Webdev Bootcamp Documentation

Release 0.2

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

Sep 27, 2017

Contents

Perhaps you are familiar with Git, Django, jQuery, Python, JS, CSS, HTML, RabbitMQ, Celery and **the DOM**.

Despite all that, web developing at Mozilla can still be challenging. The *Webdev Bootcamp* is an attempt to clarify how things are sometimes done.

See also:

If you are doing Django development for Mozilla, much of our Django behavior is encapsulated in Playdoh.

Note: This documentation is in Github, so if you find any mistakes or omissions please fork it and submit a pull request.

Warning: This document is strictly a guide. If the documentation told you to jump off a cliff, would you? Likewise, if you can do something better or if you think what's been documented is not right, challenge it and make life better for your webdev siblings.

CHAPTER 1

Accounts You'll Need

Most of the Mozilla web development process takes place between Bugzilla, IRC, and GitHub. There are some other accounts you might need too. Here's everything you should request when you start at Mozilla:

List of All Accounts (and How to Get Them)

Use the links to each app to get to their signup form.

- *Email* and other communication accounts

  See *Communications* for details.

- Bugzilla — use your @mozilla.com email to get 1337 access, and put your IRC nick or whatever you go by in your Bugzilla username for easier searching (example: `Matthew Riley MacPherson [:tofumatt]`).

  See *Bugzilla* for more details.

- GitHub — a free account is all that's required to develop on Mozilla web apps/sites.

  Ping `wenzel`, `jsocol`, or your manager on IRC to get added to the right groups.

  See *Git and Github* for more details.

- IRC — check out the wiki to learn about Mozilla IRC. You should register your main nickname on IRC so it doesn't get stolen and so you can access any channels that require you to `IDENTIFY`.

  See *IRC* for more details.

- Mercurial and `svn` — Sometimes things aren't in GitHub and you need to use `hg` or `svn`. Locale files in particular are stored in Subversion for various reasons. Check out Becoming a contributor and be sure to ask for "Level 2 Mercurial Access".

- VPN — You'll have access to Mozilla-MV (office network) by default, though it does require some setup. Access to Mozilla-MPT is by request only however, and you'll need it.

  See *VPN* for more details.

CHAPTER 2

Development Process

This document outlines the development process for most projects.

Note: **Not every project follows this**—notably *Socorro* and *Elmo*. Check with your team lead or manager for clarification and then fork this.

Release Cycles

Teams work on 1, 2 or 3-week release cycles. Ultimately teams want a continuous development process, where code can be developed and placed in production immediately.

A Bugs Life

- Most of our work is encapsulated in Bugzilla.
- Bugs represent tasks (bugs or enhancements).
- A developer lead will typically group the bugs into milestones.
  - Each milestone represent a release.
  - All work done on a project belong in that milestone.
- A developer assigned to a bug will typically:
  - make a "bug branch" in `git`
  - make code changes
  - commit the code changes
  - push them to a personal GitHub repository

  - request a review (`r?`):
    - In bugzilla
    - in IRC
    - over email
    - via GitHub's pull request system
      - `http://github.com/davedash/project/compare/mybugbranch` let you see differences between your work and the `origin`'s `master`
  - After a positive review (`r+`) code will be merged into `master` and pushed to origin.
  - Most projects avoid `merge` commits unless they are necessary.
  - The bug is marked fixed and a comment to the GitHub commit is left in the bug.
- QA will verify all bugs that have been marked fixed.

QA

QA will verify that bugs are fixed. If you are working on a bug that does not need QA verification mark it with `[qa-]` in the whiteboard status.

QA will re-open a bug if they feel it's not complete. They will file new bugs if regressions are found within the current milestone.

Deployment

Deployment varies heavily by product. A typical project will branch `master` into `prod` and tag the release with the milestone.

It will then deploy anything in `prod`.

A typical push consists of:

- Branching and tagging.
- Notifying the `l10n` volunteers.
- Filing an IT request (a "push bug").
- Including an etherpad of special instructions if needed.
- Upon QA and Dev approval code is pushed to production
- QA verifies production is working properly
- Hotfixes are made if needed
- Or a rollback happens.

