Webdev Bootcamp Documentation

Release 0.3beta

Dave Dash, Laura Thompson, Jeff Balogh, Mozilla Webdev Team

Sep 27, 2017

Contents

Hello! Welcome to the Mozilla Webdev Bootcamp, an informal guide to how web development is done at Mozilla.

If you're a new contributor / new employee, you can start out by reading the *New Contributor Guide*. If you're already a Webdev contributor, you can find generally-useful info, such as style guides, in the *Reference*.

CHAPTER 1

New Contributor Guide

First things first: Welcome to Mozilla! We're glad to have you here, whether you're considering volunteering as a contributor or are employed by Mozilla.

Our goal here is to get you up and running for contributing as a web developer. This guide attempts to be generic enough to be useful to any webdev project, but some details (especially around software you need installed) may vary among projects. Any project you're contributing to should have documentation that explains these details in, uh, detail.

Let's get started!

The Mozilla Webdev Group

Webdev is an informal group of people across the Mozilla project that are interested in or work in web development. Webdev isn't linked to a specific team and welcomes everyone who wants to be a part of it to join.

As a new contributor, the primary use of the Webdev group will be to help answer questions you have or to help you find interesting projects to work on. Later on, you can start attending some of our online meetings to meet the rest of the group and keep up to date with what everyone else is up to.

How to Talk to Us

There's two main channels of communication for the group:

- The #webdev channel on irc.mozilla.org.
- The dev-webdev@lists.mozilla.org mailing list.

If you ever have any question, regardless of how difficult it is, you can share it on those channels and someone should help you out. *Don't be afraid to ask questions!* You're trying to help us, so it's only fair that we try to help you in return.

Find a Mentor!

It's also a good idea to find someone to mentor you as a new contributor. Having someone you can personally ask for help from is incredibly helpful in finding your way around.

If you're a new employee, your mentor may be your manager, or it may be a coworker. If you're a volunteer, ask someone who works on the project you want to contribute to, and if they cannot mentor you themselves, they should be able to direct you to someone who can.

Accounts

We use a few websites to manage our code and bugs, and while it is possible to get by without signing up for these sites, it's *strongly* recommended that you create accounts on these websites.

Github

Most Webdev projects are hosted on Github. Github provides hosting for our source code, as well as tools we use for collaboration and code review. Github is based on git, a distributed version control system that lets us track the changes we make to our code.

Once you've created a Github account, you can check out the Github help site for guides on the basics of using git and Github.

See also:

Mozilla on Github Mozilla's organization account on Github.

Bugzilla

Bugzilla is the issue-tracking system that the entire Mozilla Project uses. The vast majority of Webdev projects use Bugzilla to keep track of any planned changes or bugs with our websites.

As a new contributor, Bugzilla is a useful tool for finding known issues that you can help fix or finding planned work you want to take on. In order to assign a bug to yourself or to post a comment on a bug, you'll need to create a Bugzilla account. An account also allows you to "CC" yourself on bugs that you are interested in, so that you receive emails when those bugs are changed.

After you've been using Bugzilla for a while as a community member, it's worthwhile applying for expanded permissions. The editbugs permission allows you to assign bugs to yourself and resolve them, for example. See the Bugzilla Permissions Page for details. Note that new employees get this permission automatically, no need to ask for it.

Note: It's highly recommended to add your IRC nickname to your real name within Bugzilla to make it easy for others to auto-complete your name.

The standard format is to follow your real name with your IRC name, preceeded by a colon, surrounded by square brackets. For example: `Cave Johnson [:withthelemons]`.

Software and Tools

The software you'll need to download and install on your computer in order to contribute varies between projects; please refer to the documentation for the project you want to contribute to for details.

The following information is a generic description of software or tools that you'll most likely need regardless of the project you work on.

Operating Systems: Windows, Linux, or OS X?

Most of our sites are developed on Mac OS X or Linux, and deployed on servers running Linux. If you are a Windows user, you may want to use a program like VirtualBox to create a virtual machine running a Linux-based operating system. The rest of this guide assumes you are using an OS X or Linux-based operating system.

If you are running Mac OS X, most of the software mentioned here can be installed using the Homebrew package manager.

Git

Git is a distributed version control system. It tracks the history of changes we make to our code, which allows us to see how the code has changed over time. Git also makes it very easy for multiple people to work on the same code at the same time and merge their changes together at the end.

See also:

help.github.com A great guide to getting start with Git and Github, which hosts most of our git repositories.

Github for Windows A Windows program for interacting with Github as an alternative to using git in a terminal. Useful if you are not used to using a terminal yet.

Github for Mac A Mac OS X program for interacting with Github as an alternative to using git in a terminal. Useful if you are not used to using a terminal yet.

Python

Python is a programming language that many of our websites use for their backend code. Most of our Python-based sites are implemented using Django, a Python-based framework for making websites.

Most of our Python-based sites are developed to run under Python 2, and most of our servers run the sites on Python 2.

See also:

The Hitchhiker's Guide to Python A useful guide for beginner and expert Python developers. If you need to install Python on your computer, this guide will help!

Node.js

Node.js is a JavaScript-based runtime for building network applications, including websites. An increasing amount of Mozilla projects are being written as Node applications.

See also:

nodejs.org Downloads The official Node.js download page, which includes installers for Windows and Mac OS X.

Miscellaneous

The following is a list of software you probably need that don't merit a heading:

- GNU gettext: Used for localization support on many websites. Can usually be installed via your package manager under the name `gettext`.

Finding a Project

Before you can contribute to Mozilla, you need to find a project that you're interested in contributing to. There are many different projects in Webdev, both full websites and libraries to help others make websites.

If you don't already have a project in mind, there are a few ways to find projects to contribute to:

- The GetInvolved page lists a curated selection of projects as well as information on who to get in contact with if you want to help.
- If you email the dev-webdev@lists.mozilla.org mailing list with some information on what your skills are and where your interests lie, someone may reply with a recommendation.

Getting set up

Once you've identified the project you want to work on, you should set up a development instance of the site. All projects have (or should have) a README file that either describes how to set up the site for development, or links to documentation that does.

Find a mentor

You may also find it useful to find someone who is working on or responsible for the project you want to contribute to and asking if they can help you find a task to work on and answer any other questions you have. If there's no information in the README for a project about who works on it, you can check the commit history (available in Github by clicking "# commits" near the top of the page) to find who recently worked on the project, or by asking the *Webdev group* who is responsible.

