
NewCryptoBlock
BLOCKCHAIN INNOVATION
ERC 20
Primer
GUIDE TO BUILD YOUR
SMART CONTRACT TOKEN
Dejan Jovanovic

Blank Page

CONTENTS

INTRODUCTION 4

SO HOW IT WORKS 14

TOKEN TRANSFER 19

Introduction

So what is ERC20 token and why do we need this. ERC stands for "Ethereum Request for Comment". This is Ethereum version of Request for Comments (RFC), concept created by Internet Engineering Task Force. It is used for capturing technical and organizational notes. In the case of ERC, this includes technical guidelines for buildout of the Ethereum network.

So how it works? Developer submits an Ethereum Improvement Proposal (EIP), this includes protocol specification and contract standards. Once EIP is approved by the Ethereum committee and finalized, it becomes and ERC.

Released and finalized EIPs are providing the Ethereum developers with a set of implementable standards. ERC-20 is one of the most well-known of all standards and most tokens issued on top of the Ethereum platform complies to it.

What is the ERC-20 Standard?

Originaly ERC-20 was proposed back in 2015 and officially formalized in September 2017. It is a great starting point in token standardization. The ERC-20 token standard includes the following functions:

totalSupply() : returns total token supply

balanceOf(address _owner) : returns account balance of _owner's account

transfer(address _to, uint256 _value) : takes in number _value and transfers that amount of tokens to provided address _toand triggers transfer event

transferFrom(address _from, address _to, unit256 _value) : transfer _value amount of tokens from the address _from to the address _to, and triggers the transfer event

approve(address _spender, uint256 _value) : allows _spender to any number of tokens up to _value amount

allowance(address _owner, address _spender) : returns amount which the _spender is allowed to withdraw from the _owner

The following events are triggered based on the listed ERC-20 functions:

transfer(address indexed _from, address indexed _to, uint256 _value) : this is triggered whenever tokens are transferred

approval(address indexed _owner, address indexed _spender, uint256 _value) : this is triggeredon any call to approve() function

There are some cases when additional functionality or improvements are needed, so additionaly to ERC-20 the following ERC standards are proposed:

ERC-223

ERC-233 proposal addresses the case when tokens are transferred to a smart contract. ERC-223 is backwords compatible with ERC-20 token. Status of this ERC is open and proposed on March 5th, 2017.

There are two different ways to transfer ERC-20 tokens, depending on whethere you intend to send the tokens directly or delegate the transfer to another smarth contract:

1. Invoking *transfer()* function
2. Or by invoking *approve()* function and then calling *transferFrom()* function

So what happens when you transfer your tokens, by mistake to a contract address that is unaware or doesn't expect these tokens? Problem that we have with first approach is that smart contract will not notify recipient of the transfer of the transaction, therefore he/she will not be able to recognize the incoming transaction. First example is that if you send for example 100 ERC20 tokens to a contract that is not intended to work with ERC20 tokens, the smart contract will not reject a transaction because it can't recognize an incoming transaction request. As a result of that your tokens will get stuck at the contracts balance.

Second example is in the case that the recipient is a smart contract, users must transfer their tokens using the second option (*approve()* + *transferFrom()*). In the case that recipient is a externaly owned account address, user must transfer their tokens using *transfer()* function. If user makes mistake and choose the wrong function the token will be lost inside the smart contract.

There are already number of tokens held by token contracts that did not expect any token transfer. These tokens are not accessable as there is no withdraw fynction available. Here are some examples:

310.067	GNT	Stuck in Golem contract
242	REP	Stuck in Augur conttract
814	DGD	Stuck in Digix DAO contract
14.506	1ST	Stuck in FirstBlood contract
30	MLN	Stuck in Melonport contract

The proposed solution, covered under ERC-223 is to use following for transfers of tokens to smart contract:

```
transfer(address _to, uint _value, bytes _data) returns (bool success)
```

This method is responsible for transferring tokens and invoking method:

```
tokenFallback(address _from, uint256 _value, bytes data)
```