You can also check out useful documentation from IT about how we deploy websites (requires an LDAP account).

We want a "one button" push process that automates the above steps.

CHAPTER 3

Developing Locally

For many sites (AMO, SUMO, Input, etc), developing locally is an easy achievement.

Developing locally allows you to be able to work anywhere without relying on network connectivity or being.

Homebrew (Mac OS X)

Most web developers choose a MacBook Pro as their development machine. Therefore we recommend Homebrew. Almost all the required development tools can be acquired via `brew`:

- git
- sphinx search
- mysql
- redis
- memcached
- python
- gettext (you'll want the GNU version; the BSD version that comes with OS X won't play well with playdoh)

If you need something that `brew` doesn't support, but you could otherwise compile, you can create your own recipes.

Xcode (Mac OS X)

You need Apple's Xcode to compile packages. If you're on the Mozilla network (in the Mountain View office or on OfficeVPN) you can connect to fs2 (`smb://fs2`) using your LDAP account and find a recent version of Xcode 3 in the "mac" folder.

You can also get Xcode from http://developer.apple.com/xcode/, if you have an Apple Developer account. There are currently problems using Homebrew with Xcode 4, so make sure you get a recent version of Xcode 3 if you're downloading it from Apple's site. You can install Xcode 3 and 4 alongside each other (in separate folders) if you already use Xcode 4 for other things.

If you're in the Mozilla Mountain View office (or you have VPN access to it via Mozilla-MV) you can grab a recent, compatible version of Xcode 3 from our `fs2` file server by connecting to `smb://fs2` as a guest and selecting the `public` volume. Xcode 3.2 is in the `mac` folder. See more about *VPN*.

Note: If you don't have an Apple Developer account, it might be easier to simply grab Xcode from the shared volume on `fs2`. Navigating the Apple Developer site, especially if your account is a free account, can be a bit of a maze.

Homework

Install Homebrew and install `git`, `sphinx`, `mysql`, `gettext`, and `python`. You'll need at least these items for most projects.

For gettext, you'll need to `brew link gettext` so your system uses the GNU version of gettext instead of the BSD version Apple provides.

CHAPTER 4

Bugzilla

Almost all webdev tasks take place in Bugzilla. See *A Bugs Life*.

The Hacks

In order to receive email notifications for specific components you must follow the appropriate QA contacts.

In order to let easily autocomplete usernames enter your name as:

```
My Name [:username]
```

This allows people to simply type `:username` in Assignment and CC fields and be assured they get you.

IT Requests

IT requests are a special type of bug that the IT team can follow up. You can file a request for Website pushes as well as Desktop support.

Searches

Searches in Bugzilla can be saved. You can also share searches with others and you can keep other people's shared searches in your Bugzilla view.

Making life better

Bugzilla is a wild beast that cannot be tamed. There are a few things that can make life easier:

- The Bugzilla API.

- The Bugzilla JS Jetpack.
- Bugzilla filters for gmail.

You can forward some or all of your email to gmail and make use of a rich set of filters.

CHAPTER 5

Git and Github

Unless you have a good reason you should be using `git` and GitHub for version control. One notable exception is many of our projects rely on SVN for localizers. We'll be attempting to phase that out.

Git Resources

If you don't know `git` or haven't used it in a team, fear not! There are lots of awesome sites for git newbies. We recommend:

- Help.Github can help you get started with `git` regardless of your operating system. If you haven't used GitHub before, it's the perfect crash course. There's also some good info about `git` itself. You can ignore the "deploy" section, as we have our own deployment process at Mozilla.
- Pro Git is probably the best `git` resource in existence. It covers pretty much everything you'd want to know about `git`, so it's quite lengthy, but it's a great read to get to know the basics or to use as a reference. Pro Git is written by one of the developers at GitHub.
- There's a good list of git resources on StackOverflow. It lists tools, tutorials, reference guides, etc. A lot of handy stuff there.

