dapp Documentation

Release 0.1

DappHub

May 06, 2017

Contents:

stable Ethereum building blocks

CHAPTER 1

DS-Auth

DS-Auth is the most fundamental building block of the Dappsys framework. It introduces the concept of contract ownership with two types that work together: `DSAuth` and `DSAuthority`. All the contracts in your system that require some level of authorization to access at least one of their functions should inherit from the `DSAuth` type. This is because this type introduces a public `owner` member of type `address`, a public `authority` member of type `DSAuthority`, and a function modifier called `auth`.

Any function that is decorated with the `auth` modifier will perform an authorization check before granting access to the function. It will perform these checks in order and grant access if any are true:

- If `msg.sender` is the contract itself. This will be the case if a contract makes an external call to one of its own functions (e.g. `this.foo()`)
- If `msg.sender` is the contract's `owner`
- If the contract's `authority` member returns `true` when making this call:

```
authority.canCall(msg.sender, this, msg.sig)
```

The authority will return `true` if `msg.sender` is authorized to call the function identified by `msg.sig` and `false` otherwise.

This is an extremely powerful design pattern because it creates a separation of concerns between authorization and application business logic. The authority could have any number of complex rules that the application contract doesn't need to worry about. For example the authority could be:

- A simple whitelist:

```
address 0x123abc canCall function mint on 0xdef456
```

- A timelocked whitelist:

```
address 0x123abc canCall function mint on 0xdef456
two days after proposing the action to the authority
```

- A role-based permissioning system:

```
address 0x123abc is a member of group 1
which canCall function mint on 0xdef456
```

- A voter veto system:

```
address 0x123abc canCall function mint on 0xdef456
two days after proposing the action to the authority
unless 50% of these token holders veto the action
```

From the application contract's point of view, it's just asking if `msg.sender` `canCall` the function it is trying to call. It doesn't need to worry about all these different schemes that the authority contract might be using. Because the authority member is updateable, this means that more complex authorization/governance logic can be introduced to the system later. Conversely, access can be removed once the system is finished and ready to "lockout" privileged administrators.

Updateability is one of the key benefits offered by DS-Auth. Consider a system where backend-contract A is calling an `auth`-controlled function on another backend-contract B, both owned by authority-contract X. Replacing B with backend-contract C would proceed as follows:

- Create contract C and set its authority to X
- Store data in X that allows contract A to call `auth`-controlled functions on C
- Change the pointer in A to point at C instead of B
- Store data in X that disallows anyone from calling B

This ensures that your production system is always consistent and can easily be rolled back to previous configurations.

TL;DR: If you use just one package from the Dappsys framework, make it DS-Auth. Your system will remain manageable as it grows in size, each individual component will become much easier to understand, and it will integrate seamlessly with the numerous tools that DappHub is building to work with DS-Auth controlled systems.

See also: DS-Guard, DS-Roles

DSAuth

Your contract should inherit from the `DSAuth` type if you want it to have functions that can only be called by specifically authorized addresses.

Import

```
import ds-auth/auth.sol
```

Parent Types

None

API Reference

event LogSetAuthority

This event is logged when setting the contract's `authority` member.

```
event LogSetAuthority (address indexed authority)
```

event LogSetOwner

This event is logged when setting the contract's `owner` member.

```
event LogSetOwner (address indexed owner)
```

function DSAuth

The constructor function sets `msg.sender` to be the initial `owner` of the contract. It does not need to be explicitly called because it doesn't take any parameters.

```
function DSAuth()
```

function authority

Returns the contract's public `authority` member.

```
DSAuthority public authority
```

function owner

Returns the contract's public `owner` member.

```
address public owner
```

function setAuthority

This function sets the `authority` member that your contract calls when executing the `auth` modifier. It is itself `auth` controlled.

```
function setAuthority(DSAuthority authority_) auth
```

function setOwner

This function sets the `owner` member that automatically has access to all the contract's functions. It is itself `auth` controlled.

```
function setOwner(address owner_) auth
```

