# Contributing

<!-- SPDX-License-Identifier: (MIT OR CC-BY-3.0+) -->

Feedback and contributions are very welcome!

Here's help on how to make contributions, divided into the following sections:

\* general information,

\* [vulnerability reporting](#how\_to\_report\_vulnerabilities),

\* documentation changes,

\* translations,

\* criteria changes,

\* code changes,

\* how to check proposed changes before submitting them,

\* reuse (supply chain for third-party components, including updating them), and

\* keeping up with external changes.

## General information

For specific proposals, please provide them as

[pull requests](https://github.com/coreinfrastructure/best-practices-badge/pulls)

or

[issues](https://github.com/coreinfrastructure/best-practices-badge/issues)

via our

[GitHub site](https://github.com/coreinfrastructure/best-practices-badge).

For general discussion, feel free to use the

[cii-badges mailing list](https://lists.coreinfrastructure.org/mailman/listinfo/cii-badges).

The "doc/" directory has information you may find helpful, for example:

- [governance.md](doc/governance.md) describes our governance model

(how we decide things)

- [implementation.md](doc/implementation.md) provies implementation details

- [background.md](doc/background.md) provides background info on criteria

You can see the entire set of "passing" criteria in the generated file

[criteria.md](doc/criteria.md).

The documentation for higher-level badge criteria is in

[other.md](doc/other.md).

If you want \*change\* the criteria, see below.

The [INSTALL.md](doc/INSTALL.md) file explains how to install the program

locally (highly recommended if you're going to make code changes).

It also provides a quick start guide.

If you're new to the project (or FLOSS in general), the

[Up-for-grabs](https://github.com/coreinfrastructure/best-practices-badge/labels/up-for-grabs)

issues are smaller tasks that may typically take 1-3 days.

You are welcome aboard!

The [roadmap.md](doc/roadmap.md) file provides an overview of future plans.

See [CODE OF CONDUCT](./CODE\_OF\_CONDUCT.md) for our code of conduct;

in short, "Be excellent to each other".

### Pull requests and different branches recommended

Pull requests are preferred, since they are specific.

For more about how to create a pull request, see

<https://help.github.com/articles/using-pull-requests/>.

We recommend creating different branches for different (logical)

changes, and creating a pull request when you're done into the master branch.

See the GitHub documentation on

[creating branches](https://help.github.com/articles/creating-and-deleting-branches-within-your-repository/)

and

[using pull requests](https://help.github.com/articles/using-pull-requests/).

### How we handle proposals

We use GitHub to track proposed changes via its

[issue tracker](https://github.com/coreinfrastructure/best-practices-badge/issues) and

[pull requests](https://github.com/coreinfrastructure/best-practices-badge/pulls).

Specific changes are proposed using those mechanisms.

Issues are assigned to an individual, who works it and then marks it complete.

If there are questions or objections, the conversation area of that

issue or pull request is used to resolve it.

### Two-person review

Our policy is that at least 50% of all proposed modifications will be reviewed

before release by a person other than the author,

to determine if it is a worthwhile modification and free of known issues

which would argue against its inclusion

(per the Gold requirement two\_person\_review).

We achieve this by splitting proposals into two kinds:

1. Low-risk modifications. These modifications are being proposed by

people authorized to commit directly, pass all tests, and are unlikely

to have problems. These include documentation/text updates

(other than changes to the criteria) and/or updates to existing gems

(especially minor updates) where no risk (such as a security risk)

have been identified. The project lead can decide that any particular

modification is low-risk.

2. Other modifications. These other modifications need to be

reviewed by someone else or the project lead can decide to accept

the modification. Typically this is done by creating a branch and a

pull request so that it can be reviewed before accepting it.

### Developer Certificate of Origin (DCO)

All contributions (including pull requests) must agree to

the [Developer Certificate of Origin (DCO) version 1.1](doc/dco.txt).

This is exactly the same one created and used by the Linux kernel developers

and posted on <http://developercertificate.org/>.

This is a developer's certification that he or she has the right to

submit the patch for inclusion into the project.

Simply submitting a contribution implies this agreement, however,

please include a "Signed-off-by" tag in every patch

(this tag is a conventional way to confirm that you agree to the DCO).

You can do this with <tt>git commit --signoff</tt> (the <tt>-s</tt> flag

is a synonym for <tt>--signoff</tt>).

Another way to do this is to write the following at the end of the commit

message, on a line by itself separated by a blank line from the body of

the commit:

````

Signed-off-by: YOUR NAME <YOUR.EMAIL@EXAMPLE.COM>

````

You can signoff by default in this project by creating a file

(say "git-template") that contains

some blank lines and the signed-off-by text above;

then configure git to use that as a commit template. For example:

````sh

git config commit.template ~/cii-best-practices-badge/git-template

````

It's not practical to fix old contributions in git, so if one is forgotten,

do not try to fix them. We presume that if someone sometimes used a DCO,

a commit without a DCO is an accident and the DCO still applies.

### License (MIT)

All (new) contributed material must be released

under the [MIT license](./LICENSE).