Next time you start a project, use `git`/GitHub! Working on a project by yourself is a bit different than working with others, but start with some basic `git` commands (clone, branch, merge) and some of the more wild stuff (multiple origins, rebasing, etc.) will make more sense.

Git Practices at Mozilla

- Read about the git-flow model. We work similarly to this at Mozilla, except we use `master` as our development branch, `prod` for our production branch, and `bug-$BUG_NUMBER` as our feature branches. Once you get to know `git`, understanding how to use/manage branches effectively will allow you to keep different bug fixes and features in their own branches. This is **really awesome**, especially if regressions crop up!

- We use `git submodule` for our libraries. This git submodules explained article helps you understand how they work.
- We often use `git rebase` to combine and fix commits before merging to mozilla origin repositories. This helps code reviews and keeps commit history clean. GitHub has a good rebase article.

github.com/mozilla

New projects for Mozilla websites should start in the Mozilla account.

Contact `jsocol`, `wenzel` or `peterbe` to be added to individual projects you want to have your way with. They hang out in **#webdev** on IRC, which is a fine place to ask for access when you start at Mozilla.

Service Hooks

GitHub has some service hooks that are helpful to Mozilla projects.

- Bugzilla - posts comments on bugzilla when commit messages reference a bug by id, and closes bugs when commit message says 'fix' or 'close'
- IRC - announces repository pushes in an irc channel

Contact `davedash` or `wenzel` to get access parameters for the hooks.

Working on projects

In order to work on a project:

- Fork it into your own account (do not develop directly in `origin`)
- Make a branch for your work
- Submit a pull request for review
- Merge your commit into `master` which should track the `origin/master`
- `git push`
- Place a link to the commit (as it appears in the origin repository) in the relevant bug.

Commit Messages

- Follow these guidelines.
- Should begin with a 50 character summary, with details if needed below.
- Should contain `bug 1234` somewhere in the summary.

Keeping master in sync

You will want to keep your local `master` branch in sync. Typically you will rebase your branches with your `master` and ultimately you will push your `master` to `origin/master`.

Let's assume you've defined your `origin` remote properly in GitHub. E.g. for Zamboni.

```
origin      git@github.com:jbalogh/zamboni.git
```

You will want your `.gitconfig` to have the following:

```
[branch "master"]
    remote = jbalogh
    merge = master
    rebase = true
```

Making life easier

Git Tools

shell

In order to make life easier we maintain a repository of `git-tools`. These are shell scripts or python scripts that commit all kinds of magic.

Here's a sampling:

- `git here` will tell you the name of your branch, this is an excellent building block
- `git bugbranch $BUGNUM` will copy your current branch to an appropriately named bug branch. This uses the *Bugzilla API*.
- `git compare` with the appropriate `git.config` settings will give you a Github compare URL for your branch (you'll need to push to Github on your own).
- `git url` with the appropriate `git.config` settings gives you the last commit's URL on Github.

Put these in your path and then fork and make your own tools and share.

vim

fugitive.vim may very well be the best Git wrapper of all time.

hub

hub is a git wrapper (or standalone tool) that allows deep integration of github into your command-line git workflow. You can easily clone, fork, pull-request, and checkout pull-requests locally. Read the page and install it now.

Oh My Zsh

Oh My Zsh <https://github.com/robbyrussell/oh-my-zsh> is an excellent collection of zshell scripts that can make your *zsh* environment amazing. It includes a collection of plugins, including ones for `git` and Github.

Some of these overlap with `git-tools`. Additionally by using Oh My Zsh you can easily display your current branch and it's dirtiness on your prompt.

Here is my prompt:

```
dash@awesomepants in ~/Projects/bootcamp/the_code/docs
(bootcamp) ±                                                    on master!
```

Where:

- `bootcamp` is my active *virtualenv*.
- ± signifies that I'm in a `git` repository.

- `master` is the branch I am in.
- `!` indicates that there are uncommitted things in my branch.