function isAuthorized

This function returns `true` if the `src` address is allowed to call the `sig` function(s) on this contract. It is mainly used internally by the `auth` and `authorized` modifiers. This function first checks if `src` is equal to the `owner` member, otherwise it calls `authority.canCall(src, this, sig)` and returns the result.

```
function isAuthorized(address src, bytes4 sig) internal returns (bool)
```

modifier auth

This function modifier is the main entrypoint into the logic of `DSAuth`. Decorate your functions with this modifier when you want to control what addresses can call them. It calls `isAuthorized(msg.sender, msg.sig)` and asserts that the return value is `true`, otherwise it throws an exception.

```
modifier auth
```

modifier authorized

DS-Auth also offers a slightly more complex modifier called `authorized`. This modifier takes an arbitrary `bytes4` value instead of the standard `msg.sig` that is used by `auth`. This means that you can group numerous functions under one `sig` that will all be controlled by the same line of authorization data. An example of the difference:

```
// this contract needs two separate entries
// in the owning DSAuthority contract

contract UsingAuth is DSAuth {

    // calling approveAction will cause
    // authority.canCall(msg.sender, this, "approveAction")
    // to be called

    function approveAction() auth {
        // business logic
    }

    // calling approveAction will cause
    // authority.canCall(msg.sender, this, "executeAction")
    // to be called

    function executeAction() auth {
        // business logic
    }
}

// this contract needs only one entry
// in the owning DSAuthority contract

contract UsingAuthorized is DSAuth {

    // calling approveAction will cause
    // authority.canCall(msg.sender, this, "actions")
    // to be called

    function approveAction() authorized("actions") {
        // business logic
    }

    // calling approveAction will cause
    // authority.canCall(msg.sender, this, "actions")
    // to be called
    function executeAction() authorized("actions") {
        // business logic
    }
}
```

The developer should be aware of the design tradeoff here: using `auth` is simpler and less prone to human-error, while using `authorized` is more convenient for large systems but requires more thorough review to ensure that functions are being grouped together properly.

```
modifier authorized(bytes4 sig)
```

DSAuthority

`DSAuthority` is an interface that declares just one function: `canCall`. Contracts that are of this type store authorization data about what addresses can call what specific functions on contracts that are under their authority. Each contract of type `DSAuth` consults its `DSAuthority authority` member when granting access to its functions.

You should extend `DSAuthority` if you want to make new business logic to control access to your system.

Import

`import ds-auth/auth.sol`

Parent Types

None

API Reference

function canCall

This function returns `true` if the `src` address can call the `sig` function(s) on the `dst` contract.

```
function canCall(
    address src, address dst, bytes4 sig
) constant returns (bool)
```

CHAPTER 2

DS-Guard

DS-Guard is an implementation of the *DSAuthority* interface. It is a box contract (meaning it should be deployed to the blockchain without modification) that offers a simple implementation of a whitelist, with the ability to grant addresses the authorization to call individual functions on specific contracts. This is stored in a three-dimensional mapping data structure:

```
mapping (bytes32 => mapping (bytes32 => mapping (bytes32 => bool))) acl
```

It is easy to understand the semantics of this mapping when comparing it against the definition of the `canCall` function that makes up the `DSAuthority` interface:

```
function canCall(address src, address dst, bytes4 sig)
```

Thus, the key of the outermost mapping is the `src` (or calling) address, the key of the middle mapping is the `dst` (or destination) address, and the key of the innermost mapping is the `sig` function.

DS-Guard also has a public constant called `ANY` that allows one to create blanket authorizations. One can create specific `(src, dst, sig)` triples or blanket authorizations to grant access to groups of addresses or functions. These are all the different authorization statements that one can make:

- A specific address can call a specific function on a specific contract
- A specific address can call ANY function on a specific contract
- A specific address can call a specific function on ANY contract
- A specific address can call ANY function on ANY contract
- ANY address can call a specific function on a specific contract
- ANY address can call ANY function on a specific contract
- ANY address can call a specific function on ANY contract
- ANY address can call ANY function on ANY contract

These different statements are all combined with the OR operator, meaning that the most open permission will take precedence. As you can see, this list is divided between giving super-user access to an address and making functions public (which essentially removes the functionality of the `auth` modifier). Both of these types of authorization can be dangerous, so the developer is encouraged to think carefully before using the `ANY` permission. For example, suppose a `DSGuard` controlled access to the `mint` function of a specific contract. If a permission was then added that allowed ANY address to call the `mint` function of ANY contract, it would effectively invalidate the original access control.

