

Google Summer of Code 2018
Improving the development workflow
(Oppia)

Personal Details:

Name: Apurv Bajaj

Email: apurvabajaj007@gmail.com

GitHub Handle: [apb7](#)

University: Birla Institute of Technology and Science, Pilani, Pilani Campus

Program: B.E. in Computer Science (Dual Degree)

I am currently in the second year of my degree in Computer Science at BITS Pilani, Pilani Campus. BITS Pilani is one of the premier engineering institutes in India. I have a cumulative GPA of 9.83 out of 10 (Top 1% in a batch of ~950 students). I am also a merit scholar at my university. I have been working with Python for the last two years. I am a backend developer and work around primarily with Django and Node.js. I have been involved in web development since the past year.

Project Details:

Project Name: Improving the development workflow ([Link](#)).

Why I am interested in working with Oppia:

Oppia's mission is "to help anyone learn anything they want in an effective and enjoyable way". I want to contribute to this mission in my own small way and work for a better educational scenario where each student has access to quality education, across the world. I find the explorations very interactive in a way that they stimulate the natural thinking process which most schools lack today.

Also, the team at Oppia is very friendly. I have been warmly helped at each step of my contributions, whether be it with the code, the release testing or the audio translations for the RCT. I have tried to pick up this "always helping" attitude since I became a member of the organization. The workflow is completely dynamic and I always get to learn new skills. The best part for all first-time contributors would surely be getting a personal mail from Sean and the Oppia team before they begin their journey!

What interests me about this project and why it is worth doing:

The project, improving the development workflow, as the name suggests, emphasizes on the following aspects:

- Reducing the reviewer time by automating lint checks and tests for developers and code reviewers.

- Catching errors with the help of non-flaky and thorough end-to-end tests before they end up in production as well as incorporating multiple browsers and mobile viewports.
- Automating routine processes like CLA checking, closing stale PRs and helping resolve merge conflicts and build failures by notifying the maintainer-on-duty and the pull request authors.

This project needs a variety of skills, namely, proficiency in Python, bash scripting as well as fair amount of Javascript knowledge. The project also requires the creation of a bot for automating routine tasks. Since the bot would be hosted on a server, it would require decent knowledge about server functioning. Therefore, I feel that I would get to learn a lot of things while working on this project. It would be challenging but at the same time, attainable for me. Also, having participated in a few code reviews, I understand the importance of the above mentioned points.

This project, when completed, would produce a noticeable difference in the review time and would ease the lives of all members of the organization, including the maintainers. It would also help in maintaining a standard coding style for Oppia which would be followed by everyone, including the first-time contributors.

Prior Experience:

I have worked on the following projects:

- Developed a simulation of the stock market for an event in the technical fest of BITS Pilani, APOGEE 2018. The simulation used Django as the backend framework and was hosted on Pythonanywhere.
- Worked as a backend developer for a startup, Qbox which ventures into competitive blogging. I implemented the backend using Node.js for their website.
- Wrote solutions to standard problems on Kaggle using the python scikit-learn library.
- Wrote python scripts to solve Project Euler problems. I am currently on level 2.
- Created a bot-enabled Tic-Tac-Toe GUI. The bot worked on the minimax algorithm and also utilised alpha-beta pruning.
- Headed a Special Interest Group for backend development at my college. I conducted a lecture explaining the basics of requests and APIs.

Source code for all my projects can be found at my [GitHub profile](#).

Links to important PRs:

I have been contributing to Oppia since the past 5 months and have had a number of pull requests (24 PRs) merged into the codebase. The most important ones include:

1. Implement isort: PR [#4343](#).
2. Add check for trailing white spaces in HTML: PR [#4704](#).
3. Add eqeqeq rule: PR [#4573](#).
4. Add space checks for definitions in Python: PR [#4522](#).
5. Add check for keyword arguments in Python: PR [#4752](#).

All PRs authored by me can be found here: [Merged](#) and [Open](#). I have also opened the following issues: Issue [#4429](#), Issue [#4450](#) and Issue [#4835](#).

Project Plan and implementation strategy:

Milestones:

Milestone 1 (May 14 - June 11):

Complete all Python, Javascript, CSS and HTML related lint checks by the end of this milestone. This also includes the lint checks for AngularJS.

1.0: (Start during bonding period: April 23 - May 14) Finalize the technical document containing all the rules, extensions and custom checks which are likely to be useful in case of code reviews.

1.1: Implement the Python lint checks including both Pylint and Pycodestyle rules, extensions and custom checks. Also, perform fixes in the codebase as and when the lint checks are enabled. (~ 1 week: May 15 - May 22)

1.1.1: Cover all Pycodestyle rules.

1.1.2: Implement all in-built Pylint rules.

1.1.3: Write custom rules using Pylint and the corresponding tests.

1.2: Implement Javascript and AngularJS checks. Also, perform fixes in the codebase as and when the lint checks are enabled. (~ 10 days: May 23 - June 2)

1.2.1: Enable all in-built eslint rules.

1.2.2: Enable all in-built eslint/angular plugin rules.

1.2.3: Write and implement custom rules. (This has been discussed later in the doc under the [Custom Rules](#) section.)

1.3: Implement CSS and HTML related lint checks. Also, perform fixes in the codebase as and when the lint checks are enabled. (~ 10 days: June 3 - June 12)

a. CSS lint checks:

1.3.1: Separate the CSS for different HTML pages from oppia.css and structure it under various related folders (Discussed in PR [#4654](#)).

1.3.2: Enable Stylelint for all CSS files (Implemented in PR [#4643](#)).

1.3.3: Employ the [HTML processor](#) so that it can lint CSS within our HTML files. (Discussed in Issue [#1977](#)).

b. HTML lint checks:

1.3.4: Enable htmlhint with the selected rules.

June 11 - June 15: Buffer time for Milestone 1

Milestone 2 (June 15 - July 9):

This milestone comprises of documenting, extending and organizing end-to-end tests.

2.1: Fix the flakiness occurring in end-to-end tests, focussing primarily on stateEditor.js and editorAndPlayer.js. (See issue [#4044](#)). (~ 2 weeks: June 15 - June 29)

2.2: Extend the end-to-end tests for Firefox as well as mobile viewports: (~ 1 week: June 29 - July 6)

2.1.1: Extend the tests to work on Firefox version 47. Version 47 is preferred over other versions (Please see this [answer](#) on StackOverflow and the [browser support](#) documentation for Protractor).

