NatSpec Format

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).

Note

NatSpec was inspired by Doxygen. While it uses Doxygen-style comments and tags, there is no intention to keep strict compatibility with Doxygen. Please carefully examine the supported tags listed below.

This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler. Also detailed below is output of the Solidity compiler, which extracts these comments into a machine-readable format.

NatSpec may also include annotations used by third-party tools. These are most likely accomplished via the `@custom:<name>` tag, and a good use case is analysis and verification tools.

Documentation Example

Documentation is inserted above each `contract`, `interface`, `library`, `function`, and `event` using the Doxygen notation format. A `public` state variable is equivalent to a `function` for the purposes of NatSpec.

- For Solidity you may choose `///` for single or multi-line comments, or `/**` and ending with `*/`.
- For Vyper, use `"""` indented to the inner contents with bare comments. See the Vyper documentation.

The following example shows a contract and a function using all available tags.

Note

The Solidity compiler only interprets tags if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

This may change in the future.

open in Remix

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
/// @custom:experimental This is an experimental contract.
contract Tree {
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return Age in years, rounded up for partial years
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Returns the amount of leaves the tree has.
    /// @dev Returns only a fixed number.
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}

contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// Return the amount of leaves that this specific kind of tree has
    /// @inheritdoc Tree
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

Tags

All tags are optional. The following table explains the purpose of each NatSpec tag and where it may be used. As a special case, if no tags are used then the Solidity compiler will interpret a `///` or `/**` comment in the same way as if it were tagged with `@notice`.

Tag		Context
`@title`	A title that should describe the contract/interface	contract, library, interface
`@author`	The name of the author	contract, library, interface
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event
`@return`	Documents the return variables of a contract's function	function, public state variable
`@inheritdoc`	Copies all missing tags from the base function (must be followed by the contract name)	function, public state variable
`@custom:...`	Custom tag, semantics is application-defined	everywhere

If your function returns multiple values, like `(int quotient, int remainder)` then use multiple `@return` statements in the same format as the `@param` statements.

Custom tags start with `@custom:` and must be followed by one or more lowercase letters or hyphens. It cannot start with a hyphen however. They can be used everywhere and are part of the developer documentation.

Dynamic expressions

The Solidity compiler will pass through NatSpec documentation from your Solidity source code to the JSON output as described in this guide. The consumer of this JSON output, for example the end-user client software, may present this to the end-user directly or it may apply some pre-processing.

For example, some client software will render:

open in Remix

```
/// @notice This function will multiply `a` by 7
```

to the end-user as:

```
This function will multiply 10 by 7
```

if a function is being called and the input `a` is assigned a value of 10.

Inheritance Notes

Functions without NatSpec will automatically inherit the documentation of their base function. Exceptions to this are:

- When the parameter names are different.
- When there is more than one base function.
- When there is an explicit `@inheritdoc` tag which specifies which contract should be used to inherit.

Documentation Output

When parsed by the compiler, documentation such as the one from the above example will produce two different JSON files. One is meant to be consumed by the end user as a notice when a function is executed and the other to be used by the developer.

If the above contract is saved as `ex1.sol` then you can generate the documentation using:

```
solc --userdoc --devdoc ex1.sol
```

And the output is below.

Note

Starting Solidity version 0.6.11 the NatSpec output also contains a `version` and a `kind` field. Currently the `version` is set to `1` and `kind` must be one of `user` or `dev`. In the future it is possible that new versions will be introduced, deprecating older ones.

User Documentation

The above documentation will produce the following user documentation JSON file as output:

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculate tree age in years, rounded up, for live trees"
    }
  },
  "notice" : "You can use this contract for only the most basic simulation"
}
```

Note that the key by which to find the methods is the function’s canonical signature as defined in the Contract ABI and not simply the function’s name.

Developer Documentation

Apart from the user documentation file, a developer documentation JSON file should also be produced and should look like this:

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "All function calls are currently implemented without side effects",
  "custom:experimental" : "This is an experimental contract.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "The Alexandr N. Tetearing algorithm could increase precision",
      "params" :
      {
        "rings" : "The number of rings from dendrochronological sample"
      },
      "return" : "age in years, rounded up for partial years"
    }
  },
  "title" : "A simulator for trees"
}
```