Solium Documentation

Release 1.0.0

Raghav Dua <duaraghav8@gmail.com>

Sep 14, 2019

Contents

Solium analyzes your Solidity code for style & security issues and fixes them.

Standardize Smart Contract practices across your organisation. Integrate with your build system. Deploy with confidence!

Solium does not strictly follow Solidity Style Guide. The practices it enforces by default are best practices for the community at large.

Keyword Index, Search Page

CHAPTER 1

User Guide

1.1 Quickstart

- `npm install -g solium`
- `cd myDapp`
- `solium --init`
- `solium -d contracts/` or `solium -d .` or `solium -f myContract.sol`

Fix code

- `solium -f myContract.sol --fix`
- `git diff myContract.sol`

1.2 Installation

Since this documentation is for Solium `v1`, we're going to neglect `v0`. Furthermore, `v0` is now deprecated and we highly recommend you to move to `v1`.

Use `npm install -g solium`.

Verify that all is working fine using `solium -V`.

1.2.1 Install from the snap store

In any of the supported Linux distros, `sudo snap install solium --edge`

Note: If you're using vim with syntastic, and prefer to use a locally installed version of Solium (rather than a global version), you can install syntastic local solium to automatically load the local version in packages that have installed

their own.

1.3 Usage

`cd` to your DApp directory and run `solium --init`. This will produce `.soliumrc.json` and `.soliumignore` files in your root directory. Both are to be commited to version control.

Now run `solium --dir .` to lint all `.sol` files in your directory and sub-directories.

If you want to run the linter over a specific file, use `solium --file myContract.sol`.

You can also run solium so it watches your directory for changes and automatically re-lints the contracts: `solium --watch --dir contracts/`.

By default, solium looks for the `.soliumrc.json` configuration file in your current directory. You can override this setting by using the `--config`/`-c` option like `solium -d contracts/ -c ../configs/.soliumrc.json`.

Solium supports multiple output formats:

- `Pretty` (Default): `--reporter pretty`
- `GCC`: `--reporter gcc`

Use `solium --help` for more information on usage.

Note: If all your contracts reside inside a directory like `contracts/`, you can instead run `solium --dir contracts`.

Note: `-d` can be used in place of `--dir` and `-f` in place of `--file`.

You can specify rules or plugins to apply as commandline options. If you specify one, it overrides its corresponding configuration in the soliumrc file.

```
solium --plugin zeppelin --rule 'indentation:  ["error", 4]' -d contracts/
```

Use `--no-soliumrc` and `--no-soliumignore` if you want to run solium in any arbitrary folder without looking for the config files.

```
solium --no-soliumrc --no-soliumignore --plugin zeppelin --rule 'indentation:
["error", 4]' -f contract.sol
```

After linting over your code, Solium produces either warnings, errors or both. The tool exits with a non-zero code only if 1 or more errors were found. So if all you got was warnings, solium exits with `0`.

Whether an issue should be flagged as an error or warning by its rule is configurable through `.soliumrc.json`.

1.4 Configuring the Linter

Think of Solium as a 2-sided engine. 1 side accepts the Solidity smart contracts along with a configuration and the other a set of rule implementations.

Solium's job is to **execute the rules on the contracts based on the configuration**!

You can configure solium in several ways. You can choose which all rules to apply on your code, what should their severity be (either *error* or *warning*) and you can pass them options to modify their behavior. Rule implementations will **always** contain default behavior, so its fine if you don't pass any options to a rule.

Solium contains some core rules and allows for third party developers to write plugins.

The `.soliumrc.json` created in the initialisation phase contains some default configurations for you to get started.

```
{
        "extends": "BASE RULESET",
        "plugins": ["security"],
        "rules": {
                "RULE NAME": ["SEVERITY", "PARAMETERS"],
                "RULE NAME": "ONLY SEVERITY"
        }
}
```

- By default, soliumrc inherits `solium:recommended` (starting `v1.1.3`, prior to which it was `solium:all`) - the base ruleset which enables all non-deprecated rules recommended for general audience. You can replace the value by a sharable config's name (see *Sharable Configs*).
- A few rules are passed additional configuration, like double quotes for all strings, 4 spaces per indentation level, etc.
- Solium comes bundled with its official security plugin. By default, this plugin is enabled. **We recommend that you keep the security plugin enabled without modifying behaviour of any of its rules.** But if you still wish to configure them or remove the plugin altogether, you can.

Note: soliumrc must contain at least one of `extends`, `plugins` and `rules` properties.

Note: Severity can be expressed either as a string or integer. `error` = `2`, `warning` = `1`. `off` = `0`, which means the rule is turned off.

1.4.1 Configuring with comments

Comment Directives can be used to configure Solium to ignore specific pieces of code. They follow the pattern `solium-disable<optional suffix>`.

If you only use the directive, Solium disables all rules for the marked code. If that's not desirable, specify the rules to disable after the directive, separated by comma.

- Disable linting on a specific line

```
contract Foo {
        /* solium-disable-next-line */
        function() {
                var bar = 'Hello world';        // solium-disable-line quotes

                // solium-disable-next-line security/no-throw, indentation
                                                throw;
        }

        function func(string foo) {
                /**
```

(continues on next page)

(continued from previous page)

```
                 * Below if block is intentionally empty but needs to be preceded by
                 * a doc comment.
                 */
                if (condition) {
                        // solium-disable-previous-line no-empty-blocks
                }
        }
}
```

- Disable linting on entire file

```
/* solium-disable */

contract Foo {
        ...
}
```

- Disable linting over a section of code with `solium-enable` directive

```
/* solium-disable */

contract Foo {
        ...
}

/* solium-enable */

contract Bar {
        ...
}

// solium-disable security/no-throw, indentation
contract Baz {
        throw;
        // solium-enable security/no-throw
}
```

1.5 Automatic code formatting

For the times when you're feeling lazy, just run `solium -d contracts/ --fix` to fix your lint issues. This doesn't fix all the code problems but all lint issues that CAN be fixed WILL be fixed, if the rule implementation that flags the issue also contains a fix for it.

Alternatively, you can use the `--fix-dry-run` option to see the list of changes the linter intends to apply to your code. This option is only supported with the `pretty` reporter, which is enabled by default.

Warning: Solium fixes your code in-place, so your original file is over-written. It is therefore recommended that you use this feature after ensuring that your original files are easily recoverable (recovering can be as simple as `git checkout`). You have been warned.

Note: It is not guaranteed that all the fixes will be applied to your contract code. Below is a brief explanation of why it is so. Skip to the next section if you don't wish to know the details, they're not necessary as long as you accept the

idea.

