Wrapped bitcoin token smart contract

Introduction

This document describes the current implementation and architecture of the wrapped bitcoin token (WBTC) smart contracts and its minting and burning functionalities. We refer the reader to the whitepaper to get a broader scope of the system, e.g., legal agreements between parties, interaction between merchants and end users, and auditing process.

The WBTC ecosystem consists of custodians (e.g., bitgo), merchants (e.g., kyber and republic) and end users. The token adhere the ERC20 token standard with the extension that tokens can be minted and burned. Standard ERC20 functionalities (i.e., beside miniting and burning) are fully exposed to the end user.
Conceptually, WBTC tokens are minted when a merchant deposit BTC in the custodian vault. Merchant can burn a portion of its WBTC tokens and in return receive BTC. In practice however, as currently bitcoin transfers cannot be monitored by an ethereum smart contract, the minting process requires one merchant to init a minting request and one custodian to approve.

In the rest of the document, we describe the technical requirements, present an overview of the implementation architecture, and then give details for every component in the system.

Technical requirements

1. ERC20 compliant token.
2. Only merchants can burn tokens.
3. Minting occur upon merchant request and custodian approval. Namely, in order for minting to happen a single custodian and a single merchant have to agree.
4. The list of merchants and custodians is controlled by a DAO, and the DAO implementation is outside the scope of the contract implementation. Initially a multisig will control it.
5. Upgradability. The following components are upgradable by a DAO. We note that the DAO may or may not be the same as the DAO of item 4.
    a. The token transfers are paused, and the token contract is hard forked.
    b. The burning and miniting protocols are upgradable.
    c. The process of adding new custodians and merchants is upgradable.
    d. The DAO is upgradable. Meaning the DAO can transfer ownership to a new DAO.
6. Easy auditability. The minting and burning process should be auditable by anyone with an access to Ethereum and Bitcoin blockchains. The protocol support a behavior that

satisfy the following invariant: **At any point in time, the amount of BTC in custodian wallet(s) is greater or equal to the total supply of WBTC**. (assuming no unreasonable reorgs in both blockchains)
7. Automation of the minting and burning process. If needed, the process can be done automatically on both merchant and custodian side, and more over, with the blockchain(s) being the only communication channels between the two.

Trust Model

Custodians and Merchants are bind to a legal agreement and also curated and audited by the DAO governance system. However, in the smart contract scope we restrict the trust model to the following assumptions:

Strict trust

1. The DAO (as a collective) is fully trusted.
2. Custodian is being trusted to confirm valid mint requests. I.e, a request followed (or preceded) by a BTC deposit to custodian deposit address with greater or equal amount.
3. Custodian is being trusted to confirm burn requests and send BTC to merchant deposit address (as depicted in the burn request).