How to contribute

Once you're set up to work on a project, you'll have to find a task to work on and get to work! Each project should have some information on where their tasks are tracked, whether it be in Bugzilla, Github issues, or some other system. If you're having trouble finding this information, try looking:

- For a `CONTRIBUTNG` file in the repository. Many projects use this file to store instructions on how to start as a contributor.
- For a contribute.json file, which contains a machine-readable dump of links and information on how to start contributing to a project.

Development Process

While the details vary, there is a general framework for the development process at Mozilla which describes how a change goes from an idea in someone's head to deployed code on a production webserver. This document attempts to describe that process.

Filing a Bug

The first thing that happens is that a bug is filed in the bug tracker of choice for the project. A well-written bug includes:

- A description of the issue, possibly including steps to reproduce or a link to an example if the issue is a problem with the project.
- Information or links to any conversation that is happening outside of the bug, such as a mailing list thread.
- If appropriate, a specified mentor to help new contributors work on the issue.

Depending on the project, the bug may be triaged and assigned a priority and/or milestone, or it may be added to another system for tracking work, such as a kanban board.

Working on the Bug

Either someone will voluntarily take a bug, or, in the case of projects with assigned developers actively working on them, it will be assigned to a developer.

The process of fixing a bug involves:

- Marking the bug as assigned to you so that others do not work on the bug at the same time.
- Creating a feature branch in your version control system to isolate your work from the work of others.
- Making the changes required to fix the issue or implement the new feature.
- Writing automated tests to ensure that your changes work as expected, as well as manually testing on your personal development instance of the project.
- Submitting your changes for review by another developer on the project, and updating your changes in response to the review.
- Merging your feature branch back into the main branch used for development.

Git and Github

For projects using Git and Github (which is most Webdev projects), the process can be explained in more detail:

- On Github, ensure you have forked the repository for your project to your own account and have added it as a remote to your repository.
- Identify the main development branch for your project. This is usually the `master` branch.
- Make sure the current branch is the development branch and create a new branch off of it for your feature.
- Once your work is committed and ready for review, push the branch to your fork on Github and submit a pull request.
- If you know who should review your change, add a comment to your pull request with their `@Username` in it and ask for a review (often abbreviated as `r?`).

See also:

Glossary A glossary of specialized terms used within Webdev, including some abbreviations used for code review, such as `r?`, `r+`, and `r-`.

Github Flow A process for branching, reviewing, and merging code that is very similar to the process above.

Testing and Deployment

Once a change is merged into the codebase, it needs to be tested on an actual server and then deployed to production. Typically the lead developer on a project will handle this process and let you know if any effort on your end is required.

A bug is usually marked as resolved when it is merged into the codebase. Depending on the issue tracker being used, the bug may also be marked as verified once the changes are tested and approved.

Next steps

At this point you should have all the information and tools you need to make your first contribution to Mozilla! Once you've submitted your work and gotten it merged, it's time to celebrate: you've earned it!

As you continue to contribute, you may want to check out the *Reference* to find generally-useful information for contributors of all levels.

Good luck!

Best Practices

Git and Github

- Create a separate pull request for each bug.
- If you have multiple commits in one pull request, squash them into one. For complex patches, having multiple commits might make the review easier.
- If you are working with multiple bugs at a time, make sure you create a separate branches for each bug.
- If you need to make changes to the commit, perhaps after feedback, amend the original commit and comment in the pull request what you've changed. Don't create a new pull request or commit.
- Include the bug number in your commit & pull request title. If possible follow this style -*Some text description (bug XXXXXXX)*.
- Don't work on an assigned bug. If you see the bug assignee hasn't responded for a while (two weeks), then ask for permission if you can work on that bug.
- For frontend pull requests, attach screenshots if relevant.
- Put a link to the PR in the bug as a comment.

CHAPTER 2

Reference

This is where all the useful information goes that doesn't fit into the *New Contributor Guide*.

Python Style Guide

This document is a brief set of guidelines for writing Python code for Mozilla Webdev projects. Individual projects may override these rules; make sure you know the standards for your project!

General Guidelines

- Follow PEP8.
- Follow Pocoo's extensions to PEP8, although these are a little less strictly enforced across Mozilla projects.
- Check your code against a linting tool. flake8 is highly recommended for this.

Import Statements

We expand on PEP8's suggestions for import statements.

Import one module per import statement:

```
import os
import sys
```

not:

```
import os, sys
```

Separate imports into groups with a line of whitespace: standard library; third-party; and local imports:

```
import os
import sys

from django.conf import settings

import pyquery

from myapp import models, views
```

Alphabetize your imports, it will make your code easier to scan. See how terrible this is:

```
import cows
import kittens
import bears
```

A simple sort:

```
import bears
import cows
import kittens
```

Imports on top, `from`-imports below:

```
import x
import y
import z
from bears import pandas
from xylophone import bar
from zoos import lions
```

That's loads easier to read than:

```
from bears import pandas
import x
from xylophone import bar
import y
import z
from zoos import lions
```

Lastly, when importing things into your namespace from a package use an alphabetized `CONSTANT`, `Class`, `var` order:

```
from models import DATE, TIME, Dog, Kitteh, upload_pets
```

If possible though, it may be easier to import the entire package, especially for methods as it help answers the question, "where did `you` come from?"

Bad:

```
from foo import you


def my_code():
    you()  # wait, is this defined in this file?
```

Good:

```
import foo


def my_code():
    foo.you()  # oh you...
```

See also:

baked A tool for automatically checking the import order rules listed above.

Whitespace matters

- Use 4 spaces, not 2—it increases legibility considerably.
- Never use tabs—history has shown that we cannot handle them.

Use single quotes unless double (or triple) quotes would be an improvement:

```
'this is good'

'this\'s bad'

"this's good"

"this is inconsistent, but ok"

"""this's sometimes "necessary"."""

'''nobody really does this'''
```

To continue a new line use a `()` not `\`.

Indenting code should be done in one of two ways: a hanging indent, or 4 space indent on the next line.