2.1.2: Extend the tests for Android/Chrome.

2.1.3: Extend the tests for iOS/Safari.

2.3: Document the process of writing new tests and specify the components which should be emphasized in case of new tests. (~ 3 days: July 6 - July 8)

2.4: Organize (Please see [PR #4896 \[comment\]](#)) and structure the tests so that the end-to-end tests can be easily extended by developers. (~ 4 days: July 8 - July 11)

July 9 - July 13: Buffer time for Milestone 2

Milestone 3 (July 13 - August 6):

Implement our Oppia-bot to the main repository. I plan to do most of the work pertaining to this milestone during the pre-GSoC time so that there is no rush in the last few days.

3.0: (Start during bonding period: April 23 - May 14) Finalize the response of the bot for each particular action. Here I will be focusing on a few particular questions:

- a. When should the bot respond, that is, which actions will trigger the bot?
- b. What should be the response message/comment of the bot in different situations?

3.1: Design the Oppia-bot. (~ 12 days: July 13 - July 24)

3.2: Test the bot intensively and ensure its comments matches the action of the user: This can be done using the [Jest](#) framework. The probot documentation explains [writing tests](#) using an example (Please refer to the [tests section](#)). (~ 1 week: July 24 - July 31)

3.3: Create a manual of the Oppia-bot for the developers with the following sections: (~ 2 days: July 31 - August 2)

- Response of the bot to various actions.
- Instructions for deployment on Heroku.

- Adding plugins to the existing bot.
- FAQ section.

3.4: Install the Oppia-bot as a GitHub app on the main repository, “oppia”. (~ 1 day: August 3)

3.5: Fix any problems/errors, if any. (~ 6 days: August 3 -August 9)

3.6: Install it on other active repositories as well, such as, “oppia-ml”, if time permits.

3.7: Implement JSON lint checks , if time permits.

August 6 - August 14: Buffer time for Milestone 3

Technical Design and Implementation:

1. Lint Checks:

I will be dividing this into 4 sections, one each for Python, Javascript, CSS and HTML and a section for JSON (which will be implemented if time permits).

a. Python :

Currently in use: Pylint and Pycodestyle

Details:

We have almost exploited all of the apt rules provided by Pylint except for few rules, extensions and custom checkers (discussed later). Pycodestyle had been added relatively recently to the repository in PR [#4522](#).

I'll first cover up the in-built Pycodestyle rules which we can use and then move on to custom checkers using Pylint, covering the in-built Pylint rules towards the end.

• Pycodestyle:

This module is currently being used in [_check_spacing](#) in the pre-commit linter. The enabled pycodestyle rules are specified in tox.ini. Currently enabled rules are E231 (missing whitespace after ',', ';', or ':'), E301 (expected 1 blank line, found 0), E302 (expected 2 blank lines, found 0) and E305 (expected 2 blank lines after end of function or class).

The complete set of rules can be found [here](#).

The ones we can implement are:

(The rules have been assigned a priority based on their occurrence in pull requests as observed during code reviews).

- High
- Medium
- Low

Code	Sample Message	Priority (as observed during code reviews)
<i>E1</i>	<i>Indentation</i>	
E101	indentation contains mixed spaces and tabs	Medium
E111	indentation is not a multiple of four	Medium
E112	expected an indented block	High
E113	unexpected indentation	High
E114	indentation is not a multiple of four (comment)	Medium
E115	expected an indented block (comment)	High
E116	unexpected indentation (comment)	High
E121	continuation line under-indented for hanging indent	High
E122	continuation line missing indentation or outdented	High
E126	continuation line over-indented for hanging indent	High
E131	continuation line unaligned for hanging indent	High
E133	closing bracket is missing indentation	High
<i>E2</i>	<i>Whitespace</i>	
E201	whitespace after '('	High
E202	whitespace before ')'	High
E203	whitespace before ':'	High
E211	whitespace before '('	High

E221	multiple spaces before operator	High
E222	multiple spaces after operator	High
E223	tab before operator	Medium
E224	tab after operator	Medium
E225	missing whitespace around operator	High
E226	missing whitespace around arithmetic operator	High
E227	missing whitespace around bitwise or shift operator	Medium
E228	missing whitespace around modulo operator	High
E231	<i>Already in use</i>	
E241	multiple spaces after ','	High
E242	tab after ','	Medium
E251	unexpected spaces around keyword / parameter equals	High
E266	too many leading '#' for block comment	Medium
E271	multiple spaces after keyword	High
E272	multiple spaces before keyword	High
E273	tab after keyword	Medium
E274	tab before keyword	Medium
E275	missing whitespace after keyword	High
<i>E3</i>	<i>Blank Line</i>	
E301	<i>Already in use</i>	
E302	<i>Already in use</i>	
E303	too many blank lines (3)	Low
E304	blank lines found after function decorator	Low

E305	<i>Already in use</i>	
E306	expected 1 blank line before a nested definition	Low
<i>E7</i>	<i>Statement</i>	
E701	multiple statements on one line (colon)	Low
E702	multiple statements on one line (semicolon)	Low
E703	statement ends with a semicolon	Low
E714	test for object identity should be 'is not'	High; See here .
E722	do not use bare except, specify exception instead	Medium
<i>W2</i>	<i>Whitespace warning</i>	
W292	no newline at end of file	High
W293	blank line contains whitespace	High
W3	Blank line warning	High
W391	blank line at end of file	High

The above table states all the rules we need to implement using pycodestyle.

Another thing is that we can do away with `_check_newline_character` function for all types of files since there already exists some rule or the other to check for newline at end of file. (W292 in Pycodestyle for python files and [eol-last](#) in Eslint for javascript files). This will increase the efficiency and reduce the time taken to check files.

```
def _check_newline_character(all_files):
    """This function is used to check that each file
    ends with a single newline character.
```


.....

