CRYPTO4ALL

TABLE OF CONTENTS

Audit context

Ethereum Blockchain provides a decentralized, trustless, transparent, traceable and secure execution of smart contract. All those characteristics are respected only if the implementation is done in the state of the art.

In this way, smart contracts are the implementation of the logic/trust between the parties.

We analyzed in this report the effectiveness and safety of the code in order to bring you our technical recommendations and implementations suggests.

Besides, we have not received any functional and technical documentations which describe the architecture implementation about your system. We made our possible to figure out through the code the behavior described within the Trecento Whitepaper provided on your Gitlab repository.

In consideration, we cannot be liable for any security flaws related to interoperability between the Front-end and Back-end system.

Moreover, we notice that the ICO will be launched without automatizing the investment process for cryptocurrencies described on the whitepaper (p35) by the use of a token sale smart contract. Smart contract auditing cannot unveil all existing flaws and/or vulnerabilities in the same way as an audit in which no flaws and/or vulnerabilities are found is not a guarantee for a secure smart contract. The objective of the audit is to find out flaws and/or vulnerabilities that were unobserved during development stage. Various types of issues can be detected: some may affect the smart contract application while others might induce lack protection in certain areas. In this perspective, we carry out a source code analysis to identify code that need to be patched. We have performed widespread auditing in order to discern as many flaws and/or vulnerabilities as possible.

The smart contract audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model or any other statements.

High security flaws

NONE

Medium vulnerabilities

We have detected mathematical operations within the TOTTokens contract without the use of SafeMath library operations involving a risk of overflow issues. As recommended below you should update the code to prevent it by using mul and div operations.

Besides we suggest you to use the latest stable version of solidity 0.4.24 to enforce the compiler and prevent from any previous bugs/flaws detected on previous versions.

Low vulnerabilities

NONE

Smart Contracts

1. TOTToken.**SOL**

```
pragma solidity ^0.4.23;
```

```
2.
3.
4.  /**
5.   * @title SafeMath
6.   * @dev Math operations (only add and sub here) with safety checks that throw on error
7.   */
8.  library SafeMath {
9.
10.  /**
11.  * @dev Multiplies two numbers, throws on overflow.
12.  */
13.  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
14.    if (a == 0) {
15.      return 0;
16.    }
17.    uint256 c = a * b;
18.    assert(c / a == b);
19.    return c;
20.  }
21.
22.  /**
23.  * @dev Integer division of two numbers, truncating the quotient.
24.  */
25.  function div(uint256 a, uint256 b) internal pure returns (uint256) {
26.    // assert(b > 0); // Solidity automatically throws when dividing by 0
27.    uint256 c = a / b;
28.    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
29.    return c;
30.  }
31.
32.  /**
33.  * @dev Substracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).
34.  */
35.  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
36.    assert(b <= a);
37.    return a - b;
38.  }
39.
40.  /**
41.  * @dev Adds two numbers, throws on overflow.
```

```
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;


  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);


  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the sender
   * account.
   */
  constructor() public {
    owner = msg.sender;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    require(newOwner != address(0));
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }

}


/**
 * @title Pausable
 * @dev Base contract which allows children to implement an emergency stop mechanism.
 */
contract Pausable is Ownable {
  event Pause();
  event Unpause();

  bool public paused = false;