How the autofix mechanism works is:

- All rule implementations (either core or plugin) supply their fixes via the `fix()` method
- All rules are executed on your solidity code and their provided fixes registered
- The supplied fixes are then sorted. Starting from the 1st line & 1st character, the fix that wishes to manipulate code earlier gets applied earlier. So if fix `A` wants to start make changes from Line 1, Char 7 to Line 1 Char 15 and fix `B` starts at Line 2 Char 19, the order of fixes applied is `A` -> `B`.
- In case of overlapping fixes, the **one that comes later is discarded**. If fix `C` wishes to make changes starting at Line 1 Char 9, it will result in a conflict with fix `A` in the previous point. In this case, `A` gets applied but `C` doesn't. So even though we have a total of 3 fixes, only 2 get applied.

Note: In case of the `A`, `B`, `C` example, its easy to conclude that if you run the linter with autofixing twice, you will have applied all 3 fixes. The first run applies `A` and `B`, whilst the second run will apply `C` as well, because this time there is no `A` to conflict with.

Note: Fixes for all possible errors have not been implemented yet. Whichever rules have the fix mechanism (for eg- the `quotes` rule does) will apply it. More fixes will be added in future, you can see the list of rules below to know which rules are currently able to apply fixes. So if you see a warning/error despite using `--fix`, its because that issue wasn't resolved by the autofix mechanism (either because the fix doesn't exist at the moment or due to a conflict).

1.6 Sharable Configs

The list of rules in Solium will keep growing over time. After a point, its just overkill to spend time specifying rules, their severities and options in your soliumrc every time you create a new Solidity Project. At that time, you can either choose to inherit `solium:recommended` or `solium:all` configuration or borrow configurations written by others.

A Sharable Config allows you to borrow someone else's soliumrc configuration. The idea is to simply pick a style to follow and focus on your business problem instead of making your own style specification.

Even if there are 1 or 2 rules that you disagree with in someone else's sharable config, you can always inherit it and override those rules in your soliumrc!

Sharable Configs are installed via NPM. All solium SCs will have a prefix `solium-config-`. Distributors of sharable configs are encouraged to add `solium` and `soliumconfig` as tags in their NPM modules to make them more discoverable.

Suppose Consensys releases their own sharable config called `solium-config-consensys`. Here's how you'd go about using it, assuming you already have solium globally installed:

- Run `npm install -g solium-config-consensys`
- Now, in your `.soliumrc.json`, set the value of `extends` key to `consensys` and remove the `rules` key altogether. Your config file should now look something like:

```
{
        "extends": "consensys"
}
```

Note: The above assumes that you completely follow consensys's style spec. If, say, you don't agree with how they've configured a rule `race-conditions`. You can override this rule and add your own spec inside the `rules` key. This way, you follow all rules as specified in consensys' sharable config except `race-condition`, which you specify yourself.

```
{
        "extends": "consensys",
        "rules": {
                "race-condition": ["error", {"reentrancy": true, "cross-function":␣
↪false}, 100, "foobar"]
        }
}
```

That's it! Now you can run `solium -d contracts/` to see the difference.

Note that you **didn't have to specify the prefix of the sharable config**. Whether you're specifying a config or a plugin name, you should omit their prefixes (`solium-config-` for configs & `solium-plugin-` for plugins). So if you have installed a config `solium-config-foo-bar`, you should have `"extends": "foo-bar"` in your `.soliumrc.json`. Solium will resolve the actual npm module name for you.

Note: Internally, Solium simply `require()` s the config module. So as long as require() is able to find a module named `solium-config-consensys`, it doesn't matter whether you install your config globally or locally and link it.

Note: 1 limitation here is that Sharable configs can currently not import Plugins. This means SCs can only configure the core rules provided by Solium. Plugin importing is a work in progress, please be patient!

1.7 Plugins

Plugins allow Third party developers to write their own rules and re-distribute them via NPM. Every solium plugin module has the prefix `solium-plugin-`. Plugin developers are encouraged to include the tags `solium` and `soliumplugin` in their modules for easy discoverability.

Once you install a plugin, you can specify it inside `plugins` array and configure its rules inside `rules` exactly like how you configure solium's core rules. Plugin rules too can contain fixes if the developer supplies them. There's no special way of applying these fixes. Simply lint with the `--fix` option and fixes for both core rules and pugin rules will be applied to your code.

Coming back to our previous example - Consensys' `solium-plugin-consensys`:

- Install the plugin using `npm install -g solium-plugin-consensys`
- Add the plugin's entry into your `.soliumrc.json`:

```
{
        "extends": "solium:recommended",
        "plugins": ["consensys"]
}
```

Note: Just like in sharable configs, don't specify the plugin prefix. Simply specify the plugin name. So if a plugin exists on NPM by the name of `solium-plugin-foo-bar`, you need only specify `"plugins": ["foo-bar"]`.

- In the `rules` object, you can configure the plugin's rules by adding an entry `"<PLUGIN NAME>/<RULE NAME>": "<SEVERITY>"` or `"<PLUGIN NAME>/<RULE NAME>": ["<SEVERITY>", "<OPTIONS>"]`.

```
{
        "extends": "solium:recommended",
        "plugins": ["consensys"],
        "rules": {
                "consensys/race-conditions": "error",
                "consensys/foobar": [1, true, "Hello world"]
        }
}
```

- The above configuration means you've applied all the rules supplied by the plugin and modified the behaviour of 2 of them. Try running the linter using `solium -d contracts/`.

If you simply specify a plugin and do not configure any of its rules, all the rules provided by the plugin are applied on your code with their default severities and no additional options. **If you wish to change the behaviour of any of the rules of a plugin, you have to configure them inside "rules".**

You should check the plugin's documentation provided by the plugin developer to know the list of rules provided and the options they accept.

Note: Just like in sharable configs, solium internally `require()` s the plugin module. So as long as require() is able to find a module named `solium-plugin-consensys`, it doesn't matter whether you install your plugin globally or locally and link it.

1.7.1 Recommended Security Plugin

Starting `v1.0.1`, Solium comes pre-installed with its official security plugin (view on NPM) containing lint rules for best security practices. These rules have been taken from Consensys recommended practices and Solium's Rule Wishlist thread.

You can get information about all the rules this plugin supplies on its README.

When you run `solium --init`, the `.soliumrc.json` created for you contains the entry `"plugins": ["security"]`. This means all security rules will by default be applied during linting.

We recommend that you keep the security plugin applied without modifying behaviour of any of its rules. But if you still wish to configure them or remove the plugin altogether, you can.