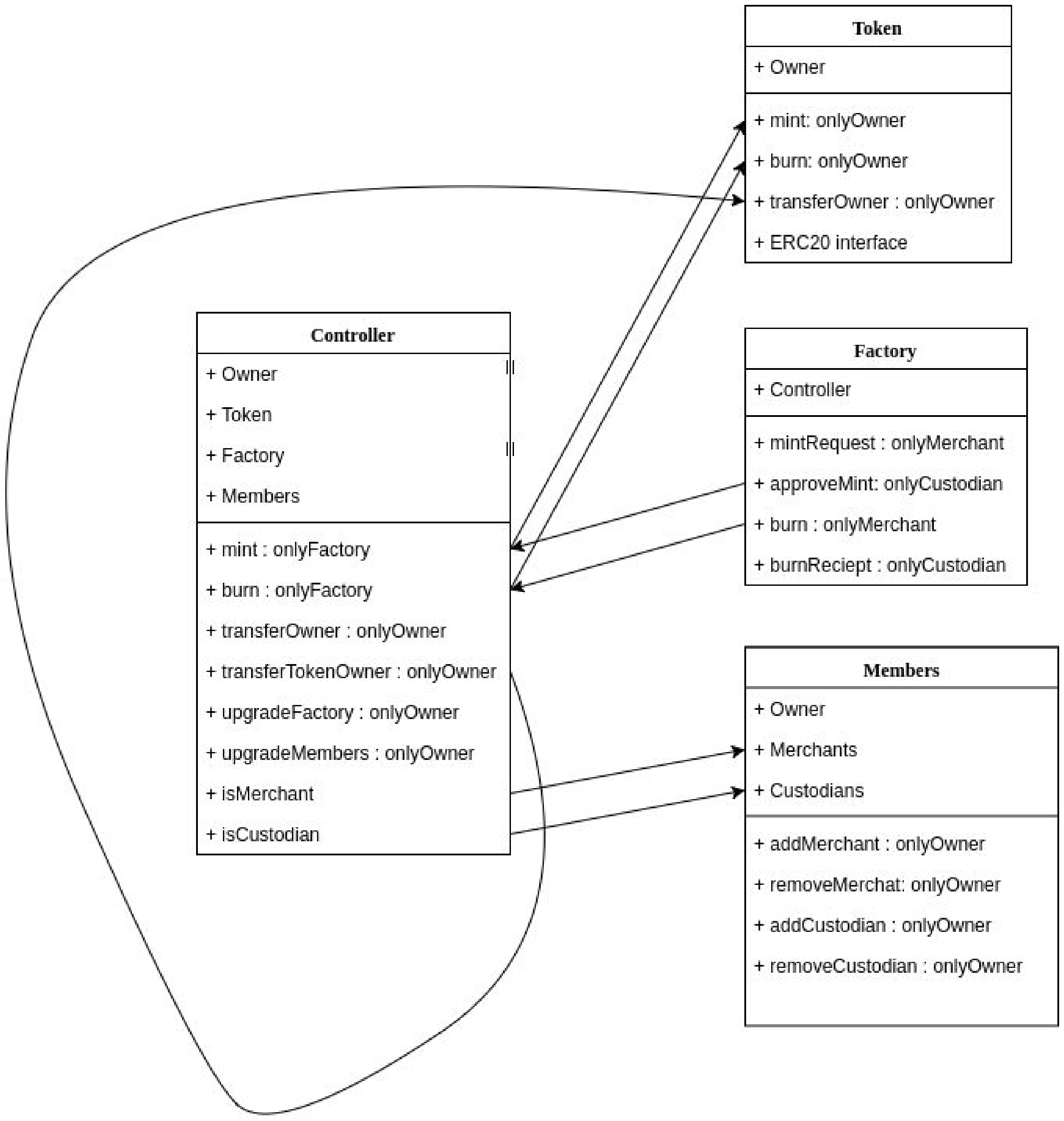
Soft trust

The following is enforced by off-chain agreements and/or being monitored by the DAO.
1. Merchant is trusted not to spam the custodian with dust requests.
2. Custodian is trusted not to frequently change his BTC deposit addresses.

Architecture Overview

The end user interacts only with the ***WBTC*** component which expose an ERC20 interface. The implementation also have an interface and implementation for minting and burning. The owner of the token is the ***Controller*** contract.
The ***Controller*** contract is owned by a DAO (e.g., a multisig wallet) which has the authority to:
1. Pause the token transfers
2. Upgrade the minting and burning protocols
3. Upgrade the protocol for adding (and removing) new members, namely, merchants and custodians.

The initial implementation of the burning and minting protocol is defined in the ***Factory*** contract. The initial implementation of the member selection is implemented in the **Members** contract. The connection between the different contracts is depicted in the figure below.