```

```
101.
102.            /**
103.             * @dev Modifier to make a function callable only when the contract is not paused.
104.             */
105.            modifier whenNotPaused() {
106.              require(!paused);
107.              _;
108.            }
109.
110.            /**
111.             * @dev Modifier to make a function callable only when the contract is paused.
112.             */
113.            modifier whenPaused() {
114.              require(paused);
115.              _;
116.            }
117.
118.            /**
119.             * @dev called by the owner to pause, triggers stopped state
120.             */
121.            function pause() onlyOwner whenNotPaused public {
122.              paused = true;
123.              emit Pause();
124.            }
125.
126.            /**
127.             * @dev called by the owner to unpause, returns to normal state
128.             */
129.            function unpause() onlyOwner whenPaused public {
130.              paused = false;
131.              emit Unpause();
132.            }
133.          }
134.
135.
136.          /**
137.           * @title ERC20 interface
138.           * @dev see https://github.com/ethereum/EIPs/issues/20
139.           */
140.          contract ERC20 {
141.            function totalSupply() public view returns (uint256);
142.            function balanceOf(address who) public view returns (uint256);
143.            function transfer(address to, uint256 value) public returns (bool);
144.            function allowance(address owner, address spender) public view returns (uint256);
145.            function transferFrom(address from, address to, uint256 value) public returns (bool);
146.            function approve(address spender, uint256 value) public returns (bool);
147.
148.            event Transfer(address indexed from, address indexed to, uint256 value);
149.            event Approval(address indexed owner, address indexed spender, uint256 value);
150.          }
151.
152.
153.          contract TOTToken is ERC20, Pausable {
154.
155.            using SafeMath for uint256;
156.
157.            string public name = "Trecento";      //  token name
158.            string public symbol = "TOT";         //  token symbol
159.            uint256 public decimals = 8;          //  token digit
```

```
160.
161.
162.            // Token distribution, must sumup to 1000
163.            uint256 public constant SHARE_PURCHASERS = 750;
164.            uint256 public constant SHARE_FOUNDATION = 50;
165.            uint256 public constant SHARE_TEAM = 150;
166.            uint256 public constant SHARE_BOUNTY = 50;
167.
168.            // Wallets addresses
169.            address public foundationAddress = 0x0;
170.            address public teamAddress = 0x0;
171.            address public bountyAddress = 0x0;
172.
173.            uint256 totalSupply_ = 0;
174.            uint256 public cap = 20000000 * 10 ** decimals; // Max cap 20.000.000 token
175.
176.            mapping(address => uint256) balances;
177.
178.            mapping (address => mapping (address => uint256)) internal allowed;
179.
180.            bool public mintingFinished = false;
181.
182.            event Burn(address indexed burner, uint256 value);
183.            event Mint(address indexed to, uint256 amount);
184.            event MintFinished();
185.
186.            modifier canMint() {
187.              require(!mintingFinished);
188.              _;
189.            }
190.
191.            /**
192.             * @dev Change token name, Owner only.
193.             * @param _name The name of the token.
194.            */
195.            function setName(string _name) onlyOwner public {
196.              name = _name;
197.            }
198.
199.            function setWallets(address _foundation, address _team, address _bounty) public onlyOwner
    canMint {
200.              require(_foundation != address(0) && _team != address(0) && _bounty != address(0));
201.              foundationAddress = _foundation;
202.              teamAddress = _team;
203.              bountyAddress = _bounty;
204.            }
205.
206.            /**
207.             * @dev total number of tokens in existence
208.            */
209.            function totalSupply() public view returns (uint256) {
210.              return totalSupply_;
211.            }
212.
213.            /**
214.             * @dev Gets the balance of the specified address.
215.             * @param _owner The address to query the the balance of.
216.             * @return An uint256 representing the amount owned by the passed address.
217.            */
```

```
218.              function balanceOf(address _owner) public view returns (uint256 balance) {
219.                return balances[_owner];
220.              }
221.
222.              /**
223.               * @dev transfer token for a specified address
224.               * @param _to The address to transfer to.
225.               * @param _value The amount to be transferred.
226.              */
227.              function transfer(address _to, uint256 _value) public whenNotPaused returns (bool success) {
228.                require(_to != address(0));
229.                require(_value <= balances[msg.sender]);
230.                // SafeMath.sub will throw if there is not enough balance.
231.                balances[msg.sender] = balances[msg.sender].sub(_value);
232.                balances[_to] = balances[_to].add(_value);
233.                emit Transfer(msg.sender, _to, _value);
234.                return true;
235.              }
236.
237.              /**
238.               * @dev Transfer tokens from one address to another
239.               * @param _from address The address which you want to send tokens from
240.               * @param _to address The address which you want to transfer to
241.               * @param _value uint256 the amount of tokens to be transferred
242.              */
243.              function transferFrom(address _from, address _to, uint256 _value) public whenNotPaused
      returns (bool) {
244.                require(_to != address(0));
245.                require(_value <= balances[_from]);
246.                require(_value <= allowed[_from][msg.sender]);
247.
248.                balances[_from] = balances[_from].sub(_value);
249.                balances[_to] = balances[_to].add(_value);
250.                allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
251.                emit Transfer(_from, _to, _value);
252.                return true;
253.              }
254.
255.              /**
256.               * @dev Approve the passed address to spend the specified amount of tokens on behalf of
      msg.sender.
257.               *
258.               * Beware that changing an allowance with this method brings the risk that someone may use
      both the old
259.               * and the new allowance by unfortunate transaction ordering. One possible solution to
      mitigate this
260.               * race condition is to first reduce the spender's allowance to 0 and set the desired value
      afterwards:
261.               * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
262.               * @param _spender The address which will spend the funds.
263.               * @param _value The amount of tokens to be spent.
264.              */
265.              function approve(address _spender, uint256 _value) public whenNotPaused returns (bool) {
266.                allowed[msg.sender][_spender] = _value;
267.                emit Approval(msg.sender, _spender, _value);
268.                return true;
269.              }
270.
271.              /**
```

```
272.              * @dev Function to check the amount of tokens that an owner allowed to a spender.
273.              * @param _owner address The address which owns the funds.
274.              * @param _spender address The address which will spend the funds.
275.              * @return A uint256 specifying the amount of tokens still available for the spender.
276.             */
277.             function allowance(address _owner, address _spender) public view returns (uint256) {
278.               return allowed[_owner][_spender];
279.             }
280.
281.             /**
282.              * @dev Increase the amount of tokens that an owner allowed to a spender.
283.              *
284.              * approve should be called when allowed[_spender] == 0. To increment
285.              * allowed value is better to use this function to avoid 2 calls (and wait until
286.              * the first transaction is mined)
287.              * From MonolithDAO Token.sol
288.              * @param _spender The address which will spend the funds.
289.              * @param _addedValue The amount of tokens to increase the allowance by.
290.             */
291.             function increaseApproval(address _spender, uint _addedValue) public whenNotPaused
    returns (bool) {
292.               allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add(_addedValue);
293.               emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
294.               return true;
295.             }
296.
297.             /**
298.              * @dev Decrease the amount of tokens that an owner allowed to a spender.
299.              *
300.              * approve should be called when allowed[_spender] == 0. To decrement
301.              * allowed value is better to use this function to avoid 2 calls (and wait until
302.              * the first transaction is mined)
303.              * From MonolithDAO Token.sol
304.              * @param _spender The address which will spend the funds.
305.              * @param _subtractedValue The amount of tokens to decrease the allowance by.
306.             */
307.             function decreaseApproval(address _spender, uint _subtractedValue) public whenNotPaused
    returns (bool) {
308.               uint oldValue = allowed[msg.sender][_spender];
309.               if (_subtractedValue > oldValue) {
310.                 allowed[msg.sender][_spender] = 0;
311.               } else {
312.                 allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
313.               }
314.               emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
315.               return true;
316.             }
317.
318.             /**
319.              * @dev Function to       mint tokens
320.              * @param _to The address that will receive the minted tokens.
321.              * @param _amount The amount of tokens to mint.
322.              * @return A boolean that indicates if the operation was successful.
323.              */
324.             function mint(address _to, uint256 _amount) onlyOwner canMint public returns (bool) {
325.               require(totalSupply_.add(_amount) <= cap);
326.               require(_to != address(0));
327.               totalSupply_ = totalSupply_.add(_amount);
328.               balances[_to] = balances[_to].add(_amount);
```

```
329.                emit Mint(_to, _amount);
330.                emit Transfer(address(0), _to, _amount);
331.                return true;
332.              }
333.
334.              /**
335.               * @dev Function to stop minting new tokens.
336.               * @return True if the operation was successful.
337.               */
338.              function finishMinting() onlyOwner canMint public returns (bool) {
339.
340.                require(foundationAddress != address(0) && teamAddress != address(0) && bountyAddress
      != address(0));
341.                require(SHARE_PURCHASERS + SHARE_FOUNDATION + SHARE_TEAM + SHARE_BOUNTY ==
      1000);
342.
343.                // before calling this method totalSupply includes only purchased tokens
344.                uint256 onePerThousand = totalSupply_ / SHARE_PURCHASERS; //ignore (totalSupply mod
      617) ~= 616e-8,
345.
346.                uint256 foundationTokens = onePerThousand * SHARE_FOUNDATION;
347.                uint256 teamTokens = onePerThousand * SHARE_TEAM;
348.                uint256 bountyTokens = onePerThousand * SHARE_BOUNTY;
349.
350.                mint(foundationAddress, foundationTokens);
351.                mint(teamAddress, teamTokens);
352.                mint(bountyAddress, bountyTokens);
353.
354.                mintingFinished = true;
355.                emit MintFinished();
356.                return true;
357.              }
358.
359.
360.              /**
361.                * @dev Burns a specific amount of tokens.
362.                * @param _value The amount of token to be burned.
363.              */
364.              function burn(uint256 _value) public whenNotPaused {
365.                require(_value <= balances[msg.sender]);
366.                // no need to require value <= totalSupply, since that would imply the
367.                // sender's balance is greater than the totalSupply, which *should* be an assertion failure
368.
369.                address burner = msg.sender;
370.                balances[burner] = balances[burner].sub(_value);
371.                totalSupply_ = totalSupply_.sub(_value);
372.                emit Burn(burner, _value);
373.                emit Transfer(burner, address(0), _value);
374.              }
375.
376.
377.              /**
378.                * @dev This is an especial owner-only function to make massive tokens minting.
379.                * @param _data is an array of addresses
380.                * @param _amount is an array of uint256
381.              */
382.              function batchMint(address[] _data,uint256[] _amount) public onlyOwner canMint {
383.                for (uint i = 0; i < _data.length; i++) {
384.                        mint(_data[i],_amount[i]);
```

```
385.              }
386.            }
387.
388.
389.          }
```

The global structuration of the code is composed with only one solidity file (.sol) on the folder provided to us. However, on the code we can see the use of imported standards smart contracts like “Safemath, Pausable, Ownable and ERC20”.