1.8 List of Style Rules

Note: See security plugin if you're looking for documentation on Solium's security rules.

Below is the list of style rules supplied by Solium. By default, `solium:recommended` is extended by your soliumrc, which enables all lint rules recommended for general audience (See solium-recommended). You can choose to

further configure their severities inside your soliumrc itself. If you choose `solium:all` instead, all core rules are enabled **except for the deprecated ones**. Enabling a deprecated rule will display a warning message on Solium CLI.

These rules may or may not contain fixes. Their fixes will be applied on the code if you use the `--fix` flag in your lint command. Some rules even take options that can modify their behavior.

For eg- your choice of indentation might be Tab or 4 spaces or 2 spaces. What indentation is enforced is configurable.

Name	Description	Options	Defaults	Fixes
imports-on-top	Ensure that all import statements are on top of the file	•		YES
variable-declarations	Ensure that names 'l', 'O' & 'I' are not used for variables	Array of strings representing forbidden names. This overwrites the default names.	['l', 'O', 'I']	
array-declarations	Ensure that array declarations don't have space between the type and brackets	•		YES
operator-whitespace	Ensure that operators are surrounded by a single space on either side	•		
conditionals-whitespace	Ensure that there is exactly one space between conditional operators and parenthetic blocks	•		
comma-whitespace	Ensure that there is no whitespace or comments between comma delimited elements and commas	•		
semicolon-whitespace	Ensure that there is no whitespace or comments before semicolons	•		
function-whitespace	Ensure function calls and declaration have (or don't have) whitespace in appropriate locations	•		
lbrace	Ensure that every if, for, while and do statement is followed by an opening curly brace '{' on the same line	•		

Continued on next page

Table 1 – continued from previous page

mixedcase	Ensure that all variable, function and parameter names follow the mixedCase naming convention	•		
camelcase	Ensure that contract, library, modifier and struct names follow CamelCase notation	•		
uppercase	Ensure that all constants (and only constants) contain only upper case letters and underscore	•		
no-empty-blocks	Ensure that no empty blocks exist. Exempts fallback and payable functions and payable constructors.	•		
no-unused-vars	Flag all the variables that were declared but never used	•		
quotes	Ensure that all strings use only 1 style - either double quotes or single quotes	Single option - either “double” or “single”	double	YES
blank-lines	Ensure that there is exactly a 2-line gap between Contract and Funtion declarations	•		YES
indentation	Ensure consistent indentation of 4 spaces per level	either “tab” or an integer representing the number of spaces	4 spaces	
arg-overflow	In the case of 4+ elements in the same line require they are instead put on a single line each	Single integer representing the number of args to allow per line	4	
whitespace	Specify where whitespace is suitable and where it isn’t	•		
deprecated-suicide	Suggest replacing deprecated ‘suicide’ for ‘selfdestruct’	•		YES

Continued on next page

Table 1 – continued from previous page

pragma-on-top	Ensure a) A PRAGMA directive exists and b) its on top of the file	•		YES
function-order	Ensure order of functions in a contract: constructor,fallback,external,public,internal,private	Functions to ignore (https://github.com/duaraghav8/Solium/issues/335)	(See below)	
emit	Ensure that emit statement is used to trigger a solidity event	•		YES
no-constant	Ensure that view is used over deprecated constant in function declarations	•		YES
value-in-payable	Ensure 'msg.value' is only used in functions with the 'payable' modifier	•		
no-experimental	Ensure that experimental features are not used in production	•		
max-len	Ensure that a line of code doesn't exceed the specified number of characters	Single integer representing the number of characters to allow per line of code	145	
error-reason	Ensure that error message is provided for revert and require statements	Object with "revert" and "require" booleans & "errorMessageMaxLength" unsigned int	{ "revert": true, "require": true, "errorMessageMaxLength": 76 }	
visibility-first	Ensure that the visibility modifier for a function should come before any custom modifiers	•		
linebreak-style	Ensure consistent linebreak style	linebreak style (either "windows" or "unix")	unix	YES
constructor	Ensure that the deprecated style of constructor declaration is not used			YES
no-trailing-whitespace	Ensure that lines do not contain trailing whitespaces	Object specifying whether to ignore blank lines or comments	{ "skipBlankLines": false, "ignoreComments": false }	

The following is an example of a configuration object passed to the `function-order` rule. See https://github.com/duaraghav8/Solium/issues/235 to understand its purpose and usage.

```
{
    "rules": {
        "function-order": [
            "error",
            {
                "ignore": {
                    "constructorFunc": true,
                    "fallbackFunc": true,
                    "functions": ["foo", "myFunc"],
                    "visibilities": ["private"]
                }
            }
        ]
    }
}
```

Deprecated rules:

Name	Description	Options	Defaults	Fixes
double-quotes	Ensure that string are quoted with double-quotes only. Replaced by "quotes".	•		
no-with	Ensure no use of with statements in the code	•		

1.9 IDE & Editor Integrations

- VS Code: Solidity with Solium linting by Beau Gunderson
- ethereum/emacs-solidity with Solium support by Lefteris Karapetsas
- VS Code: Solidity with Solium linting by Juan Blanco
- VS Code: Solidity with Solium linting by CodeChain.io
- Sublime Solium Gutter by Florian Sey
- Sublime Solium Linter by Alex Step
- Atom Solium Linter by Travis Jacobs
- Syntastic local solium by Brett Sun
- Solium Ale Integration by Jeff Sutherland
- Solium Neomake Integration by Beau Gunderson
- Solium official plugin for Embark Framework

1.10 Migrating to v1.0.0

If you're currently using Solium `v0` and wish to migrate to `v1`, then this section is for you.

Note: If you simply upgrade to Solium v1 right now and lint your project with v0's configuration files, it will work fine (but will give you a deprecation warning) since v1 has been built in a backward-compatible manner. The only 2 exception to this are the discontinuation of `custom-rules-filename` attribute and `--sync` option - these features provided negligible benefit.

1.10.1 What you need to do

Let's say your current `.soliumrc.json` looks like this:

```
{
  "custom-rules-filename": null,
  "rules": {
    "imports-on-top": false,
    "variable-declarations": false,
    "array-declarations": true,
    "operator-whitespace": true,
    "lbrace": true,
    "mixedcase": true,
    "camelcase": true,
    "uppercase": true,
    "no-empty-blocks": true,
    "no-unused-vars": true,
    "quotes": true,
    "indentation": true,
    "whitespace": true,
    "deprecated-suicide": true,
    "pragma-on-top": true
  }
}
```

Please change it to this:

```
{
  "extends": "solium:recommended",
  "rules": {
    "imports-on-top": 0,
    "variable-declarations": 0,
    "indentation": ["error", 4],
    "quotes": ["error", "double"]
  }
}
```

You:

- Only had to specify those rules separately whose behaviour you need to change. Set a rule to `0` or `off` to turn it off. Other values can be `1`/`warning` or `2`/`error`.
- Set up the indentation rule to enforce 4 spaces (replace `4` with any other integer or `tab`).
- Instructed Solium to enforce double quotes for strings (change that to `single` if you so desire).
- Instructed Solium to import all other non-deprecated rules and enable them by default.