Good, using hanging indent. Note that the next line is lined up with the previous line delimiter:

```
log.msg('Something long log message and some vars: {0}, {1}'
        .format(variable_a, variable_b))
```

Good using 4 spaces:

```
accounts = PaymentAccounts.objects.filter(
    accounts__provider__type=2,
    something_else=True
)

# A more compact alternative.
accounts = PaymentAccounts.objects.filter(
    accounts__provider__type=2, something_else=True)

accounts = (PaymentAccounts.objects
    .filter(accounts__provider__type=2)
    .exclude(something_else=False)
)
```

Remember that comprehensibility is the goal here. If following one of the rules above would result in less readable code, don't follow it!

HTML Style Guide

This document is a brief set of guidelines for writing HTML for Mozilla Webdev projects. Individual projects may override these rules; make sure you know the standards for your project!

General Guidelines

- Use the HTML5 doctype.
- Make sure your code validates.
- No inline CSS or JavaScript.
- Be semantic.
- Use doublequotes for attributes:

```
<a href="#">Good</a>
<a href='#'>Less Good</a>
```

JS Style Guide

First and Foremost

ALWAYS use JSHint on your code.

Note: There are some exceptions for which JSHint complains about things in node that you can ignore, like how it doesn't know what 'const' is and complains about not knowing what 'require' is. You can add keywords to ignore to a *.jshintrc* file.

Variable Formatting:

```
// Classes: CapitalizedWords
var MyClass = ...

// Variables and Functions: camelCase
var myVariable = ...

// Constants: UPPER_CASE_WITH_UNDERSCORES
// Backend
const MY_CONST = ...

// Client-side
var MY_CONST = ...
```

Indentation

4-space indents (no tabs).

For our projects, always assign var on a newline, not comma separated:

```
// Bad
var a = 1,
    b = 2,
    c = 3;

// Good
var a = 1;
var b = 2;
var c = 3;
```

Use `[]` to assign a new array, not `new Array()`.

Use `{}` for new objects, as well.

Two scenarios for `[]` (one can be on the same line, with discretion and the other not so much):

```
// Okay on a single line
var stuff = [1, 2, 3];

// Never on a single line, multiple only
var longerStuff = [
    'some longer stuff',
    'other longer stuff'
];
```

Never assign multiple variables on the same line

Bad:

```
var a = 1, b = 'foo', c = 'wtf';
```

DO NOT line up variable names

Bad:

```
var wut    = true;
var boohoo = false;
```

Semi-colons

Use them.

Not because ASI is black-magic, or whatever. I'm sure we all understand ASI. Just do it for consistency.

Conditionals and Loops

```
// Bad
if (something) doStuff()

// Good
if (something) {
    doStuff();
}
```

Space after keyword, and space before curly

```
// Bad
if(bad){

}

// Good
if (something) {

}
```

Functions

Named Functions

There's no need to explicitly name a function when you're already assigning it to a descriptively named symbol:

```
var updateOnClick = function() { ... };
```

...or...

```
var someObject = {updateOnClick: function() { ... }
```

Most modern JS engines will infer the name *updateOnClick* for the above anonymous function and use it in tracebacks.

Of course, if you're passing a nontrivial function as an argument, you should still contrive to name it somehow. The meaning here would be needlessly obscured if the anonymous function were, for example, 10 lines long:

```
.forEach(function() { ... })
```

In such cases, either name the function, or pass a descriptively named symbol that points to a function.

Whitespacing Functions

No space between name and opening paren. Space between closing paren and brace:

```
var method = function(argOne, argTwo) {

};
```

Anonymous Functions

Anonymous functions are fine if they have a small amount of code in them. See the *Named Functions* section for info about inferred function names for anonymous functions.

Operators

Always use ===.

Only exception is when testing for null and undefined.

Example:

```
if (value != null) {

}
```

Quotes

Always use single quotes: `'not double'`

Only exception: `"don't escape single quotes in strings. use double quotes"`

Comments

For node functions, always provide a clear comment in this format:

```
/* Briefly explains what this does
 * Expects: whatever parameters
 * Returns: whatever it returns
 */
```

If comments are really long, also do it in the `/* ... */` format like above. Otherwise make short comments like:

```
// This is my short comment and it ends in a period.
```

Ternaries

Try not to use them.

If a ternary uses multiple lines, don't use a ternary:

```
// Bad
var foo = (user.lastLogin > new Date().getTime() - 16000) ? user.lastLogin - 24000 :
↪'wut';

// Good
return user.isLoggedIn ? 'yay' : 'boo';
```

General Good Practices

If you see yourself repeating something that can be a constant, refactor it as a single constant declaration at the top of the file.

Cache regex into a constant.

Always check for truthiness:

```
// Bad
if (blah !== false) { ...

// Good
if (blah) { ...
```

If code is really long, try to break it up to the next line or refactor (try to keep within the 80-col limit but if you go a bit past it's not a big deal). Indent the subsequent lines one indent (2-spaces) in.

If it looks too clever, it probably is, so just make it simple.

CSS Style Guide

Terminology

Just so we all know what we're talking about, a CSS *rule* comprises one or more *selectors* followed by a *declaration block* consisting of one or more *declarations*. A declaration comprises a *property* and a *value* (some properties accept multiple values).

A rule in CSS looks like:

```
selector {
    property: value;
}
```

The basics (tl;dr)

- Multi-line rules, not single line.
- Spaces, not tabs.
- Four space indentation.
- Order declarations alphabetically (with some exceptions).
- Use the simplest, least specific selector possible.
- Make meaningful names, not presentational.
- All lowercase for classes and IDs, no camelCase.
- Separate words in classes and IDs with hyphens, not underscores.
- ID selectors are allowed but use them sparingly and appropriately.
- Don't use `!important`.
- You can use pixels for `font-size`, but you don't have to.
- Use unitless `line-height`.
- Group related rules into sections.
- Order sections and rules from general to specific.
- Use Stylus but write it like plain CSS.

General guidelines

If a length value is `0`, do not specify units; `0px` and `0in` are exactly equal because zero is zero.

Omit leading zeroes in decimal units, e.g. `.75em`, not `0.75em`.

When using experimental properties with vendor prefixes, always include the unprefixed declaration as well, and always last in the list. An exception would be a strictly vendor-specific property with no standard implementation, like `-webkit-font-smoothing`.

When declaring gradient backgrounds, you don't need to include the old Webkit syntax unless, for some reason, you need to target old versions of Safari.