- Pylint:

We are currently using Pylint version [1.7.1](#) which was released on 17/04/2017. The latest version is [2.0.0](#). The 2.0 is a major release and therefore here is what I plan to do:

1. Update the Pylint version to 2.0.0
2. Ensure proper migration and working of the previous rules in the 2.0.0 version.
3. Implement other rules, enable new extensions and write custom checkers wherever necessary.

Steps 1 and 2:

Updating to version 2.0.0 will not be much of a problem. We need to change the version in `install_third_party.sh`, something like this:

```
echo Checking if pylint is installed in $TOOLS_DIR/pip_packages
if [ ! -d "$TOOLS_DIR/pylint-2.0.0" ]; then
  echo Installing Pylint

  pip install pylint==2.0.0 --target="$TOOLS_DIR/pylint-2.0.0"
  # Add __init__.py file so that pylint dependency backports are resolved
  # correctly.
  touch $TOOLS_DIR/pylint-2.0.0/backports/__init__.py
fi
```

This will take care of the updation part. We can check the rules by running them over the complete files.

Step 3:

In-built Rules:

All in-built rules can be found out [here](#).

Some specific rules are mentioned in `.pylintrc`:

```
# TODO(sll): Consider re-enabling the following checks:
# abstract-method
# arguments-differ
# broad-except
# duplicate-code
# fixme
# missing-docstring
```

```

# no-member
# no-self-use
# redefined-variable-type
# too-many-arguments
# too-many-boolean-expressions
# too-many-branches
# too-many-instance-attributes
# too-many-lines
# too-many-locals
# too-many-public-methods
# too-many-statements
# and fix those issues.

```

These rules need to be considered.

Code	Name	Explanation	Priority (as observed during code reviews)
W0223	abstract-method	Used when an abstract method (i.e. one that raises <code>NotImplementedError</code>) is not overridden in concrete class.	Low
W0221	arguments-differ	Used when a method has a different number of arguments than in the implemented interface or in an overridden method.	Low
W0703	broad-except	Used when an except catches <code>Exception</code> instances. (Same as E722 of Pycodestyle)	Medium
R0801	duplicate-code	Used when same lines of code are repeated. (There is some problem with this rule. See here .)	Low
C0111	missing-docstring	Used when a module, function, class or method has no docstring. Some special methods like <code>__init__()</code> don't require a docstring and for those, this message is not raised if they have no docstring. <u>Note</u> : This rule will be enabled once we resolve Issue #4374 completely.	High

E1101	no-member	Used when an object (variable, function, ...) is accessed for a non-existent member.	Medium
R0201	no-self-use	Used when there is no reference to the class, suggesting that the method could be used as a static function instead.	High; This might turn out to be a good check.
R0204	redefined-variable-type	Used when the type of a variable changes inside a method or a function.	Medium; Again a good check.
R0914	too-many-locals	Used when a method or function uses more than 15 variables in the namespace.	Low
C0302	too-many-lines	Used when a module has more lines than the limit specified in the max-module-lines option.	Low; Not preferred

I will be picking up the High and Medium priority checks from the above table.

Pylint Extensions:

Optional Pylint checkers are documented [here](#). This is the complete list of the extensions which will be in place after this project is completed:

1. [Parameter Documentation Checker](#):

Need: See PR [#4604](#) where the author has missed the “Raises” part of the docstring and PR [#4605](#) where the author had initially missed “Returns”.

Priority: High

This checker verifies that all function, method, and constructor docstrings include documentation of the:

- parameters and their types
- return value and its type
- exceptions raised

and can handle docstrings in

- Sphinx Style
- Google Style
- Numpy Style

Since our docstrings follow Google style, I will be choosing that style.

2. [Docstyle-Checker](#):

This checker is already in use. See PR [#4572](#) for implementation.

Need: See PR [#4458](#) where a blank line has been wrongly placed.

Priority: High

This checker checks for two things: first, that each docstring ends in triple quotes and second, that there is no blank line at the start of each docstring.

Custom Checkers:

We need to design a few custom checkers for certain issues which might spring up during code reviews. We also need to write certain tests for these to ensure that they work properly. These tests need to be automatically detected by the bash script written for backend tests, *run_backend_tests.sh*.

We will shape the tests similar to the one in PR [#4752](#).

Issue	Solution	Tests	Priority	Status
Keyword args should be explicitly named in calling functions.	A custom check using astroid, pylint.checkers and pylint.interfaces. (I have already implemented this. See PR #4752)	Completed	High	Completed
Args parts of docstrings should match the actual arg names passed to the method/function.	A custom check using astroid, pylint.checkers and pylint.interfaces.	Required	High	
Args, Returns and Raises parts of docstrings should include type information.	A custom check using astroid, pylint.checkers and pylint.interfaces. This might be accomplished using pylint.extensions.docparams (Stated above under Parameter Documentation Checker)	Required	Low (if the inbuilt extension works else High)	
If something within parens extends across multiple lines, break after the opening '('.	A custom check using astroid, pylint.checkers and pylint.interfaces.	Required	High	

Do not import classes directly. Use modules instead to refer to the required classes.	A custom check using astroid, pylint.checkers and pylint.interfaces. (Please see this discussion in PR #4752)	Required	Medium	
---	---	----------	--------	--

b. Javascript (including checks for AngularJS):

For Javascript, I had collaboratively worked with Sandeep. We had created a plan of action and divided this task into four sub-milestones.

b.0 Upgrade Eslint from version 3.18.0 to version 4.18.2:

We currently use Eslint 3.18.0. This version is behind the current (4.x) by one major release, but uses the same rule format. Rules written for this version will not require changes if updated to 4.x in the future.

b.1 Enable built-in rules provided by Eslint.

b.2 Enable built-in rules provided by eslint-plugin-angular (and also eslint-plugin-html).

b.3 Write custom rules for some checks.