This function will allow receiving contract to decline the token or trigger further actions. This can be used in place of the *approve()* function. Please see example:

```
contract ERC223ReceivingContract {
/**
 * @dev Standard ERC223 function that will handle incoming token transfers.
 *
 * @param _from  Token sender address.
 * @param _value Amount of tokens.
 * @param _data  Transaction metadata.
 */
    function tokenFallback(address _from, uint _value, bytes _data);
}

contract ERC223Token is ERC223Interface {
    using SafeMath for uint;

    mapping(address => uint) balances; // List of user balances.

    /**
     * @dev Transfer the specified amount of tokens to the specified address.
     *      Invokes the `tokenFallback` function if the recipient is a contract.
     *      The token transfer fails if the recipient is a contract
     *      but does not implement the `tokenFallback` function
     *      or the fallback function to receive funds.
     *
```

```
    * @param _to    Receiver address.
    * @param _value Amount of tokens that will be transferred.
    * @param _data  Transaction metadata.
    */
   function transfer(address _to, uint _value, bytes _data) {
      // Standard function transfer similar to ERC20 transfer with no _data .
      // Added due to backwards compatibility reasons .
      uint codeLength;

      assembly {
         // Retrieve the size of the code on target address, this needs assembly .
         codeLength := extcodesize(_to)
      }

      balances[msg.sender] = balances[msg.sender].sub(_value);
      balances[_to] = balances[_to].add(_value);
      if(codeLength>0) {
         ERC223ReceivingContract receiver = ERC223ReceivingContract(_to);
         receiver.tokenFallback(msg.sender, _value, _data);
      }
      Transfer(msg.sender, _to, _value, _data);
   }
   ...
}
```

Reference implementation available at: *https://github.com/Dexaran/ERC223-token-standard/tree/master/token/ERC223*

ERC-621

ERC-621 proposal addresses the case when tokerns supply needs to be changed.

Status of this ERC is open and proposed on May 1th, 2017.

In order to address this need proposition is to add following functions to ERC-20 token:

increaseSupply(uint256 _value, address _to) : allows at any time to inscrease existing supply for specified address

decreaseSupply(uint256 _value, address _from) : supply can be decrised at any time by specified _value for the owner _from

This proposal is mostly relevant to digitized assets managemend.

ERC-721

ERC-721 addresses totally different need then what ERC-20 and ERC-223 are addressing. It describes nonfungible token. Fungible is being something (such as monew or commodity) of such nature that one part or quantity in paying a debt or settling an account, per Merriam Webster dictionary. So fungability is characteristic of an asset or token. But if you are tracking collectables such as Pokiman cards where each card has it own chanracteristics (attributes) and potentially are different from each other from any perspective including price then we are looking at nonfungible token.

This means that eact token is totally different and each one can have totally different values to different users. Each one is its own separate commodity whose values are based on its own rarity and desirability by users.

The standard definition of functions is:

name() constant returns (string name) : this function returns the name. Interfaces and other smart contracts must not depend on existence of this function.

symbol() constant returns (string symbol) : this method returns the symbol referencing the entire collection of non fungable tokens

totalSupply() constant return (uint256 totalSupply) : returns the number of non fungable tokens currently tracked by the contract

balanceOf(address _owner) constant returns (uint256 balance) : returns the number of non fungable tokens assigned to address _owner

ownerOf(uint256 _tokenId) constant returns (address owner) : this method returns the owner of _tokenId

approve(address _to, uint256 _tokenId) : this method grants the approval of the address _to to take possession of the non fungable token _tokenId. Only one address can have approvalat any given time, calling method *approveTransfer* with a new address revokes approval for the previous address. Successful completion of this method creates the *Approve* event.

takeOwnership(uint256 _tokenId) : this function is responsible for assigning the ownership of the

non fungible token to the message sender or it throws exception if message sender does not have approval for _tokenId, or _tokenId does not represent non fungable token currently tracked by the smart contract, or message sender already has ownership of the _tokenId.

transfer(address _to, uint256 _tokenId) : this method assignes the ownership of the non fungable token with id that is _tokenId to _to if and only as message sender is also a owner of the token (msg.sender == ownerOf(tokenid)). A successful transfer must fire the Transfer event. In the case trat transfer is not successfult it will throw the exception. Exception is thrown in following cases: message sender is not a owner of _tokenId, or _tokenId does not represent currently tracked non fungable token, or _to is 0.

tokenOfOwnerByIndex(address _owner, uint256 _index) constant return (uint tokenId) : this method is optional and it is recommended to be implemented in order to enhance usability with wallets and exchnages. It returns non fungable token that is assigned to the address of the _owner, with value of the _index argument. This method throws exception in the case that _index >= balanceOf(owner).

tokenMetadata(uint256 _tokenId) constant returns (string infoUrl) : this method is optional but it is recommended to be implemented for enhanced usability with wallets and exchanges. This method returns multiaddress stringreferencing an external resource bundle that contains metadata about non fungable token associated with _tokenId. The string must be an IPFS of HTTP/HTTPS base path.

