can be used. For this it is recommended to use the ProxyFactory or the DelegatingConstructorProxy.

Initialization

After the Safe proxy is deployed it needs to be initialized. As the general proxy has no constructor it is necessary to initialize the contract using a function call.

For this the **GnosisSafe.sol** contract exposes the method `setup`. It requires four parameters:

1. owners - List of Safe owners.
2. threshold - Number of required confirmations for a Safe transaction.
3. to - Contract address for optional delegate call.
4. data - Data payload for optional delegate call.

Using `to` and `data` it is possible to nest advanced initialization. One of the use cases would be to initialize a Safe with some default modules. This can be done using the **CreateAndAddModules.sol** library.

The module assumes that Proxies are used for all modules. The library will use a **ProxyFactory** to deploy a Proxy for each module. So when triggering the `setup` method `to` would be the address of the deployed libary contract and `data` the call to the method `createAndAddModules`. This method has two parameters:

1. proxyFactory - Address of the Proxy factory used to create the Proxy for each module
2. data - Modules initialization payload. This is the data for each proxy factory call concatinated.

For a complete example see the CreateAndAddModules test

Trustless deployment with ERC20 Tokens

Using the ProxyFactory or deploying a proxy requires that the user has Ether on an externally owned account. To make it possible to pay for the creation with any token or Ether the following flow is used.

1. Create deployment transaction. The PayingProxy enables the payment in any ERC20 token. Once the proxy is deployed it will refund a predefined address with the funds present at the address where it was deployed.
2. To make the deployed address deterministic it is necessary to use a known account and calculate the target address. To make this trustless it is recommended to use a random account that has nonce 0. This can be done by creating a random signature for the deployment transaction. From that transaction it is possible to derive the sender and the target address.
3. The user needs to transfer at least the amount required for the payment to the target address.
4. Once the payment is present at the target address the relay service will fund the sender with Ether required for the creation transaction.
5. As soon as the sender is funded the creation transaction can be submitted.

For more details on the Safe deployment process please checkout the DappCon 2018 presentation

Planned usage of Create2

The described approach for trustless deployment requires 3 transactions:

1. Fund calculated Safe address
2. Fund address that will deploy the Proxy contract
3. Deploy the Proxy contract

Using the new `create2` opcode makes it possible to use a factory without having to worry about the nonce of the factory. By using a factory it is possible to eliminate one of the transactions required by described flow. The adjusted flow would be the following:

1. Fund calculated Safe address (address is based on **factory address**, the **init code** of the contract that is deployed and a **salt**)
2. Trigger the Proxy factory

To make sure that the correct contract will be deployed the Safe configuration should be part of the **init code** and the **salt** should be generated in a way that all parties involved can verify that it was not manipulated.

1.1.3 Transaction Execution

To execute a transaction with the Gnosis Safe the `execTransaction` methods needs to be called with the following parameters:

- to, value, data - Safe transaction information
- operation - Operation that should be used for the Safe Transaction. Can be `CALL` (uint8 - `0`), `DELEGATECALL` (uint8 - `1`) or `CREATE` (uint8 - `2`)
- safeTxGas - minimum gas provided for the Safe transaction. In case of `CALL` and `DELEGATECALL` this is also the maximum available gas (gas limit).
- dataGas - base gas for the execution the Safe transaction
- gasPrice - price used to calculate the gas costs that are refunded to the relayer.
- gasToken - Token used for gas cost payment. If `0x0` then Ether is used. Gas costs are calculated by `(dataGas + txGas) * gasPrice`
- refundReceiver - Address of receiver of gas payment (or 0 if tx.origin should be used).
- signatures - hex encoded signatures (`execTransaction` expects that the signatures are sorted by owner address. This is required to easily validate no confirmation duplicates exist)

There need to be enough signatures to reach the threshold configured on Safe setup. To generate a signature a Safe owner generates a hash based on EIP-712 and generates signatures for it.

The `nonce` of the Safe contract is a public variable and increases after each execution of a Safe transaction (every time `execTransaction` is executed).

When a transaction was submitted the contract will store the gas left on method entry. Based on this the contract will calculate the gas used that the user needs to pay.