Note: Alternatively, you can back up your current `.soliumrc.json` and `.soliumignore` (if you made changes to it), then run `solium init` (after installing v1). You can then make changes to the new `.soliumrc.json`.

A complete list of changes made in `v1` are documented below.

1.10.2 Custom Rule injection is now deprecated

v0 allows you to inject custom rule implementations using the `custom-rules-filename` attribute in your `.soliumrc.json`. This feature is now deprecated. If you specify a file, the linter would simply throw a warning informing you that the custom rules supplied will not be applied while linting.

Custom rule injection has now been replaced by Solium *Plugins*.

1.10.3 Deprecated rules

Following rules have been deprecated:

- `double-quotes` has been replaced by `quotes`.
- `no-with`

1.10.4 soliumrc configuration has a new format

A fully fledged example of v1's `.soliumrc.json` is:

```
{
        "extends": "solium:recommended",
        "plugins": ["consensys", "foobar"],
        "rules": {
                "consensys/race-conditions": "error",
                "consensys/foobar": [1, true, "Hello world"],
                "foobar/baz": 1
        }
}
```

To learn about the new format, please see *Configuring the Linter*.

Note that v1 still accepts the old soliumrc format but throws a format deprecation warning.

1.10.5 Rule implementation has a new format

Note: Unless you're developing rules (whether core or plugins) for Solium, you can skip this part.

The new format of a rule implementation is:

```
module.exports = {
        meta: {
                docs: {
                        recommended: true,
                        type: 'warning',
```

(continues on next page)

(continued from previous page)

```
                        description: 'This is a rule'
                },
                schema: [],
                fixable: 'code'
        },

        create(context) {
                function lintIfStatement(emitted) {
                        context.report({
                                node: emitted.node,
                                fix(fixer) {
                                        // magic
                                }
                        });
                }

                return {
                        IfStatement: lintIfStatement
                };
        }
};
```

See an example on github.

Learn how to develop a Solium rule on the Developer Guide.

1.10.6 Additions in Solium API

There have been additions in the Solium API. However, there are no breaking changes.

- When using the `lint(sourceCode, config)` method (where `config` is your soliumrc configuration), you can now pass an `options` object inside `config` to modify Linter behavior. You can specify the `returnInternalIssues` option whose value is Boolean. If `true`, solium returns internal issues (like deprecation warnings) in the error list. If `false`, the method behaves exactly like in `v0`, and doesn't spit out any warnings (even if, for eg, you're using deprecated rules).

```
const mySourceCode = '...';;
const config = {
        extends: "solium:recommended",
        plugins: ["security"],
        rules: {
                "double-quotes": "error"
        },
        options: {
                returnInternalIssues: true
        }
};

const errors = Solium.lint(mySourceCode, config);
// Now errors list contains a deprecated rule warning since "double-quotes" is␣
↪deprecated.
// If returnInternalIssues were false, we wouldn't receive this warning.
```

- The API now exposes another method `lintAndFix()`. Guess what it does? Please refer to the developer guide on how to use this method to retrieve lint errors as well as the fixed solidity code along with a list of fixes applied.

1.10.7 –sync has been removed

v0's CLI allowed the `--sync` flag so a user could sync their `.soliumrc.json` with the newly added rules after updating solium. sync was not a great design choice and so we've removed it. v1 is designed in a way such that core developers can keep adding more rules to solium and a user doesn't need to do anything apart from installing an update in order to use that rule. It gets applied automatically.

CHAPTER 2

Developer Guide

2.1 Architecture

Solium is organized as a module that exposes an API for any javascript application to use. The user would supply a source code string along with a configuration object that determines what exactly solium does with the code.

`Solium` refers to an engine (a middleman) that accepts user input (source code & configuration) from one side and rule implementations from another. A rule implementation can refer to any piece of code that operates on the given solidity code's Abstract Syntax Tree, points out flaws and suggests fixes.

In a sense, Solium is a generic engine that operates on any given solidity code. The linter itself is a special use case of this engine where the "analyzer" refers to a set of rule implementations that tell whether something in the code looks right or wrong. Because you can write plugins and get access to the complete solidity code, its AST and a solium-exposed set of utility functions to operate on the AST, you can build anything on top of solium that you can imagine!

Solium has a set of core rules for the purpose of linting code.

The frontend of the app is a CLI that a user uses to interact with the Solium module to get things done. The module exposes 2 main functions for usage: `lint()` and `lintAndFix()`.

Architecture will be explained in more detail in future.

2.2 Installation & Setting up the Development Enviroment

Make sure you have Node.js and NPM installed on your system. Install Solium `v1` as a local module using `npm install --save solium`.

You can now use Solium like:

```
const Solium = require('solium'),
        sourceCode = 'contract fOO_bar { string hola = \'hello\'; }';
```

(continues on next page)

(continued from previous page)

```
const errors = Solium.lint(sourceCode, {
        "extends": "solium:recommended",
        "plugins": ["security"],
        "rules": {
                "quotes": ["error", "double"],
                "double-quotes": [2],   // returns a rule deprecation warning
                "pragma-on-top": 1
        },

        "options": { "returnInternalIssues": true }
});

errors.forEach(console.log);
```

- Source Code can be either a string or a buffer object
- `lint()` takes in the source code, followed by the soliumrc configuration object.
- `returnInternalIssues` option tells solium to return internal issues (like rule deprecation) in addition to the lint issues. If this option is `false`, Solium only returns lint issues. It is recommended that you set it to `true`, otherwise you're missing out on a lot ;-)
- `lint()` returns an array of error objects. The function's output looks something like:

```
[
        {
                type: 'warning',
                message: '[DEPRECATED] Rule "double-quotes" is deprecated. Please use
↪"quotes" instead.',
                internal: true,
                line: -1,
                column: -1
        },
        {
                ruleName: 'quotes',
                type: 'error',
                node: { type: 'Literal', value: 'hello', start: 79, end: 86 },
                message: '\'hello\': String Literals must be quoted with double␣
↪quotes only.',
                line: 7,
                column: 15,
                fix: { range: [Array], text: '"hello"' }
        }
]
```