Practice progressive enhancement! Include solid fallback colors for old browsers that don't support `rgba()` or gradients:

```
.widget {
    background: #ccc;
    background: linear-gradient(rgba(155, 155, 155, .25), rgba(155, 155, 155, .5));
}
```

Hiding content

Consider screen readers when hiding content. Screen readers will not read content that is `display: none;` or `visibility: hidden;`. Hiding something visually but not from screen readers requires a bit more CSS. Be conscientious when choosing your hiding technique.

Simple selectors

Use the least specific selector required to do the job.

Favor classes over IDs.

IDs aren't forbidden, but reserve them for either major blocks (site header, main nav, etc) or very specific singletons that are truly unique. Hanging styles from ID selectors can lead to specificity wars requiring ever more powerful selectors to override previous styling.

Avoid qualifying class names with type selectors. E.g. `.widget` is better than `div.widget`.

Avoid adjoining classes unless there's a good reason to do it.

Sometimes different elements share a class but have an additional modifier class that extends the meaning and changes the styling. E.g. `.message.error` and `.message.success`. You could simply take advantage of the cascade order and declare the `.error` and `.success` classes after the `.message` class, but you can't always ensure classes will be kept in the proper cascade order (rules get moved around as style sheets are refactored, or they appear in different style sheets imported at different points, etc). In those cases you might prefer to create a single, more explicit modifier class rather than rely on adjoined classes, e.g. `.message-error` and `.message-success`.

However, don't try to CLASS ALL THE THINGS by creating a unique class for every single element just for an easy style hook, or by creating oodles of generic classes to apply fine-grained styling at the expense of requiring a string of classes on each element in the markup.

Bad:

```
/* Too specific */
.module-news-title-main {
    font-family: 'League Gothic', sans-serif;
}

.module-news-title-sub {
    font-family: Georgia, serif;
}

/* Too generic (and presentational) */
```

```
.size20 {
    font-size: 20px;
}

.size16 {
    font-size: 16px;
}
```

It's usually better to style elements based on their context than to try to make every possible style rule free-standing and every element 100% reusable in any context on any page. Use descendant selectors judiciously but keep them simple.

Good:

```
.module-news h2 {
    font: 20px 'League Gothic', sans-serif;
}

.module-news h3 {
    font: 16px Georgia, serif;
}
```

Avoid `!important` in CSS unless absolutely necessary, **which it almost never is**.

Some off-the-shelf frameworks/libraries/plugins include `!important` styles of their own that you might have to override with another `!important` style, or they write out inline styling into the DOM that you have to override in a style sheet with `!important`. (One could consider these transgressions to be warning signs of a poorly made framework/library/plugin and you might want to seek better options that don't force you to junk up your CSS.)

Fonts and typography

It's alright to use pixels for `font-size`.

For many years CSS authors eschewed pixels and favored relative units for font sizing because IE 5 and 6 couldn't scale text set in absolute units (like `px`). All modern browsers can scale text in any unit (or zoom the entire page) so this is no longer a driving concern, unless you're catering to versions of IE from the previous century.

There are times when it's better to use relative `font-size` units like *em's or percentages. You may have a bit of text that should be sized proportionally to a parent element whose font size is unknown. Some responsive designs call for globally resizing text in different layouts (e.g. globally bigger text for mobile), in which case it's simpler to change a single base size on a parent than to re-declare the absolute ''font-size'* of each element.

Just remember that relative font sizes inherit and cascade so you can end up with magic numbers like `.6875em`. The `rem` unit (root em) can avoid the cascade problems, but older browsers don't support rems and IE9 and 10 don't support them in shorthand `font` declarations (fixed in IE11). It's always something.

If you use `rem``s for font sizing, include a ``px` or other fallback for older browsers.

Use unit-less line-height. It doesn't inherit a percentage value of its parent element, but instead is based on a multiplier of the font-size, whatever that may be. E.g. `line-height:  1.4;` or in a shorthand `font` property: `font: 14px/1.4 sans-serif;`. Don't use an absolute unit like `px` for `line-height`; it creates more problems than it solves.

Use "bulletproof font syntax" for webfonts. You usually don't need to include SVG font files unless your project needs to target older versions of WebKit. For modern browsers, TTF + WOFF is sufficient, as well as EOT for older versions of IE (which may also be optional, depending on your target audience). Example:

```
@font-face {
    font-family: 'Open Sans';
    font-style: normal;
    font-weight: normal;
    src: url(/media/fonts/OpenSans-Bold-webfont.eot?#iefix) format('embedded-opentype
↪'),
         url(/media/fonts/OpenSans-Bold-webfont.woff) format('woff'),
         url(/media/fonts/OpenSans-Bold-webfont.ttf) format('truetype');
}
```

Formatting CSS

When a rule has a group of selectors separated by commas, place each selector on its own line.

The opening brace ({) of a rule's declaration block should be on the same line as the selector (or the same line as the last selector in a group of selectors).

Use a single space before the opening brace ({) in a rule, after the last selector.

Put each declaration on its own line.

Indent the declaration block one level relative to its selector.

Use a colon (:) immediately after the property name, followed by a single space, then the value.

Terminate each declaration with a semicolon (;), including the last declaration in a block.

Put the closing brace (}) on its own line, aligned with the rule's selector.:

```
.selector-1,
.selector-2 {
    property: value;
    property: value;
}

.selector-3 {
    property: value;
}
```

When you have a block of related rules, each with one or two declarations, you can use a single-line format without any blank lines between rules. It makes the block of related rules a bit easier to scan. In this case include a single space after the opening brace and before the closing brace. Add spaces after the selector to align the values.:

```
.message-success { color: #080; }
.message-error   { color: #f00; }
.message-notice  { color: #00f; }
```

Or:

```
@keyframes bounce {
    0%   { bottom: 300px; }
    25%  { bottom: 30px; }
    50%  { bottom: 100px; }
    100% { bottom: 30px; }
}
```

When possible, limit line lengths to 80 characters. This improves readability, minimizes horizontal scrolling, makes it possible to view files side by side, and produces more useful diffs with meaningful line numbers. There will be

exceptions such as long URLs or gradient syntax but most rules in CSS should fit well within 80 characters even with indentation.