The `DSGuard` type inherits from both the `DSAuth` and `DSAuthority` types. This means that it has its own `owner`/`authority`, and it can be an authority for other `DSAuth`-controlled contract systems. It is possible to control one `DSGuard` contract with another `DSGuard` contract!

DSGuard

You should deploy a `DSGuard` contract if you want to control other `DSAuth` contracts with a simple whitelist that holds entries for each access-controlled function. This is the simplest implementation of the `DSAuthority` interface.

Import

`import ds-guard/guard.sol`

Parent Types

DSAuth, *DSAuthority*

API Reference

event LogOkay

This event is logged when modifying the contract's authorization data by calling the `okay` function.

```
event LogOkay(bytes32 src, bytes32 dst, bytes32 sig, bool yes)
```

function ANY

This will return the contract's public `ANY` constant, which is an alias for the maximum possible `uint256` and is used in `DSGuard` to assign blanket permissions (see the introduction above).

```
bytes32 constant public ANY = bytes32(uint(-1))
```

function canCall

This function definition is inherited from *DSAuthority* and will return `true` if the `DSGuard` holds authorization data that allows the `src` address to call the `sig` function on the `dst` address. This will either be an explicit authorization entry for the specific `(src, dst, sig)` triple in question, or a blanket ANY permission (see the introduction above for the different types of ANY permissions).

```
function canCall(
    address src, address dst, bytes4 sig
) constant returns (bool)
```

function okay

This function has three signatures and is controlled by the `authorized` modifier. It allows the caller to edit the whitelist that is consulted by the `canCall` function.

The first signature will write the value of `yes` to the `(src, dst, sig)` triple in the whitelist.

```
function okay(bytes32 src, bytes32 dst, bytes32 sig, bool yes) authorized("okay")
```

This signature is an alias for `okay(src, dst, sig, true)`

```
function okay(address src, address dst, bytes32 sig)
```

This signature is an alias for `okay(src, dst, ANY, true)`

```
function okay(address src, address dst)
```

CHAPTER 3

DS-Math

DS-Math provides arithmetic functions for the common numerical primitive types of Solidity. You can safely add, subtract, multiply, and divide `uint256` and `uint128` numbers without fear of integer overflow. You can also conveniently find the minimum and maximum of two `uint256`, `uint128`, or `int256` numbers.

Additionally, this package provides arithmetic functions for new two higher level numerical concepts called Wad and Ray. These are used to represent decimal numbers using a `uint128`, as the Solidity compiler does not yet support fixed-point mathematics natively (e.g. representing the number `3.141592` as `3141592`).

A Wad is a decimal number with 18 digits of precision and a Ray is a decimal number with 27 digits of precision. These functions are necessary to account for the difference between how integer arithmetic behaves normally, and how decimal arithmetic should actually work. A brief example using `wmul`, which returns the product of two Wads:

```
1.1 * 2.2 = 2.24

Regular integer arithmetic adds orders of magnitude:

110 * 220 = 22400

Wad arithmetic does not add orders of magnitude:

wmul(1100000000000000000, 2200000000000000000) = 2240000000000000000
```

Naming Convention:

The standard functions are considered the `uint256` set, so their function names are not prefixed: `add`, `sub`, `mul`, `div`, `min`, and `max`.

Since `uint128` is half the size of the standard type, `h` is the prefix for this set: `hadd`, `hsub`, `hmul`, `hdiv`, `hmin`, and `hmax`.

The `int256` functions have an `i` prefix: `imin`, and `imax`.

Wad functions have a `w` prefix: `wadd`, `wsub`, `wmul`, `wdiv`, `wmin`, and `wmax`.

Ray functions have a `r` prefix: `radd`, `rsub`, `rmul`, `rdiv`, `rmin`, and `rmax`.

DSMath

Your contract should inherit from this type if you want to perform safe arithmetic functions on `uint256`, `uint128`, `int256` primitive types, or decimal numbers being represented with unsigned integers.