- You can use the `lintAndFix()` function as demonstrated in the following example:

```
const Solium = require('solium'),
        sourceCode = 'contract fOO_bar { string hola = \'hello\'; }';

const result = Solium.lintAndFix(sourceCode, {
        "extends": "solium:recommended",
        "plugins": ["security"],
        "rules": {
                "quotes": ["error", "double"],
                "double-quotes": [2],   // returns a rule deprecation warning
                "pragma-on-top": 1
```

(continues on next page)

(continued from previous page)

```
        },

        "options": { "returnInternalIssues": true }
});

console.log(result);
```

The output of `lintAndFix()` look like:

```
{
        originalSourceCode: 'pragma solidity ^0.4.0;\n\n\nimport "./hello.sol";
↪\n\ncontract Foo {\n\tstring hola = \'hello\';\n}\n',
        fixesApplied:[
                {
                        ruleName: 'quotes',
                        type: 'error',
                        node: [Object],
                        message: '\'hello\': String Literals must be quoted with␣
↪double quotes only.',
                        line: 7,
                        column: 15,
                        fix: [Object]
                }
        ],
        fixedSourceCode: 'pragma solidity ^0.4.0;\n\n\nimport "./hello.sol";
↪\n\ncontract Foo {\n\tstring hola = "hello";\n}\n',
        errorMessages: [
                {
                        type: 'warning',
                        message: '[DEPRECATED] Rule "double-quotes" is deprecated.␣
↪Please use "quotes" instead.',
                        internal: true,
                        line: -1,
                        column: -1 },
                        { ruleName: 'double-quotes',
                        type: 'warning',
                        node: [Object],
                        message: '\'hello\': String Literals must be quoted with
↪"double quotes" only.',
                        line: 7,
                        column: 15
                }
        ]
}
```

Note: The input supplied to `lint()` and `lintAndFix()` is the same. Its the output format that differs.

To work with Solium:

- clone the repository to your local machine using, for eg, `git clone git@github.com:duaraghav8/Solium.git`.
- Move into its directory using `cd Solium`.
- Install all dependencies **and** dev dependencies using `npm install --dev`.
- To ensure that everything works fine, run `npm test`. If you've cloned the `master` branch, there should be no

test failures. If there are, please raise an issue or start a chat on our Gitter channel.

2.3 Writing a Core Rule

To write a core rule for Solium, please start by raising an issue on github describing your proposal. You can check out some of the rules in the roadmap in our Rules Wishlist.

Note: You are allowed (even encouraged) to write any code you wish to contribute in `ES6`.

Say you want to develop a new rule `foo-bar`. Here's how you'd go about it:

2.3.1 Creating a core rule

Create a file `foo-bar.js` inside lib/rules. This is the main implementation of your rule. Use the below template to implement your core rule:

```
module.exports = {
        meta: {
                docs: {
                        recommended: true,
                        type: 'warning',        // 'warning' | 'error' | 'off'
                        description: 'This is my foobar rule'
                },
                schema: [],
                fixable: 'code'
        },

        create(context) {
                function lintIfStatement(emitted) {
                        const { node } = emitted;

                        if (emitted.exit) {
                                return;
                        }

                        context.report({
                                node,
                                fix(fixer) {
                                        // fix logic
                                },
                                message: 'Oh snap! A lint error:('
                        });
                }

                return {
                        IfStatement: lintIfStatement
                };
        }
};
```

Your rule should expose an object that contains 2 attributes - `meta` object which describes the rule and `create()` function that actually lints over the given solidity code.

`meta`

- Contains `docs` object used to describe the rule.
- The `schema` object is used to describe the schema of options the user can pass to this rule via soliumrc config (see AJV). This ensures that a valid set of options are passed to your rule. You can see the schema of quotes rule to understand how to write the schema for your rule.
- The `fixable` attribute can have value as either `code` or `whitespace`. Set this attribute if your rule also contains fixes for the issues you report. Use `whitespace` if your rule only add/removes whitespace from the code. Else use `code`.
- When a rule needs to be deprecated, we can add `deprecated: true` inside meta. We can add `replacedBy: ["RULE NAME"]` inside meta.docs if this rule is to be replaced by a new rule (see deprecated example).

Note: `replacedBy` doesn't force the linter to apply the new rule. Instead, it only throws a warning to the user, notifying them that they're using a deprecated rule and should consider moving to the new rule(s) specified inside `replacedBy` array. Try adding `double-quotes: "error"` inside `rules` inside your `.soliumrc.json` and running the linter.

`create()`

This function is responsible for actual processing of the contract code, determining whether something is wrong or not, reporting an issue and suggesting fixes. create() must return an object whose Key is an AST node type, and value is the function to execute on that node (i.e., the *handler function*). So, for example, `IfStatement` is the type of the AST node representing an `if` clause and block in solidity.

Note: To know which node type you need to capture, install solparse, parse some sample code into AST, then examine the particular node of interest for its `type` field. Specify that type as your return object key. You can see any rule implementation to understand what create()'s return object looks like.

The create() function receives a `context` object, which allows you to access the solidity code to be linted and many other things to help your rule work its magic.

- `context.options` - `undefined` if user doesn't supply any options to your rule through soliumrc. An Array of options otherwise. Solium ensures that the options passed inside the array are fully compliant with the `schema` you define for each of them in `meta`. So if a user specifies `foo-bar: ['error', 'hello', 110, {a: [99]}]`, then `foo-bar` rule's `context.options` contains the array `['hello', 110, {a: [99]}]` (all but the first item, because the first is the severity of the rule). See options example.
- `context.getSourceCode()` - returns a SourceCode object that gives you access to the solidity code and several functions to operate on it and AST nodes.

The functions exposed by SourceCode object are as follows:

1. `getText (node)` - get source code for the specified node. If no arguments given, it returns the complete source code
2. `getTextOnLine (lineNumber)` - get the complete text on the specified line number (lineNumber is an Integer)
3. `getLine (node)` - get the line number on which the specified node's code starts
4. `getEndingLine (node)` - get the line number on which the specified node's code ends
5. `getColumn (node)` - get column no. of the first character of the specified node's code
6. `getEndingColumn (node)` - get column no. of the last character of the specified node's code
7. `getParent (node)` - get the parent node of the specified node

8. `getNextChar (node)` - get 1 character after the code of specified node
9. `getPrevChar (node)` - get 1 character before the code of specified node
10. `getNextChars (node, charCount)` - get charCount no. of characters after the code of specified node
11. `getPrevChars (node, charCount)` - get charCount no. of characters before the code of specified node
12. `isASTNode (arg)` - Returns true if the given argument is a valid (Spider-Monkey compliant) AST Node
13. `getStringBetweenNodes (prevNode, nextNode)` - get the complete code between 2 specified nodes. (The code ranges from prevNode.end (inclusive) to nextNode.start (exclusive))
14. `getLines ()` - get the source code split into lines
15. `getComments()` - get the list of AST nodes representing comments in the code. Call `getSourceCode()` inside your *handler function* if you wish to use this method.