1. AngularJS:

<u>Issue</u>	<u>Solution</u>	<u>Status</u>
Missing semicolons at end of lines	Enable built-in rule: semi	Done #4576
Use of '==' rather than '==='	Enable built-in rule: equeqeq	Done PR #4573
Spurious console.log() statements	Enable built-in rule: no-console	Done #4564
Directives should have an explicit scope key and it should not be scope: true since this leads to hard-to-maintain direct-from-parent imports	Custom rule.	
All directives should have restrict: 'E'	Already available with eslint-plugin-angular directive-restrict	

For function args within parens, indent follow-on lines by 2 additional spaces	Enable built-in rule: Indent with FunctionExpression (This rule becomes more strict in V4.0.0 and might cause errors: see indent-legacy in that case.)	Done PR #4588
Always use templateUrl instead of template	Eslint-plugin-angular: no-inline-template	
Align line breaks of angular dependencies with those in the stringified list just below them, and check that the dependencies match exactly	Eslint-plugin-angular: di	
Unused Angular dependencies injected into a controller	Eslint-plugin-angular: di-unused	
If something within parens extends across multiple lines, break after the opening '('	Enable built-in rule: function-paren-newline This rule was introduced from 4.6.0 and is not available for 3.18.0	
Injected dependencies should be sorted alphabetically	Eslint-plugin-angular: di-ordered	
Use '\$log' instead of console methods	Eslint-plugin-angular: log	
All of the file names should match the angular component name	Eslint-plugin-angular: file-name (need some modification)	

2. Javascript:

enforce consistent spacing inside array brackets	Enable built-in rule: array-bracket-spacing	
enforce consistent brace style for blocks	Enable built-in rule: brace-style	

require or disallow newline at the end of files	Enable built-in rule: eol-last	
enforce consistent spacing between keys and values in object literal properties	Enable built-in rule: key-spacing	
enforce consistent spacing before and after keywords	Enable built-in rule: keyword-spacing	
disallow multiple empty lines	Enable built-in rule: no-multiple-empty-lines	
disallow all tabs	Enable built-in rule: no-tabs	
disallow trailing whitespace at the end of lines	Enable built-in rule: no-trailing-spaces	
disallow whitespace before properties	Enable built-in rule: no-whitespace-before-property	
enforce the consistent use of either backticks, double, or single quotes	Enable built-in rule: quotes	
require quotes around object literal property names	Enable built-in rule: quote-props	
enforce consistent spacing after the // or /* in a comment	Enable built-in rule: spaced-comment	
require object keys to be sorted	Enable built-in rule: sort-keys	
Indent the continuation line by two spaces	Custom rule (Please see this discussion in PR #4820)	

Custom Rules:

As of now, we have two cases where we need custom rules. Both are related to AngularJS. So I think it would be better to write these rules using `eslint-plugin-angular` framework that way it would be simple to extend to any rules we might need because

eslint-plugin-angular is an open source project and provides [detailed documentation](#) for writing such rules.

Let's take an example of [directive-name](#) rule, which says that all directives should have a name starting with the parameter defined in the config object:

```
/**
 * require and specify a prefix for all directive names
 *
 * All your directives should have a name starting with the parameter
 you can define in your config object.
 * The second parameter can be a Regexp wrapped in quotes.
 * You can not prefix your directives by "ng" (reserved keyword for
 AngularJS directives) ("directive-name": [2, "ng"])
 *
 * @styleguideReference {johnpapa} `y073` Provide a Unique Directive
 Prefix
 * @styleguideReference {johnpapa} `y126` Directive Component Names
 * @version 0.1.0
 * @category naming
 * @sinceAngularVersion 1.x
 */
'use strict';

var utils = require('./utils/utils');

module.exports = {
  meta: {
    docs: {
      url:
'https://github.com/Gillespie59/eslint-plugin-angular/blob/master/doc
s/rules/directive-name.md'
    },
    schema: [{
      type: ['string', 'object']
    }]
  },
  create: function(context) {
    if (context.settings.angular === 2) {
```



```

        return {};
    }

    return {
        CallExpression: function(node) {
            var prefix = context.options[0];
            var convertedPrefix; // convert string from JSON
.eslintrc to regex

            if (prefix === undefined) {
                return;
            }

            convertedPrefix = utils.convertPrefixToRegex(prefix);

            if (utils.isAngularDirectiveDeclaration(node)) {
                var name = node.arguments[0].value;

                if (name !== undefined && name.indexOf('ng') ===
0) {
                    context.report(node, 'The {{directive}}
directive should not start with "ng". This is reserved for AngularJS
directives', {
                        directive: name
                    });
                } else if (name !== undefined &&
!convertedPrefix.test(name)) {
                    if (typeof prefix === 'string' &&
!utils.isStringRegexp(prefix)) {
                        context.report(node, 'The {{directive}}
directive should be prefixed by {{prefix}}', {
                            directive: name,
                            prefix: prefix
                        });
                    } else {
                        context.report(node, 'The {{directive}}
directive should follow this pattern: {{prefix}}', {

```

```
directive: name,  
  prefix: prefix.toString()  
});  
}  
}  
}  
};  
}  
};
```

All the custom rules will be built around these lines. Each rule will use the utilities provided by the context object.

c. CSS and HTML:

I had already prepared a [design document](#) for CSS linting (Issue [#1977](#)). The design doc is complete in itself and has been reviewed by Sean. I'll be following the exact document with the given milestones.

c.1 CSS:

We will be using [Stylelint](#) for the process (discussed in the issue thread).

As suggested by Sean, we should “bite off small pieces at a time, instead of a big one!”, we'll divide the process into three sub-milestones:

c.1.0:

Lay down the rules for CSS linting which need to be followed by all developers. These rules need to be compliant with the current CSS files.

(Discussed in detail later on in the document)

c.1.1:

Separate selectors used in different pages from oppia.css. (Please see PR [#4654](#) for the approach to be followed here).

c.1.2:

Setup Stylelint and implement the laid out rules only for the main css file, that is, oppia.css for now.

Steps involved:

1. Install stylelint by using the stylelint cli process in third_party.sh.
2. Create a .stylelintrc file (at the project level) with the decided rules.
3. Design a function: _lint_css_files in pre_commit_linter.py.

These are the broad steps involved in this milestone.

c.1.3:

- Expand the lint check all other files as well.
- Take care of the threading issues (if any).
- Employ the [HTML processor](#) following the steps as given [here](#).

Rules for CSS linting:

In Stylelint, all rules are turned off by default. We'll have to set them up manually according to our requirements.