Following events are supported:
Transfer(address indexed _from, address indexed _to, uint256 _tokenId) – this event must be triggered when non fungable tokens ownership is transferred. Additionaly in the case of creation of new non fungable token this event is triggered with _from address of 0 and a _to address matching the owner of new non gungable token, and also in the case of deletion enet is trigerred with _to address of 0 and a _from address of yhe owner.

Approval(address indexed _owner, address indexed _approved, uint256 _tokenId) – this event is trigerred on any successful call to approve(address _spender, uint256 _value)

Following table presents the cases when Approve event is triggered:

Action	Prior State	_to address	New State	Event
Clear unset approval	Clear	0	Clear	None
Set new approval	Clear	X	Set to X	Approval(owner, X, tokenId)
Change approval	Set to X	Y	Set to Y	Approval(owner, Y, tokenId)
Reaffirm approval	Set to X	X	Set to X	Approval(owner, X, tokenId)
Clear approval	Set to X	0	Clear	Approval(owner, 0, tokenId)

ERC-827

ERC-827 presents another extension to ERC-20, and it allows for the transfer of tokens and also allows tokens to be approved by by the holder in order to be spent by third party. It can be used to sole the same problem as ERC-223 is resolving, preventing ERC-20 tokens to be stuck, moreover it offers flexibility to transfer data along with tokens to smart contract and execute them. ERC-827 has been included in Zeppelin open source contract. On Ethereum blockchain tokens can be reused by other applications from wallets to decentralized exchanges.

To understand ERC-827 token lets go through shoping cart example.

Let say that shopping cart has items waiting to be bought. In order to complete the transation user has to transfer tokens to the shopping cart with some data passed with the cart which will call another smart contract that is responsible for checking whether the cart owner has enough tokens to cover the balance and proceed to checkout for the items saved in the shopping cart.

Here is OpenZeppelin implementation

```
contract ERC827 is ERC20 {
   function approve( address _spender, uint256 _value, bytes _data ) public returns (bool);
   function transfer( address _to, uint256 _value, bytes _data ) public returns (bool);
   function transferFrom( address _from, address _to, uint256 _value, bytes _data )
        public returns (bool);

}
```

The standard definition of specified ERC-827 methods is:

approve(address _from, uint256 _value, byte _data) public returnes (bool) : this method allows to approve the transfer of values and execute a call with the data sent during the approve function call. Potential risk is that allowance can be changed within this method and and the old one. One possible solution to mitigate the risk is to first reduce the spenders allowance to 0 and set the desire value afterwards. Here is example implementation:

```
/**
@dev Addition to ERC20 token methods. It allows to
approve the transfer of value and execute a call with the sent data.

Beware that changing an allowance with this method brings the risk that
someone may use both the old and the new allowance by unfortunate
transaction ordering. One possible solution to mitigate this race condition
is to first reduce the spender's allowance to 0 and set the desired value
afterwards:
https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

@param _spender The address that will spend the funds.
@param _value The amount of tokens to be spent.
@param _data ABI-encoded contract call to call `_to` address.

@return true if the call function was executed successfully
*/
function approve(address _spender, uint256 _value, bytes _data) public returns (bool) {
        require(_spender != address(this));

        require(super.approve(_spender, _value));

        require(_spender.call(_data));

        return true;
}
```

transfer(address _to, uint256 _value, bytes _data) public returns (bool) : this method transfers tokens from specified address and executes a call with the sent data on the same transaction. Method will return true if the called function is executed successfully. Example implementation:

```
/**
@dev Addition to ERC20 token methods. Transfer tokens to a specified address and execute a
call with the sent data on the same transaction
```

```
@param _to address The address which you want to transfer to
@param _value uint256 the amout of tokens to be transfered
@param _data ABI-encoded contract call to call `_to` address.

@return true if the call function was executed successfully
*/
function transfer(address _to, uint256 _value, bytes _data) public returns (bool) {
        require(_to != address(this));

        require(super.transfer(_to, _value));

        require(_to.call(_data));
        return true;
 }
```