Note: The recommended way to use the `getSourceCode()` method is inside the *handler function* in which you will be calling the functions the SourceCode object provides. If you call `getSourceCode()` inside the main `create()` function, some functions will return empty results because the data hasn't been populated yet. This is by design. If you see some rule implementations calling the function outside of their handler functions, it means that the SourceCode object functions they use are unaffected by whether you call them inside or outside the handler functions.

- `context.report()` - Lastly, the context object provides you with a clean interface to report lint issues:

```
context.report({
        node,   // the AST node retrieved through emitted.node (see below)
        fix(fixer) {    // [OPTIONAL]
                if (wantToApplyFix) {
                        return [fixer.replaceText(node, "hello world!!")];
                }

                return null;
        },
        message: 'Lint issue raised yayy!',
        location: {     // [OPTIONAL]
                line: 9,        // [OPTIONAL]
                column: 20      // [OPTIONAL]
        }
});
```

See report with fix example and report with location example.

Note: If you're supplying the `fix()` function, make sure you specify the `fixable` attribute in `meta`.

Your `fix()` function will receive a `fixer` object that exposes several functions so you can tell Solium **how** to fix the raised lint issue. Every fixer function you call returns a fixer packet. Solium understands how to work with this packet. Your fix function must return either a single fixer packet, an array of fixer packets or `null`.

Note: Returning a `null` results in the particlar fix function being ignored. This is convenient when, under certain conditions, you don't want to apply any fixes. This means that `fix(fixer) { return null; }` is equivalent to not supplying a `fix()` function in the error object at all. See the `context.report()` example above.

Warning: Multiple fixer packets inside the array must not overlap, else Solium throws an error. For eg- the first packet tries to remove the first 10 characters from the solidity code, whereas another packet tries to replace them by, say, "hello world". This results in an overlap and hence the complete fix is not valid. However, if the replacement begins at the 11th character, then there is no conflict and so your fix is valid!

Below is the list of functions exposed by the `fixer` object:

1. `insertTextAfter (node, text)` - inserts text after the given node
2. `insertTextAfterRange (range, text)` - inserts text after the given range
3. `insertTextBefore(node, text)` - inserts text before the given node
4. `insertTextBeforeRange(range, text)` - inserts text before the given range
5. `remove (node)` - removes the given node
6. `removeRange(range)` - removes text in the given range
7. `replaceText(node, text)` - replaces the text in the given node
8. `replaceTextRange(range, text)` - replaces the text in the given range
9. `insertTextAt(index, text)` - inserts text at the given position in the source code

Where `range` is an array of 2 unsigned integers, like `[12, 19]`, `node` is a valid AST node retrieved from `emitted.node` (see below), `text` is a valid string and `index` is an unsigned integer like `69`.

`emitted`

As mentioned earlier, `create()` should return an object. The function specified as the value for a key is responsible for operating over that AST node, so it gets passed an `emitted` object. This object's properties are as follows:

- `emitted.exit` - Solium passes an AST node to a rule twice - once when it enters the node during its Depth-first traversal and second when its leaving it. exit property, if true, means Solium is leaving the node. So if you only want your rule to execute once over a node, you can specify `if(emitted.exit) { return; }`.

Note: A common use case for `exit` is when you want your rule to access the whole contract's AST Node (type `Program`) at the end, ie, when all other rules are done reporting their rules. Then you could specify `if(!emitted.exit) { return; }`.

- `emitted.node` - is the AST Node object of type specified as the key in your return object. So if, for eg, your create() returns `{ ForStatement: inspectForLoop }`, then you can access the AST Node representing the `for` loop in solidity like:

```
create(context) {
        function inspectForLoop(emitted) {
                const {node} = emitted;
                console.log (node.type);        // prints "ForStatement" and the node
↪has appropriate properties of 'for' statement
        }

        return { ForStatement: inspectForLoop };
}
```

See emitted node example

You now have all the required knowledge to develop your core rule `lib/rules/foo-bar.js`. Its now time to write tests.

2.3.2 Testing your Core rule

- Inside the `test/lib/rules`, creating a new directory `foo-bar` and a file inside this directory `foo-bar.js` (see test examples).
- Now paste the below template in `test/lib/rules/foo-bar/foo-bar.js`:

```
/**
 * @fileoverview Description of the rule
 * @author YOUR NAME <your@email>
 */

'use strict';

const Solium = require('../../../../lib/solium'),
        wrappers = require('../../../utils/wrappers');
const { toContract, toFunction } = wrappers;

// Solium should only lint using your rule so only issues flagged by your rule are␣
↪reported
// so you can easily test it. Replace foo-bar with your rule name.
const config = {
        "rules": {
                "foo-bar": "error"      // alternatively - ["error" OR "warning",␣
↪options according to meta.schema of rule]
        }
};

describe('[RULE] foo-bar: Rejections', () => {
        it('should reject some stuff', done => {
                const code = 'contract Blah { function bleh() {} }',
                        errors = Solium.lint(code, config);

                // YOUR TESTS GO HERE. For eg:
                errors.should.be.size(2);       // If you're expecting your rule to␣
↪flag 2 lint issues on the given code.

                Solium.reset();
                done();
        });
});

describe('[RULE] foo-bar: Acceptances', () => {
        it('should accept some stuff', done => {
                // YOUR LINTING & TESTS GO HERE. For eg:

                Solium.reset();
                done();
        });
});
```

You're now ready to write your tests (see shouldjs documentation).

After writing your tests, add an entry for your rule `foo-bar` in solium json. You also need to add your rule's entry to the `List of Style Rules` section in User Guide.

Finally, add an entry for your rule in solium all and solium recommended rulesets: `foo-bar: <SEVERITY>` where severity should be how your rule should be treated by default (as an error or warning). Severity should be same as what you specified in your rule's `meta.docs.type`.

Now run `npm run lint` to let eslint work its magic. Resolve any lint issues you might see in your rule & test files. Run `npm test` and resolve any failures.

Once everything passes and there are no lint issues, you're ready to make a Pull Request :D

Note: ESLint allows us to disable linting on specific pieces of code. This should only be used after a brief discussion about why it's suitable.