Token
+ Owner
+ mint: onlyOwner
+ burn: onlyOwner
+ transferOwner : onlyOwner
+ ERC20 interface
Controller
+ Owner
+ Token
+ Factory
+ Members
+ mint : onlyFactory
+ burn : onlyFactory
+ transferOwner : onlyOwner
+ transferTokenOwner : onlyOwner
+ upgradeFactory : onlyOwner
+ upgradeMembers : onlyOwner
+ isMerchant
+ isCustodian
Factory
+ Controller
+ mintRequest : onlyMerchant
+ approveMint: onlyCustodian
+ burn : onlyMerchant
+ burnReciept : onlyCustodian
Members
+ Owner
+ Merchants
+ Custodians
+ addMerchant : onlyOwner
+ removeMerchat: onlyOwner
+ addCustodian : onlyOwner
+ removeCustodian : onlyOwner

Detailed implementation

Directories structure

1. **token** the token implementation.

2. **controller** the controller implementation.
3. **factory** the factory and members implementation.
4. **utils** auxiliary util contracts.
    a. **IndexedMapping.sol** an iterable data structure that support O(1) insertions and deletions. Used to keep an iterable and easy auditable list of current merchants and custodians.
    b. **OwnableContract.sol** a basis of all contracts to inherit from. Combined open-zepplin ownable scheme with open-zepplin scheme that allows to extract stuck tokens, and secure open-zepplin ownership transfer scheme.
    c. **OwnableContractOwner.sol** a basis for a contract that owns other contracts.

Token

Overview:
ERC20 compatible token. Its code fully taken from open-zepplin.

Things to notice:

1. Multiple inheritance. Contract inherit from multiple contracts that inherit from **ownable**. And also from **Claimable**, which overrides **ownable**. While this is a bad practice in modern programming languages, open-zepplin implementation dictates the use of such an approach. We made sure **Claimable** is the first in the inheritance hierarchy.
2. Burning functionality is enabled only for owner.
3. Finish minting functionality is disabled.
4. **public** (rather than **external**) visibility is used to maintain consistency with open-zepplin code.
5. Inconsistency in return value type. Open zepplin code is sometimes inconsistent. For example the **burn** function does not have a return value, while other functions such as **finishMinitng** return a boolean value.

Controller

Overview:
Owner of the token and provides upgradable interface for members (merchants/custodian) query and token minting/burning.

Things to notice:

1. Even though merchants and custodians are selected in a different contract, the controller has, conceptually, a final say, as it can upgrade/change the selection process.
2. **getWBTC()** function return a value that is already visible via a public variable, however it is needed in order to include it in the interface.

Factory.sol

Overview:

Implements the minting and burning protocol. To be precise the initial process. It could later be upgraded by the controller. A demo document on the process is given in the appendix.

Things to notice:

1. The protocol is tailored for a setting with a single custodian and multiple merchants. It does have a limited support for multiple custodian entities, however we assume they all have the same deposit address. A precise protocol to support multiple custodians is still not defined in the whitepaper.
2. To prevent erroneous minting requests, we force the merchant to input the custodian deposit address when initiating a new mint request which also include the bitcoin tx hash. This can give rise to a situation in which the custodian changed his deposit address after the merchant bitcoin transaction was done but prior to the smart contract new mint request submission. We propose two ways to mitigate it:
    a. Submitting the minting request, with the bitcoin tx hash **before** broadcasting it to the bitcoin blockchain. This is only partial mitigation as it might change the trust model in the merchant side. E.g., in a multisig setting, it will assume some trust in the tx sender (on the bitcoin side). In addition, this operation might not be easy for novice users.
    b. Off chain settlement. We expect the custodian to notify to the merchant ahead of time on address changes. And in case he didn't merchant can submit an alternative, up to date, deposit address along with the minting request, and handle things with the custodian off-chain.
3. The above issue is solved for the burn process by logging current merchant deposit address in the burn request.
4. Burn request confirmation is only for auditing process. Burn requests cannot be rejected or cancelled (as oppose to minting requests). However, custodian can choose to ignore it if it is a dust request.
5. Requests id. For the purpose of easy auditability we place the minting and burning request in an append only array. Each request has a unique nonce which stands for its position in the array. However, to mitigate blockchain reorg we require the custodian to identify and confirm the requests according to a unique hash over the request data, rather then by their nonce.
    a. A burn request hash record is expected to change after a **confirmBurnRequest** operation (beacuse of btcTxid field change), but after the confirmation the hash serves no additional purpose.
6. The owner of the contract is the controller, however owner role is limited to extracting struck tokens.