Import

```
import ds-math/math.sol
```

Parent Types

None

API Reference

function add

This function will return `x + y` unless it results in a `uint256` overflow, in which case it will throw an exception.

```
function add(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function sub

This function will return `x - y` unless it results in a `uint256` overflow, in which case it will throw an exception.

```
function sub(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function mul

This function will return `x * y` unless it results in a `uint256` overflow, in which case it will throw an exception.

```
function mul(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function div

This function will return `x / y` unless `y` is equal to 0, in which case it will throw an exception.

```
function div(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function min

This function returns the smaller number between `x` and `y`.

```
function min(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function max

This function returns the larger number between `x` and `y`.

```
function max(uint256 x, uint256 y) constant internal returns (uint256 z)
```

function hadd

This function will return `x + y` unless it results in a `uint128` overflow, in which case it will throw an exception.

```
function hadd(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function hsub

This function will return `x - y` unless it results in a `uint128` overflow, in which case it will throw an exception.

```
function hsub(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function hmul

This function will return `x * y` unless it results in a `uint128` overflow, in which case it will throw an exception.

```
function hmul(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function hdiv

This function will return `x / y` unless `y` is equal to 0, in which case it will throw an exception.

```
function hdiv(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function hmin

This function returns the smaller number between `x` and `y`.

```
function hmin(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function hmax

This function returns the larger number between `x` and `y`.

```
function hmax(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function imin

This function returns the smaller number between `x` and `y`.

```
function imin(int256 x, int256 y) constant internal returns (int256 z)
```

function imax

This function returns the larger number between `x` and `y`.

```
function imax(int256 x, int256 y) constant internal returns (int256 z)
```

function wadd

Alias for *hadd*.

```
function wadd(uint128 x, uint128 y) constant internal returns (uint128)
```

function wsub

Alias for *hsub*.

```
function wsub(uint128 x, uint128 y) constant internal returns (uint128)
```

function wmul

This function will multiply two Wads and return a new Wad with the correct level of precision. A Wad is a decimal number with 18 digits of precision that is being represented as an integer. To learn more, see the introduction to DS-Math above.

```
function wmul(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function wdiv

This function will divide two Wads and return a new Wad with the correct level of precision. A Wad is a decimal number with 18 digits of precision that is being represented as an integer. To learn more, see the introduction to DS-Math above.

```
function wdiv(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function wmin

Alias for *hmin*.

```
function wmin(uint128 x, uint128 y) constant internal returns (uint128)
```

function wmax

Alias for *hmax*.

```
function wmax(uint128 x, uint128 y) constant internal returns (uint128)
```

function radd

Alias for *hadd*.

```
function radd(uint128 x, uint128 y) constant internal returns (uint128)
```

function rsub

Alias for *hsub*.

```
function rsub(uint128 x, uint128 y) constant internal returns (uint128)
```

function rmul

This function will multiply two Rays and return a new Ray with the correct level of precision. A Ray is a decimal number with 27 digits of precision that is being represented as an integer. To learn more, see the introduction to DS-Math above.

```
function rmul(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function rdiv

This function will divide two Rays and return a new Ray with the correct level of precision. A Ray is a decimal number with 27 digits of precision that is being represented as an integer. To learn more, see the introduction to DS-Math above.

```
function rdiv(uint128 x, uint128 y) constant internal returns (uint128 z)
```

function rpow

This function will raise a Ray to the n^th power and return a new Ray with the correct level of precision. A Ray is a decimal number with 27 digits of precision that is being represented as an integer. To learn more, see the introduction to DS-Math above.

```
function rpow(uint128 x, uint64 n) constant internal returns (uint128 z)
```

function rmin

Alias for *hmin*.

```
function rmin(uint128 x, uint128 y) constant internal returns (uint128)
```

function rmax

Alias for *hmax*.

```
function rmax(uint128 x, uint128 y) constant internal returns (uint128)
```

function cast

This function will transform a `uint256` into a `uint128` and return it after asserting that it is equal to the original parameter `x`.

```
function cast(uint256 x) constant internal returns (uint128 z)
```

CHAPTER 4

DS-Note

The `DSNote` type is a way to generically log function calls as events. To do this, it provides a `LogNote` event and a `note` modifier. Functions that are decorated with the `note` modifier, will log a `LogNote` event that contains this list of data:

- `msg.sig`
- `msg.sender`
- The first parameter that the function is taking
- The second parameter that the function is taking
- `msg.value`
- `msg.data`

The first four items are indexed, making them queryable by blockchain clients. This covers *most* of the usecases for events, making this package useful to quickly add event logging functionality to your dapp.