Long, comma-separated property values – such as multiple background images, gradients, transforms, transitions, webfonts, or text and box shadows – can be arranged across multiple lines (indented one level from their property).:

```
.selector {
    background-image:
        linear-gradient(#fff, #ccc),
        linear-gradient(#f3c, #4ec);
    box-shadow:
        1px 1px 1px #000,
        2px 2px 1px 1px #ccc inset;
    transition:
        border-color .5s ease-in,
        opacity .1s ease-in;
}
```

For vendor prefixed properties, use spaces to align the values, keeping the property names left-aligned as usual:

```
.selector {
    -webkit-box-shadow: 1px 2px 0 #ccc;
    -moz-box-shadow:    1px 2px 0 #ccc;
    -ms-box-shadow:     1px 2px 0 #ccc;
    -o-box-shadow:      1px 2px 0 #ccc;
    box-shadow:         1px 2px 0 #ccc;
}
```

Or, when the value has the prefix:

```
.selector {
    background: -webkit-linear-gradient(to bottom, #fff, #000);
    background:    -moz-linear-gradient(to bottom, #fff, #000);
    background:     -ms-linear-gradient(to bottom, #fff, #000);
    background:      -o-linear-gradient(to bottom, #fff, #000);
    background:         linear-gradient(to bottom, #fff, #000);
}
```

Also notice this implies a specific order for vendor prefixes from longest to shortest, mostly just for readability and consistency. It's convenient that the unprefixed version, which always appears last, is shortest by default.

Whitespace

Use spaces (or soft-tabs) with a four space indent. Never use tabs.

Eliminate trailing whitespace at the end of lines. Blank lines should have no spaces.

Include one blank line between rules.

Include a single blank line at the end of files.

Property ordering

Order declarations alphabetically by property name (from A to Z), with a few exceptions:

- Keep vendor prefixed properties together and ordered by length, with the unprefixed property last (see the earlier example).

- Keep positioning properties together, namely `position`, `top`, `right`, `bottom`, `left`, and `z-index`.
- You can optionally keep `width` and `height` together if you're declaring both.
- You can optionally keep some type-related properties together when that's sensible, such as `font-size`, `text-transform`, and `letter-spacing`.

Many developers settle into their own system for ordering declarations based on relevance, logical groupings, line length, or just semi-random as they're added. Although alphabetical ordering can defy any other logical ordering – adjacent properties may have nothing in common while closely related properties can be spread far apart – at least there's no ambiguity about the alphabet and it's easy to enforce the guideline across a team.

After all that, it's actually pretty rare for a single rule to hold so many declarations that ordering becomes too much of a hassle. When in doubt, alphabetize.

Naming conventions

Names should be semantically meaningful, descriptive of the element's content, purpose, or function, not its presentation.

Bad: `.big-blue-button`, `.right-column`, `.small`
Good: `.button-submit`, `.content-sub`, `.field-note`

Many CSS frameworks, such as Twitter's Bootstrap and Zurb's Foundation, define a lot of presentational classes for things like column widths, font sizes, and button styles. If you're using such a framework, you can use those classes as mixins in a preprocessed style sheet, rather than littering markup with presentational names.

Bad:

```
<div class="author-bio col-md-3 col-md-offset-2">
```

Better:

```
.author-bio {
    .col-md-3;
    .col-md-offset-2;
}
```

Note: For very large and complex sites, excessively repeating common declarations can lead to a lot of redundancy and CSS bloat. In those cases you can get better performance with some presentational classes if it leads to a significantly lighter style sheet. E.g. it can speed up a site considerably to specify column widths with a class in a few dozen HTML templates than to repeat the same width, float, and margin declarations a thousand times in CSS. We don't have many sites operating on the kind of scale that warrants that approach, but there are always exceptions.

Names should be as short as possible and as long as necessary. Clarity is key. E.g. `.prime-nav` is better than `.primary-navigation`, but `.article-author` is better than `.art-auth`.

Avoid overly abstract names that require a cheat sheet to understand.

Bad: `.color12`, `.r2-c6`, `.v`

Names should be all lower case, no camelcase.

Bad: `.badClassName`, **Better:** `.betterclassname`

Separate words with hyphens, not underscores.

Bad: `.bad_class_name`, **Best:** `.best-class-name`

Use US English spellings (sorry, rest of the world). CSS itself follows US English so it's inconsistent to mix standard spellings like `color: #000;` with classes like `.colour-picker`.

Style sheet organization

It's hard to standardize on a particular structure for style sheets, especially when it comes to preprocessors and other tools that import and concatenate separate files. But that doesn't mean we can't try to stick to some basic principles:

- Group related rules into sections.
- Give each section a title in a comment.
- Order rules in a section from general to specific (remember the cascade).
- Order sections in a style sheet from general to specific.
- Add three blank lines between the last rule in a section and the next section's title (clear separation between sections makes scanning easier).

A typical style sheet might be structured from top to bottom like so (only an example):

1. A preamble comment with a table of contents and other info.
2. *Fonts* (webfonts need to be declared first so you can reference them further down the cascade).
3. *Reset* (global resets should be first so you can override them later).
4. *Base elements* (no IDs or classes here, just general elements like links, headings, lists, forms).
5. *Base layout* (setting up the general page layout for the entire site, arranging basic blocks like a global header, global footer, main content areas and sidebars).
6. *Global components/modules* (general purpose widgets that will be reused like button links, a sidebar menu, pagination, breadcrumbs, footnotes, a search form, error messages).
7. *Specific page layout* (pages that deviate from the base layout and need more more specific styling, like a home page, contact page, gallery page).
8. *Specific components/modules* (less generic, self-contained widgets that need more specific styling like a download button, a contact form, or a carousel).

Many (most) websites end up with a few one-off pages or subsets of pages that require more specific styling, rules used only on those pages and nowhere else. To avoid dumping everything into a single ever-expanding CSS file, it's usually best practice to split it into separate style sheets and combine them server-side so each page gets just the rules it needs.

For responsive layouts, collect all the rules for a given medium/viewport into a single media query rather than repeat the same media query several times throughout a style sheet.

Preprocessors

All of the above guidelines (those relating to formatting and organization, at least) apply equally to vanilla CSS and to style sheets authored for a preprocessor. Here are some additional guidelines specific to preprocessors:

Keep nesting simple

Nested rules in pre-processed CSS turn into descendant selectors in the generated style sheet. The deeper the nesting, the more complex and specific the selector will be. Don't nest rules unless necessary for context and specificity, and don't nest rules just to group them together (use sectioning comments for grouping).

All the declarations for the parent element should come before the nested rules. Include a blank line before each nested rule to separate it from the rule or declaration above it.