Members.sol

Overview:
Implements a naive add/remove functionality for merchants and custodians. Future logic/constraints could be implemented in the controlling dao.

Things to notice:

1. The list of members and custodians is implemented with an iterable data structure that supports O(1) additions and removal. This is done for easier auditability.
2. **getMerchants()** and **getCustodians()** should be used only by offchain calls and are only useuable for quick audit when number of members is small.

IndexedMapping.sol

Overview:
A data structure is used to hold the merchants and custodians lists, so these mappings could be iterable.

OwnableContract.sol and OwnableContractOwner.sol

Things to notice:
We decided to exclude the use of open-zeppelin **HasNoEther** contract as it would make the implementation of OwnableContractOwner.sol more difficult.

Appendix

WBTC Demo

Basic scenario: mint and burn wbtc.
If time permits: adding another merchant.

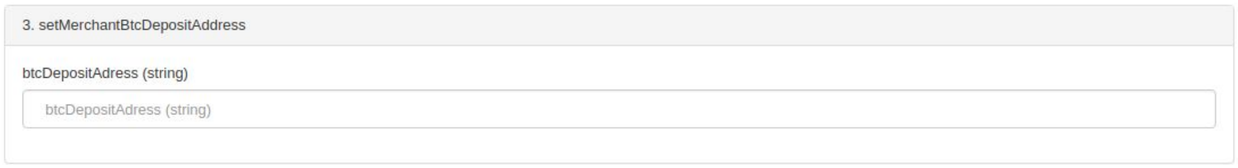
Basic addresses

Republic merchant = 0x00655ea989254c13e93c5a1f74c4636b5b9926b5
Bitgo custodian = 0xbcf59c97276be264df3310ca7e5cc5a8f4c26962

Factory contract = 0x6bC567F37C2e5A6d0B89906Fe5BefC199a57C5A2 (take abi from here: https://kovan.etherscan.io/address/0x6bc567f37c2e5a6d0b89906fe5befc199a57c5a2#code)

Basic setup

1. Republic: Set merchant BTC deposit address. This is the address republic is expecting



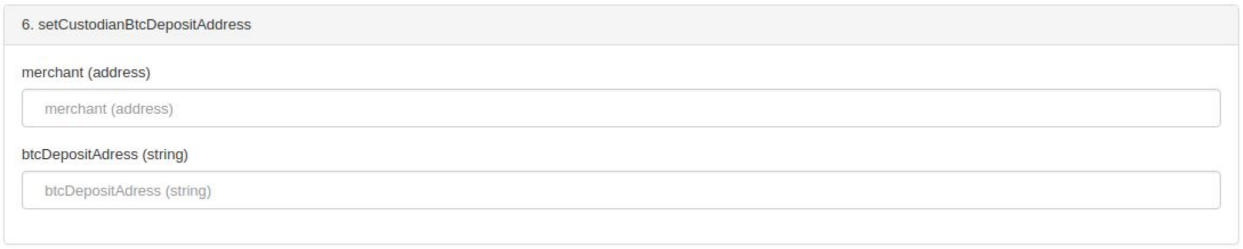
3. setMerchantBtcDepositAddress

btcDepositAdress (string)

btcDepositAdress (string)

to get its BTC from bitgo, after burning wbtc tokens.
Function: setMerchant**BtcDepositAddress**

2. Bitgo: set the deposit address to which republic should send its bitcoin in minting.



6. setCustodianBtcDepositAddress

merchant (address)

merchant (address)

btcDepositAdress (string)