All new contributed material

that is not executable, including all text when not executed,

is also released under the

[Creative Commons Attribution 3.0 International (CC BY 3.0) license](https://creativecommons.org/licenses/by/3.0/) or later.

See the section on reuse for their license requirements

(they don't need to be MIT, but all required components must be

open source software).

### We are proactive

In general we try to be proactive to detect and eliminate

mistakes and vulnerabilities as soon as possible,

and to reduce their impact when they do happen.

We use a defensive design and coding style to reduce the likelihood of mistakes,

a variety of tools that try to detect mistakes early,

and an automatic test suite with significant coverage.

We also release the software as open source software so others can review it.

Since early detection and impact reduction can never be perfect, we also try to

detect and repair problems during deployment as quickly as possible.

This is \*especially\* true for security issues; see our

[security information](doc/security.md) for more.

### No trailing whitespace

Please do not use or include trailing whitespace

(spaces or tabs at the end of a line).

Since they are often not visible, they can cause silent problems

and misleading unexpected changes.

For example, some editors (e.g., Atom) quietly delete them by default.

## <span id="how\_to\_report\_vulnerabilities">Vulnerability reporting (security issues)</a>

If you find a significant vulnerability, or evidence of one,

please send an email to the security contacts that you have such

information, and we'll tell you the next steps.

For now, the security contacts are:

David A. Wheeler <dwheelerNOSPAM@dwheeler.com>,

Jason Dossett <jdossettNOSPAM@utdallas.edu>,

and

Dan Kohn <danNOSPAM@linuxfoundation.com>

(remove the NOSPAM markers).

Please use an email system (like Gmail) that supports

hop-to-hop encryption using STARTTLS when reporting vulnerabilities.

Examples of such systems include Gmail, Outlook.com, and runbox.com.

See [STARTTLS Everywhere](https://starttls-everywhere.org/)

if you wish to learn more about efforts to encourage the use of STARTTLS.

Your email client should use encryption to communicate with

your email system (i.e., if you use a web-based email client then use HTTPS,

and if you use email client software then configure it to use encryption).

Hop-to-hop encryption isn't as strong as end-to-end encryption,

but we've decided that it's strong enough for this purpose

and it's much easier to get everyone to use it.

We will gladly give credit to anyone who reports a vulnerability

so that we can fix it.

If you want to remain anonymous or pseudonymous instead,

please let us know that; we will gladly respect your wishes.

## Documentation changes

Most of the documentation is in "markdown" format.

All markdown files use the .md filename extension.

Where reasonable, limit yourself to Markdown

that will be accepted by different markdown processors

(e.g., what is specified by CommonMark or the original Markdown)

In practice we use

the version of Markdown implemented by GitHub when it renders .md files,

and you can use its extensions

(in particular, mark code snippets with the programming language used).

This version of markdown is sometimes called

[GitHub-flavored markdown](https://help.github.com/articles/github-flavored-markdown/).

In particular, blank lines separate paragraphs; newlines inside a paragraph

do \*not\* force a line break.

Beware - this is \*not\*

the same markdown algorithm used by GitHub when it renders

issue or pull comments; in those cases

[newlines in paragraph-like content are considered as real line breaks](https://help.github.com/articles/writing-on-github/);

unfortunately this other algorithm is \*also\* called

GitHub rendered markdown.

(Yes, it'd be better if there were standard different names

for different things.)

The style is basically that enforced by the "markdownlint" tool.

Don't use tab characters, avoid "bare" URLs (in a hypertext link, the

link text and URL should be on the same line), and try to limit

lines to 80 characters (but ignore the 80-character limit if that would

create bare URLs).

Using the "rake markdownlint" or "rake" command

(described below) implemented in the development

environment can detect some problems in the markdown.

That said, if you don't know how to install the development environment,

don't worry - we'd rather have your proposals, even if you don't know how to

check them that way.

Do not use trailing two spaces for line breaks, since these cannot be

seen and may be silently removed by some tools.

Instead, use <tt>&lt;br&nbsp;/&gt;</tt> (an HTML break).

## Translations

Please help us have \*good\* native language translations -

we love to have them!

Native language translations

help both developers and potential users understand the state

of a software project, even if they speak a variety of different languages.

Each translation is led by one or more trusted translators.

If you want to be a trusted translator, you basically need to convince

us that you're trustworthy and will maintain a \*good\* idiomatic translation.

If you are trustworthy we would love to have you!

Trusted translators are given access to the

[translation.io](https://translation.io/) service, which provides a

simple web interface for translating every English phrase into the

language(s) the trusted translator manages.

If you want to propose specific \*changes\* to a translation, and you are

not a trusted translator, there are two main options:

\* The usual option is to open an issue and simply propose the text changes.

Make sure you tell us which locale you're referrring to!

\* You \*can\* propose changes as edits to the appropriate files in

`config/locales`, but unlike most changes that will not work directly.

One of the trusted translators will then need to hand-copy

each change into the `translation.io` website

(our workflow (see below) currently does \*not\* support copying from GitHub

to the translation.io site; it only copies the other direction).

If we merely accepted translation changes to `config/locales`, the changes

would be overwritten the next time we sync'ed with translation.io,

and we don't want to lose good changes!