We recommend you to create a solidity file for each called smart contract to improve readability and manage easily future updates without impacting the entire behavior during the compilation.

```
/**
   * @dev Change token name, Owner only.
   * @param _name The name of the token.
 */
 function setName(string _name) onlyOwner public {
   name = _name;
 }
```

We have noticed that you use a specific function to modify the name of the ERC20 Token after the deployment. It will be also useful to update the symbol “TOT by creating a dedicated function because we don’t find any reasons to only update the name. In that case, we advise you to use this function with awareness even more when TOT tokens will be tradable to not impact token transfers for your users or partners like Exchanges.

```
// Wallets addresses
 address public foundationAddress = 0x0;
 address public teamAddress = 0x0;
 address public bountyAddress = 0x0;

 uint256 totalSupply_ = 0;
```

We noticed an initialization using “0x0” and we suggest you to initialize it with dedicated addresses for more transparency. There is no concrete impact to define it after the deployment because you have made control to prevent the distribution of tokens to “0x0” addresses. By initializing it during the deployment it will cost you less in GAS.

```
uint256 totalSupply_ = 0;

 mapping(address => uint256) balances;

 mapping (address => mapping (address => uint256)) internal allowed;

  function totalSupply() public view returns (uint256) {
   return totalSupply_;
 }

 function balanceOf(address _owner) public view returns (uint256 balance) {
   return balances[_owner];
```

```
}
function allowance(address _owner, address _spender) public view returns (uint256) {
  return allowed[_owner][_spender];
```

The attributes totalSupply_, balances and allowed should be “private” as recommended by the OpenZeppelin Standard. Indeed "private" means that other contracts can't directly query balances but data is still viewable to other parties on blockchain. Making something private only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

Always be careful about overflow attacks with numbers. Besides you have already declared public getter functions to query the value of these attributes.

```
function mint(address _to, uint256 _amount) onlyOwner canMint public returns (bool) {
  require(totalSupply_.add(_amount) <= cap);
  require(_to != address(0));
  totalSupply_ = totalSupply_.add(_amount);
  balances[_to] = balances[_to].add(_amount);
  emit Mint(_to, _amount);
  emit Transfer(address(0), _to, _amount);
  return true;
}
```

The mint function call is using an uint256 so be sure that the variable _amount respects the 8 decimals format to avoid any conversion problem to not impact the calculation of the totalSupply..

Moreover, the value of the variable _amount must be different from zero to prevent any call with “0” value. We recommend you to add a control to prevent this situation require(_amount != 0)

```
function finishMinting() onlyOwner canMint public returns (bool) {

  require(foundationAddress != address(0) && teamAddress != address(0) && bountyAddress != address(0));
  require(SHARE_PURCHASERS + SHARE_FOUNDATION + SHARE_TEAM + SHARE_BOUNTY == 1000);

  // before calling this method totalSupply includes only purchased tokens
  uint256 onePerThousand = totalSupply_ / SHARE_PURCHASERS; //ignore (totalSupply mod 617) ~= 616e-8,

  uint256 foundationTokens = onePerThousand * SHARE_FOUNDATION;
  uint256 teamTokens = onePerThousand * SHARE_TEAM;
  uint256 bountyTokens = onePerThousand * SHARE_BOUNTY;

  mint(foundationAddress, foundationTokens);
  mint(teamAddress, teamTokens);
  mint(bountyAddress, bountyTokens);

  mintingFinished = true;
  emit MintFinished();
  return true;
}
```

We have noticed that you make a control of constant values before the execution of token calculation. In this case the “require” is consuming GAS for checking constants that cannot be publicly writable. We recommend you to remove the require control.

For mathematical operations the code is calling Safemath library allowing to prevent any overflow issues. However is not used on your code so we recommend you to apply “mul” and “div” operations instead of “*” and “/”

Moreover be sure that you have finished everything before calling externally the function finishMinting() because you cannot go back once. By design the choice is respectable because you have decided to have fungible tokens preventing the future creation of new TOT tokens which is in adequation with the Whitepaper.

Conclusion

We have not found critical issues which can impact the distribution of ERC20 Tokens. However, some improvements could be done by using Safemath library to avoid overflow risks.

Unit tests

Unit tests are a critical part of testing any project. The contracts described above currently have unit tests, which are marked below. We have executed all javascripts files and also conclude that your coverage is 100% of nominal scenarios.