Note: Running `npm test` also prints coverage stats at the bottom of the CLI output. It creates the `coverage` directory whose `index.html` can be opened in any browser to view the same. Write enough tests to keep the coverage for the rule above `90%`.

2.4 Developing a Sharable Config

The purpose of a sharable config is for an organisation to just pick up a solidity style spec to work with and focus on the coding part instead of getting into a tabs vs. spaces debate. You install the SC and specify its name without prefix as value of the `extends` key in your soliumrc config. Something like:

```
{
        "extends": "foobar"
}
```

(See full documentation in User Guide)

Sharable configs are distributed as modules via NPM. You are encouraged to include `solium`, `solidity` and `soliumconfig` tags in your `package.json`. Say, you want to call your config `foobar`. Then your module's name must be `solium-config-foobar`. The prefix is mandatory for solium to recognise the module as a sharable config.

Note: For reasons discussed on our blog, we have reserved a few NPM solium config module names. If you find your organisation's name in the list in the blog, please follow the instructions at the bottom of the blog to claim your module.

Start by creating a directory to contain your module

- `mkdir solium-config-foobar`
- `cd solium-config-foobar`
- `npm init` Fill in the appropriate details and don't forget to add the tags mentioned above!
- Create your `index.js` file (or whichever you specified as your entry point file). This file must expose an object like below:

```
module.exports = {
        rules: {
                quotes: ["error", "double"],
                indentation: ["warning", 4],
                "pragma-on-top": 1,
                ...
        }
};
```

- Specify the `peerDependencies` attribute in your `package.json` like:

```
{
        ...
        "peerDependencies": {
                "solium": "^1.0.0"
        }
}
```

Read about Peer Dependencies on NPM. You're now ready to test your config.

2.4.1 Testing your Sharable Config

Solium internally simply `require()` s the config you extend from in your soliumrc. So as long as require() can resolve the name `solium-config-foobar`, it doesn't care where the config is installed.

The simplest way to test is to first link your config and make it globally available. Traverse to your config directory and run `npm link`. You can verify that your config is globally available by going to any random directory, opening a node REPL and running `require('solium-config-foobar')`.

Next, go to your dapp directory that contains the `.soliumrc.json` file. Open this file and set `"extends": "foobar"` (**only the config name, not the prefix**). You can omit the entire `rules` object.

Now run `solium -d contracts/`. The linter should behave according to the severities & rule options provided by you.

That's it! You're now ready to `npm publish` your Sharable Config.

Note: It is a good practice to specify **all** the rules in your sharable config. This ensures that you decided how each rule is to be treated and that you didn't forget about any of them. If you wish to turn a rule off, simply specify its value as `off` or `0`. See list of all rules on User Guide. See example configuration solium all ruleset.

Note: It is good practice to turn off all the deprecated rules. See the Rule List in User Guide to know which rules are now deprecated.

2.5 Developing a Plugin

Plugins allow third party developers to write rule implementations that work with solium and re-distribute them for use. Plugins too are distributed via NPM, have the prefix `solium-plugin-` and should, as a best practice, have the tags `solium`, `solidity` and `soliumplugin`.

As an example, you can check out Solium's official Security Plugin.

Note: For reasons discussed on our blog, we have reserved a few NPM solium plugin module names. If you find your organisation's name in the list in the blog, please follow the instructions at the bottom of the blog to claim your module.

Start by creating a directory to contain your plugin (lets call the plugin `baz`)

- `mkdir solium-plugin-baz`
- `cd solium-plugin-baz`

- `npm init` Fill in the appropriate details and don't forget to add the tags mentioned above
- Specify the `peerDependencies` attribute in your `package.json` like:

```
{
        ...
        "peerDependencies": {
                "solium": "^1.0.0"
        }
}
```

Read about Peer Dependencies on NPM.

- Create your `index.js` file (or whichever you specified as your entry point file). This file must expose an object like below:

```
module.exports = {
        meta: {
                description: 'Plugin description'
        },
        rules: {
                foo: {
                        meta: {
                                docs: {
                                        recommended: true,
                                        type: 'warning',
                                        description: 'Rule description'
                                },
                                schema: []
                        },
                        create(context) {
                                function inspectProgram (emitted) {
                                        if (emitted.exit) {
                                                return;
                                        }
                                        context.report ({
                                                node: emitted.node,
                                                message: 'The rule baz/foo reported␣
↪an error successfully.'
                                        });
                                }
                                return {
                                        Program: inspectProgram
                                };
                        }
                }
        }
};
```

Note: In the above example, you can set the `type` property to `off`. The effect of this is that the rule exists in your plugin but is **disabled by default**. This feature can be used when you require that a user only purposely enable the rule (probaby because it may not be desirable for general audience).

Notice that every rule you define inside the `rules` object has the exact **same schema as the core rule** described above. So if you know how to implement a core rule, you need not learn anything new to implement a plugin rule.

2.5.1 Testing your Plugin

Inside your main plugin directory itself:

- Install solium v1 as a dev dependency using `npm install --save-dev solium`.
- Run `npm install --save-dev mocha chai should` to install the devDependencies for testing purposes.
- In your `package.json`, add the following key:

```
"scripts": {
        "test": "mocha --require should --reporter spec --recursive"
},
```

- Run `npm link` to make this plugin globally available. (You can confirm that it worked by going to any random directory in your system, firing up Nodejs REPL and run `require('solium-plugin-baz')`).
- Write your tests inside the `test/` directory following the below pattern:

```
const Solium = require ('solium');
/**
 * If you require any other modules like lodash, install them.
 * If the module is only being used in your tests, then it should go in your dev
↪dependencies.
 * If being used by any of your rules, then it must go into dependencies.
 */
const config = {
        plugins: ['baz'],
        rules: {
                'baz/foo': 'warning'
        },
        // This returns internal warnings, like deprecation notices
        options: {
                returnInternalIssues: true
        }
};
describe ('Rule foo: Acceptances', () => {
        it ('should accept some stuff and reject other stuff', done => {
                const code = 'contract BlueBerry { function foo () {} }';
                const errors = Solium.lint (code, config);
                // If your rules also contain fix()es you'd like to test, use:
                // var errors = Solium.lintAndFix (code, config);
                console.log ('Errors:\n', errors);
                // Now you can test the error objects returned by Solium.
                // Each item in errors array represents a lint error produced by the
↪plugin's rules foo & bar
                errors.should.be.Array ();
                errors.should.have.size (2);
                // Add further tests to examine the error objects
                // Once your tests have finished, call below functions to safely exit
                Solium.reset ();
                done ();
        });
});
```

Notice that the **schema of plugin rule tests is the same as that of core rule tests**.