Really Bad:

```
.wrapper {
    #sidebar {
        .modules {
            .module-news {
                background: #ccc;
                h2 {
                    font-size: 18px;
                }
                padding: 10px;
            }
        }
        width: 320px;
        float: right;
    }
}
```

Good:

```
.module-news {
    background: #ccc;
    padding: 10px;

    h2 {
        font-size: 18px;
    }
}
```

Try to limit nesting to one or two levels. If you find yourself nesting rules deeper than three levels, you probably need to reconsider your approach.

If you wouldn't need to use a descendent selector in vanilla CSS, you probably don't need to nest it in a pre-processed style sheet.

```
/* Unnecessary nesting; the nested class doesn't need the specificity */
.module {
    background: #ccc;
    padding: 10px;

    .module-title {
        font-size: 18px;
    }
}
```

```
/* Two rules for two elements */
.module {
    background: #ccc;
    padding: 10px;
}

.module-title {
    font-size: 18px;
}
```

If the parent rule has no declarations, nesting isn't necessary at all. If you need the specificity, use an ordinary descendant selector.

```
/* Especially unnecessary nesting */
.breadcrumbs {
    ul {
        li {
            display: inline;
            list-style: none;
        }
    }
}

/* Better */
.breadcrumbs ul li {
    display: inline;
    list-style: none;
}

/* Best */
.breadcrumbs li {
    display: inline;
    list-style: none;
}
```

LESS vs. Stylus

Many current and past Mozilla websites use LESS as a CSS preprocessor. However, LESS appeared to be stagnating for a time and some projects moved toward Stylus as an emerging contender under more active development (and also because Stylus has some extra features and shares some traits with Python). LESS has since resumed more active development, but in an effort to standardize across Mozilla webdev, we're making the call: it's Stylus for us.

New Mozilla webdev projects should use Stylus for CSS preprocessing (or stick with vanilla CSS). Sites currently using LESS should work toward converting to Stylus as soon as practically feasible (tools can help). LESS isn't forbidden, but prefer Stylus if you have a choice.

A Few Words About Stylus

On the Stylus website, right at the top of the home page, the creators crow a lot about how all these required CSS syntax bits, like braces and colons and semicolons, are optional in Stylus, as if they're a great annoyance that we've all been clamoring to abolish for years.

Well, Stylus still generates ordinary CSS in the end, and inserts all those optional doodads on your behalf anyway because they're *still required in CSS*. Just because Stylus makes them optional doesn't mean we should omit them,

especially if they make style sheets easier to read. For the sake of readability and smoother collaboration, we should try to make CSS look like CSS.

Format your Stylus-flavored pre-processed files as if you were formatting vanilla CSS. Do use mixins, variables, functions, etc. and take advantage of all the flexible goodness Stylus offers, but it should still read like a CSS document.

- Use CSS syntax (Stylus allows it).
- Include colons, semi-colons, and braces.
- Identify variables with a dollar sign (`$`). It's optional in Stylus but makes variables easier to spot by humans.

Bad (though valid in Stylus):

```
.module
    background light-background
    h2
        font-size h-medium
```

Good (and still valid in Stylus):

```
.module {
    background: $light-background;
    h2 {
        font-size: $h-medium;
    }
}
```

A Note on Sass/SCSS/Compass

Very few (if any?) Mozilla projects use Sass because it requires Ruby. While Sass is a fine tool, and can be awesome in combination with Compass, adding Ruby to our dev stack is a bridge too far. Sorry Rubyists; we're a Python shop.

Even so, all the same formatting and organizational guidelines can apply just as well to Sass/SCSS. Live long and prosper.

Validate!

Validate your CSS with the W3C's online tool or equivalent.

Validation tools may report errors or give warnings for vendor prefixes, as they should. It's something to be mindful of but it's perfectly fine to use prefixed properties if you're doing it right.

Validation *warnings* are very different from validation *errors*. You should take warnings under consideration and address them if needed, but errors are real problems that you need to fix.

If you're using a preprocessor you'll obviously only be able to validate the generated plain CSS, which can make it harder to track down where the errors appear in the source files. A well organized style sheet can ease the pain.

A Note on CSS Lint

CSS Lint is a useful tool and we recommend it, but take its results with a grain of salt. Many of Lint's rules are phrased like absolute edicts when they're more like soft warnings of things to be mindful of (e.g. "Don't use too many floats"). Lint also forbids some things we expressly allow in our own guidelines (e.g. "Don't use ID selectors"). If your file gets a slew of warnings from CSS Lint that doesn't mean it's bad, just be able to justify your decisions.

This shortcut to CSS Lint disables some of the more stringent rules we don't necessarily abide.

FAQ

Q: [insert question]

A: It depends.

Internationalization (i18n) and Localization (L10n)

Mozilla is a global community, and most of our websites are localized, meaning that they are available in multiple languages for users across the world.

Internationalization is the process of designing software so that it may be adapted to different languages and regions, whereas localization is the actual process of adapting that software to a specific locale.

This document describes what you need to be aware of while developing software intended for a global community.

Note: The examples in this document are using Jinja2 syntax. While the concepts are generally the same, the exact syntax may differ between projects.

There are also a few jingo-specific filters being used. Again, be sure to check the syntax for your specific project.

See also:

Mozilla Wiki: Localizing Projects and Content A short guide on how to start the process of getting a new project localized by the Mozilla L10n team.

How does L10n work?

In practice, most of Webdev's L10n work involves making sure all English strings that are shown to users are marked for L10n by wrapping them in a translation function. Typically, a translation function takes the English string you want to display as an argument, and when the website is rendered, the function returns the translated version of that string for the user's locale.

Here's an example of some HTML marked up for translation:

```
<p>{{ _('I am a short translated string!') }}</p>
<p>
  {% trans %}
    Sometimes text that is much longer can be put in a special tag that
    is easier to read!
  {% endtrans %}
</p>
<p>{{ _('There is %s variable in this string.')|fe(1) }}</p>
<p>
  {% trans replacement='variables' %}
    Larger blocks can have {{ replacement }} too!
  {% endtrans %}
</p>
```

In this case, the _ function, as well as the `{% trans %}` block, both display different text depending on the user's locale. The text that is shown is stored in a file that maps the English text to the translated text. It is this file that localizers produce when translating a site.

There are several different file formats for translation files, but the most common ones are:

- PO files are a format supported by the GNU gettext program, and are very common and well-supported by tools.