Development Process

See *A Bugs Life*

Looking at someone's code

Sometimes you need to run someone else's code locally. If they've given you a pull request, or a commit hash this is what you need to do to see there code:

```
git remote add davedash git@github.com:davedash/zamboni.git
git fetch davedash
git co davedash/branch
```

Note:

- The above assumes that someone else was me.
- The first line defines a "remote". A remote is simply an alias to a repository.
- The second line fetches all my commit hashes that you don't already have. Usually this is just branches, and commits, but in theory it can be anything.
- In the third line I can check out your branch. If you just gave me a commit hash I would do `git co $COMMIT_HASH`.

CHAPTER 6

Jenkins: Continuous Integration

We have a public instance of Jenkins (formerly Hudson). For most projects it runs their python test suite. Optionally we use it to do JS testing as well as any menial tasks that need to be done regularly, like packaging. If you break things you will be warned in IRC.

Adding a new Project

If you've got tests (which you should), and you are deploying to production, you might want to add your project to Jenkins. This let's the world know just how wonderful you are at writing tests.

Asumming you're working on `mozilla/awesome_project`, you'll need to:

- On Jenkins:
    1. Log in via LDAP to Jenkins.
    2. Start a new project.
    3. Copy Affiliates.
    4. Update the notification settings (IRC, email, etc).
    5. Update the Github project to point to `https://github.com/mozilla/awesome_project/`
    6. Update the Git repository to point to `git://github.com/mozilla/awesome_project.git`
    7. Check the "Build when a change is pushed to GitHub" checkbox
- In your repo:
    1. Copy the `bin/jenkins.sh` script from playdoh if you don't have it.
    2. Make sure it's got `+x` permissions.
- On Github:
    1. Go to `https://github.com/mozilla/awesome_project/admin/hooks`
    2. The hook URL is `https://ci.mozilla.org/github-webhook/`

You may need to make some adjustments according to the needs of your particular project.

Interacting with Jenkins on IRC

You can get a Jenkins bot in your channel to send him commands.

1. Go to the general configuration https://ci.mozilla.org/configure
2. Add your channel in the IRC Notification section

You can now run commands such as `jenkins:  build awesome_project now`

CHAPTER 7

How to Code

In general, follow the languages established best practices, and fill in the gaps where there are holes.

General Guidelines

- *Style matters*

  How code is aligned matters, because code is reviewed, edited, and public. Code that is uneasy to read does not align with the spirit of open source.

- *Be consistent*

  If you do something a certain way, be able to justify it. Don't mix *camelCase* with *underscore_words* unless you have good reason.

- *Follow code around you*

  If you don't know what you're doing try to follow what others have done.

Testing

In languages and frameworks that provide easy test-ability, *write tests!*

Go for 80% or more coverage, especially in the following areas:

- privacy - e.g. test that private things are private
- heavily used code - e.g. landing pages, library level code
- re-opened bugs

Tests last longer than code, as they tend to define the products' functionality. They are valuable because they allow us to quickly make changes without fear of hindering functionality.

The other half of testing is continuous integration. We should be running our tests at every check in and be able to say with certainty that the code is correct to the best of our knowledge. See *Jenkins: Continuous Integration*.

Python

We do what others in the Python community have established:

- We follow PEP8.
- We test using *flake8* which combines *pep8.py* and *pyflakes*.
- We follow Pocoo's extensions of PEP8 as they are well thought out.
- You might also consider *baked* which enforces the import order in this guide.

Import Statements

We expand on PEP8's suggestions for import statements. These greatly improve ones ability to ascertain what is and isn't available in a given file.