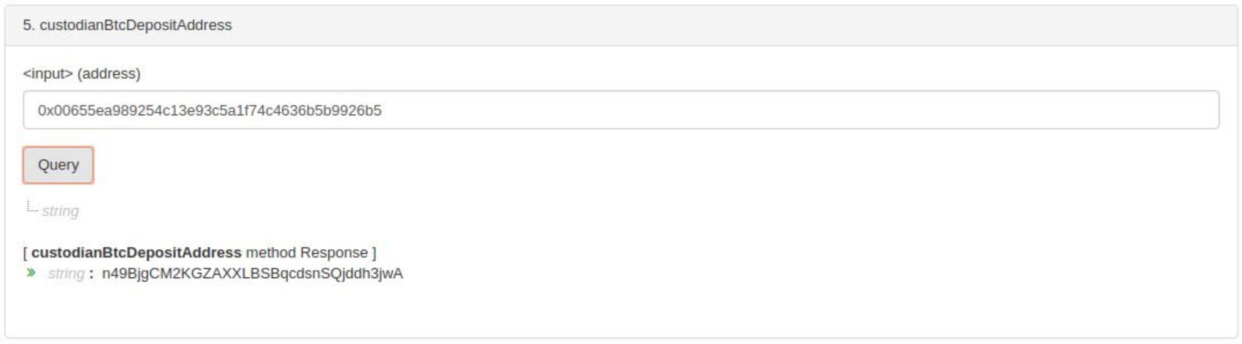
btcDepositAdress (string)

Function: **setCustodianBtcDepositAddress**

Minting

Republic want to mint 1 wbtc

1. It query bitgo deposit address. Function: **custodianBtcDepositAddress**



5. custodianBtcDepositAddress

<input> (address)

0x00655ea989254c13e93c5a1f74c4636b5b9926b5

Query

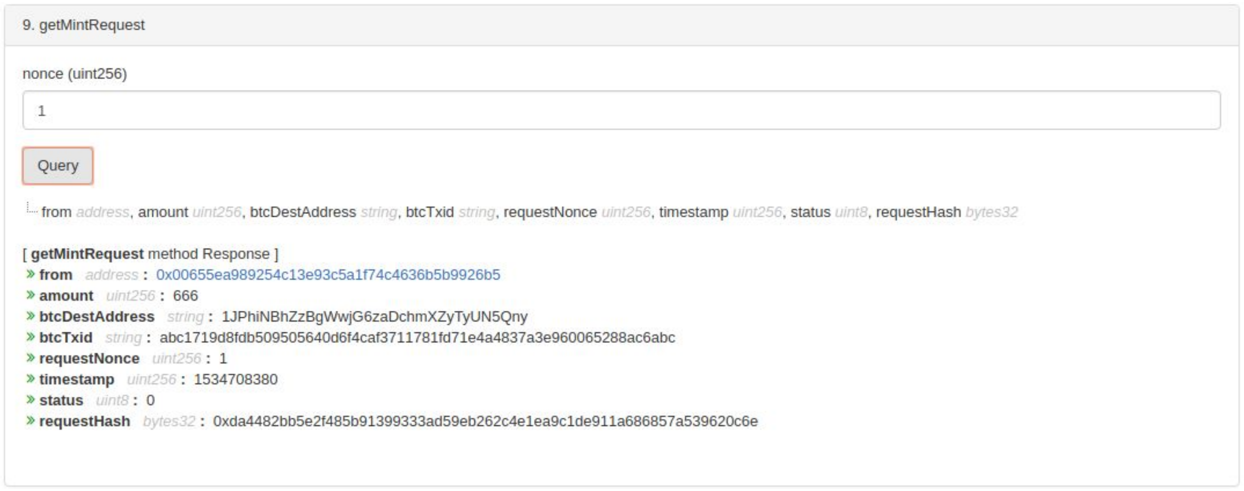
string

[**custodianBtcDepositAddress** method Response]
» string : n49BjgCM2KGZAXXLBSBqcdsnSQjddh3jwA

2. Republic generates a bitcoin tx to bitgo address for 1.005 BTC. But does not send it yet.
3. Republic calls **addMintRequest** with:
    1. amount = 1e8