TOTToken Testing (TTOToken.test.js)

```
import assertRevert from './helpers/assertRevert';
import BigNumber from 'bignumber.js'
const TOTToken = artifacts.require('TOTToken');

contract('TOTToken', function ([_, owner, recipient, anotherAccount]) {
 const ZERO_ADDRESS = '0x0000000000000000000000000000000000000000';

 const batchListAccount = [
  '0xf17f52151ebef6c7334fad080c5704d77216b732',
  '0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef',
  '0x821aea9a577a9b44299b9c15c88cf3087f3b5544'
 ];

 const TOKEN_DECIMAL = 8;
 const MAX_TOKEN_SUPPLY = new BigNumber(20000000 * 10 ** TOKEN_DECIMAL);



 beforeEach(async function () {
  this.token = await TOTToken.new({ from: owner });
 });

 describe('mint', function () {
  it('should start with the correct cap', async function () {
   let _cap = await this.token.cap();
   assert(MAX_TOKEN_SUPPLY.eq(_cap));
  });

  it('should fail to mint if not owner', async function () {
   await assertRevert(this.token.mint(owner, 100, { from: anotherAccount }));
  });

  describe('when the requested amount of token is less than cap', function () {
   it('should mint', async function () {
    const result = await this.token.mint(owner, 100, { from: owner });
    assert.equal(result.logs[0].event, 'Mint');
   });
  });

  describe('when the requested amount of token exceeds the cap', function () {
   it('should fail to mint and revert', async function () {
    await this.token.mint(owner, MAX_TOKEN_SUPPLY.minus(1), { from: owner });
    await assertRevert(this.token.mint(owner, 100, { from: owner }));
   });
  });

  describe('After the cap is reached', function () {
   it('should fail to mint and revert', async function () {
```

```
      await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
      await assertRevert(this.token.mint(owner, 1));
    });
  });
 });

 describe('finishMinting', function () {
  const PURCHASER_AMOUNT = new BigNumber(10000000 * 10 ** TOKEN_DECIMAL);
  const ONE_PER_THOUSAND = PURCHASER_AMOUNT.dividedToIntegerBy(750);
  it('should revert when finishMinting when wallets are not set', async function () {
   await this.token.mint(owner, PURCHASER_AMOUNT, { from: owner });
   await assertRevert(this.token.finishMinting({ from: owner }));
  });
  it('should allocate Foundation, Team and Bounty ', async function () {
   await this.token.mint(owner, PURCHASER_AMOUNT, { from: owner });
   await this.token.setWallets('0xBa893462c1b714bFD801e918a4541e056f9bd924',
'0x2418C46F2FA422fE8Cd0BF56Df5e27CbDeBB2590',
'0x84bE27E1d3AeD5e6CF40445891d3e2AB7d3d98e8',{ from: owner });
   await this.token.finishMinting({ from: owner });
   let bountyBalance = await this.token.balanceOf('0x84bE27E1d3AeD5e6CF40445891d3e2AB7d3d98e8');
   assert(bountyBalance.eq(ONE_PER_THOUSAND.mul(50)));

   let foundationBalance = await this.token.balanceOf('0xBa893462c1b714bFD801e918a4541e056f9bd924');
   assert(foundationBalance.eq(ONE_PER_THOUSAND.mul(50)));

   let teamBalance = await this.token.balanceOf('0x2418C46F2FA422fE8Cd0BF56Df5e27CbDeBB2590');
   assert(teamBalance.eq(ONE_PER_THOUSAND.mul(150)));


assert(teamBalance.plus(foundationBalance).plus(bountyBalance).plus(ONE_PER_THOUSAND.mul(750)).eq(ON
E_PER_THOUSAND.mul(1000)));
  });
  it('should revert when minting after finishMinting ', async function () {
   await this.token.mint(owner, PURCHASER_AMOUNT, { from: owner });
   await this.token.setWallets('0xBa893462c1b714bFD801e918a4541e056f9bd924',
'0x2418C46F2FA422fE8Cd0BF56Df5e27CbDeBB2590',
'0x84bE27E1d3AeD5e6CF40445891d3e2AB7d3d98e8',{ from: owner });
   await this.token.finishMinting({ from: owner });
   await assertRevert(this.token.mint(owner, 1));
  });
 });

 describe('total supply', function () {
  it('returns the total amount of tokens', async function () {
   await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
   const totalSupply = await this.token.totalSupply();
   assert(MAX_TOKEN_SUPPLY.eq(totalSupply));
  });
 });

 describe('balanceOf', function () {
  beforeEach(async function () {
   await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
  });

  describe('when the requested account has no tokens', function () {
   it('returns zero', async function () {
    const balance = await this.token.balanceOf(anotherAccount);
```

```
      assert.equal(balance, 0);
    });
  });

  describe('when the requested account has some tokens', function () {
    it('returns the total amount of tokens', async function () {
      const balance = await this.token.balanceOf(owner);

      assert(MAX_TOKEN_SUPPLY.eq(balance));
    });
  });
});

describe('transfer', function () {
  describe('when the recipient is not the zero address', function () {
    beforeEach(async function () {
      await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
    });

    const to = recipient;

    describe('when the sender does not have enough balance', function () {
      const amount = MAX_TOKEN_SUPPLY.plus(1);

      it('reverts', async function () {
        await assertRevert(this.token.transfer(to, amount, { from: owner }));
      });
    });

    describe('when the sender has enough balance', function () {
      const amount = MAX_TOKEN_SUPPLY;

      it('transfers the requested amount', async function () {
        await this.token.transfer(to, amount, { from: owner });

        const senderBalance = await this.token.balanceOf(owner);
        assert.equal(senderBalance, 0);

        const recipientBalance = await this.token.balanceOf(to);
        assert(amount.eq(recipientBalance));
      });

      it('emits a transfer event', async function () {
        const { logs } = await this.token.transfer(to, amount, { from: owner });

        assert.equal(logs.length, 1);
        assert.equal(logs[0].event, 'Transfer');
        assert.equal(logs[0].args.from, owner);
        assert.equal(logs[0].args.to, to);
        assert(logs[0].args.value.eq(amount));
      });
    });
  });
});

describe('batchMint', async function () {

  describe('when the recipient is not the zero address', function () {
```

```
    describe('when max cap is exceeded', function () {
      const batchListAmount = [
        5000000 * 10 ** TOKEN_DECIMAL,
        10000000 * 10 ** TOKEN_DECIMAL,
        10000000 * 10 ** TOKEN_DECIMAL
      ];

      it('reverts', async function () {
        await assertRevert(this.token.batchMint(batchListAccount, batchListAmount, { from: owner }));
        const totalSupply = await this.token.totalSupply();
        assert.equal(totalSupply, 0);
      });
    });

    describe('when the max cap is not exceeded', function () {
      const batchListAmount = [
        5000000 * 10 ** TOKEN_DECIMAL,
        5000000 * 10 ** TOKEN_DECIMAL,
        10000000 * 10 ** TOKEN_DECIMAL
      ];

      it('mints the requested amount', async function () {

        await this.token.batchMint(batchListAccount, batchListAmount, { from: owner });

        const totalSupply = await this.token.totalSupply();
        assert(MAX_TOKEN_SUPPLY.eq(totalSupply));

        const recipientBalance1 = await this.token.balanceOf(batchListAccount[0]);
        assert.equal(recipientBalance1, batchListAmount[0]);

        const recipientBalance2 = await this.token.balanceOf(batchListAccount[1]);
        assert.equal(recipientBalance2, batchListAmount[1]);

        const recipientBalance3 = await this.token.balanceOf(batchListAccount[2]);
        assert.equal(recipientBalance3, batchListAmount[2]);
      });

      it('emits the Mint and Transfer events', async function () {
        const { logs } = await this.token.batchMint(batchListAccount, batchListAmount, { from: owner });

        assert.equal(logs.length, 6);

        assert.equal(logs[0].event, 'Mint');
        assert.equal(logs[0].args.to, batchListAccount[0]);
        assert(logs[0].args.amount.eq(batchListAmount[0]));

        assert.equal(logs[1].event, 'Transfer');
        assert.equal(logs[1].args.to, batchListAccount[0]);
        assert(logs[1].args.value.eq(batchListAmount[0]));
      });
    });
  });

  describe('when one recipient is the zero address', function () {
    const batchListAmount = [25000000 * 10 ** TOKEN_DECIMAL, 25000000 * 10 ** TOKEN_DECIMAL,
50000000 * 10 ** TOKEN_DECIMAL];

    it('reverts', async function () {
```

```
      await assertRevert(this.token.batchMint([batchListAccount[0], ZERO_ADDRESS, batchListAccount[2]],
batchListAmount, { from: owner }));
            const totalSupply = await this.token.totalSupply();
      assert.equal(totalSupply, 0);
    });
  });