Stylelint also provides us with a set of recommended as well as standard rules which we can extend using our .stylelintrc file:

Recommended Rules:

<u>Rule</u>	<u>Explanation</u>	<u>Priority</u>
at-rule-no-unknown	Disallow unknown at-rules.	High
block-no-empty	Disallow empty blocks.	High
color-no-invalid-hex	Disallow hex colors.	High
comment-no-empty	Disallow empty comments.	High
declaration-block-no-duplicate-properties	Disallow duplicate properties within declaration blocks.	High
declaration-block-no-shorthand-properties-overrides	Disallow shorthand properties that override related longhand properties within declaration blocks.	Medium
font-family-no-duplicate-names	Disallow duplicate font family names.	High

font-family-no-missing-generic-family-keyword	Disallow missing generic families in lists of font family names.	High
function-calc-no-unspaced-operator	Disallow an unspaced operator within calc functions.	High
function-linear-gradient-no-nonstandard-direction	Disallow direction values in linear-gradient() calls that are not valid according to the standard syntax .	Medium
keyframe-declaration-no-important	Disallow !important within keyframe declarations.	High
no-descending-specificity	Disallow selectors of lower specificity from coming after overriding selectors of higher specificity.	High
no-duplicate-at-import-rules	Disallow duplicate @import rules within a stylesheet.	Medium
no-duplicate-selectors	Disallow duplicate selectors.	High
no-empty-source	Disallow empty sources.	High
no-extra-semicolons	Disallow extra semicolons.	High
no-invalid-double-slash-comments	Disallow double-slash comments (//...) which are not supported by CSS.	Medium
property-no-unknown	Disallow unknown properties.	High
selector-pseudo-class-no-unknown	Disallow unknown pseudo-class selectors.	High
selector-pseudo-element-no-unknown	Disallow unknown pseudo-element selectors.	High
selector-type-no-unknown	Disallow unknown type selectors.	High

string-no-newline	Disallow (unescape) newlines in strings.	High
Custom rule	For CSS in directive HTML files, each CSS selector should be prefixed by the name of the directive, in order to make sure that the CSS selector is scoped correctly and doesn't affect other parts of the codebase.	High
unit-no-unknown	Disallow unknown units.	Medium

These can be found [here](#).

Standard Rules:

These rules are build upon the recommended rules and are available [here](#).

I suggest, we should go by the recommended rules, make necessary changes to the rules, add more rules, turn off unpreferred rules, so that we have more flexibility.

Also, there is an interesting website, which can help us understand which rules refer to which CSS element: [CSS Vocabulary](#).

Status:

Completed: c.1.0

I've setup Stylelint locally and enabled the recommended configuration. Our main css file: oppia.css produces the following errors:

1. **No-descending-specificity:**

- Major cause of errors with recommended settings.
- This rule disallows selectors of lower specificity from coming after overriding selectors of higher specificity.
- Currently, we have around 1073 such errors to be precise.
- Fix all the errors first and then enable the rule.

(I am attaching the error report pertaining to this error for reference)[[Link](#)]

2. Selector-type-no-unknown:

- Enable this rule with ignore: ["custom-elements"]
 - Reasons:
 - We currently use a number of custom type-selectors: md-input-group, md-card, md-content, oppia-parameter, oppia-expression-error-tag, create-activity-button, uib-accordion, md-chip, md-chips and profile-link-image. If we chose not to ignore custom templates, "Unexpected unknown type selector" error is produced. Also, we can do away with "default-namespace" in ignore.

3. No-duplicate-selectors:

- Does not allow duplication of a selector within a stylesheet.
- Caused due to repetition of navbar-nav, oppia-delete-interaction-button, md-card.oppia-dashboard-title, oppia-clickable-navbar-element:hover.

Sample Fix:

Current code:

```
.oppia-delete-interaction-button,  
.oppia-close-popover-button {  
  background: none;  
  border: 0;  
  color: #000;  
  cursor: pointer;  
  height: 30px;  
  opacity: 0.5;  
  width: 30px;  
}  
.oppia-delete-interaction-button,  
.oppia-close-popover-button {  
  position: absolute;  
}  
.oppia-delete-interaction-button,  
.oppia-close-popover-button {
```

```
right: 8px;  
top: 8px;  
}
```

Proposed Fix:

```
.oppia-delete-interaction-button,  
.oppia-close-popover-button{  
  background: none;  
  border: 0;  
  color: #000;  
  cursor: pointer;  
  height: 30px;  
  opacity: 0.5;  
  width: 30px;  
  position: absolute;  
  right: 8px;  
  top: 8px;  
}
```

Current code:

```
.oppia-clickable-navbar-element: hover {  
  display: inline-block;  
}
```

```
.oppia-clickable-navbar-element: hover {  
  display: block;  
}
```

Proposed code:

```
.oppia-clickable-navbar-element: hover {  
  display: block;  
}
```

4. [Declaration-block-no-shorthand-property-overrides](#):

- Fixed
- Changed order of “padding” and “padding-right”

5. [Function-calc-no-unspaced-operator](#);

- Fixed
- Added space before and after “/” operator

6. [Font-family-no-missing-generic-family-keyword](#):

- Fixed
- Added generic-family name (sans-serif)

(The end result after milestone c.1.2 will be similar in structure to PR [#4643](#)).

c.2 [HTML](#):

I plan to use [htmlhint](#) for linting HTML. Since htmlhint does not provide any built-in interface, I will be using [htmlhint-cli](#) to work with it.

This process will involve the following steps:

Step 1:

Create a .htmlintrc file as specified [here](#).

Step 2:

Decide the rules which need to be implemented. The complete set of rules can be found [here](#). The following table lists the set of rules I will be implementing here:

Rule	Explanation/Solution	Priority
attr-name-style	Attribute names must conform to the “dash” naming style format.	High
attr-no-dup	The same attribute cannot be repeated within a single tag.	High
attr-validate	Attributes in a tag must be well formed.	High
indent-style	Set to “spaces”; Only spaces should be used for indentation.	High
indent-width	Spaces used to indent must in multiples of the set number. Set to: 2	High
line-max-len	The length of each line must not exceed the set number. Set to:	High

	80 to be in conformity with the codebase.	
tag-close	All opening tags must be closed.	High
tag-name-lowercase	Tag names must be lowercase.	Medium
tag-name-match	Tag names must match (including the case).	Medium
title-no-dup	The title tag should not be repeated within the head.	Medium
All directive files need to end with <code>_directive.html</code>	Custom rule. The approach to be followed here will be to write a function in <code>pre_commit_linter.py</code> which scans the JS files, extract the HTML directive files from them and then check their names.	High

d. JSON:

To implement JSON lint checks, I plan to use the [jsonlint](#) npm package and the [jsonlint-cli](#). The rules will be placed in a `.jsonlintrc` file:

```
{
  "validate": "", // a JSON schema to use for validation
  "ignore": ["node_modules/**/*"], // glob patterns to ignore
  "indent": "", // indent to use for pretty-printed output
  "pretty": true // pretty-print formatted json if quiet is false
}
```

(This is a sample configuration file. Since we do not require a json schema, we can leave that option out. For indent, we can specify the indent according to our requirements. Also, the pretty option does not modify the file. It “pretty” prints the output to STDOUT. If quiet is set to true, no output is printed to STDOUT).