Before executing the Safe transaction the contract will check the signatures of the Safe, to ensure that the transaction was authorized by the Safe owners, and check that enough gas is left to fullfill the gas requested for the Safe transaction (`safeTxGas`). If these checks fail the transaction triggering `execTransaction` will also fail. This means the relayer will not be refunded.

After the execution of the Safe transaction the contract calculates the gas that has been used based on the start gas. If the `gasPrice` is set to 0 no refund transaction will be triggered.

Refunds are not included in the calculated gas costs, since the contract uses `gasLeft()` to calculate how much gas has been used.

Transaction Hash

The transaction hash is generated based on EIP-712 and the following EIP712 domain object is used:

```
{
    EIP712Domain: [
        { type: "address", name: "verifyingContract" }
    ]
}
```

The following object describes the typed data that is signed:

```
{
    SafeTx: [
        { type: "address", name: "to" },
```

(continues on next page)

(continued from previous page)

```
        { type: "uint256", name: "value" },
        { type: "bytes", name: "data" },
        { type: "uint8", name: "operation" },
        { type: "uint256", name: "safeTxGas" },
        { type: "uint256", name: "dataGas" },
        { type: "uint256", name: "gasPrice" },
        { type: "address", name: "gasToken" },
        { type: "address", name: "refundReceiver" },
        { type: "uint256", name: "nonce" },
    ]
}
```

For an example take a look at the eth_signTypedData test in the Safe contracts repository.

On chain approvals

It is not always possible to generate a ECDSA signature for a transaction hash (e.g. a smart contract is the owner of a Safe). In this case it is possible that an owner approves the hash on-chain.

`approveHash` can be used with the generated transaction hash to mark it as approved by an owner. This is stored on-chain in the Safe contract. Once the transaction with this hash has been executed the approval will be removed (to free the storage).

Note: There is no method to revert the approval of the transaction hash without the transaction being executed.

The relayer of a transaction can automatically approve the Safe transaction if he is an owner. For more information on the different types of signatures see Signatures

Failing Safe Transactions

If the execution of a Safe transaction fails the contract will emit the `ExecutionFailed` event that contains the transaction hash of the failed transaction. The transaction triggering `execTransaction` will not fail, since the relayer should still be refunded in this case.

Safe Transaction Gas Limit Estimation

The user should set an appropriate `safeTxGas` to define the gas required by the Safe transaction, to make sure that enough gas is send by the relayer with the transaction triggering `execTransaction`. For this it is necessary to estimate the gas costs of the Safe transaction.

To correctly estimate the call to `execTransaction` it is required to generate valid signatures for a successful execution of this method. This opens up potential exploits since the user might have to sign a very high `safeTxGas` just for estimation, but the signatures used for the estimation could be used to actually execute the transaction.

One way to estimate Safe transaction is to use `estimateGas` and with the following parameters:

```
{
    "from": <Safe address>,
    "to": <`to` of the Safe transaction>,
    "value": <`value` of the Safe transaction>,
    "data": <`data` of the Safe transaction>,
}
```

While it is possible to estimate a normal transactions (where `operation` is `CALL` or `CREATE`) like this, it is not possible to estimate `DELEGATECALL` transactions this way. Also the value returned by `estimateGas` might include refunds (e.g. Ganache) and the base transaction costs.

For a more accurate estimate it is recommended to use the `requiredTxGas` method of the Safe contract. The method takes `to`, `value`, `data` and `operation` as parameters to calculate the gas used in the same way as `execTransaction`. Therefore it will not include any refunds or base transaction costs.

To avoid that this method can be used inside a transaction two security measures have been put in place:

1. The method can only be called from the Safe itself
2. The response is returned with a revert

The value returned by `requiredTxGas` is encoded in a revert error message (see solidity docs at the very bottom). For retrieving the hex encoded uint value the first 68 bytes of the error message need to be removed.

Safe Transaction Data Gas Estimation

The `dataGas` parameter can be used to include additional gas costs in the refund. This could include the base transaction fee of 21000 gas for a normal transaction, the gas for the data payload send to the Safe contract and the gas costs for the refund itself.

To correctly estimate the gas costs for the data payload without knowing the signatures, it is suggested to generate the transaction data of `execTransaction` with random signatures and `dataGas` set to 0. The costs for this data are 4 gas for each zero-byte and 68 gas for each non-zero-byte.

Transactions without refund

As it is not always required to refund the relayer of the transaction it is possible to simplify the parameters in that case.