- Now run the tests using `npm test` and resolve any failures that occur.

- As another (optional) test, you can also go to your DApp directory and add your plugin's entry in `.soliumrc.json` to see if its working properly:

```
{
        "plugins": ["baz"],
        "rules": {
                "baz/foo": "error"
        }
}
```

And run the linter.

Once all tests pass, you can remove the global link of your plugin using `npm unlink` inside your plugin directory and then `npm publish` it!

See a sample plugin for solium.

2.6 Building this documentation

This documentation is built with Sphinx and written in RST.

- To make changes in it, start by cloning Solium to your workstation with `git clone`.
- `cd` into the `docs/` directory. This dir is responsible for containing all rst files, sphinx confguration and builds.
- Make sure you have all Sphinx dependencies installed (see getting started with readthedocs).

Note: This documentation builds successfully with Sphinx `v1.5` but fails with `v1.6`. Although we haven't yet fully investigated whether its a problem with our docs or Sphinx, we recommend you to install `v1.5` in order to see the changes you've made.

- Make the changes to the docs as you see fit, then run `make html` while still inside `docs/`. If there were no RST errors, the docs should build successfully.
- Open up `docs/_build/html/index.html` in your favourite browser to see the changed.
- Once you're satisfied, you can commit the changes you made in the RST docs and send a PR.

CHAPTER 3

Experimental Features

At any given time, Solium might have a few experimental features in production. They're experimental in order to determine whether and to what extent they're beneficial. And we could really use your feedback on experimental features!

These features haven't been mentioned anywhere in the developer and user guides to ensure that you're aware when you're using an experimental feature in your workflow.

Only this section contains the list of the features. They are subject to change or even removal in subsequent releases. You can either use them temporarily or freeze the Solium version in your app to ensure thay always work for you (not recommended).

As a rule of thumb, **never use the features listed in this section in your production apps**.

3.1 List of Experimental features

3.1.1 v1.0.8

- Intuitive Util methods to aid rule devs determine node types. PR: 149 Trial ends on: **25th Dec '17**

See experimental features issue for discussions on them.

CHAPTER 4

Known Issues

While Solium is being actively maintained, a few major issues are still lurking around and we thought it best to make you aware of them so you don't spend time discovering them instead.

- Solium is currently **file-aware instead of being project-aware**. What this means is that while linting, Solium doesn't have the context of all the contracts and how they may be using the contract currently being linted. A consequence of this is that the linter currently flags a state variable as unused if it doesn't find its usage in the same contract, whereas its clearly possible that you're `import` ing the contract elsewhere to use that variable (See issue). This is a fairly critical problem and will be resolved in a future release. We believe a codebase-aware linter would be much more powerful because of its broader context.
- The linter's internal parser supports Solidity `v0.5`. This means that it supports the calldata storage location specifier, but in a non-backward-compatible manner. If you're currently using Solidity version < 0.5 and have used `calldata` as a name for a variable or function parameter, you might see false lint issues because `calldata` is treated as location and hence, the variable name is seen as `null`. Regardless of whether you use Solium or not, it is a good idea to rename all such variables to keep your code compatible with Solidity 0.5.
- When installing the Linter from the `ethlint` NPM package, you might see the following warning:

```
npm WARN solium-plugin-security@0.1.1 requires a peer of solium@^1.0.0 but none is␣
↪installed. You must install peer dependencies yourself.
```

You can safely ignore this warning.

Solium was recently renamed to Ethlint and the linter is available for download from both `solium` and `ethlint` NPM packages. Ethlint comes shipped with its Security plugin. This plugin checks to ensure whether `solium` NPM package is installed or not.

There is currently no way in NPM to *allow any one of the specified packages to satisfy as peer dependency*, so we can't specify `solium OR ethlint`. We also cannot change `solium` to `ethlint` in `peerDependencies` because its a potential breaking change. See the original issue.

- There is a limitation when using the `solium-enable` comment directive: You cannot disable all rules (using `// solium-disable` for example) and then enable a select few (using `// solium-enable rule1, rule2` for example). The enabling part doesn't work and rules remain disabled even after using the `enable` directive. This is due to how the linter internally represents disabling **all** rules.

In the below example, the `security/no-throw` rule will **not** be enabled on the `throw;` statement, against the expectations.

```
contract Foo {
    // solium-disable
    function b11d() {
        // solium-enable security/no-throw
        throw;
    }
}
```

CHAPTER 5

Contributing

We're constantly looking out for awesome people to join our community and help make Solium a world-class static analyser that keeps production code in check. There are various opportunities for you to contribute to Solium, regardless of whether you're new to the project or deeply familiar with it.

A few areas where we could use some help are:

- All the issues on our repository tagged with `Contributors needed` and `Help wanted`. These include adding new rules, moving the codebase to ES6, fixing some of the existing rules (like whitespace or indentation).
- If you're, by now, pretty familiar with Solium's codebase, you could also fix the un-tagged issues.
- Feedback & Suggestions - always welcome. The author of solium sucks at User experience and would love to hear about what pain points still exist in your smart contract development workflow (regardless of whether they're directly related to solium or not).
- This documentation! Yep, we could certainly use more eyeballs that can correct typos, paraphrase instructions, introduce diagrams or simply make the docs much cleaner.

CHAPTER 6

About

Solium was authored by Raghav Dua in 2016.

It borrows ideas from ESLint, Solidity Parser and other such ambitious projects.

The linter was initially designed to strictly follow Solidity's official Style Guide, but has since evolved into a completely customizable tool focused on style and security of smart contract code (read our v1 release blog).

6.1 Community

Anyone who has contributed to strengthening this Project is a community member.

- The Ethereum Foundation (see Ethereum Inaugral Grants announcement)
- The Augur Project (see Augur Bounties)
- Beau Gunderson
- Nicolas Feignon
- Simon Hajjar
- Mitchell Van Der Hoeff
- Jack Peterson
- Joseph Krug
- Micah Zoltu
- Tom Kysar
- Artem Litchmanov
- Michelle Pokrass
- Tristan H
- Federico Bond

- Elena Dimitrova
- Christopher Gewecke
- Ulrich Petri
- Leo Arias
- Alex Chapman
- Chih Cheng Liang
- Jooraj Bednar
- Juan Blanco
- Florian Sey
- Alex Step
- Travis Jacobs
- Remco Bloemen
- Brett Sun
- Franco Victorio
- Gabriel Alacchi
- Utkarsh Gupta
- Ivan Mushketyk
- Bernd Bohmeier
- Donatas Stundys

Index