DSNote

Your contract should inherit from the `DSNote` type if you want a simple repeatable way to log function calls as events.

Import

```
import ds-note/note.sol
```

Parent Types

None

API Reference

event LogNote

This event will log information about functions that are decorated with the `note` modifier. The parameters correspond to these data fields:

- `sig` is `msg.sig`
- `guy` is `msg.sender`
- `foo` is the first parameter that the function is taking
- `bar` is the second parameter that the function is taking
- `wad` is `msg.value`
- `fax` is `msg.data`

```
event LogNote(
    bytes4   indexed  sig,
    address  indexed  guy,
    bytes32  indexed  foo,
    bytes32  indexed  bar,
    uint              wad,
    bytes             fax
) anonymous
```

modifier note

Decorating functions with the `note` modifier will cause useful information to be logged when they are called. See `LogNote` above for the specific information that gets logged.

```
modifier note
```

CHAPTER 5

DS-Token

The DS-Token package is for creating useful ERC20 tokens. There are two types included in this package, `DSTokenBase` and `DSToken`. If you just want a standard ERC20 template to extend with your business logic, inherit from the `DSTokenBase` type.

If you already working with a *DSAuth*-controlled system however, we highly recommend using the `DSToken` type. This type is meant to serve as a box contract (meaning it should be deployed to the blockchain without modification) and then controlled with other admin contracts. This is because `DSToken` introduces three new features that cover almost all *direct* usecases for tokens: `mint`, `burn`, and `stop`/`start`. As one might guess, the `mint` function creates new tokens, the `burn` function destroys existing tokens, and the `stop`/`start` functions disable/enable the normal functionality of the token. All three of these features are `auth` controlled, which means *a separate contract can control the token and its supply according to unique business logic*.

This is what makes `DSToken` so useful at its core. It is obvious that all the different ERC20 tokens of Ethereum will have special unique business logic, but at their core they almost always boil down to creating and destroying tokens according to some condition. The `DSToken` building block can be deployed just once and still fit into an upgradeable system thanks to the separation of concerns afforded by DS-Auth. A brief example:

```
// An interface that your unique business logic implements
contract SystemRules {

    function canCashOut(address user);

    function serviceFee() returns (uint128);
}

// A basic controller that handles cashing out
// from appTokens to some underlying collateral

contract AppTokenController is DSAuth, DSMath {

    ERC20 deposit;
    DSToken appToken;

    SystemRules rules;
    function cashOut(uint128 wad) {
        assert(rules.canCashOut(msg.sender));

        // Basic idea here is that prize < wad
        // with the contract keeping the difference as a fee.
        // See DS-Math for wdiv docs.

        uint prize = wdiv(wad, rules.serviceFee());

        appToken.pull(msg.sender, wad);

        // only this contract is authorized to burn tokens
        appToken.burn(prize);

        deposit.transfer(msg.sender, prize);
    }

    function newRules(SystemRules rules_) auth {
        rules = rules_;
    }
}
```

As you can see, the actual operations on the token are completely covered by the `DSToken` feature set, there is some app-specific logic around charging fees, and the system is completely updateable. This is what makes the `DSToken` type so useful.

DSTokenBase

Your contract should inherit from the `DSTokenBase` type if you want it to have standard ERC20 functionality.