  describe('when 50 recipient ', function () {

    let batch50Account = [50];
    for (var i = 0; i < 50; i++) {
      let address = '0x2bdd21761a483f71054e14f5b827213567971c' + (i + 16).toString(16);
      batch50Account[i] = address;
    }
    let batchListAmount = [50];
    for (var i = 0; i < 50; i++) {
      batchListAmount[i] = 100 * 10 ** TOKEN_DECIMAL;
    }

    it('should perform batch for 50 accounts that should have 100 TOT each', async function () {

      await this.token.batchMint(batch50Account, batchListAmount, { from: owner });
      for (var i = 0; i < batch50Account.length; i++) {
        const tokenBalance = await this.token.balanceOf(batch50Account[i]);
        assert.equal(tokenBalance, 100 * 10 ** TOKEN_DECIMAL);
      }
    });
  });

  describe('when 100 recipient ', function () {

    let batch100Account = [100];
    for (var i = 0; i < 100; i++) {
      let address = '0x2bdd21761a483f71054e14f5b827213567971c' + (i + 16).toString(16);
      batch100Account[i] = address;
    }
    let batchListAmount = [100];
    for (var i = 0; i < 100; i++) {
      batchListAmount[i] = 100 * 10 ** TOKEN_DECIMAL;
    }

    it('should perform batch for 100 accounts that should have 100 TOT each', async function () {

      await this.token.batchMint(batch100Account, batchListAmount, { from: owner });
      for (var i = 0; i < batch100Account.length; i++) {
        const tokenBalance = await this.token.balanceOf(batch100Account[i]);
        assert.equal(tokenBalance, 100 * 10 ** TOKEN_DECIMAL);
      }
    });
  });
});

describe('approve', function () {
  beforeEach(async function () {
    await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
  });

  describe('when the spender is not the zero address', function () {
    const spender = recipient;
```

```
describe('when the sender has enough balance', function () {
  const amount = MAX_TOKEN_SUPPLY;

  it('emits an approval event', async function () {
    const { logs } = await this.token.approve(spender, amount, { from: owner });

    assert.equal(logs.length, 1);
    assert.equal(logs[0].event, 'Approval');
    assert.equal(logs[0].args.owner, owner);
    assert.equal(logs[0].args.spender, spender);
    assert(logs[0].args.value.eq(amount));
  });

  describe('when there was no approved amount before', function () {
    it('approves the requested amount', async function () {
      await this.token.approve(spender, amount, { from: owner });

      const allowance = await this.token.allowance(owner, spender);
      assert(amount.eq(allowance));
    });
  });

  describe('when the spender had an approved amount', function () {
    beforeEach(async function () {
      await this.token.approve(spender, 1, { from: owner });
    });

    it('approves the requested amount and replaces the previous one', async function () {
      await this.token.approve(spender, amount, { from: owner });

      const allowance = await this.token.allowance(owner, spender);
      assert(amount.eq(allowance));
    });
  });
});

describe('when the sender does not have enough balance', function () {
  const amount = MAX_TOKEN_SUPPLY.plus(1);

  it('emits an approval event', async function () {
    const { logs } = await this.token.approve(spender, amount, { from: owner });

    assert.equal(logs.length, 1);
    assert.equal(logs[0].event, 'Approval');
    assert.equal(logs[0].args.owner, owner);
    assert.equal(logs[0].args.spender, spender);
    assert(logs[0].args.value.eq(amount));
  });

  describe('when there was no approved amount before', function () {
    it('approves the requested amount', async function () {
      await this.token.approve(spender, amount, { from: owner });

      const allowance = await this.token.allowance(owner, spender);
      assert(amount.eq(allowance));
    });
  });
```

```
    describe('when the spender had an approved amount', function () {
      beforeEach(async function () {
        await this.token.approve(spender, 1, { from: owner });
      });

      it('approves the requested amount and replaces the previous one', async function () {
        await this.token.approve(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert(amount.eq(allowance));
      });
    });
  });
});

describe('when the spender is the zero address', function () {
  const amount = MAX_TOKEN_SUPPLY;
  const spender = ZERO_ADDRESS;

  it('approves the requested amount', async function () {
    await this.token.approve(spender, amount, { from: owner });

    const allowance = await this.token.allowance(owner, spender);
    assert(amount.eq(allowance));
  });

  it('emits an approval event', async function () {
    const { logs } = await this.token.approve(spender, amount, { from: owner });

    assert.equal(logs.length, 1);
    assert.equal(logs[0].event, 'Approval');
    assert.equal(logs[0].args.owner, owner);
    assert.equal(logs[0].args.spender, spender);
    assert(logs[0].args.value.eq(amount));
  });
});
});

describe('transfer from', function () {
  beforeEach(async function () {
    await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
  });

  const spender = recipient;

  describe('when the recipient is not the zero address', function () {
    const to = anotherAccount;

    describe('when the spender has enough approved balance', function () {
      beforeEach(async function () {
        await this.token.approve(spender, MAX_TOKEN_SUPPLY, { from: owner });
      });

      describe('when the owner has enough balance', function () {
        const amount = MAX_TOKEN_SUPPLY;

        it('transfers the requested amount', async function () {
          await this.token.transferFrom(owner, to, amount, { from: spender });
```

```
      const senderBalance = await this.token.balanceOf(owner);
      assert.equal(senderBalance, 0);

      const recipientBalance = await this.token.balanceOf(to);
      assert(amount.eq(recipientBalance));
    });

    it('decreases the spender allowance', async function () {
      await this.token.transferFrom(owner, to, amount, { from: spender });

      const allowance = await this.token.allowance(owner, spender);
      assert(allowance.eq(0));
    });

    it('emits a transfer event', async function () {
      const { logs } = await this.token.transferFrom(owner, to, amount, { from: spender });

      assert.equal(logs.length, 1);
      assert.equal(logs[0].event, 'Transfer');
      assert.equal(logs[0].args.from, owner);
      assert.equal(logs[0].args.to, to);
      assert(logs[0].args.value.eq(amount));
    });
  });

  describe('when the owner does not have enough balance', function () {
    const amount = MAX_TOKEN_SUPPLY.plus(1);

    it('reverts', async function () {
      await assertRevert(this.token.transferFrom(owner, to, amount, { from: spender }));
    });
  });
});

describe('when the spender does not have enough approved balance', function () {
  beforeEach(async function () {
    await this.token.approve(spender, MAX_TOKEN_SUPPLY.minus(1), { from: owner });
  });

  describe('when the owner has enough balance', function () {
    const amount = MAX_TOKEN_SUPPLY;

    it('reverts', async function () {
      await assertRevert(this.token.transferFrom(owner, to, amount, { from: spender }));
    });
  });

  describe('when the owner does not have enough balance', function () {
    const amount = MAX_TOKEN_SUPPLY.plus(1);

    it('reverts', async function () {
      await assertRevert(this.token.transferFrom(owner, to, amount, { from: spender }));
    });
  });
 });
});

describe('when the recipient is the zero address', function () {
  const amount = MAX_TOKEN_SUPPLY;
```

```
    const to = ZERO_ADDRESS;

    beforeEach(async function () {
      await this.token.approve(spender, amount, { from: owner });
    });

    it('reverts', async function () {
      await assertRevert(this.token.transferFrom(owner, to, amount, { from: spender }));
    });
  });
});

describe('decrease approval', function () {
  beforeEach(async function () {
    await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
  });

  describe('when the spender is not the zero address', function () {
    const spender = recipient;

    describe('when the sender has enough balance', function () {
      const amount = MAX_TOKEN_SUPPLY;

      it('emits an approval event', async function () {
        const { logs } = await this.token.decreaseApproval(spender, amount, { from: owner });

        assert.equal(logs.length, 1);
        assert.equal(logs[0].event, 'Approval');
        assert.equal(logs[0].args.owner, owner);