If the `gasPrice` is set to `0` there will be no transfer triggered to refund the relayer. This makes it unncessary to specify `dataGas`, `gasToken` or `refundReceiver` (can be set to `0`)

In addition if also the `safeTxGas` is set to `0` all available gas will be used for the execution of the Safe transaction. With this it is also unnecessary to estimate the gas for the Safe transaction.

1.1.4 Signatures

The Safe supports different types of signatures. All signatures are combined into a single `bytes` and transmitted to the contract when a transaction should be executed.

Encoding

Each signature has a constant length of 65 bytes. If more data is necessary it can be appended to the end of concatenated constant data of all signatures. The position is encoded into the constant length data.

Constant part per signature: `{(max) 64-bytes signature data}{1-byte signature type}`

All the signatures are sorted by the signer address and concatenated.

ECDSA Signature

`signature type > 26`

To be able to have the ECDSA signature without the need of additional data we use the signature type byte to encode `v`.

Constant part:

`{32-bytes r}{32-bytes s}{1-byte v}`

`r`, `s` and `v` are the required parts of the ECDSA signature to recover the signer.

Contract Signature (EIP-1271)

`signature type == 0`

Constant part:

`{32-bytes signature verifier}{32-bytes data position}`

Signature verifier - Padded address of the contract that implements the EIP 1271 interface to verify the signature
Data position - Position of the start of the signature data (offset relative to the beginning of the signature data)

Dynamic part:

`{bytes signature data}`

Signature data - Signature bytes that are verified by the signature verifier

The method `signMessage` can be used to mark a message as signed on-chain.

Pre-Validated Signatures

`signature type == 1`

Constant Part:

`{32-bytes hash validator}`

Hash validator - Padded address of the account that pre-validated the hash that should be validated. The Safe keeps track of all hashes that have been pre validated. This is done with a **mapping address to mapping of bytes32 to boolean** where it is possible to set a hash as validated by a certain address (hash validator). To add an entry to this mapping use `approveHash`. Also if the validator is the sender of transaction that executed the Safe transaction it is **not** required to use `approveHash` to add an entry to the mapping. (This can be seen in the Team Edition tests)

Examples

Assuming that three signatures are required to confirm a transaction where one signer uses an EOA to generate a ECDSA signature, another a contract signature and the last a pre-validated signature:

We assume that the following addresses generate the following signatures:

1. `0x3` (EOA address) -> `bde0b9f486b1960454e326375d0b1680243e031fd4fb3f070d9a3ef9871ccfd5` (r) + `7d1a653cffb6321f889169f08e548684e005f2b0c3a6c06fba4c4a68f5e00624` (s) + `1c` (v)
2. `0x1` (EIP-1271 validator contract address) -> `000000000000000000000000000000000000000000000000000000000000000` (address) + `00000000000000000000000000000000000000000000000000000000000000c3` (dynamic position) + `00` (signature type)
    - The contract takes the following `bytes` (dynamic part) for verification `00000000000000000000000000000000000000000000000000000000deadbeef`
3. `0x2` (Validator address) -> `0000000000000000000000000000000000000000000000000000000000000002` (address) + `0000000000000000000000000000000000000000000000000000000000000000` (padding - not used) + `01` (signature type)

The constant parts need to be sorted so that the recovered signers are sorted **ascending** (natural order) by address (not checksummed).

The signatures bytes used for `execTransaction` would therefore be the following:

```
"0x" +

↪"000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000000000000
↪" + // encoded EIP-1271 signature

↪"000000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000000000000
↪" + // encoded pre-validated signature

↪"bde0b9f486b1960454e326375d0b1680243e031fd4fb3f070d9a3ef9871ccfd57d1a653cffb6321f889169f08e548684e0
↪" + // encoded ECDSA signature

↪"000000000000000000000000000000000000000000000000000000000000000800000000000000000000000000000000000000
↪"     // length of bytes + data of bytes
```

1.1.5 Meta transactions relays and the Safe

To improve the user experience of the clients Gnosis is offering a relay service. There are multiple endpoints on this service for the purpose of deploying a Safe contract, estimating transactions and submitting transactions.