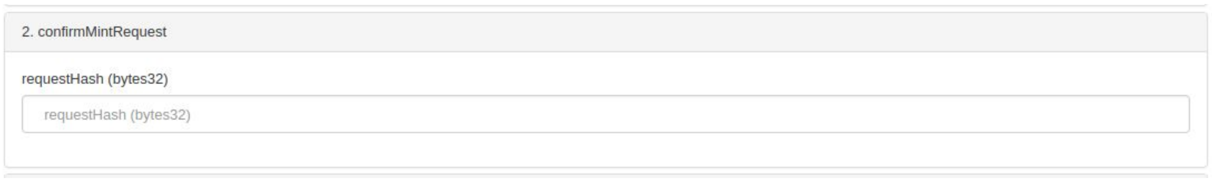
2. btcTxid = the tx id
3. btcDepositAdress = bitgo deposit address

4. Republic sends the btc tx. And give bitgo the minting request nonce.
5. Bitgo reads the nonce info **getMintRequest**



```
9. getMintRequest

nonce (uint256)
1

Query

from address, amount uint256, btcDestAddress string, btcTxid string, requestNonce uint256, timestamp uint256, status uint8, requestHash bytes32

[ getMintRequest method Response ]
» from  address : 0x00655ea989254c13e93c5a1f74c4636b5b9926b5
» amount  uint256 : 666
» btcDestAddress  string : 1JPhiNBhZzBgWwjG6zaDchmXZyTyUN5Qny
» btcTxid  string : abc1719d8fdb509505640d6f4caf3711781fd71e4a4837a3e960065288ac6abc
» requestNonce  uint256 : 1
» timestamp  uint256 : 1534708380
» status  uint8 : 0
» requestHash  bytes32 : 0xda4482bb5e2f485b91399333ad59eb262c4e1ea9c1de911a686857a539620c6e
```

6. Bitgo confirms the minting request with **confirmMintRequest**

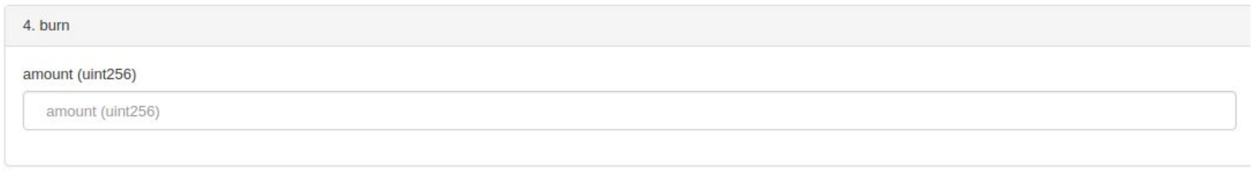


```
2. confirmMintRequest

requestHash (bytes32)
requestHash (bytes32)
```

Burning

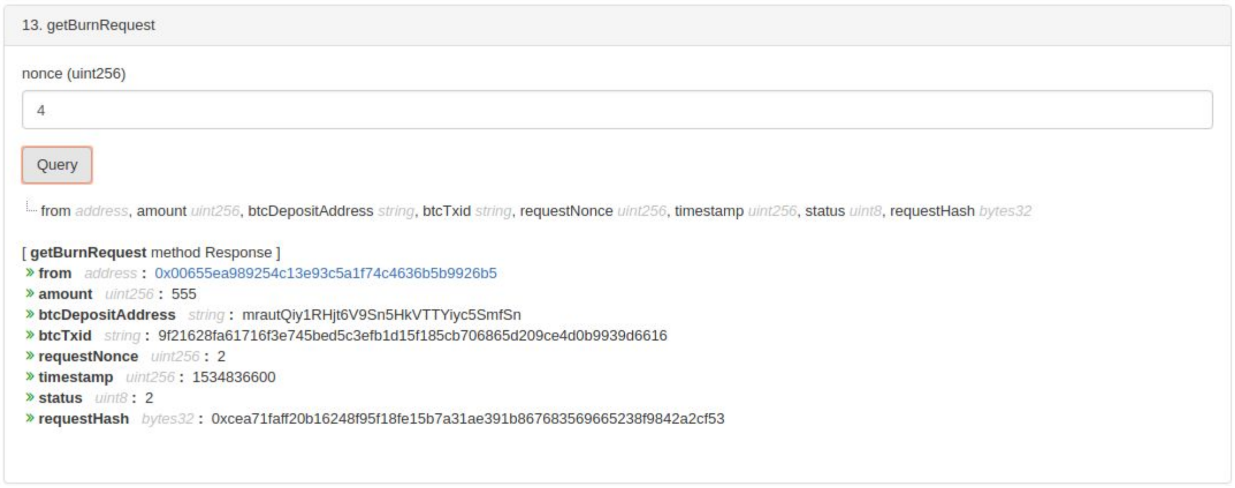
Republic wants to burn 1 wbtc. We assume it already approved allowance to the factory contract.