        assert.equal(logs[0].args.spender, spender);
        assert(logs[0].args.value.eq(0));
      });

      describe('when there was no approved amount before', function () {
        it('keeps the allowance to zero', async function () {
          await this.token.decreaseApproval(spender, amount, { from: owner });

          const allowance = await this.token.allowance(owner, spender);
          assert.equal(allowance, 0);
        });
      });

      describe('when the spender had an approved amount', function () {
        beforeEach(async function () {
          await this.token.approve(spender, amount.plus(1), { from: owner });
        });

        it('decreases the spender allowance subtracting the requested amount', async function () {
          await this.token.decreaseApproval(spender, amount, { from: owner });

          const allowance = await this.token.allowance(owner, spender);
          assert.equal(allowance, 1);
        });
      });
    });

    describe('when the sender does not have enough balance', function () {
      const amount = MAX_TOKEN_SUPPLY.plus(1);
```

```
    it('emits an approval event', async function () {
      const { logs } = await this.token.decreaseApproval(spender, amount, { from: owner });

      assert.equal(logs.length, 1);
      assert.equal(logs[0].event, 'Approval');
      assert.equal(logs[0].args.owner, owner);
      assert.equal(logs[0].args.spender, spender);
      assert(logs[0].args.value.eq(0));
    });

    describe('when there was no approved amount before', function () {
      it('keeps the allowance to zero', async function () {
        await this.token.decreaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert.equal(allowance, 0);
      });
    });

    describe('when the spender had an approved amount', function () {
      beforeEach(async function () {
        await this.token.approve(spender, amount.plus(1), { from: owner });
      });

      it('decreases the spender allowance subtracting the requested amount', async function () {
        await this.token.decreaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert.equal(allowance, 1);
      });
    });
  });
});

describe('when the spender is the zero address', function () {
  const amount = MAX_TOKEN_SUPPLY;
  const spender = ZERO_ADDRESS;

  it('decreases the requested amount', async function () {
    await this.token.decreaseApproval(spender, amount, { from: owner });

    const allowance = await this.token.allowance(owner, spender);
    assert.equal(allowance, 0);
  });

  it('emits an approval event', async function () {
    const { logs } = await this.token.decreaseApproval(spender, amount, { from: owner });

    assert.equal(logs.length, 1);
    assert.equal(logs[0].event, 'Approval');
    assert.equal(logs[0].args.owner, owner);
    assert.equal(logs[0].args.spender, spender);
    assert(logs[0].args.value.eq(0));
  });
 });
});

describe('increase approval', function () {
  beforeEach(async function () {
```

```
  await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
});
const amount = MAX_TOKEN_SUPPLY;

describe('when the spender is not the zero address', function () {
  const spender = recipient;

  describe('when the sender has enough balance', function () {
    it('emits an approval event', async function () {
      const { logs } = await this.token.increaseApproval(spender, amount, { from: owner });

      assert.equal(logs.length, 1);
      assert.equal(logs[0].event, 'Approval');
      assert.equal(logs[0].args.owner, owner);
      assert.equal(logs[0].args.spender, spender);
      assert(logs[0].args.value.eq(amount));
    });

    describe('when there was no approved amount before', function () {
      it('approves the requested amount', async function () {
        await this.token.increaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert(amount.eq(allowance));
      });
    });

    describe('when the spender had an approved amount', function () {
      beforeEach(async function () {
        await this.token.approve(spender, 1, { from: owner });
      });

      it('increases the spender allowance adding the requested amount', async function () {
        await this.token.increaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert(amount.plus(1).eq(allowance));
      });
    });
  });

  describe('when the sender does not have enough balance', function () {
    const amount = MAX_TOKEN_SUPPLY.plus(1);

    it('emits an approval event', async function () {
      const { logs } = await this.token.increaseApproval(spender, amount, { from: owner });

      assert.equal(logs.length, 1);
      assert.equal(logs[0].event, 'Approval');
      assert.equal(logs[0].args.owner, owner);
      assert.equal(logs[0].args.spender, spender);
      assert(logs[0].args.value.eq(amount));
    });

    describe('when there was no approved amount before', function () {
      it('approves the requested amount', async function () {
        await this.token.increaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
```

```
        assert(amount.eq(allowance));
      });
    });

    describe('when the spender had an approved amount', function () {
      beforeEach(async function () {
        await this.token.approve(spender, 1, { from: owner });
      });

      it('increases the spender allowance adding the requested amount', async function () {
        await this.token.increaseApproval(spender, amount, { from: owner });

        const allowance = await this.token.allowance(owner, spender);
        assert(amount.plus(1).eq(allowance));
      });
    });
  });
 });

 describe('when the spender is the zero address', function () {
  const spender = ZERO_ADDRESS;

  it('approves the requested amount', async function () {
    await this.token.increaseApproval(spender, amount, { from: owner });

    const allowance = await this.token.allowance(owner, spender);
    assert(amount.eq(allowance));
  });

  it('emits an approval event', async function () {
    const { logs } = await this.token.increaseApproval(spender, amount, { from: owner });

    assert.equal(logs.length, 1);
    assert.equal(logs[0].event, 'Approval');
    assert.equal(logs[0].args.owner, owner);
    assert.equal(logs[0].args.spender, spender);
    assert(logs[0].args.value.eq(amount));
  });
 });
});

describe('burn', function () {
 beforeEach(async function () {
  await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
 });

 const from = owner;

 describe('when the given amount is not greater than balance of the sender', function () {
  const amount = 100;

  it('burns the requested amount', async function () {
    await this.token.burn(amount, { from });

    const balance = await this.token.balanceOf(from);
    assert(MAX_TOKEN_SUPPLY.minus(100).eq(balance));
  });

  it('emits a burn event', async function () {
```

```
      const { logs } = await this.token.burn(amount, { from });
      const ZERO_ADDRESS = '0x0000000000000000000000000000000000000000';
      assert.equal(logs.length, 2);
      assert.equal(logs[0].event, 'Burn');
      assert.equal(logs[0].args.burner, owner);
      assert.equal(logs[0].args.value, amount);

      assert.equal(logs[1].event, 'Transfer');
      assert.equal(logs[1].args.from, owner);
      assert.equal(logs[1].args.to, ZERO_ADDRESS);
      assert.equal(logs[1].args.value, amount);
    });
  });

  describe('when the given amount is greater than the balance of the sender', function () {
    const amount = MAX_TOKEN_SUPPLY.plus(1);

    it('reverts', async function () {
      await assertRevert(this.token.burn(amount, { from }));
    });
  });
});
describe('pause', function () {
  describe('when the sender is the token owner', function () {
    const from = owner;

    describe('when the token is unpaused', function () {
      it('pauses the token', async function () {
        await this.token.pause({ from });

        const paused = await this.token.paused();
        assert.equal(paused, true);
      });

      it('emits a paused event', async function () {
        const { logs } = await this.token.pause({ from });

        assert.equal(logs.length, 1);
        assert.equal(logs[0].event, 'Pause');
      });
    });

    describe('when the token is paused', function () {
      beforeEach(async function () {
        await this.token.pause({ from });
      });

      it('reverts', async function () {
        await assertRevert(this.token.pause({ from }));
      });
    });
  });

  describe('when the sender is not the token owner', function () {