transferFrom(address _from, address _to, uint256 _value, bytes _data) public returns (bool) : this method like *transfer()* it also transfers tokens from one address to another and makes a contract call on the same transaction. The only difference is that this method can be used in coordination with *approve()* method and execute different logic while approving and transferring tokens. Example implementation:

```
 /**
@dev Addition to ERC20 token methods. Transfer tokens from one address to another and
make a contract call on the same transaction

@param _from The address which you want to send tokens from
@param _to The address which you want to transfer to
@param _value The amout of tokens to be transferred
@param _data ABI-encoded contract call to call `_to` address.

@return true if the call function was executed successfully
*/
function transferFrom(address _from, address _to, uint256 _value, bytes _data)
        public returns (bool) {
        require(_to != address(this));

                super.transferFrom(_from, _to, _value);

        require(_to.call(_data));
        return true;
}
```

So How It Works

ERC-20 pprovides a formalized specification of the Token Smart Contract. Token Smart Contract is a smart contract that contains the map of account addresses and related balances. The balance represents a value that is defined by the contract creator. Here is the example of the smart contract balance:

HOLDER ADDRESS	BALANCE
0X000.....000000	0
0X3H9.....87223A	100
0X33B.....11C238	100
0X1F8.....02AA3H	100
0XD4F.....0BFC33	100
0X2FF.....0004D21	100
0X00D.....13332A	100
0XDFF.....AAF321	100
0X2BB.....9D8F11	100
0X1FC.....000032	100
0XFFC.....000FC2	100
TOTAL SUPPLY	1000

When tokens are transferred from one account to another the token contract updatres the balance of two accounts. For example, if from address 0x1F8…….02AA3h we transfer 10 tokens to address 0x2bb….9d8f11 it will result in balance being updated to following:

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	100
0X33B……11C238	100
0X1F8……02AA3H	90
0XD4F……0BFC33	100
0X2FF……0004D21	100
0X00D……13332A	100
0XDFF……AAF321	100
0X2BB……9D8F11	110
0X1FC……000032	100
0XFFC……000FC2	100
TOTAL SUPPLY	1000

If token contract allows there are two ways how to change the total supply of tokens. As we have mentioned ERC-621 provides the specification for two methods: increase supplies (minting) and decries supplies (burning). If token contract has implemented *increaseSupply(uint256 _value, address _to)* method then following calling *increaseSupply(100,* 0x2ff….0004d21*)* will result in following balance update:

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	100
0X33B……11C238	100
0X1F8……02AA3H	90
0XD4F……0BFC33	100
0X2FF……0004D21	200
0X00D……13332A	100
0XDFF……AAF321	100
0X2BB……9D8F11	110
0X1FC……000032	100
0XFFC……000FC2	100
TOTAL SUPPLY	1100

By calling method *decreaseSupply(uint256 _value, address _to)* total supply can be decreased. This is also called burning. So for eample if we call *decreaseSupply(70,* 0xdff……aaf321*)* this will result in the following balance:

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	100
0X33B……11C238	100
0X1F8……02AA3H	90
0XD4F……0BFC33	100
0X2FF……0004D21	200
0X00D……13332A	100
0XDFF……AAF321	30
0X2BB……9D8F11	110
0X1FC……000032	100
0XFFC……000FC2	100
TOTAL SUPPLY	1030

So, as we understand how balace of tokens works let explore how is ERC-20 implementation work. There are few items that we need to define before we start with implementation.
There are few attributes that needs to be definied:

```
string public constant name
string public constant symbol
uint8 public constant decimals
string public version
```

First, we need to select token name. We need to understand that there is no central repository that will guaranty your token name uniqness. There is no restriction of how long the name can be but still keep it short as some wallets will truncate it anyways. For example, it can be “Hive Project Token”.

Token symbol is the symbol by which token contract will be known, for example “HVN”. It should be no longer then 3 to 4 characters.