1. Republic calls **burn** with amount 1e8. And let bitgo know the nonce of the request.



```
4. burn

amount (uint256)
amount (uint256)
```

2. Bitgo reads the burn request **getBurnRequest**


13. getBurnRequest

nonce (uint256)

4

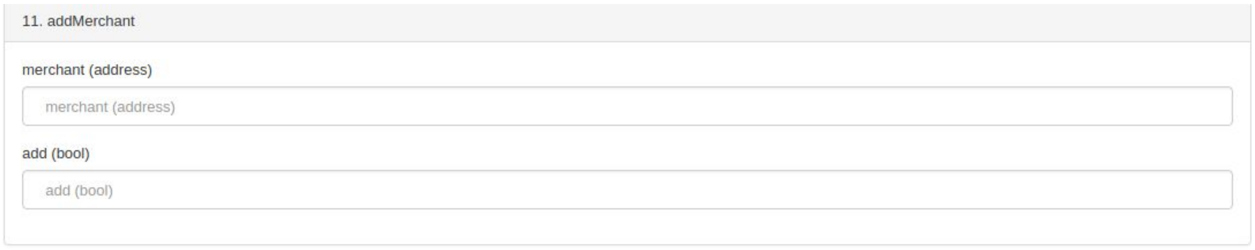
```
from address, amount uint256, btcDepositAddress string, btcTxid string, requestNonce uint256, timestamp uint256, status uint8, requestHash bytes32

[ getBurnRequest method Response ]
» from   address : 0x00655ea989254c13e93c5a1f74c4636b5b9926b5
» amount   uint256 : 555
» btcDepositAddress   string : mrautQiy1RHjt6V9Sn5HkVTTYiyc5SmfSn
» btcTxid   string : 9f21628fa61716f3e745bed5c3efb1d15f185cb706865d209ce4d0b9939d6616
» requestNonce   uint256 : 2
» timestamp   uint256 : 1534836600
» status   uint8 : 2
» requestHash   bytes32 : 0xcea71faff20b16248f95f18fe15b7a31ae391b867683569665238f9842a2cf53
```

3. Bitgo sends 0.995 BTC to the deposit address of the request.
4. Bitgo calls **confirmBurnRequest** with the burn request hash and the BTC tx hash

Adding new Merchant

Republic suggest to add new merchant, namely, kyber in address 0x6bC567F37C2e5A6d0B89906Fe5BefC199a57C5A2

This is done by calling the **addMerchant** function in **members.sol** contract (0x6A778ba60A98774D9CB370b8609e3442F6c1f514).


11. addMerchant

merchant (address)

add (bool)

However, this function is callable only by the contract owner, which is a multisig. (0xFFc4E87e971bb9000F4F042ED4A9A0E349415D98).

1. Republic init the request with a call to **submitTransaction**
2. Bitgo and republic approve the request by calling **confirmTransaction** (note that last caller gas limit should be higher than what MEW recommends)