    const from = anotherAccount;

    it('reverts', async function () {
      await assertRevert(this.token.pause({ from }));
    });
```

```
  });
});

describe('unpause', function () {
  describe('when the sender is the token owner', function () {
    const from = owner;

    describe('when the token is paused', function () {
      beforeEach(async function () {
        await this.token.pause({ from });
      });

      it('unpauses the token', async function () {
        await this.token.unpause({ from });

        const paused = await this.token.paused();
        assert.equal(paused, false);
      });

      it('emits an unpaused event', async function () {
        const { logs } = await this.token.unpause({ from });

        assert.equal(logs.length, 1);
        assert.equal(logs[0].event, 'Unpause');
      });
    });

    describe('when the token is unpaused', function () {
      it('reverts', async function () {
        await assertRevert(this.token.unpause({ from }));
      });
    });
  });

  describe('when the sender is not the token owner', function () {
    const from = anotherAccount;

    it('reverts', async function () {
      await assertRevert(this.token.unpause({ from }));
    });
  });
});

describe('pausable token', function () {
  beforeEach(async function () {
    await this.token.mint(owner, MAX_TOKEN_SUPPLY, { from: owner });
  });

  const from = owner;

  describe('paused', function () {
    it('is not paused by default', async function () {
      const paused = await this.token.paused({ from });

      assert.equal(paused, false);
    });

    it('is paused after being paused', async function () {
      await this.token.pause({ from });
```

```
    const paused = await this.token.paused({ from });

    assert.equal(paused, true);
  });

  it('is not paused after being paused and then unpaused', async function () {
    await this.token.pause({ from });
    await this.token.unpause({ from });
    const paused = await this.token.paused();

    assert.equal(paused, false);
  });
});

describe('transfer', function () {
  it('allows to transfer when unpaused', async function () {
    await this.token.transfer(recipient, MAX_TOKEN_SUPPLY, { from: owner });

    const senderBalance = await this.token.balanceOf(owner);
    assert.equal(senderBalance, 0);

    const recipientBalance = await this.token.balanceOf(recipient);
    assert(MAX_TOKEN_SUPPLY.eq(recipientBalance));
  });

  it('allows to transfer when paused and then unpaused', async function () {
    await this.token.pause({ from: owner });
    await this.token.unpause({ from: owner });

    await this.token.transfer(recipient, MAX_TOKEN_SUPPLY, { from: owner });

    const senderBalance = await this.token.balanceOf(owner);
    assert.equal(senderBalance, 0);

    const recipientBalance = await this.token.balanceOf(recipient);
    assert(MAX_TOKEN_SUPPLY.eq(recipientBalance));
  });

  it('reverts when trying to transfer when paused', async function () {
    await this.token.pause({ from: owner });

    await assertRevert(this.token.transfer(recipient, MAX_TOKEN_SUPPLY, { from: owner }));
  });
});

describe('approve', function () {
  it('allows to approve when unpaused', async function () {
    await this.token.approve(anotherAccount, 40, { from: owner });

    const allowance = await this.token.allowance(owner, anotherAccount);
    assert.equal(allowance, 40);
  });

  it('allows to transfer when paused and then unpaused', async function () {
    await this.token.pause({ from: owner });
    await this.token.unpause({ from: owner });

    await this.token.approve(anotherAccount, 40, { from: owner });
```

```
    const allowance = await this.token.allowance(owner, anotherAccount);
    assert.equal(allowance, 40);
  });

  it('reverts when trying to transfer when paused', async function () {
    await this.token.pause({ from: owner });

    await assertRevert(this.token.approve(anotherAccount, 40, { from: owner }));
  });
});

describe('transfer from', function () {
  beforeEach(async function () {
    await this.token.approve(anotherAccount, 50, { from: owner });
  });

  it('allows to transfer from when unpaused', async function () {
    await this.token.transferFrom(owner, recipient, 40, { from: anotherAccount });

    const senderBalance = await this.token.balanceOf(owner);
    assert(MAX_TOKEN_SUPPLY.minus(40).eq(senderBalance));

    const recipientBalance = await this.token.balanceOf(recipient);
    assert.equal(recipientBalance, 40);
  });

  it('allows to transfer when paused and then unpaused', async function () {
    await this.token.pause({ from: owner });
    await this.token.unpause({ from: owner });

    await this.token.transferFrom(owner, recipient, 40, { from: anotherAccount });

    const senderBalance = await this.token.balanceOf(owner);
    assert(MAX_TOKEN_SUPPLY.minus(40).eq(senderBalance));

    const recipientBalance = await this.token.balanceOf(recipient);
    assert.equal(recipientBalance, 40);
  });

  it('reverts when trying to transfer from when paused', async function () {
    await this.token.pause({ from: owner });

    await assertRevert(this.token.transferFrom(owner, recipient, 40, { from: anotherAccount }));
  });
});

describe('decrease approval', function () {
  beforeEach(async function () {
    await this.token.approve(anotherAccount, 100, { from: owner });
  });

  it('allows to decrease approval when unpaused', async function () {
    await this.token.decreaseApproval(anotherAccount, 40, { from: owner });

    const allowance = await this.token.allowance(owner, anotherAccount);
    assert.equal(allowance, 60);
  });

  it('allows to decrease approval when paused and then unpaused', async function () {
```

```
      await this.token.pause({ from: owner });
      await this.token.unpause({ from: owner });

      await this.token.decreaseApproval(anotherAccount, 40, { from: owner });

      const allowance = await this.token.allowance(owner, anotherAccount);
      assert.equal(allowance, 60);
    });

    it('reverts when trying to transfer when paused', async function () {
      await this.token.pause({ from: owner });

      await assertRevert(this.token.decreaseApproval(anotherAccount, 40, { from: owner }));
    });
  });

  describe('increase approval', function () {
    beforeEach(async function () {
      await this.token.approve(anotherAccount, 100, { from: owner });
    });

    it('allows to increase approval when unpaused', async function () {
      await this.token.increaseApproval(anotherAccount, 40, { from: owner });

      const allowance = await this.token.allowance(owner, anotherAccount);
      assert.equal(allowance, 140);
    });

    it('allows to increase approval when paused and then unpaused', async function () {
      await this.token.pause({ from: owner });
      await this.token.unpause({ from: owner });

      await this.token.increaseApproval(anotherAccount, 40, { from: owner });

      const allowance = await this.token.allowance(owner, anotherAccount);
      assert.equal(allowance, 140);
    });

    it('reverts when trying to increase approval when paused', async function () {
      await this.token.pause({ from: owner });

      await assertRevert(this.token.increaseApproval(anotherAccount, 40, { from: owner }));
    });
  });
 });
});
```

```
    ✓ reverts
  pausable token
    paused
      ✓ is not paused by default
      ✓ is paused after being paused (61ms)
      ✓ is not paused after being paused and then unpaused (74ms)
    transfer
      ✓ allows to transfer when unpaused (85ms)
      ✓ allows to transfer when paused and then unpaused (190ms)
      ✓ reverts when trying to transfer when paused (58ms)
    approve
      ✓ allows to approve when unpaused (48ms)
      ✓ allows to transfer when paused and then unpaused (116ms)
      ✓ reverts when trying to transfer when paused (50ms)
    transfer from
      ✓ allows to transfer from when unpaused (72ms)
      ✓ allows to transfer when paused and then unpaused (144ms)
      ✓ reverts when trying to transfer from when paused (68ms)
    decrease approval
      ✓ allows to decrease approval when unpaused (58ms)
      ✓ allows to decrease approval when paused and then unpaused (120ms)
      ✓ reverts when trying to transfer when paused (46ms)
    increase approval
      ✓ allows to increase approval when unpaused (50ms)
      ✓ allows to increase approval when paused and then unpaused (115ms)
      ✓ reverts when trying to increase approval when paused (65ms)


80 passing (16s)
```