Import

`import ds-token/base.sol`

Parent Types

ERC20, *DSMath*

API Reference

function totalSupply

Returns the outstanding supply of all tokens.

```
function totalSupply() constant returns (uint256)
```

function balanceOf

Returns the balance of the `src` address.

```
function balanceOf(address src) constant returns (uint256)
```

function allowance

Returns the amount of tokens that `guy` can withdraw from the `src` address via the `transferFrom` function.

```
function allowance(address src, address guy) constant returns (uint256)
```

function approve

Approves `guy` to withdraw `wad` tokens from `msg.sender` via the `transferFrom` function. Throws on uint overflow.

```
function approve(address guy, uint256 wad) returns (bool)
```

function transfer

Transfers `wad` tokens from `msg.sender` to the `dst` address. Throws on uint overflow.

```
function transfer(address dst, uint wad) returns (bool)
```

function transferFrom

Assumes sufficient approval set by the `approve` function. Transfers `wad` tokens from the `src` address to the `dst` address and decrements `wad` from `approvals[src][msg.sender]`. Throws on uint overflow.

```
function transferFrom(address src, address dst, uint wad) returns (bool)
```

DSToken

You should deploy a `DSToken` contract if you want standard ERC20 functionality, plus the ability to create and destroy tokens and disable the token's functionality from authorized addresses. This complete functionality covers almost all basic token usecases, allowing you to separate app-specific business logic into adminstrative controller contracts with elevated permissions on the token.

Import

```
import ds-token/token.sol
```

Parent Types

DSTokenBase(0), *DSStop*, *DSAuth*, *DSNote*

API Reference

function DSToken

The constructor function only assumes you want a custom symbol. For custom `name`, see the `setName` function. For custom `decimals`, please think carefully about what you're doing and then override the type if you still need them.

```
function DSToken(bytes32 symbol_)
```

function symbol

Returns the value of the public `symbol` variable. Used to identify the token.

```
string public symbol
```

function decimals

Returns `18`. For custom `decimals`, please think carefully about what you're doing and then override the type if you still need them.

```
uint256 public decimals = 18
```

function mint

Creates `wad` tokens and adds them to the balance of `msg.sender`, increasing the total supply. Throws on uint overflow.

```
function mint(uint128 wad) auth stoppable note
```

function burn

Removes `wad` tokens from the balance of `msg.sender` and destroys them, reducing the total supply. Throws on uint overflow.

```
function burn(uint128 wad) auth stoppable note
```

function push

Alias for `transfer(dst, wad)`.

```
function push(address dst, uint128 wad) returns (bool)
```

function pull

Alias for `transferFrom(src, msg.sender, wad)`.

```
function pull(address src, uint128 wad) returns (bool)
```

function transfer

Identical functionality to its parent function in `DSTokenBase`. Adds the `stoppable` and `note` modifiers.

```
function transfer(address dst, uint wad) stoppable note returns (bool)
```

function transferFrom

Identical functionality to its parent function in `DSTokenBase`. Adds the `stoppable` and `note` modifiers.

```
function transferFrom(
    address src, address dst, uint wad
) stoppable note returns (bool)
```

function approve

Identical functionality to its parent function in `DSTokenBase`. Adds the `stoppable` and `note` modifiers.

```
function approve(address guy, uint wad) stoppable note returns (bool)
```

function name

Returns the value of the public `name` variable. Used to identify the token. Defaults to empty string and can be set via `setName`.

```
bytes32 public name = ""
```

function setName

Sets the token's `name` variable. Controlled by `auth`.

```
function setName(bytes32 name_) auth
```

CHAPTER 6

DS-Stop

DS-Stop is a simple mixin type that allows an authorized account to disable and enable functions on deriving types via a `stoppable` modifier. It is useful in situations where one needs to halt a system for maintenance, in case of emergency, or simply to wind it down after a temporary lifespan.

DSStop

Your contract should inherit from the `DSStop` type if you want an admin address to be able to disable/enable any of its functions.

Import

```
import ds-stop/stop.sol
```

Parent Types

DSAuth, *DSNote*

API Reference

function stopped

Returns the value of the public `stopped` variable, which is set to `true` when the token's `stoppable` functions are disabled.

```
bool public stopped
```

function stop

Sets `stopped` to `true`, which disables normal token behavior.

```
function stop() auth note
```

function start

Sets `stopped` to `false`, which enables normal token behavior.

```
function start() auth note
```

modifier stoppable

Asserts that `stoppable` is equal to `false`, allowing an admin account to disable normal token operations.

```
modifier stoppable
```