As all the parameters required for execution are part of the submitted transaction it is possible that miners front-run the original relayer to receive the reward. In the long run that behaviour would be appreciated, since it would allow that anybody submits these transactions with `gasPrice` of the transaction triggering `execTransaction` set to 0. Miners could pick up these transactions and claim the rewards. This is why it is possible to specify the account that should receive the refund.

The vision for submitting Safe transactions would be the following:

1. Acquire estimates for a Safe transaction from potential relayers
2. Choose a relayer and generate the signatures for the Safe transaction (the chosen relayer should be set to receive the rewards to prevent front-running)
3. Submit the Safe transaction with all required paramters to the chosen relayer.

Currently this is implemented as a simple REST API. But the idea would be to make use of an open decentralized system that supports different relayers (see MetaCartel).

Rationale of parameters

The `execTransaction` has quite some parameters which might be unnecessary for other projects that want to use a relay service. To better understand why this set of paramters was chosen the rationale of them will be outlined.

`operation`

The Safe contract allows the execution different types of transactions. These are differentiated by the operation. Currently the Safe supports three different types of transactions: `CALL`, `DELEGATECALL` and `CREATE`.

`safeTxGas`

When a relayer submits a transaction with valid signatures it should be paid even if the Safe transaction fails. This has been done for the following reasons:

1. If the transaction fails the signatures stay valid. Therefore it would be possible to potentially replay the transaction.
2. The relayer should always be paid even if the Safe transaction fails (e.g. due to state changes)

Is is necessary that the relayer cannot make the Safe transaction to fail on purpose. This would make it possible that the relayer gets paid without performing the Safe transaction. For this the client needs to specify the minimum required gas for the Safe transaction. This is similar to the gas limit of a normal Ethereum transaction.

`dataGas`

As outlined before `safeTxGas` only specifies how much gas should be available for the Safe transaction. The gas that the client needs to pay for is determined at runtime. But there are some static costs. This includes the costs for the payload and the gas required to perform the payment transfer.

`refundReceiver`

It is possible to specify the receiver of the refund to avoid that submitted transactions can be front-run by others.

Demo (on Rinkeby)

```
mkdir safe-demo
cd safe-demo
truffle unbox gnosis/safe-demo
```

1.2 Services Overview

1.2.1 Notification Service

Allows users to send signed transaction messages between devices taking part in the signing process.

Database model

Device:

- pushToken (char, unique)
- owner (char, primary key)

DevicePair:

- authorizingDevice
- authorizedDevice

Primary key: both together

Pairing

Before a chrome extension can send push notifications to the phone a channel has to be established. Both, the chrome extension and the mobile app request a token from a notification service like firebase. They sign the token with their private key and submit the token to the notification service. To pair the chrome extension with a mobile app a QR code containing an expiry date and a signature are generated and displayed in the chrome extension. This QR contains a expiry date and a signature signing the expiry date:

```
{
    "expirationDate": "<date>",
    "signature": { // signs sha3("GNO" + <expirationDate>)
        "v": "<integer>",
        "r": "<string>", // stringified int (decimal)
        "s": "<string>" // stringified int (decimal)
    }
}
```

Note: The QR is a stringified object. The mobile app can scan the QR code and use the message to add itself as authorized device. It is then allowed to send push notifications to the chrome extension.

1.2.2 Endpoints

/auth/ POST

Pre-requirements:

- Generate local private key
- Ask firebase for push token
- Authorize notification service to send notifications to authorizing device.
- Request contains a expiry date. Notification service will only accept request in case expiry date is not in the past.

Request

```
{
    "pushToken": "<string>",
    "signature": { // signs sha3("GNO" + <pushToken>)
        "v": "<integer>",
        "r": "<string>", // stringified int
        "s": "<string>" // stringified int
    },
}
```

Response

If no previous owner exists, we create a new entry with push token and owner param Otherwise, update current Device entry.

Returns HTTP 201 if OK

```
{
    "pushToken": "<string>",
    "owner": "<string>"
}
```

Use https://firebase.google.com/docs/cloud-messaging/ in the clients to get push token

/pairing/ POST

Allows to authorize pairing of two devices. The signature of the expiry date signed by the Chrome extension is sent together with the signature of the Chrome extension ethereum account address signed by the mobile app. Pairing is only successful if expiry date is not expired.