Unit testing conclusion

All unit tests are running successfully. The test coverage represents 100% of the smart contract nominal behavior in TEST RPC network.

Deployment tests

All contracts are deployed with success on the top of the ropsten network. You can check on etherscan :

TOTToken.sol :
https://ropsten.etherscan.io/tx/0xb078c1bd85988ec9fd3e4600f609b973f6f49f76cd03e6d98d324faa4c7e5836

Interaction tests :

Here we will describe the testing process :

1. Deploy TOTToken : OK
   https://ropsten.etherscan.io/tx/0xf79a42b91374a85d840b014e21c2154e36f8984e5b56da03549cac62d1efe3e5

2. Call setName() (When the sender is not the Owner) : OK

https://ropsten.etherscan.io/tx/0xe9bc62a846551909a493722fe2ad8accf29065f5df0644c6d371f2f33354cced

3. Call setWallet() : OK

@bountyAddress : 0x08D1896Ed3029C93Db708eA026670695e29236f8

@teamAddress : 0xb13a10cD2F7eCC3085a311dBEAC2a3F28d2CedCd

@foundationAddress : 0x24202079C04feC583944d01166F1CdD67c09f930

https://ropsten.etherscan.io/tx/0x9f181cdafb838be36b66f74577adf20fc68f75a18df7b212454999ea1ec4a978

4. batchMint() : OK

@contributor1 : 0x96022C4667aF840528BB95eaBe8Cd3da705452c4 (amount = 37500000000)

@contributor2 : 0x23669cD2061FdfF6f1E04B43709324A4E8120C21 (amount = 37500000000)

https://ropsten.etherscan.io/tx/0x05d6db32b6f354c2d3cf114c155a56049bd7287f6e65957a1656a09871741ed8

5. Check TotalSupply = 75000000000 OK

6. Call transferFrom() (from contributo1 to contributor2 when

paused = true) : OK

https://ropsten.etherscan.io/tx/0xeefaa9d6b69587a6def745de54a6838b6da5ca3809bfe721e5e30e3215f4b804

7. Call FinishMinting() : OK

https://ropsten.etherscan.io/tx/0x1bac93787019c297121c87009e4b69090e9c96e865f851c063f9e199d262a06b

8. Check balance of bountyAddress, teamAddress and foundationAddress : OK

@balanceOf("0x08D1896Ed3029C93Db708eA026670695e29236f8") = 5000000000

@balanceOf("0xb13a10cD2F7eCC3085a311dBEAC2a3F28d2CedCd") = 15000000000

@balanceOf("0x24202079C04feC583944d01166F1CdD67c09f930") = 5000000000

9. Check TotalSupply = 100000000000 OK

Deployment & interaction tests conclusion

We have deployed and tested the interactions of your smart contracts on Ethereum Ropsten Testnet. The result of the test done above was done successfully and could be checked on the Ethereum Ropsten Testnet.

Tests don't show any issues regarding an exceed consumption of GAS or problems to call public functions after the deployment.