This approach works as long as there aren't \*too\* many changes.

Here's how we handle translations.

We periodically "sync" translation.io with the development version of the

badge application using the command `rake translation:sync`.

Sync'ing loads the updated English text `config/locales/en.yml`

to the translation.io website,

and also copies the translations in the translation.io website

into the badge application directory `config/locales/`.

When this updated version of the badge application is later

put into production, the updated translations are used.

This makes it very easy for translators to translate text.

Note that there are intermediate steps between the updating a translation

and deployment to users, so there's a delay between editing the translation

and seeing it deployed.

If you want to become a trusted translator in a locale we don't already support,

there are a few steps we have to take to add the locale.

In particular, we need to know the locale code.

If the language is not written left-to-right (e.g., Arabic and Hebrew),

there are some additional steps we will need to take.

And of course, you need to convince us that you'll produce trustworthy

translations, because others are counting on you.

## Criteria changes

Changing \*criteria\* can have a much larger impact on participating

projects than simply changing the supporting software, so we have special

rules about them. For those rules, see

[governance.md](doc/governance.md).

For the technical details on how to implement new criteria,

or modify existing criteria (including their text, details, rationale,

and scoring criteria), see

[implementation.md](doc/implementation.md).

## Code changes

To make changes to the "BadgeApp" web application that implements the criteria,

you may find the following helpful; [INSTALL.md](doc/INSTALL.md)

(installation information) and [implementation.md](doc/implementation.md)

(implementation information).

The code should strive to be DRY (don't repeat yourself),

clear, and obviously correct.

Some technical debt is inevitable, just don't bankrupt us with it.

Improved refactorizations are welcome.

Always ensure that all JavaScript and CSS styles are

in \*separate\* files, do not embed them in the HTML.

That includes any generated HTML.

That way we can use CSP entries

that harden the program against security attacks.

Below are guidelines for specific languages.

### Ruby

The web application is primarily written in Ruby on Rails.

Please generally follow the

[community Ruby style guide](https://github.com/bbatsov/ruby-style-guide)

and the complementary

[community Rails style guide](https://github.com/bbatsov/rails-style-guide).

Our continous integration setups runs Rubocop on each commit to ensure they're

being followed.

For example, in Ruby:

\* [use two-space indents](https://github.com/bbatsov/ruby-style-guide#spaces-indentation)

\* [use Unix line-endings](https://github.com/bbatsov/ruby-style-guide#crlf)

\* [use single-quoted strings when you don't need string interpolation or special symbols](https://github.com/bbatsov/ruby-style-guide#consistent-string-literals)

\* [Use the Ruby 1.9 hash literal syntax when your hash keys are symbols.](https://github.com/bbatsov/ruby-style-guide#hash-literals)

\* [Prefer symbols instead of strings as hash keys](https://github.com/bbatsov/ruby-style-guide#hash-literals)

In Ruby,

[prefer symbols over strings (especially as hash keys)](https://github.com/bbatsov/ruby-style-guide#symbols-as-keys)

when they do not potentially come from the user.

Symbols are typically faster, with no loss of readability.

There is one big exception:

Data from JSON should normally be accessed with strings,

since that's how Ruby normally reads it.

Rails normally uses the type ActiveSupport::HashWithIndifferentAccess,

where the difference between symbols and strings is ignored,

but JSON results use standard Ruby hashes where symbols and strings are

considered different; be careful to use the correct type in these cases.

We have designed the application to be thread-safe (Rails is itself

thread-safe, including its caching mechanism).

Please follow the guidelines in

[How Do I Know Whether My Rails App Is Thread-safe or Not?](https://bearmetal.eu/theden/how-do-i-know-whether-my-rails-app-is-thread-safe-or-not/);

see also

[How to test multithreaded code](http://www.mikeperham.com/2015/12/14/how-to-test-multithreaded-code/).

In short: Each concurrent request runs in its own thread.

Threads typically create objects within their own thread; they may also

read shared objects with impunity.

However, be extremely cautious about \*changing\* shared objects.

You should normally only change shared objects through mechanisms

designed for the purpose (e.g., the database or internal Rails cache).

In Ruby please prefer the String operations that do not have side-effects

(e.g., "+", "sub", or "gsub"), and consider freezing strings.

Do \*not\* modify a String literal in-place

(e.g., using "<<", "sub!", or "gsub!") until you have applied ".dup" to it.

There are current plans that

[Ruby 3's string literals will be immutable](https://twitter.com/yukihiro\_matz/status/634386185507311616).

See [issue 11473](https://bugs.ruby-lang.org/issues/11473) for more.

Even if this doesn't happen, freezing string literals is both faster and

reduces the risk of accidentally modifying a shared string.

Use "dup" on a string literal to produce a mutable string;

since "dup" is already permitted in the language,

this provides a simple backwards-compatible way for us to indicate

that the String is mutable in this case.

For example, if you want to build a string using append, do this:

~~~~ruby

''.dup << 'Hello, ' << 'World'

~~~~

You may use the safe navigation operator '&amp;.' added in

[Ruby version 2.3.0](https://www.ruby-lang.org/en/news/2015/12/25/ruby-2-3-0-released/).