Attribute decimals are common source of confusion. Depends on what is contract for value can varie. Let's go throught the cases:

1. If the smart contract is tracking number of software licenses that customers have then the decimal value is going to be 0 as the values that accounts have a re going to be round numbers. See the below example:

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	75
0X33B……11C238	10
0X1F8……02AA3H	5
0XD4F……0BFC33	1
0X2FF……0004D21	1
0X00D……13332A	1
0XDFF……AAF321	2
0X2BB……9D8F11	1
0X1FC……000032	1
0XFFC……000FC2	3
TOTAL SUPPLY	100

   In this example total nimber of available licenses is 100. Owner of the contract holds the majority of licenses. As users purchases a license a single token will be transferred from the holding account (in our case 0x3h9……87223A) to the purchaser.

2. Second example uses GoldToken, the token contract that represents ownership of physical gold. The token creator wants unit of to represent 1 kg of gold, but in the same time wants to allow users to hold and trade amounts of gold down to the gram level. In this case value of decimals is set to 3. Do not forget Ethereum doesn't support decimal numbers so we deal with this on the code level.

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	15246
0X33B……11C238	741
0X1F8……02AA3H	16502
0XD4F……0BFC33	15511
0X2FF……0004D21	2000
TOTAL SUPPLY	50000

So for total supply of 50 kg of gold represented (1 g per token * 50000 tokens). However, as decimals attribute is set to 3 the view to the users will be as follows:

HOLDER ADDRESS	BALANCE
0X000……000000	0
0X3H9……87223A	15,246
0X33B……11C238	0.741
0X1F8……02AA3H	16,502
0XD4F……0BFC33	15,511
0X2FF……0004D21	2,000
TOTAL SUPPLY	50,000

3. And at last if you are creating utility token it is recommended to use 18 decimals. In some cases contract owners are using 8 decimals, but I will recommend to stay away from it. The idea of divisibility is to allow for token contract to represent fine grained decimal value.

In summary when selecting a sutable value for decimals following rules should be followed:

- Does the token contract represent an invisible entity, then set deimals to 0
- Does the token contract represent an item with a fixed number of decimals, then set decimals to that number
- If neither of the above apply set decimals to 18

It is important to understand impact of decimal point on the token creation. Following equatia shows the calculation of total supply of tokens that token contract is going to have:

$$totalSupply = requiredNumberOfTokens * 10^{decimals}$$

And finaly string public version attribute is there to track the version of the contract and to ensure that test results provided with a contract are the one that are correspondent to the version of the token contract.

Token Transfer

There are two mechanisms for transferring tokens from one account to another. First case is simply using *transfer()* method. The *transfer()* method transfer the number of tokens from directly from message sender to specified another account. This method doesn't make any checks on recipient address so all responsibility layes on sender to make sure that provided information is correct.

Lets now emplore the cases when interaction is between two smart contracts. Eample will be based on sevice smart contract ServiceContract that is responsible for sellin services to consumers. SeviceContract has method called provideService() that requires 10 SCT (ServiceContract Tokens) in order to operate.

The consumer has in his account 100 SCT. So, her is the two step process that needs to be executed. First step is that consumer provides approval for ServiceContract address to transfer

required number of SCT tokens. The consume, in this case also a token holder call the method approve() to provide this information.

HOLDER ADDRESS	ALLOWANCE ADDRESS	ALLOWANCE
0X3H9……87223A	**0x33B……1AC230**	50
0X33B……11C238	**0x321……B1C22B**	25
0X1F8……02AA3H	**0x211……11C233**	20
0XD4F……0BFC33	**0x343……1FF238**	250
0X2FF……0004D21	**0x98A……98BBF8**	10

In our example consumers address is 0x2ff…….0004d21 and ServiceContract address is 0x98A…….98BBF8. From the allowance table abowe we can see that consumer has already sett allowance of 10 SCT for ServiceContract. Once this is done ServiceContract can transfer allocated tokens for the operation. So when consumer now calls method provideService(), provideService() will call transferFrom() method in order to transfer 10 SCT for the service. In the case that consumer do not have sufficient founds provideService() method will fail.

Important Notes: it is important to notice that allowances are preliminary one, "soft", as both individual and cumulative allowances can exceed and address balance. Any contract using allowance() must consider the balance of the consumer, holder when calculating the number of tokens available to it.

Smart Contract service payments with tokens

In this section we are going to go over case when tokens are used to pay for a secice provided by another smart contract.