Request

```
{
    "temporaryAuthorization": {
    "expirationDate": "<date>",
    "signature": { // signs sha3("GNO" + <expirationDate>)
        "v": "<integer>",
        "r": "<string>", // stringified int
        "s": "<string>" // stringified int
    },
    },
    "signature": { // signs sha3("GNO" + <chrome-extension-address>)
        "v": "<integer>",
        "r": "<string>", // stringified int
        "s": "<string>" // stringified int
    }
}
```

IMPORTANT: addresses must be in checksum and date in ISO format without microseconds and with timezone (you must use UTC)

Example date: 2018-04-18T14:46:09+00:00

Response

We create two entries for DevicePair, in both directions.

Returns HTTP 201 if OK

```
{
    "devicePair": [
        "<string>", "<string>"
    ]
}
```

/pairing/ DELETE

Allows to delete an authorization for a device for sending push notifications.

Request

```
{
    "device": "<address>",
    "signature": { // signs sha3("GNO" + <address>)
        "v": "<integer>",
        "r": "<string>", // stringified int
        "s": "<string>" // stringified int
    }
}
```

We remove the DevicePair where authorized

- device = address and authorizing = signer address
- authorizing = address and device = signer address

Response

Returns HTTP 204 if OK

/notifications/ POST

Allows to send notifications to multiple devices. If one of the pairs is not allowed, the service sends notifications to the others anyways.

Signature address cannot be contained in devices list.

Request

```
{
    "devices": ["<checksumed_address>", ...],
    "message": "<string>",
    "signature": { // signs sha3("GNO" + <message>)
        "v": "<integer>",
```

(continues on next page)

(continued from previous page)

```
        "r": "<string>", // stringified int
        "s": "<string>" // stringified int
    }
}
```

Response

- HTTP 204 if at least one notification is sent
- HTTP 404 if no device pair is found

Push Notification Types (message parameter)

Send Safe address to Chrome Extension

```
{
  "type": "safeCreation",
  "safe": "<address>",
}
```

Send transaction from chrome extension to app

```
{
  "type": "sendTransaction",
  "hash": "<hex-string>", // Hash of the safe transaction
  "safe": "<address>",
  "to": "<address>",
  "value": "<stringified-int>",
  "data": "<hex-string>",
  "operation": "<stringified-int>",
  "txGas": "<stringified-int>",
  "dataGas": "<stringified-int>",
  "operationalGas": "<stringified-int>",
  "gasPrice": "<stringified-int>",
  "gasToken": "<address>",
  "nonce": "<stringified-int>",
  // Signature of hash (DO NOT USE the GNO prefix or any additional hashing as this␣
↪signature is used with the safe smart contract)
  "r": "<stringified-int>",
  "s": "<stringified-int>",
  "v": "<stringified-int>"
}
```

The parameters txGas, dataGas, operationalGas and gasPrice can be retrieved from the Relay Service operationalGas is just used to display a more accurate estimate. gasToken is address(0) for ETH or the token that should be used

Request confirmation from chrome extension

```
{
  "type": "requestConfirmation",
  "hash": "<hex-string>", // Hash of the safe transaction
  "safe": "<address>",
  "to": "<address>",
  "value": "<stringified-int>",
  "data": "<hex-string>",
  "operation": "<stringified-int>",
  "txGas": "<stringified-int>",
  "dataGas": "<stringified-int>",
  "operationalGas": "<stringified-int>",
  "gasPrice": "<stringified-int>",
  "gasToken": "<address>",
  "nonce": "<stringified-int>"
}
```

Once the extension receives this push it can validate the hash and sign it. The signature then can be send back to the app

The transaction hash can be calculated with:

```
keccak256(byte(0x19), byte(0), this, to, value, data, operation, safeTxGas, dataGas,␣
↪gasPrice, gasToken, nonce)
```

Confirm transaction from chrome extension

```
{
  "type": "confirmTransaction",
  "hash": "<hex-string>", // Hash of the safe transaction
  // Signature of hash (DO NOT USE the GNO prefix or any additional hashing as this␣
↪signature is used with the safe smart contract)
  "r": "<stringified-int>",
  "s": "<stringified-int>",
  "v": "<stringified-int>"
}
```

Reject transaction from chrome extension

```
{
  "type": "rejectTransaction",
  "hash": "<hex-string>", // Hash of the safe transaction
  // Signature of sha3(GNO + hash + type)
  "r": "<stringified-int>",
  "s": "<stringified-int>",
  "v": "<stringified-int>"
}
```