Our static analysis tools' parsers can now handle this syntax.

This means that this application \*requires\* Ruby version 2.3.0 or later to run.

When making new tests, if you need to modify the setup or teardown methods for a

test class, please use callbacks instead of overwrites; i.e. use "setup do"

instead of "def setup." This preserves any changes to those methods that

may have been made in test\_helper.rb.

\*Never\* include user email addresses in the internal Rails cache.

Caches are stored for reuse later, and there's always the risk that

they will accidentally presented to another user not authorized to see it.

Otherwise, please \*do\* use caches to speed repeated responses where they

make sense. Caches are one of the key mechanisms we use to provide

rapid responses to users.

Please use

[# frozen\_string\_literal: true](https://bugs.ruby-lang.org/issues/8976)

at the beginning of each file.

This 'magic comment' (added in Ruby 2.3.0) automatically freezes

string literals, increasing speed, preventing accidental changes, and

will help us get ready for the planned Ruby transition

to immutable string literals.

Each source file should include a copyright and license statement

the beginning. Here is our standard header:

~~~~ruby

# frozen\_string\_literal: true

# Copyright 2015-2017, the Linux Foundation, IDA, and the

# CII Best Practices badge contributors

# SPDX-License-Identifier: MIT

~~~~

### JavaScript

There is a small amount of application-specific client-side JavaScript;

by convention custom client-side JavaScript is in "app/assets/javascripts/".

This is written in JavaScript, not CoffeeScript;

it's only a small amount of JavaScript, so the advantages of

CoffeeScript aren't

obvious, and far more people know basic JavaScript than CoffeeScript.

Our JavaScript coding style is based on the

[Node.js style guide](https://github.com/felixge/node-style-guide).

In particular, we use

2-space indents, terminating semicolons, camelCase, required braces,

and '===' (never '==') for string comparison,

These coding style rules are checked by ESLint

(see .eslintrc for the rule list).

Always put JavaScript (and CSS styles) in \*separate\* files, do not

embed JavaScript in the HTML. That way we can use CSP entries

that harden the program against security attacks.

Historically we used

[jquery-turbolinks](https://github.com/kossnocorp/jquery.turbolinks)

to try to make Rails' "turbolinks" mechanism work

with standard JavaScript ready events.

Rails' turbolinks gem claims that it

["works great with the jQuery framework"](https://github.com/rails/turbolinks),

but this is misleading.

[Turbolinks breaks $(document).ready](http://guides.rubyonrails.org/working\_with\_javascript\_in\_rails.html#page-change-events)

(an extremely common construct)

and by default requires you to use a nonstandard on..."page:change".

These incompatibilities led to unreliability, particularly on the

/project\_stats page.

We have (for now) abandoned turbolinks, and that made everything reliable.

### Shell

There's a small amount of Bourne shell code

(the script that sets up a new development install).

If you modify it, make sure it passes shellcheck

(a static analysis tool for shell).

### HTML

Try to avoid using "target=" in the "a" tag. If you must use target=,

always use rel="noopener" as well, for security reasons.

For more information, see

["Target="\\_blank" - the most underestimated vulnerability ever" by Alex Yumashev, May 4, 2016](https://www.jitbit.com/alexblog/256-targetblank---the-most-underestimated-vulnerability-ever/).

### Automated tests

When adding or changing functionality, please include new tests for them as

part of your contribution.

We require the Ruby code to have at least 90% statement coverage;

please ensure your contributions do not lower the coverage below that minimum.

The Ruby code uses minitest as the test framework, and we use

'vcr' to record live data for replaying later.

Additional tests are very welcome.

We encourage tests to be created first, run to ensure they fail, and

then add code to implement the test (aka test driven development).

However, each git commit should have both

the test and improvement in the \*same\* commit,

because 'git bisect' will then work well.

\*WARNING\*: It is possible that some tests may intermittently fail, even though

the software works fine.

If tests fail, restart to see if it's a problem with the software

or the tests. On CircleCI you can choose to rebuild.

Where possible, try to find and fix the problem; we have worked to

eliminate this, and at this point believe we have fixed it.

If you use an old version of PhantomJS (e.g., if you use

Ubuntu 14.04 and install PhantomJS via apt-get), you'll see this

message during tests:

> You're running an old version of PhantomJS,

> update to >= 2.1.1 for a better experience.

You can eliminate the warnings about old versions of PhantomJS

by uninstalling the old version and installing a

[current version of PhantomJS](http://phantomjs.org/download.html).

### Security, privacy, and performance

Pay attention to security, and work \*with\* (not against) our

security hardening mechanisms. In particular, put JavaScript and CSS

in \*separate\* files - this makes it possible to have very strong

Content Security Policy (CSP) rules, which in turn greatly reduces

the impact of a software defect. Be sure to use prepared statements

(including via Rails' ActiveRecord).

Protect private information, in particular passwords and email addresses.

Avoid mechanisms that could be used for tracking where possible

(we do need to verify people are logged in for some operations),

and ensure that third parties can't use interactions for tracking.