For paying for service function in Ether code looks like this:

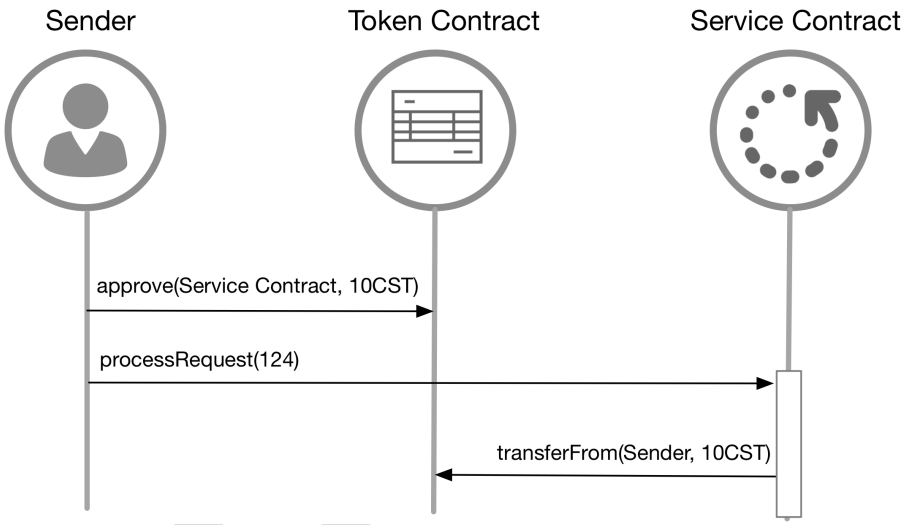
```
function processRequest(unit256 payload) public payable {
        require(msg.value == 2 ether);
        process(msg.sender, payload);
}
```

Unfortunately, in the case that we have our own token, this is not possible as payable is always in Eth and can not be in amount of tokens. Instead consumer and service provide must communicate with ServiceContract in order to move tokens from consumer to service provider.

There are three different ways how to achive this transfer:

1. approve() and transferFrom()

   Bothe functions are part of the ERC-20 standard and should be part of any compliant token implementation. The role of the approve() method is to authorize a third party to transfer tokens from the sender's account. Fod detail sequence diagram please see below diagram:

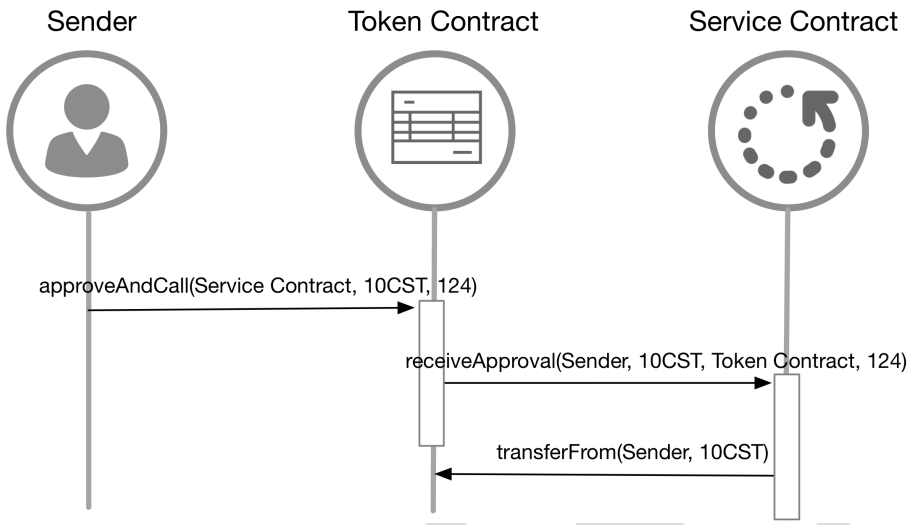


   transferFrom() is called as a part of processRequest() implementation. Example implementation of the processRequest() method is:

```
function processRequest(unit256 payload) public payable {
        require(tokenContract.transferFrom(msg.sender, address(this), 10));
        process(msg.sender, payload);
}
```

2. approveAndCall()

Previos case requires two separate requests from the sendrr. In the case of approveAndCall() implementation, it allowes sender to carry out the same work with a single transaction. Detail sequence diagram is shown below:



receiveApproval() method is called inside the implamentation of approveAndCall() method implementation. This requires approveAndCall() methid to I be implemented as following:

```
function approveAndCall(address _recipient, uint256 _value, bytes _extraData){
        approve(_recipient, _value);
        TokenRecipient(_recipient).receiveApproval(msg.sender, _value,
                                                   address(this), _extraData);
}
```