Send Transaction Hash

```
{
  "type": "sendTransactionHash",
```

(continues on next page)

(continued from previous page)

```
  "hash": "<hex-string>", // Hash of the safe transaction
  "chainHash": "<hex-string>" // Hash of transaction on chain
}
```

1.2.3 Transaction Relay Service

This service allows us to have owners of the Safe contract that don't need to hold any ETH on those owner addresses. How is this possible? The transaction relay service acts as a proxy, paying for the transaction fees and getting it back due to the transaction architecture we use.

Our target user hold crypto in a centralized exchange (or on another Ethereum address) and wants to move it to a secure account. We don't want the user to trust us, for moving the funds and deploying the smart contract on their behalf. We on the other side want to prevent users from spamming our services, there shouldn't be a need to trust the user either.

That's why we came up with this solution. The user (phone app) and us (service) both generate a random signature for a valid transaction. With this valid signed transaction, anyone can submit this to the blockchain and a Safe would be created with the following specifics:

- No one knows the private key, so no previous transaction can be sent before this tx. Any former transaction would invalidate the signature due to an invalid nonce.
- The Safe address can be derived, because it is calculated from the address of the sender (the address can be derived in turn from the signed tx, cf. How is the address of an Ethereum contract computed?).
- This transaction will create the Safe contract and refund the service with funds from within the Safe contract. Thereby the service has the guarantee that once it executes the tx, which incurs tx fees, it will get them back as part of the tx execution.

Proxy used: PayingProxy

Flows

Safe creation flowchart

Transaction execution flowchart

API Endpoints

/safes/ POST

Creates new Safe Creation Transaction with random signature, generated by user and server, so no one knows the private key of the deployer address.

Note: We don't use a Chain ID to facilitate testing on different chains (cf. EIP-155. We don't need the replay protection, because no one knows the private key. The PayingProxy is used in this process. Furthermore, we use "fast" from our gas station. The First version will only support ETH as gasToken. Therefore the payment will be returned in Wei.

More info about the signature values in appendix F of the Ethereum Yellow Paper.

Request:

```
{
    "owners": ["<string>"], // Hexadecimal addresses, with checksum with 0x prefix,
    "threshold": "int", // min 1
    "s": "string", // stringified int, base 10,  (0 "< s "< secp256k1n / 2 + 1)
}
```

Returns

HTTP 201

```
{
    "signature": {
        "r": "<string>", // stringified int, base 10 (0 "< r "< secp256k1n)
        "s": "<string>", // stringified int, base 10 (0 "< s "< secp256k1n / 2 + 1)
        "v": "<int>" // (27 or 28)
    },
    "tx" : {
        "from": "<string>",
        "value":  "<string>", // stringified int, base 10 (wei) Will always be 0
        "data": "<string>",
        "gas": "<string>", // stringified int, base 10
        "gasPrice": "<string>", // stringified int, base 10 (wei)
        "nonce": 0
    },
    "safe": "<string>", // hex string with checksum
    "funder": "<string>", // hex string with checksum
    "payment": "<string>", // stringified int, base 10, it's what the service gets as␣
↪refund
}
```

HTTP 400 not valid values submitted

Note: Atomic operation, many values of s are invalid which are generated by the server.

Clients should verify the server's response with the following process:

1. Reconstruct sender address from `signature`
2. Reconstruct Safe address with sender address and `nonce = 0`
3. Verify that `Safe` checksum and reconstructed Safe addresses checksum are matching
4. Verify that `signature` is correct for hash of `tx`
5. Verify that `tx` has valid bytecode (postponed until mainnet release).
6. If all checks pass, then transaction and Safe address are valid and user can transfer at least `payment` amount of ETH to the Safe address.
7. Otherwise, the response has error or it is compromised, and it should not be used any further.

/safes/<address>/funded/ PUT

Signal funds were transferred, start Safe creation

Returns:

HTTP 202

Note: Creation has 2 txs and a check/. This is done asynchronously through a queue.