When sending an email to an existing account, use the original account

email not the claimed email address sent now; for why, see

[Hacking GitHub with Unicode's dotless 'i'](https://eng.getwisdom.io/hacking-github-with-unicode-dotless-i/).

For more about security, see [security](doc/security.md).

We want the software to have decent performance for typical users.

[Our goal is interaction in 1 second or less after making a request](https://developers.google.com/web/fundamentals/performance/rail).

Don't send megabytes of data for a request

(see

[The Website Obesity Crisis](http://idlewords.com/talks/website\_obesity.htm)).

Use caching (at the server, CDN, and user side) to improve performance

in typical cases (while avoiding making the code too complicated).

Moving all the JavaScripts to a long-lived cached page, for example,

means that the user only needs to load that page once.

See the "other tools" list below for some tools to help measure performance.

There's always a trade-off between various attributes, in particular,

don't make performance so fast that the software is hard to maintain.

Instead, work to get "reasonable" performance in typical cases.

## How to check proposed changes before submitting them

Before submitting changes, you \*must\*

run 'rake' (no options) to look for problems,

and fix the problems found.

In some cases it's okay to fix them by disabling the warning in that particular

place, but be careful; it's often better to make a real change,

even if it doesn't matter in that particular case.

### Standard checks

The specific list of tools run by default using 'rake' is listed in

[default.rake](lib/tasks/default.rake).

Currently these include at least the following rake tasks that

check the software:

\* \*bundle\* - use bundle to check dependencies

("bundle check || bundle install")

\* \*bundle\_doctor\* - sanity check on Ruby gem configuration/installation

\* \*bundle\_audit\* - check for transitive gem dependencies with

known vulnerabilities

\* \*rubocop\* - runs Rubocop, which checks Ruby code style against the

[community Ruby style guide](https://github.com/bbatsov/ruby-style-guide)

\* \*markdownlint\* - runs markdownlint, also known as mdl

(this checks for errors in the markdown text)

\* \*rails\_best\_practices\* - check Ruby against rails best practices

using the gem

[rails\_best\_practices](http://rails-bestpractices.com/)

\* \*brakeman\* - runs Brakeman, which is a static source code analyzer

to look for Ruby on Rails security vulnerabilities

\* \*license\_okay\* - runs license\_finder to check the

OSS licenses of gem dependencies (transitively).

A separate dependency on file 'license\_finder\_report.html' generates

a detailed license report in HTML format.

\* \*whitespace\_check\* - runs "git diff --check" to detect

trailing whitespace in latest diff

\*yaml\_syntax\_check\* - checks syntax of YAML (.yml) files.

Note that the automated test suite includes a number of specific

checks on the criteria/criteria.yml file.

\* \*fasterer\* - report on Ruby constructs with poor performance

(temporarily disabled until it supports Ruby 2.4)

\* \*eslint\* - Perform code style check on JavaScript using eslint

\* \*test\* - run the automated test suite

Running "rake test" (the automated test suite) will show

"Run options: --seed ...", "# Running:", and a series of dots (passing tests).

### Expected noise from 'rake'

Ruby 2.4.0 has deprecated the Fixnum and Bignum classes, but they are used

by some gems we depend on.

Because we now use Ruby 2.4.0, there may be several warnings of this form:

~~~~

FILENAME.rb:LINE: warning: constant ::Fixnum is deprecated

FILENAME.rb:LINE: warning: constant ::Bignum is deprecated

~~~~

In some cases you'll see a test retry message like this

(but it will eventually pass):

~~~~

..[MinitestRetry] retry 'test\_0002\_Can Login and edit using custom account'

count: 1, msg: Unexpected exception

~~~~

The retry messages, when they happen, come from

the few tests we use that use a full simulated web browser (via Capybara).

Sometimes these full tests cause spurious failures, so we intentionally

retry failing tests to eliminate false failure reports (to make sure the

problem is in the software under test, and not in our test framework).

### Handling test failures

Tests may fail after certain kinds of asset changes,

with odd messages such as:

> Capybara::Poltergeist::JavascriptError: One or more errors were raised

> in the Javascript code on the page. If you don't care about these errors,

> you can ignore them by setting js\_errors: false in your Poltergeist

> configuration (see documentation for details).

In many cases this is because you need to forcibly precompile the assets;

just run this:

~~~~

RAILS\_ENV=production rake assets:precompile

~~~~

### Other tools

Here are some other tools we sometimes use for checking quality or security,

though they are not currently integrated

into the default "rake" checking task:

\* OWASP ZAP web application security scanner.

You are encouraged to use this and other web application scanners to find and

fix problems.

\* Google Chrome auditor. View a web page, then select menu / more tools /

developer tools / audits. This runs a variety of tests and checks,

including some security checks of the code delivered to the browser.

Note that not all reports are relevant.

\* [Snyk](https://snyk.io/test/github/coreinfrastructure/best-practices-badge?severity=high&severity=medium&severity=low)

\* JSCS (JavaScript style checker) using the Node.js format.