The receiveApproval() method in the ServiceContract is as follows:

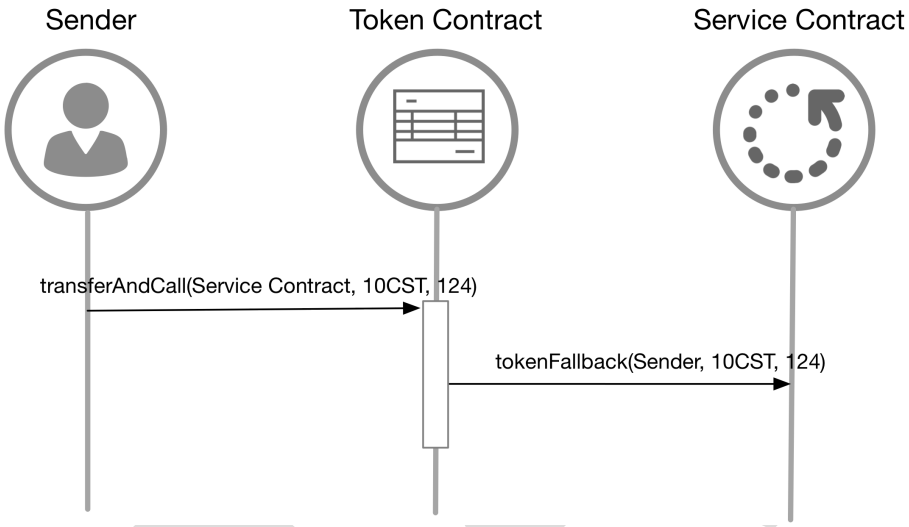
```
function receiveApproval(address _sender, uint256 _value,
                                TokenContract _tokenContract, bytes _extraData){
        require(_tokenContract == tokenContract);
        require(tokenContract.transferFrom(_sender, address(this), 10));
        unit256 payloadSize;
        uint256 payload;

        assembly{
                payloadSize := mload(_extraData)
                payload := mload(add(_extraData, 0x20))
        }
       payload = payload >> 8*(32 - payloadSize);
       process(msg.sender, payload);
}
```

First line confirms that it is token conytract that is calling this method. Second line of code obtains the required tokens. The rest of the code obtains the payload and calls the processing methid.

There are few points that needs to be notest with definition of approveAndCall() as it is not part of ERC-20 standard:

1. some token contracts do not support approveAndCall()
2. some token contracts only support a single hardcoded receiveApproval() method as the target of approveAndCall()
3. some token contracts support calling arbitrary service contract methods by adding the relevant methid signature to the approveAndCall() transmitted by the sender

3. transferAndCall()
   ERC-677 standard provides the alternative to approveAndCall() method called transferAndCall() method. transferAndCall() method carries out transfer() and then calls a function on the service contract. Following sequence diagram shows the detail of the interaction:



The transferAndCall() method implementation should look following:

```
function transferAndCall(address _recipient, uint256 _value, bytes _extraData){
        trasfer(_recipient, _value);
        require(TokenRecipient(_recipient).tokenFallback(msg.sender, _value,
                                                         _extraData));
}
```

The tokenFallback() method implementation should look like this:

```
function tokenFallback(address _sender, uint256 _value, bytes _extraData)
                                return (bool){
        require(msg.sender == tokenContract);
        require(_value == 10);
        .......
        // implementation of the code what needs to be done
       return true;
}
```

transferAndCall() method is a upgrade solution from approveAndCall() method. It provides similar functionality for less gas cost and lower complexity.

There are cases where this solution is not so good, for example when we have variable price of the transactions. In this case sender doesn't know the cost of the price in advance as price can for example depend of the service demand.

Second case where this implementation presents the limitation is in the case of repeated transactions. An example of this as a gambling contract where multiple bets can be placed across different games. So instead of calling approve() method every time, sender could call it just once and save on gas.

Conclusion: each method presented here has benefits and drawbacks. When building a service contract that accepts tokens as payment it is important to clearly define requirements as this will allow proper selection of the method that best fits the contracts's needs.