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; Django (or framework); 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...
```

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
)

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

Remember the golden rule of pep 8: **A Foolish Consistency is the Hobgoblin of Little Minds**. Generally with indenting, do what makes sense and is logically easy to read. Really dense code is as hard to read as really spread out code.

Django

Follow *Python*. There are a few things in Django that will make your life easier:

Use `resolve('myurl')` and `{{ url('myurl') }}` when linking to internal URLs. This will handle hosts, relative host names, changed end points for you. It will also noticeably break so dead-links don't linger in your code.

Indent only two spaces in templates:

```
{% if indenting %}
  <div class="example">
    <p>This is how it's done.</p>
  </div>
{% endif %}
```

This deviates from the standard four space indentation we recommend for other coding languages. HTML lends itself to a lot of nested elements and indenting each level four spaces can quickly lead to long lines and messy formatting. Indenting two spaces in templates can make it easier to manage. Use four spaces everywhere else.

Playdoh

New web-apps should be spawned from Playdoh and existing ones should follow the spirit of Playdoh. Playdoh collects lessons that several Mozilla Django projects have learned and wraps them into a single Django project template.

In the future, much of Playdoh's moving parts (Middleware, filters, etc) will be moved into a separate library so these features won't be lost.

See *Packaging and Dependency Management*.

Javascript

See *JS Style Guide*.

HTML

- Use the HTML5
- Make sure your code validates
- No CSS or JS in the HTML
- Be semantic
- Use doublequotes for attributes:

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

Todo

The previous list compiles to weird html where the list is a bunch of separate lists.

CHAPTER 8

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

You're doing it wrong. See above about named 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.

CHAPTER 9

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).
- Single quotes, not double. No quotes in `url()`.
- 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.
- IDs 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.
- Comment a lot, use KSS for structure.
- Use Stylus but write it like plain CSS.
- Consider screen readers when hiding elements.

General guidelines

Use hex color codes unless using *rgba()* or *hsla()*. Write hex values in lowercase and, if possible, the shorthand notation, e.g. `#ff0` is better than `#FFFF00`.

Use single quotation marks for property values that require quotes, such as `content: 'x';` or `font-family: 'Open Sans';`.

Also single-quote attribute values in selectors such as `input[type='search']`.

Don't use quotation marks in URLs:
No: `url('/images/dalek.png')`
Yes: `url(/images/dalek.png)`

Note: You can single-quote URLs if the file path contains spaces, as that's generally more readable than URL-encoding (*foo%20bar.png*, yuck). But better yet, don't put spaces in file names or URLs, assuming you have control over them.

Use protocol-relative URIs for any external resources, e.g. `url(//fonts.googleapis.com/css?family=Open+Sans);`

Note: If your site is secured with https and the external resource is not, omitting the protocol from a URI will result in a 404 error, which is perhaps better than a mixed content warning. Then again, it's usually best to avoid referencing external resources in CSS at all.

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 hex 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](http://webaim.org/techniques/css/invisiblecontent/).

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`.

In rare cases it may be necessary to distinguish different elements that belong to the same class but require slightly different styling, such as `a.button` and `input.button`. Most of the time a class or ID alone is sufficient, and decoupling it from a specific element makes the name more reusable (yes, an ID can be reusable, just not in the same HTML document; `#main-content` could be a `main` element on one page and an `article` element on another).

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 cases where you'll want to use relative `font-size` units like ems 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.

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 for *line-height*.

Use "bulletproof font syntax" for webfonts. However, 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 slightly different, 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: #ff0; }
.message-notice  { color: #00f; }
```

Or:

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

Long, comma-separated property values – such as multiple background images, gradients, transforms, transitions, or text and box shadows – can be arranged across multiple lines (indented one level from their property). This improves readability, minimizes horizontal scrolling, and produces more useful diffs with meaningful line numbers.:

```
.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.

Include a space after each comma in comma-separated property or function values:
Yes: `rgba(27, 34, 38, .9)`
No: `rgba(27,34,38,.9)`

Don't pad parentheses with spaces:
Yes: `url(/images/galactus.jpg)`
No: `url( /images/galactus.jpg )`

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.

Commenting

Comment profusely. Be descriptive. Write for posterity.

Write your comments for someone unfamiliar with your site or application. Tell them where each set of rules is used and why you did what what you did the way you did it.

This is the age of preprocessors and minifiers that strip comments and whitespace before it's served to the client anyway so you usually don't need to worry about saving bytes in your source files.

If you're using a preprocessor that allows it, comment lines with `//`

Give each section of a style sheet a useful title. You can flag titles with a @ to ease searching. (We like @ because it's not used much in CSS and can't be mistaken for an operator or variable.)

Use KSS to document sections, rule sets, and individual rules as needed.

Include a "preamble" at the very top of each style sheet with a title, description, table of contents, and any other useful information (license, credits, changelog) or references (font sizes, color chart, library dependencies).

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-news {
    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).

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

We don't use Sass because it requires Ruby. While Sass is a fine tool, and is especially 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 heavy pinch 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.

CHAPTER 10

Localization (l10n)

Most web apps at Mozilla are localized. We make use of `gettext`.

See also Localization (L10n) in Playdoh docs

SVN

By convention Mozilla puts locales in *locale/* and that folder is a working copy from SVN. This allows localizers to use tools like Verbatim to add new localized content.

You can create an empty subdirectory off of https://svn.mozilla.org/projects/l10n-misc/trunk/ to store your *.po* files:

```
svn mkdir \
    https://svn.mozilla.org/projects/l10n-misc/trunk/$MYPROJECTNAME \
    -m "Creating $MYPROJECTNAME"
```

Where `$MYPROJECTNAME` is the name of your project.

You'll also want to add `*.mo` to the list of `global-ignores` in your `~/.subversion/config` file. `.mo` files can be compiled at deploy time.

Warning: `.mo` files are compiled and therefore have no place in version control.

Now in your project root:

```
svn co https://svn.mozilla.org/projects/l10n-misc/trunk/$MYPROJECTNAME \
locale
```

Anytime you make changes using the `merge` or `extract` commands you'll need to commit them back to SVN:

```
cd locale/
svn add *
svn commit -m 'Locale update'
```

See Localization (L10n) in Playdoh docs for more info on the `merge` and `extract` commands

Adding new locales (non-django)

Note: See Localization (L10n) in Playdoh for django instructions

To add a new locale to an existing project, go to the Verbatim project admin page, e.g., https://localize.mozilla.org/projects/mdn/admin.html

Use the dropdown at the bottom of the list of locales to add the new locale to verbatim

ssh to the verbatim box and add the new locale to svn:

```
ssh sm-verbatim01
cd /var/lib/pootle/po/<project>
svn add <locale>
svn ci <locale> -m "Adding <locale>"
```

Warning: Only commit the new locale directory and do not update the svn working copy.

Adding a new text domain (non-django)

Note: These instructions are only for localizing text outside of django/playdoh projects. See Localization (L10n) in Playdoh for django instructions

Sometimes in a single project you need to use more than one translatable module. A text domain is a handle for each module with different .po files. For example, MDN uses separate text domains for the MDN website and the Promote MDN WordPress plugin.

You can still use Verbatim to translate these other text domains alongside the primary domain.

Generate your .pot file, and then generate a .po file for each locale e.g.,:

```
msginit --no-translator -l <locale> -i templates/LC_MESSAGES/promote-mdn.pot \
-o <locale>/LC_MESSAGES/promote-mdn.po
```

Add the .pot file and the .po files to svn e.g.,:

```
svn add */LC_MESSAGES/promote-mdn.*
svn commit */LC_MESSAGES/promote-mdn.*
```

Now go to each locale's project directory and click the "Update all from version control" button.

Make this better

This process is merely a suggestion. If you think localization can be improved or perhaps automated, by all means... **DO IT!** If your improvement takes off update this, so others can benefit.

CHAPTER 11

Packaging and Dependency Management

Python projects can incur a number of dependencies. `pip` can be handy, but we've had better luck with distributing a `vendor` library.

For the basics, read Zamboni packaging as well as pip and friends: Packaging.

Updating a Library

Let's say we want to update Django to 1.3. We already have Django setup in our `requirements/prod.txt` as well as a submodule of our vendor directory.

`prod.txt`:

```
-e git://github.com/django/django@1.2.5#egg=django
```

So we're on Django 1.2.5. We want to go to Django 1.3. We edit `prod.txt`:

```
-e git://github.com/django/django@1.3#egg=django
```

Save and quit, puts us back on the command line.:

```
$ git submodule update --recursive
$ pushd vendor
$ git pull
$ git submodule update --recursive
$ pushd src/django
$ git checkout origin/1.3
$ popd
$ git add src/django
$ git commit -m"Bug 1235 Upgrading to Django 1.3"
$ git push origin master
$ popd
$ git add vendor requirements/prod.txt
$ git commit -m"Bug 1235 Upgrading to Django 1.3"
$ git push origin master
```

Now other developers can pick up your changes into their *virtualenv* and IT can pickup your changes in *vendor* and push out to the web heads.

Upgrading Libraries

To keep up-to-date, one should occassionally do:

```
pip install --upgrade -r requirements/compiled.txt
pushd vendor
git submodule --update --init
popd
```

This will refresh the libraries you've installed with their latest tagged version.

Todo

Write tools to automate this.

CHAPTER 12

Security

While there is a web security team, building secure services is your responsibility, too. This guide will give you a quick heads-up on important topics.

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 developement 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

CHAPTER 13

Data storage and retrieval

Most sites have fairly simple data-layers. The notable exception is Socorro.

Typically we use some form of mysql with master-slave replication.

For search either Sphinx or Elastic Search are used.

For cache memcache and redis.

Socorro uses postgres and HBase.

Production Data

Sometimes having production-like data is necessary for debugging. Private data relating to users *must* remain on the Mozilla network. Therefore, there are two options for getting Production or production-like MySQL data.

Anonymous Data

In *~ddash/anonymize* On *webdev1.db.scl3.mozilla.com* on the Mozilla/MPT VPN are anonymized dumps of production data for:

- AMO
- FlightDeck
- SUMO

While the datasets are anonymous, they are not for general distribution.

Input Data

You can get a copy of the Firefox Input database by using the following script:

```
set -e
FILE=input_mozilla_com.`date +%Y.%m.%d`.sql.gz
USERNAME=username
LOCAL_DB=firefox_input

scp $USERNAME@webdev1.db.scl3.mozilla.com:~ddash/input_mozilla_com/$FILE .
zgrep -v "INSERT INTO \`feedback_term\`" $FILE > tmp.sql
cat tmp.sql |mysql -u root $LOCAL_DB
rm tmp.sql $FILE
```

Be sure to replace `username` with your actual LDAP username.

Webdev Database Cluster

Alternately, many production databases have copies running on *webdev1.db.scl3.mozilla.com* and *webdev2.db.scl3.mozilla.com*. You can connect directly to these servers.

CHAPTER 14

Servers

We have a number of servers that you'll regularly encounter as a web dev.

khan is a development server. If you choose not to develop locally, this option is available.

***.allizom.org**: all our staging servers share this domain.

webdev1.db.scl3.mozilla.com is the webdev mysql server. See *Webdev Database Cluster*.

cm-vpn01 is where our server logs are copied. Note that you need to file a bug to get access to this server.

Served Environments

There are two or three main environments for our web sites:

- **dev** (currently "stage" or "preview") which serves the latest *master*.
- **stage**
  - Currently `amo-next` and `crash-stats.stage` are our only "stage" environments.
  - This is what will go live to production.
- **production**

VPN

To get to any Mozilla servers you will need VPN access. There are two VPN networks: Mozilla-MV (Mountain View office VPN) and Mozilla-VPN (also known as the Datacenter VPN).

If you want to use Mozilla's shared network volumes (like `fs2`) you can connect to the Mozilla-MV VPN.

You'll need to connect to the Mozilla VPN if you want to access anything in the data centres (such as khan, any database server, etc). All Mozilla employees with a valid LDAP login have access to this VPN by default.

On OS X, we recommend Viscosity for VPN.

CHAPTER 15

Error Notification in Production

When a traceback occurs in production sites, you need to send it somewhere. Normally Django sends emails which can work fine until you get a few thousand error emails in a minute.

To mitigate this, you can use Arecibo to track your errors. There are currently three servers:

https://arecibo-phx.mozilla.org/ (behind LDAP) The *preferred* PHX instance.

To use it place the following in your Django settings:

```
ARECIBO_SERVER_URL = 'http://arecibo1.dmz.phx1.mozilla.com/'
```

https://arecibo-sjc.mozilla.org/ (behind LDAP) The SJC instance.

To use it place the following in your Django settings:

```
ARECIBO_SERVER_URL = 'http://arecibo1.dmz.sjc1.mozilla.com/'
```

http://amckay-arecibo.khan.mozilla.org/ Test server for local development.

To use it place the following in your Django settings:

```
ARECIBO_SERVER_URL = 'http://amckay-arecibo.khan.mozilla.org/'
```

To send errors from playdoh, see the playdoh docs.

Warning: IT needs to update the `ARECIBO_SERVER_URL` in Django's local settings. They will need to verify via curl, that the webheads can reach Arecibo:

```
curl -I http://arecibo1.dmz.phx1.mozilla.com/
```

Note: The PHX data centre server is on a dedicated box, where as SJC is in a VM. Please use the PHX one if possible.

CHAPTER 16

Communications

To contact someone, you can use the following methods:

Mailing lists

Two mailing lists are the most important:

- dev-webdev is a public mailing list for all web developers in the Mozilla community. Sign up for it yourself, and use it for all webdev group communication unless the email contains:
  - Sensitive security-related information
  - Private information about or from our partners
  - Administrivia (PTO, WFH, etc.)
- `webdev@` is an internal mailing list specific to webdevs employed by Mozilla. File a ServiceNow request to get subscribed to it and use the list for topics that would not be interesting outside the Webdev group at Mozilla.

Additionally, many teams have their own, team-specific mailing lists. Check with your manager and have her or him add you.

IRC

There's a Mozilla IRC server at `irc.mozilla.org`. The Mozilla IRC server page on the wiki talks about how to connect, how to ask questions, and other things.

We hang out on a bunch of different channels:

- #webdev - general web development
- #flux - the Rapid Development team's channel
- #sumodev - development of SUMO

- #input - development of Input
- #mozillians - development of Mozillians (Community Directory)
- #breakpad - development of socorro, crash-stats, and breakpad
- #mdndev - development of MDN
- #amo - development of AMO

Todo

Add additional channels here.

Also of interest:

- #js - javascript programming discussion
- #nodejs - nodejs discussion

Todo

Add other useful channels here. Is there one for HTML5?

Zimbra Email

Todo

Info about email access

Zimbra Calendar

Todo

Info about calendar access

Teleconferencing

Todo

Info about teleconferencing

CHAPTER 17

Documentation

Documenting Python

Use Restructured Text and PEP-257 for docstrings.

Documenting projects

Use Sphinx to document Python projects.

When doing that, follow the Restructured Text primer.

ReadTheDocs

Read The Docs hosts documentation for applications and libraries written in Python.

The Getting Started walks through getting docs on the site.

You can also set up ReadTheDocs as a post-commit hook in GitHub.

CHAPTER 18

Indices

- genindex

CHAPTER 19

Todo

- port this document - http://etherpad.mozilla.org:9000/webdev-bootcamp
- Link to jsocol's continuous deployment doc
- Link to IT Requests
- Intersphinx
- Add indexes everywhere.
- See if the anonymize directory is correct.
- Verify that the slave db is correct.
- screen shots
- explain data centers - http://blog.mozilla.com/mrz/2010/01/04/mozillas-new-phoenix-data-center/

Todo

The previous list compiles to weird html where the list is a bunch of separate lists.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/coding.rst, line 271.)

Todo

Add additional channels here.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/communications.rst, line 55.)

Todo

Add other useful channels here. Is there one for HTML5?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/communications.rst, line 65.)

Todo

Info about email access

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/communications.rst, line 84.)

Todo

Info about calendar access

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/communications.rst, line 93.)

Todo

Info about teleconferencing

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/webdev-bootcamp-osmose/checkouts/overhaul/communications.rst, line 102.)

Index