\* JSHint (JavaScript error detector)

\* W3C link checker <https://validator.w3.org/checklink>

\* W3C markup validation service <https://validator.w3.org/>

\* AChecker, <https://achecker.ca/checker/index.php>, a web accessibility

checker. This tool looks for web accessibility problems.

We typically check the following paths

(these are key forms in the system): "/", "/signup", "/login",

"/projects", and "/projects/1".

Our goal is to have no known problems and no likely problems per this tool

(this was true on 2017-06-03).

The "potential problems" list is very noisy, so we don't worry

about that list as much.

Here are some online tools we sometimes use to check for performance issues

(including time to complete rendering, download size in bytes, etc.):

\* [WebPageTest](https://www.webpagetest.org/)

\* [Varvy PageSpeed](https://varvy.com/pagespeed/)

\* [Yellow lab tools](http://yellowlab.tools/) - Examines performance.

It can notice issues like excessive accesses of the DOM from JavaScript.

It's OSS; see

([YellowLabTools on GitHub](https://github.com/gmetais/YellowLabTools))

\* [Pingdom](https://tools.pingdom.com/)

This

[article on Rails front end performance](https://www.viget.com/articles/rails-front-end-performance)

may be of use to you if you're interested in performance.

We sometimes run this to check if assets compile properly (see

[heroku\_rails\_deflate](https://github.com/mattolson/heroku\_rails\_deflate)):

~~~~

RAILS\_ENV=production rake assets:precompile

~~~~

### Testing during continuous integration

Note that we also use

[CircleCI](https://circleci.com/gh/coreinfrastructure/best-practices-badge)

for continuous integration tools to check changes

after they are checked into GitHub; if they find problems, please fix them.

These run essentially the same set of checks as the default rake task.

## Git commit messages

When writing git commit messages, try to follow the guidelines in

[How to Write a Git Commit Message](https://chris.beams.io/posts/git-commit/):

1. Separate subject from body with a blank line

2. Limit the subject line to 50 characters.

(We're flexible on this, but \*do\* limit it to 72 characters or less.)

3. Capitalize the subject line

4. Do not end the subject line with a period

5. Use the imperative mood in the subject line (\*command\* form)

6. Wrap the body at 72 characters ("<tt>fmt -w 72</tt>")

7. Use the body to explain what and why vs. how

(git tracks how it was changed in detail, don't repeat that)

## Reuse (supply chain)

### Requirements for reused components

We prefer reusing components instead of writing lots of code,

but please evaluate all new components before adding them

(including whether or not you need them).

We want to reduce our risks of depending on software that is poorly

maintained or has vulnerabilities (intentional or unintentional).

Mike Perham's [Kill Your Dependencies](http://www.mikeperham.com/2016/02/09/kill-your-dependencies/)

notes that, "every dependency in your application has the potential to

bloat your app, to destabilize your app, to inject odd behavior...

When you are considering adding a dependency to your Rails app,

it's a good idea to do a quick sanity check...".

So don't bring in gems you don't need

(if it's trivial to re-implement the required function, consider doing it).

Also, if the gem transitively depends on in many other gems,

especially if they are new additions, look for simpler alternatives

or help the upstream library remove the unnecessary dependencies.

#### Requirements for reused Ruby gems

In most cases we add reusable components by adding Ruby gems.

Here are guidelines for adding Ruby gems:

\* Before adding a Ruby gem, check its popularity on

<https://www.ruby-toolbox.com/>, and prefer "more popular" gems.

A popular gem may have unintentional or intentional vulnerabilities,

but they are less likely, and are more likely to be noticed.

\* For Ruby gems, look at its data at <https://rubygems.org/> to learn

more about it. E.G., is it still actively maintained?

(e.g., it uses semantic versioning and have a ChangeLog).

\* For some tips on how to evaluate gems, see

<a href="https://evilmartians.com/chronicles/open-source-software-whats-in-a-poke">"Back-end Open-Source Software: What is in a poke?"

by Sergey Dolganov (January 26, 2017)</a>.

\* All required reused components MUST be open source software (OSS).

It is \*not\* acceptable to insert a dependency

that \*requires\* proprietary software; making it portable so it \*can\* use

some proprietary software is gratefully welcome.

We also have to combine them legally in the way they are used.

You can review the code of a specific version of a gem.

You \*must\* review the code if you have any reason to believe that the

gem might be malicious.

That said, you need to be careful.

If you simply install the gem using bundler, that will

potentially run code, which is not what you want to do if it includes

malicious code.

Instead, create a subdirectory ("mkdir temp"), run "cd temp", and then

run this (you can omit "-v VERSION" if you're just getting the latest):

~~~~

gem fetch -v VERSION GEM\_NAME

gem unpack GEMNAME-VERSION

~~~~

Then "cd GEMNAME-VERSION" to review the code.

We presume that the Rubygems site will not insert malicious code into what

it distributes, but clearly individual gem writers can be malicious.

If you add a Ruby gem, put its \*fixed\* version number in the Gemfile file,

and please add a brief comment to explain what it is and/or why it's there.