2. Documenting, Extending and Organizing End-to-End Tests:

Here is what I plan to do:

1. Extension:

We need to extend the e2e tests to Firefox and mobile viewports. We currently use Selenium 2.53.2, Chromedriver 2.36 and Geckodriver 0.20.0. The protractor

documentation provides a [browser setup](#) guide for several browsers including Chrome, Firefox and mobile browsers.

Since the [browser support](#) guide recommends using Firefox version 47 for testing with Protractor, we will be using the same version. To accommodate the Firefox browser along with Chrome, we will have to modify **multiCapabilities** in **protractor.conf.js** (as stated in the [documentation](#)):

```
multiCapabilities: [{
  browserName: 'chrome',
  chromeOptions: {
    args: ['lang=en-EN'],
    prefs: {
      intl: {
        accept_languages: 'en-EN'
      }
    }
  },
  loggingPrefs: {
    driver: 'INFO',
    browser: 'INFO'
  }
}, {
  browserName: 'firefox',
}],
```

For mobile viewports, there exists a [mobile setup](#) guide. Here I plan to go for Appium - [Android/Chrome](#). We can also extend all the existing tests further to [iOS/Safari](#). The Appium - Android/Chrome documentation provides a step-by-step process to setup protractor. To summarize, we will add another capability object to the multicapabilities array:

```
capabilities: {
  browserName: 'chrome',
  platformName: 'Android',
  platformVersion: '7.0',
  deviceName: 'Android Emulator',
},
```

There are two things we need to take care of incase of mobile viewports:

- baseUrl is 10.0.2.2 instead of localhost because it is used to access the localhost of the host machine in the android emulator.
- Selenium address is using port 4723.

Some end-to-end tests sometimes fail without any specific reason. I will also fix this flakiness occurring in end-to-end tests, focussing primarily on stateEditor.js and editorAndPlayer.js (Please see issue [#4044](#)).

2. Documentation:

Present Scenario:

We have two Wiki pages, [one](#) which focuses on writing end-to-end tests and [another](#) page which explains [how to write end-to-end tests](#) for a new interaction. These pages offer an extremely clear step-by-step explanation and/or process for the same.

Plan:

Documentation for writing new end-to-end tests will be updated in the existing Wiki page on [how to write end-to-end tests](#) comprising of the following sections:

- What should a new end-to-end test necessarily comprise of?
- The parts that need to be stressed while designing such a test, namely, console errors and testing the complete workflow.
- How should any new test be integrated into the existing framework?

For documentation, Wiki pages seem to be a better choice here as compared to Google Docs as we already have two Wiki pages for the end-to-end tests. Also, accessing a Wiki page for quick reference is easier as compared to a Doc.

3. Organization:

Present Scenario:

The present end-to-end tests structure is:

- **core/tests/protractor/**: This directory comprises of the actual tests.
- **core/tests/protractor_utils/**: This directory comprises of all the utilities used to perform actions based on the elements from the core components of Oppia (found in core/templates/dev/head/)
- **extensions/**/protractor.js**: These files comprise of utilities for actions related to a specific extension.

Plan:

I intend to keep the structure same for the tests since there seem to be no problems in the current arrangement. Any new developer can easily get familiar with the organization of the tests by going through the Wiki pages once. Existing developers are already familiar with this arrangement and therefore it would be a bit difficult for them to adjust if we change the structure of the tests. Also, the present organization makes adding new tests quite easy - the developer loads up the utilities from the “protractor_utils” directory in a newly created file for writing the test, in “protractor” directory.

Any developer who wishes to write an end-to-end should do so before the first Saturday of each month so that they can be pushed to the release branch and can be used to test the added functionality and fix bugs. This is in accordance with our [release process](#). The tests should strongly emphasize on checking console errors and the complete workflow or sequence of actions which all possible users can perform.

3. Oppia-bot:

Present Scenario:

Oppia requires that contributors sign a CLA before they begin taking part in the development process, that is, resolving issues by submitting pull requests for them. Some first-time contributors forget to sign the CLA or skip the signing part completely. Then, when they ping on issue threads, it becomes difficult for the maintainer-on-duty or any other member to know whether they have signed the CLA. They have to wait for someone to confirm the CLA status.

The maintainer-on-duty also needs to manually keep an eye for stale PRs and ping the author to know the status. Sometimes the build fails and the author of the PR is unaware of it. The maintainer-on-duty has to ping the author telling him/her that the build has failed. This again is all manually done.

Plan:

I intend to lay the foundation for our very own, [Oppia-bot](#). This bot will be developed over the [Probot](#) framework as an independent Oppia application. The bot will automate all the above mentioned processes and will ease the work of the maintainer as well as the members of Oppia.

The Probot framework is built on Node.js and is highly configurable. Also, another good point regarding this framework is that it is continuously under development. One more fact is that, it already has some [extensions](#) ready-to-use.

The complete initial setup process can be found in the [official documentation](#) of Probot. In short, to create any GitHub app, one needs to register the app and get a private key.

Since the oppia-bot application is under development, I have not made it public, that is, it cannot be yet installed by other GitHub users. The oppia-bot repository also is private at the moment but I will make it public in the future.

Heroku VS GAE: The bot is currently deployed on Heroku. Another framework which we can go for, to host the bot is GAE.

The official Probot documentation provides an intuitive way to [deploy the bot on Heroku](#). We can easily monitor the bot's activity using the [heroku-cli](#).