/safes/<address>/funded/ GET

Get info about Safe's funding status

Returns:

HTTP 200

```
{
    "safeFunded": "<boolean>",  # Safe has enough balance to start the deploying
    "deployerFunded":  "<boolean>",  # Deployer was funded and confirmations awaited
    "deployerFundedTxHash": "<string>",  # Deployer funding tx hash
    "safeDeployed": "<boolean>",  # Safe was finally deployed
    "safeDeployedTxHash": "<string>"  # Safe tx was sent to the network
}
```

/gas-station/ GET

Similar to ETH Gas Station but with reliable availability and sufficient rate limits

Returns:

HTTP 200

```
{
    "safeLow": "<string>", // stringified int, wei
    "standard": "<string>", // stringified int, wei
    "fast": "<string>", // stringified int, wei
    "fastest": "<string>", // stringified int, wei
}
```

/safes/<address>/transactions/estimate/ POST

Estimates the gas and gasPrice for the requested Safe transaction. Safe contract needs to exist previously. To estimate transaction cost, use the following formula:

gasCosts = (safeTxGas + dataGas) * gasPrice

Request:

```
{
    "to": "<address>",
    "value": "<string>", // stringified int, in wei
    "data": "<string>", // prefixed or not prefixed hex string
    "operation": "<integer>", // enumerated from here (0 - call, 1 - delegateCall, 2 -
↪ create)
    "nonce": "<integer>" // nonce of the last tx sent for execution
}
```

Returns:

HTTP 200

```
{
    "safeTxGas": "<integer>"
    "dataGas": "<integer>"
    "gasPrice": "<integer>"
    "gasToken": "<string>" // hexadecimal address, checksummed, address(0) for now
}
```

/safes/<address>/transactions/ POST

Allows to send and pay transactions via the Transaction Relay Service. The Safe contract the tx is directed to must have enough ETH to pay tx fees and be created through the tx relay service. Safe contract needs to exist previously.

Request:

```
{
    "to": "<address>",
    "value": "<string>", // stringified int, in wei, base 10
    "data": "<string>", // prefixed or not prefixed hex string
    "operation": "<integer>", // enumerated from here
    "signatures": [{
        "v": "<integer>",
        "r": "<string>",
        "s": "<string>"
    }, ...], // Sorted lexicographically by owner address in lowercase
    "safeTxGas": "<string>" // stringified int, base 10
    "dataGas": "<string>" // stringified int, base 10
    "gasPrice": "<string>" // stringified int, base 10
    "nonce": "<string>" // stringified int, base 10
    "gasToken": "<string>" // hexadecimal address, checksummed, address(0) for now
}
```

Returns:

HTTP 201

```
{
    "transactionHash": "<string>"
}
```

Note: Atomic operation.

/tokens/ GET

Returns a paginated list of tokens. Each token has the ERC20 information (address, name, symbol, decimals) and if available additional meta information to the token (icon, website . . .). Furthermore tokens can be marked to be shown to the user by `default`.

Notes:

- Currently token info is retrieved from etherscan

Query params:

Besides pagination:

- `search`: Search words in `name`, `symbol` and `description`.
- `name`, `symbol` and `address`: Do an exact filtering based on that parameters.
- `default`: If `1` just show tokens marked to be shown by default.
- `gas`: If `1` just show tokens that can be used to pay for gas.
- `decimals__lt` and `decimals__gt`: Filter based on tokens with decimals *less than* or *greater than*.

Response

Returns HTTP 200

```
{
  "count":432,
  "next":"${host}:${port}/api/v1/tokens/?limit=100&offset=200",
  "previous":"${host}:${port}/api/v1/tokens/?limit=100",
  "results": [
    {
      "address": <hex encoded checksummed address (0xaBc1....),
      "name": <string>,
      "symbol": <string>,
      "decimals": <int>,
      "logoUri": <string>,
      "websiteUri": <string>,
      "default": <bool>,
      "gas": <bool>,  // If token can be used as gas token
    }
  ]
}
```

1.2.4 History Service

Keeps track of transactions sent via Gnosis Safe contracts and confirmed transactions.

Show on GitHub

1.3 Safe Clients

1.3.1 Android Client (Safe Personal Edition)

Show on GitHub

1.3.2 iOS Client (Safe Personal Edition)

Show on GitHub

1.3.3 Chrome Extension (Safe Personal Edition)

Show on GitHub

1.3.4 Web React Client (Safe Team Edition)

Show on GitHub