#### Requirements for reused components in general

For any reused software, here are a few general rules:

\* Prefer software that

appears to be currently maintained (e.g., has recent updates),

has more than one developer, and appears to be applying good practices

\* In general, prefer a Rails-specific gem over a generic Ruby gem, and

for JavaScript Node.js packages prefer a Ruby gem that repackages it.

The repackage will often help make it work more cleanly

with the Rails application, and it also suggests that the package is

a more common one (and thus more likely to be maintained).

\* Check if the gem is thread-safe, in particular, avoid gems that

don't control modifying objects shared between threads.

This is less of an issue today, because in many cases the objects

being modified are not being shared, and threaded implementations

have become common (Heroku encourages them).

Someday we hope to add "have one of our badges" as a preference.

#### License requirements for reused components

All \*required\* reused software \*must\* be open source software (OSS).

It's okay to \*optionally\* use proprietary software and add

portability fixes.

We use 'license\_finder' to help ensure that we're using OSS legally.

We generally use SPDX license expressions to describe licenses.

In general we want to use GPL-compatible OSS licenses.

Acceptable licenses include MIT,

BSD 2-Clause "Simplified" or "FreeBSD" License (BSD-2-Clause),

BSD 3-Clause "New" or "Revised" License (BSD-3-Clause), the

[original Ruby license](https://spdx.org/licenses/Ruby.html) (Ruby) or the

[current Ruby license](https://www.ruby-lang.org/en/about/license.txt)

(the latter includes BSD-2-Clause as an option),

GNU Library or "Lesser" General Public License (any version),

GNU General Public License version 2.0 or later (GPL-2.0+), and the

GNU General Public License version 3.0 ("or later" or not)

(either GPL-3.0 or GPL-3.0+).

We can use Apache License 2.0 (Apache-2.0)

and GPL-2.0 exactly (GNU GPL version 2.0 only),

but Apache-2.0 and GPL-2.0 (only) have potential compatibility issues.

First check if that Apache-2.0 and GPL-2.0 only components are

in separate executables (if so, no problem).

Most software licensed using the GPL version 2.0 is actually

GPL-2.0+ (GPL version 2 or later), and GPL version 3 is known to be

compatible with the Apache 2.0 license, so this is not a common problem.

For more on license decisions see doc/dependency\_decisions.yml;

you can also run 'rake' and see the generated report

license\_finder\_report.html.

Once you've checked, you can approve a library and its license with the

this command (this quickly modifies doc/dependency\_decisions.yml;

you can edit the file as well):

~~~~

license\_finder approval add --who=WHO --why=WHY GEM-NAME

~~~~

### Updating reused components

For stability we set fixed version numbers for Ruby and the Ruby gems.

Please update only one or few components in each commit, instead of

"everything at once". This makes debugging problems much easier.

In particular, if we find a problem later, we can

use "git bisect" to easily and quickly find the cause.

#### Updating Ruby gems

We use the bundler Ruby gem package management system (<http://bundler.io>);

file 'Gemfile' lists direct gem dependencies; 'Gemfile.lock' lists them

transitively.

In short, we have strong package management

over the exact versions used for each gem, and we

can easily update our dependencies.

That's important, because we transitively depend on over 150 gems.

The default 'rake' task and the variant used by our

continuous integration (CI) suite includes the rake 'bundle\_audit' task.

This reports if a Ruby gem we use has a publicly known

vulnerability listed in the National Vulnerability Database (NVD).

Thus, simply running 'rake' will immediately warn you if there is a

publicly known vulnerability in the version of a gem we use.

Obviously, if there is a known vulnerability you \*definitely\* need

to update that gem.

To find all outdated gems, use the 'bundle outdated' command.

Many of the gems named "action..." are part of rails, and thus, you should

update rails to update them.

You \*must\* review gems if you have reason to believe they might be malicious.

Occasional spotchecks are encouraged.

For more information, see the archived blog article

["Being paranoid with Ruby gems" (Gemnasium)](https://web.archive.org/web/20180130214924/https://gemnasium.com/blog/being-paranoid-with-ruby-gems/).

You can see the changes by doing the following.

Create a subdirectory ("mkdir temp"), run "cd temp", and then

run this (you can omit "-v VERSION" if you're just getting the latest):

~~~~

gem fetch -v OLD\_VERSION GEM\_NAME

gem unpack GEMNAME-OLD\_VERSION.gem

gem fetch -v NEW\_VERSION GEM\_NAME

gem unpack GEMNAME-NEW\_VERSION.gem

diff -ur GEMNAME-OLD\_VERSION GEMNAME-NEW\_VERSION

~~~~

I recommend updating in stages (instead of all at once) since this

makes it easier to debug problems (if any). Here is a suggested order,

though these are only suggestions.

In general, update one or a few related things, use rake to check it,

and then commit.

First, edit the file Gemfile to update a gem that is depended on directly.

It's best to do this one line at a time if you can (to only update a

single direct dependency), though in some

cases it may be necessary to update several at the same time.