- .lang files is a format similar to PO files but simplified. They don't support more advanced features like plural forms.

After marking text up for translation, you will typically run some sort of extraction process to update these files with the new strings you've added. Once you've successfully updated these files, you upload them to whatever service your project stores them in for translation. Refer to your project's documentation for more details.

See also:

Tower A Python library that extends Jinja's i18n extension with additional features. Most Django-based Mozilla projects use this for extraction.

Marking up text for L10n

Marking up text to be translated is pretty straightforward:

```
<p>{{ _('I am a short translated string!') }}</p>
<p>
  {% trans %}
    Sometimes text that is much longer can be put in a special tag that
    is easier to read!
  {% endtrans %}
</p>
```

If you need to insert a variable into a translated string:

```
<p>{{ _('There is %(count)s variable in this string.')|fe(count=1) }}</p>
<p>
  {% trans replacement='variables' howmany='multiple' %}
    Larger blocks can have {{ replacement }} too! Even {{ howmany }} ones!
  {% endtrans %}
</p>
```

The wording of some text may change depending on the amount of items you're talking about. Supporting strings that change depending the amount of something is called *pluralization*:

```
<p>{{ ngettext('%(num)d apple', '%(num)d apples', apples|count) }}</p>
<p>
  {% trans count=apples|count %}
    There is {{ count }} apple.
  {% pluralize %}
    There are {{ count }} apples.
  {% endtrans %}
</p>
```

You can often add notes describing a string to be translated using comments. These comments are shown to translators to help them figure out the right wording to use:

```
{# L10n: "They" refers to a group of people here. #}
<p>{{ _('They had no idea what was coming.') }}</p>
```

Things to keep in mind

- Avoid unnecessary complexity in strings. In particular, avoid including HTML in strings as much as possible. If you must include HTML, use a `<span>` or similar tag with no class, and wrap the string in another tag with any class or ID you need.

```
{# WRONG #}
{{ _('Check out the new <a href="http://mozilla.org" rel="external">website</a>!
↪') }}

{# RIGHT #}
{{ _('Check out the new <a {{ link_attrs }}>website</a>!')
   |fe('href="http://mozilla.org" rel="external"') }}
```

- Languages vary wildly in how they work. Some languages put punctuation at the beginning of sentences. Some languages have a different word for 1 item, 3 items, 10 items, and 22 or more items. Some languages use very long words with no spaces to describe things. Some languages read right to left. Some languages put the subject of a sentence at the end.

  The point is, never make any assumptions about how translated text will be structured. One example is assuming that a greeting comes before a name:

```
{# WRONG #}
<p>{{ _('Welcome back,') }} {{ user_name }}</p>

{# RIGHT #}
<p>{{ _('Welcome back, %(user_name)s')|fe(user_name=user_name) }}</p>
```

- If the text you're marking up uses any locale-specific idioms that may be confusing to people outside your locale, add a comment explaining the meaning.

```
{# L10n: "Well I'll be a monkey's uncle" is an expression that means
         "This is a surprise!" #}
<p>{{ _("Well I'll be a monkey's uncle, you've got a new badge!") }}</p>
```

- When displaying things like numbers or dates, make sure to use a library like Babel to format them properly for the user's locale. For example, many locales use spaces instead of commas to split up large numbers.

See also:

Creating localizable web applications A guide with further tips written by the Mozilla L10n team.

Security

This guide will give you a quick heads-up on important security topics to keep in mind as you work on your projects.

Involving the Security Team

The security teams can be easily involved by setting the `sec-review` flag to `?` in Bugzilla. It is highly encouraged to do that early in the development process. For bigger projects the Security Review Process should be taken into account, so that security considerations are resolved before the day of deployment dawns. If you have small questions, feel free to flag someone for `feedback` or ask in #security on IRC.

X-Frame-Options

X-Frame-Options (XFO) is a security header (i.e. in your HTTP response) that states whether your site should be framed or not. For several reasons laid out in this blog post, you should default to **DENY**.

If you do need to be framed, you can restrict this to web pages in the same origin (people sometimes think "same domain", but it's actually the protocol, domain name and port forming the security scope of a website). There are

detailed docs about XFO on MDN and there are additional resources that show you how to set up X-Frame-Options in your Django and NodeJS projects.

Content Security Policy

Note: When building web applications, it is best to incorporate Content Security Policy (CSP) early into the development process. You may find it substantially harder to apply CSP to existing projects because of the way it restrains the capabilities of your code.

Content Security Policy (CSP) is a security header which is able to mitigate some client-side attacks on web applications, like Cross-Site Scripting (XSS). Think of CSP as a whitelist of resources which are allowed to be embedded into your HTML documents. As CSP does not prevent flaws from being exploited but merely mitigates the effects, you should never solely rely on it. CSP 1.1 is already being drafted at the W3C, but you should focus on CSP 1.0 - mainly because of its wide adoption among browsers. Head on over to MDN for more information on this topic.

CSP usage

Warning: `*` wildcards pose a security risk and should be completely avoided in critical directives like `style-src`, `object-src` and `script-src`. If you are unsure about a certain case, members of the web security team will gladly help (See *Involving the Security Team*).

To avoid CSP problems, follow these guidelines:

- Don't use inline JavaScript code. This includes inline script elements (`<script>code</script>`), inline event handlers (e.g. `<button onclick="code">Click me</button>`) and the JavaScript pseudo protocol (`<a href="javascript:code">Click me</a>`).
- Don't use inline CSS code. This includes inline style elements (`<style>code</style>`) and inline style attributes (`<button style="code"></button>`).
- Don't use `eval`, `setTimeout('string', time)`, `setInterval('string', time)`, `Function('string')()` or any other eval-like construct.

Here are some strategies for avoiding common CSP errors:

Inline script elements Should go into a JS file.

Inline event handlers Attach event handler in an external JS file (addEventListener) or let event bubbling work for you (e.g. JQuery's $.live).

JS pseudo protocol Attach click event handler to the node (see above)

Inline style elements These can be easily put into an external CSS file

Inline style attributes Add classes or IDs to your markup and handle those in an external CSS file

Inline style attributes which are set via JavaScript Use the `element.style` property instead of `element.setAttribute`.

Projects simplifying the use of CSP

- Python/Django: https://github.com/mozilla/django-csp
- Node.js/Express: https://github.com/evilpacket/helmet

Testing

Testing is a very important part of the development process. It allows us to verify the functionality of our projects as well as judge the quality of our work.