Suppose any user comments on a issue thread, then heroku-cli highlights the bot activity as well:

```
2018-03-12T08:11:45.744522+00:00 app[web.1]: content-length: 8321
2018-03-12T08:11:45.751776+00:00 app[web.1]: 08:11:45.7512 INFO probot: Webhook received
2018-03-12T08:11:45.751778+00:00 app[web.1]: event: {
2018-03-12T08:11:45.751780+00:00 app[web.1]:   "id": "fc68e8d0-25cc-11e8-84c7-5218e09df451",
2018-03-12T08:11:45.751781+00:00 app[web.1]:   "event": "issue_comment.created",
2018-03-12T08:11:45.751782+00:00 app[web.1]:   "repository": "apb7/hello-world",
2018-03-12T08:11:45.751783+00:00 app[web.1]:   "installation": 92398
2018-03-12T08:11:45.751785+00:00 app[web.1]: }
2018-03-12T08:11:45.753070+00:00 app[web.1]: 08:11:45.7522 INFO probot: POST / 200 - 8.28 ms (id=a3b81a40-5014-4605-905a-82121da16274)
2018-03-12T08:11:45.753731+00:00 app[web.1]: 08:11:45.7532 TRACE probot: (id=a3b81a40-5014-4605-905a-82121da16274, res.duration=8.28)
2018-03-12T08:11:45.753733+00:00 app[web.1]: HTTP/1.1 200 OK
2018-03-12T08:11:45.753734+00:00 app[web.1]: x-powered-by: Express
2018-03-12T08:11:45.753735+00:00 app[web.1]: x-request-id: a3b81a40-5014-4605-905a-82121da16274
2018-03-12T08:11:45.753739+00:00 app[web.1]: content-type: application/json
```

Here, the image clearly shows the “event” which has triggered the bot, namely, “issue_comment.created” on the “repository”: “apb7/hello-world”.

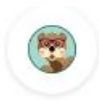
Hosting on Heroku is free of cost since the bot does not require much server computations and can be hosted on a free account.

I have hosted the app since the past ~30 days (as of March 2018) and have not faced any problem with heroku.

GAE, on the other hand, is favored since the Oppia website has been deployed using it. Since the server of the bot does not affect its working or performance, I have decided to go for Heroku.

Access Control: Another concern which springs up in case of any such application is the amount of access or control. This will not be a problem in case of the Oppia-bot.

The bot requires read and write access to issues, pull requests and repository contents to function properly:



Oppia-Bot

🕒 Installed 2 months ago

👤 Developed by [apb7](#)

🌐 <https://example.com>

A "Hello World!" bot which welcomes first-time contributors!

Permissions

✖ No access to code

✔ Read and write access to issues and pull requests

This bot will be completely under our control. It is not a fork of the framework and is an independent app. Moreover, we can specify the access during installation and can monitor the data through the heroku-cli (stated above).

The bot will be deployed on a free Heroku account. The credentials of the account could be shared with the Oppia admins. In this way, the running instance of the bot would be owned by the Oppia admins (and me). The admins would be able to make changes to the bot and push it onto the server.

Response Time: The response time of the bot when it is active is almost instantaneous (approximately ~1 to 2 seconds). After 30 minutes of inactivity, the server becomes idle. In that case the response time of the bot is approximately ~9 seconds, depicted by the image below.



apb7 commented 20 seconds ago

Owner



Hi! there!



oppia-bot bot commented 11 seconds ago



Hello! @apb7. Welcome to Oppia! Do you mind filling the CLA as given here:
<https://tinyurl.com/claformoppia> ?

The oppia-bot repository is private at the moment since it stores the private key as well as the Google authentication token to access the sheet.

As a starting point, the bot will automate the following processes:

a. CLA checking:

Status:

Completed for sample CLA form and sheet.

Technology:

Google Sheets API v4 (The latest API version for Sheets)

Details:

We need a one-time authentication token from the Gmail account of the user who stores the Sheet in the Drive. This is mandatory and is required to use the APIs to access the Sheet. I have set this up in such a way that this token is stored locally on the server in a directory ".credentials" as a JSON file.

At present, the bot picks the username of the user who has commented on any issue thread or a pull request using GitHub APIs and then checks it against each username in the Sheet using Google Sheets APIs.

This picture highlights the logic behind CLA checking:

```
.1]: GitHub Handle
.1]: 2
.1]: [ [ 'apb7' ], [ 'bugtester7' ] ]
.1]: apb7
.1]: true
.1]: userCheck details
.1]: true
.1]: Inside if
```

(In line 3 above, the user names are stored as list of list of strings. This is the response obtained on reading the values from the sheet. This might be for multiple comma-separated values in a single cell which might not happen in our case. I'll do away with this when we set the CLA sheets).

The above picture displays the list of GitHub handle of individuals who have signed the CLA. The bot then compares it with the userName extracted from the issue/pull request and operates the if block when the GitHub handle from the list matches the userName otherwise else block. It then generates a comment accordingly.

(I've set console.log here to highlight the logic. This will be more clear when viewed with the complete code.)

The "hasUserSignedCla" variable is a boolean which is set to "true" if it encounters the userName in the Sheet as well.

In case of a PR, if a user has not signed the CLA, the bot puts up a label "Needs CLA" along with a comment asking the author to sign the CLA.

Later when the author comments something like "I signed it!", the bot rechecks the CLA status and removes the label from the PR.

This works even if the PR has multiple labels attached to it:

The screenshot shows a GitHub pull request interface. At the top, a comment from user 'apb7' (commented 23 hours ago) says "No description provided." Below this, a commit titled "Create TEST-3.md" is shown, verified by user '05821bb'. The Oppia-bot (commented 23 hours ago) then adds a "Needs CLA" label. A subsequent comment from 'apb7' (commented 23 hours ago) says "Hey @oppia-bot! I have signed it." Finally, the Oppia-bot (commented 23 hours ago) removes the "Needs CLA" label. On the right side of the interface, there are settings for Reviewers (No reviews), Assignees (No one—assign yourself), Labels (None yet), Projects (None yet), and Milestone (No milestone). A notification bar at the bottom right indicates "Unsubscribe" and states "You're receiving notifications because you were mentioned."