If you see gems with names beginning with "active",

those are gems in rails; update the Gemfile to change the version of

the 'rails' gem instead. Once edited, run this to install the new

gem version and also update all of its transitive dependencies:

~~~~

bundle update GEM\_NAME && rake

~~~~

If that works commit the change with a "git comment -as" with summary

'Update gem GEM\_NAME (OLD\_VERSION\_NUMBER-&gt;NEW\_VERSION\_NUMBER)'.

Updates sometimes fail. In particular, sometimes one gem has been

update but a related gem is temporarily incompatible.

You can also update gems that are only indirectly depended on.

(These are gems listed in Gemfile.lock, but \*not\* listed in Gemfile.)

You can just run "bundle update" to update them all at once.

Then run "rake" to make sure it works;

and if it does, use "git commit -a" to commit that change.

When all the main dependencies are up-to-date, it's a good idea to

do this occasionally.

It's important to try to stay relatively up to date.

However, it's usually not possible to keep

everything perfectly up-to-date, because

different gems' specifications forbid it.

Also, the Ruby ecosystem is fairly fast-moving,

so it doesn't take long at all for even a direct dependency to go out of date.

In short, it's normal to have some gems that are not the latest.

The key is to replace gem versions that have security vulnerabilities,

and to not get \*too\* far behind, because if it's too far back then it's

harder to update.

If you are very far behind, it may be better to update in stages

(using intermediate versions), but avoid needing to do that.

Historically the gems that cause trouble updating in this app are

github\_api, octokit, and the various "pronto" gems.

If you update Rubocop you're likely to need to make a number of changes

(either to code or to disable a new checker).

If an update fails, you can use this to undo it:

~~~~

git checkout -- Gemfile Gemfile.lock

bundle update

~~~~

You can learn more about "bundle update" by running

"bundle help update".

Again, you \*must\* run 'rake' after updating; this will run the

regression tests, check the licenses (transitively, which is important because

sometimes library updates add new dependencies), and so on.

One of the main reasons we maintain a strong testsuite, and have

a number of other automated checks, is so that we can quickly and

confidently update gems.

Updates should be handled as a separate commit from functional improvements.

One exception: it's okay if the commit includes

both a component update and the minimum set of code changes to

make the update work.

Specially check any update of nokogiri or loofah;

we specially analyze those to prevent vulnerabilities in our

[security assurance case](./doc/security.md).

#### Updating Ruby (and handling Ruby updates)

Ruby itself can be updated.

You can change the Ruby version yourself, and

when you use git pull the current version of Ruby in use could change.

There are extra steps needed when Ruby is updated; here they are.

In particular, if you try to run commands and you see errors like this:

~~~~

rbenv: version \`2.3.9' is not installed (set by .../.ruby-version)

~~~~

You \*need\* to update your local installation of Ruby.

This typically happens after you've done a `git pull`;

what has happened is that the Ruby version changed in file `.ruby-version`.

To fix this, just run this command:

~~~~sh

./update-ruby

~~~~

In general, the current version of Ruby used in the project is stored in

the file `.ruby-version` in the project's top directory.

For example, file Gemfile declares that the ruby version

used is whatever is in ".ruby-version".

This `.ruby-version` file is controlled by git.

It's wise to occasionally check the

[Ruby site for the latest version of Ruby](https://www.ruby-lang.org/en/downloads/).

If you want to change the current version of Ruby used in the project,

use `cd` to go this project's top directory,

and use 'git pull' to ensure this branch is up-to-date.

You should normally use `git checkout -b NEW\_BRANCH\_NAME` for the new branch.

Then run the following command:

~~~~sh

./update-ruby NEW\_VERSION\_NUMBER

~~~~

Note at the end of this script a `git commit -as` will be initiated

if everything worked correctly. Next you should update the CircleCI build

image and `.circleci/config.yml` as indicated in the dockerfiles

[README](dockerfiles/README.md). You will then need to commit those changes.

and run `git push`.

For more details about updating Ruby versions with rbenv, see

<https://github.com/rbenv/ruby-build> and

<http://dan.carley.co/blog/2012/02/07/rbenv-and-bundler/>.

Note that 'rbenv install 2.3.0' is equivalent to the longer

<tt>ruby-build 2.3.0 $HOME/.rbenv/versions/2.3.0</tt>.

If you update Ruby but don't update the parser gem

(e.g., a new version may not be available yet), you may get a number

of warnings from the static analysis tools that we run via rake.

Where possible, consider updating the parser gem as well.

These warnings will look like these:

~~~~

warning: parser/current is loading parser/ruby22, which recognizes

warning: 2.2.x-compliant syntax, but you are running 2.3.0.

warning: please see https://github.com/whitequark/parser#compatibility-with-ruby-mri.

~~~~

Once the component update has been verified,

it can be checked in as a new commit.

## Keeping up with external changes

The installer adds a git remote named 'upstream'.

Running 'git pull upstream master' will pull the current version from

upstream, enabling you to sync with upstream.

You can reset this, if something has happened to it, using:

~~~~sh

git remote add upstream \

https://github.com/coreinfrastructure/best-practices-badge.git

~~~~

If the version of Ruby has changed (in the Gemfile),

use the 'Ruby itself can be updated' instructions.

If gems have been added or their versions changed, run

"bundle install" to install the new ones.