At Mozilla, we have multiple ways of testing our code, including:

- Unit tests and integration tests, which are automated tests that verify that pieces of code work as expected.
- End-to-end tests, automated tests which check the functionality of a project as a whole. For example, simulating clicks in a web browser to test how a site functions.
- Manual testing, which is performed by a human and involves verifying features work as expected and exploratory tests.

Assessing and managing risk

The end goal of testing is to manage the risk of something going wrong with your project. To this end, one of the first steps you should take is to assess the risk of each area of your project.

More concretely, some parts of your project are going to be more likely to fail than others. Also, some parts of your project are more important than others, and it may be more harmful for them to fail than less important parts.

A risk assessment lists out the different parts of your project (such as certain webpages or parts of an API) and ranks them based on their importance. For example, a news site rank being able to read existing articles as more important than being able to submit new articles. Ranking these parts allows you to make decisions about which to test more and what kind of tests to run.

Unit and integration tests

As a developer, the most common type of tests you will write are unit and integration tests. Unit tests test the smallest possible chunk of functionality and are isolated from eachother. Integration tests test the interaction between these chunks.

In practice, **any changes you make to a project should be tested in some automated way if it's reasonable to do so.** While each project varies, generally Webdev isn't picky about having perfect unit tests or perfect test isolation. If you're unsure, look at existing tests for the project for guidance on the preferred style.

Mozilla runs a Jenkins server for running these tests automatically for several projects. Other projects rely on Travis CI for executing their tests.

For Django projects, these tests live within the `tests` module of each included Django application. For Node-based projects, they normally live in a directory named `test` or `tests` at the root of the repository. Refer to your project's documentation for more details.

End-to-end tests

End-to-end tests simulates how your project will be used by users and verifies that it behaves as expected. This is most commonly applied to websites, where we use tools like Selenium to simulate users interacting with the website.

For many sites, these tests are written by WebQA contributors and run against the various *server environments*.

Manual testing

Manual testing is good old-fashioned human-powered testing, where a living, breathing human uses your project and checks for any errors. Typically this is either for verifying that a new feature works as expected, or for free-form exploratory testing.

In addition to writing automated tests, you almost certainly should be manually testing any changes you make to a project.

Testing tools

The following is a non-exhaustive, possibly-out-of-date list of tools and libraries that may aid you in testing your projects.

General

- Jenkins is a continuous integration server that builds and/or tests software projects continuously.
- Travis CI is a hosted continuous integration service that integrates with Github.
- Selenium is a tool for automating browsers, often for testing purposes.

Python

- nose is a highly recommended testing library for Python.
  - django-nose integrates nose into a Django test runner.
  - nose-progressive is a nose plugin that makes test output much easier to read.
- factory-boy replaces test fixtures with factories that generate test objects easily. It integrates with the Django ORM to generate model instances with a very conveninent syntax.
- Mock is one of the most popular libraries for replacing parts of the system you're testing with mock objects and asserting things about their behavior.

Node / JavaScript

- Mocha is a framework for running tests on node.js and in the browser.
- Chai is an assertion library with many interfaces to accomodate different testing styles.
- Karma allows you to execute JavaScript code in multiple real browsers.

Server Environments

Writing code is one thing, but getting it live on a public server is another. This document explains some high-level details of our server environments and how we deploy our websites.

Development, staging, and production

Most Mozilla websites are split into three environments:

- Development (AKA Dev) usually runs off of the latest code that has been committed for a project by updating regularly throughout the day. Dev is useful for testing new features and getting feedback.

  The standard pattern for dev server URLs is `project-dev.allizom.org`.

- Staging (AKA stage) is intended to be similar to production and is used for testing the deployment process itself. Stage is usually deployed before deploying to production, and is sometimes used by QA for testing purposes as well.

  The standard pattern for stage server URLs is `project.allizom.org`.

- Production (AKA prod) is the live instance of the site. It is the environment that users see when they visit your site. Admin interfaces and other sensitive areas of a site are generally locked down in this environment.

  The standard pattern for production server URLs is `project.mozilla.org`.

Git and Github

This document describes some tips and tricks for using Git and Github. There are also some best practices for *Git and Github*.

Commit Messages

See Tim Pope's blog post on git commit messages.

Rebasing Commits

While projects vary in their opinions on whether merge commits should be avoided or not, it is generally a good idea to rebase a feature branch before submitting a pull request.

Rebasing allows you to alter a series of commits, changing the history of your repository. Typically you rebase a branch to:

- Combine smaller commits made during development into larger, logical commits that are easier to understand and review, or split up larger commits into smaller commits for the same purpose.
- Alter commit messages of previous commits.
- Move a branch to be based on the latest commit of the branch you want to merge into and resolve any conflicts that occur.

These changes all make the code review process as well as the merging process easier, and are recommended for all pull requests.

> **Warning:** Rebasing code that has already been pushed to a public or shared repository makes it very difficult for others to update their local repositories. Only rebase branches that you are absolutely sure no one else is using, such as feature branches on your personal fork.

See also:

Using Git rebase A short guide on how to use the `git rebase` command.

Asking for Review

When asking someone for a code review, it is recommended to add a new comment to a pull request using the `@Username` syntax. This notifies them via email that a review has been requested. Adding the `@Username` to the pull request description will **not** send out the notification and thus isn't recommended.

You may also add an attachment to a bug in Bugzilla and paste in the URL for the pull request as the content. This will cause Bugzilla to link to the pull request and allow you to set the review bit on the attachment to track the state of the review in Bugzilla.

Owners and the Mozilla Github Organization

See the Github page on wiki.mozilla.org for information on the Mozilla organization on Github or anything that requires owner access for the organization.

Glossary

At Mozilla Webdev we use a lot of special terms that mean specific, non-obvious things to those who aren't familiar with them. This document attempts to define those terms.

pull request

PR A term for a request on Github to merge some changes into a codebase. Pull requests are the primary place where code review happens for Github projects.

r? Abbreviation for a request to review a piece of code.

r+ Abbreviation for passing a code review.

r+wc Abbreviation for passing a code review *with changes*. This means there was feedback on things to change during the review, but the requested changes are so minor that a subsequent review is not necessary after making them.

r- Abbreviation for failing a code review. This usually means that more work is needed rather than rejecting the code outright. After the requested changes are made, another code review is required.

Index