This is a [video demonstration](#) for the CLA checking done by the Oppia-bot. Also, the Oppia-bot is already installed in one of my repositories and can be tested [here](#).


I also found some inspiration for CLA checking:

Googlebot:



googlebot commented on 9 Feb + 🗨️

Thanks for your pull request. It looks like this may be your first contribution to a Google open source project. Before we can look at your pull request, you'll need to sign a Contributor License Agreement (CLA).

 Please visit <https://cla.developers.google.com/> to sign.

Once you've signed, please reply here (e.g. `I signed it!`) and we'll verify. Thanks.

- If you've already signed a CLA, it's possible we don't have your GitHub username or you're using a different email address on your commit. Check [your existing CLA data](#) and verify that your [email is set on your git commits](#).
- If your company signed a CLA, they designated a Point of Contact who decides which employees are authorized to participate. You may need to contact the Point of Contact for your company and ask to be added to the group of authorized contributors. If you don't know who your Point of Contact is, direct the project maintainer to `go/cla#troubleshoot`. The email used to register you as an authorized contributor must be the email used for the Git commit.
- In order to pass this check, please resolve this problem and have the pull request author add another comment and the bot will run again. If the bot doesn't comment, it means it doesn't think anything has changed.

b. Stale Pull Requests and Issues:

Status: Completed.

Technology: Probot's existing application, [stale](#).

Details:

The framework already has an application for this purpose, [stale](#). We can directly add this as a "plugin" to our Oppia-bot. I have read the documentation regarding this and will try implementing it. There are other bots too which extend a number of plugins together. So it is quite feasible.

To specify the the exact definition of stale, we create a ".github/stale.yml" file as explained [here](#). There are two things we need to specify, "daysUntilStale", which is used to label the PR stale and "daysUntilClose", which closes the PR after a specific number of days, after the "daysUntilStale" has passed. I suggest we set "daysUntilStale" as 60 and "daysUntilClose" as 7. "daysUntilClose" are counted after the PR is labelled as "stale". Therefore any PR will be closed after 67 days (It will marked as stale after 60 days and then closed a week after that).

The screenshot shows a sequence of events in a GitHub pull request. It starts with a comment from the bot 'oppiabot' welcoming user '@apb7' and asking them to follow instructions. This is followed by a response from '@apb7' stating they have signed the CLA. Then, the bot removes the 'Needs CLA' label. Next, the bot comments that the pull request is stale due to lack of activity and will be closed. Finally, the bot adds a 'stale' label.

c. Mentioning Maintainer on-duty and the PR author:

Status: Under development.

Technology: Travis CI [npm package](#) or [Travis CI API V3](#) directly. For merge conflicts, we can use the metadata of a PR using GitHub APIs.

Details:

In case of build failure or merge conflicts, the Oppia-bot will post a comment on the thread, with @ mention to the maintainer and/or the author. This will automatically trigger an email to the involved individuals (by GitHub itself). Regarding the maintainers list, it can be stored as a Sheet and the bot can directly access it. The comment will contain a link to redirect individuals to the appropriate wiki pages -- the first to "[If your build fails](#)" and the second to the relevant part of the PR instructions (Second sub-point of the fifth point in Instructions for making a code change).

d. Automating routine update of translations (if time permits):

Status: Under development.

Technology: GitHub APIs

Details:

At the moment, we have to manually generate [PRs](#) from the translatewiki branch to the develop branch whenever the translatewiki branch changes, once a month. This process could be automated by the Oppia-bot. The bot could generate a PR once a month and the maintainer-on-duty could then review and merge it accordingly.

Testing the bot: We will use the [jest](#) framework to test the bot. Jest can be installed by `npm install jest` (Detailed installation instructions can be found [here](#)).

The tests would be placed in the `__tests__` folder.

The tests will be built along the same lines as shown in the below example:

```
// Requiring probot allows us to mock out a robot instance
const {createRobot} = require('probot')
// Requiring our app
const app = require('.')
// Create a fixtures folder in your test folder
// Then put any larger testing payloads in there
const payload = require('./fixtures/payload')

describe('your-app', () => {
  let robot
  let github

  beforeEach(() => {
    // Here we create a robot instance
    robot = createRobot()
    // Here we initialize the app on the robot instance
    app(robot)
    // This is an easy way to mock out the GitHub API
    github = {
      issues: {
        createComment: jest.fn().mockReturnValue(Promise.resolve({
          // Whatever the GitHub API should return
        })))
      }
    }
    // Passes the mocked out GitHub API into our robot instance
    robot.auth = () => Promise.resolve(github)
  })
})
```

```
describe('your functionality', () => {
  it('performs an action', async () => {
    // Simulates delivery of a payload
    // payload.event is the X-GitHub-Event header sent by GitHub
    (for example "push")
    // payload.payload is the actual payload body
    await robot.receive(payload)
    // This test would pass if in your main code you called
    `context.github.issues.createComment`
    expect(github.issues.createComment).toHaveBeenCalled()
  })
})
})
```

Time Zone where I will primarily be during the summer:

Indian Standard Time (IST) which is ahead of UTC by 5 hours and 30 minutes.

Time which I will be able to commit to the project:

I will be able to devote approximately at least 7 to 9 hours a day on an average throughout the project and aim for 55 to 60 hours a week. The time devoted to the project may increase but will never fall below 7 hours per day on an average.

Other obligations during the summers:

I have my summer vacations from mid of May to end of July. I might have some classes for a few days (the exact date is not known as of now) but that will not affect the time I devote to the project. There might be two days of travelling, one in May, around 21st and the other in July, around 15th. Since I am already familiar with the community, I plan to begin early during the community bonding period so that I have ample of buffer time for each milestone. Also, I have planned the milestones in such a way so that I can directly begin with their implementation.

Preferred mode of communication: I am comfortable with all modes, be it Gitter or Hangouts and am willing to chose any mode used by the mentors.

How often I plan to communicate with my mentors and through which channels:

I will be in continuous touch with the mentors via email, Gitter or Hangouts. There could be biweekly (or as preferred by the mentors) meetings on Hangouts to discuss about the workflow to be followed ahead. I would also love to maintain daily logs of my progress to keep a track of everything.

Thanks!