Webdev Bootcamp Documentation

Versión 0.1_alpha1

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

27 de septiembre de 2017

Índice general

Así que términos como Git, Django, jQuery, Python, JS, CSS, HTML, RabbitMQ, Celery y **el DOM** te parecen familiares.

Aun así, el desarrollo para Mozilla Hispano puede ser un reto.

La *Documentación para el Desarrollador de Mozilla Hispano* es un intento de aclarar cómo se hacen las cosas.

Ver también:

Si estás desarrollando con Django para Mozilla, muchos de nuestros patrones están encapsulados en Playdoh.

Nota: Esta documentación está en el repo dev-docs de GitHub, así que si encuentras errores u omisiones, por favor haz un fork y envíanos un pull request.

Advertencia: Este documento es estrictamente una guía. Si la documentación te dice que saltes por un precipicio, ¿lo harías? De la misma manera, si puedes hacer algo mejor o si piensas que lo que se está documentando no es correcto, rétanos y haz que la vida sea mejor para tus vecinos del desarrollo Web.

CAPÍTULO 1

Cuentas que necesitarás

La mayor parte del proceso de desarrollo toma parte entre Bugzilla, el IRC, y GitHub. Hay algunas cuentas adicionales que puedes llegar a necesitar. Aquí está todo lo que deberías tener para comenzar a desarrollar en Mozilla Hispano:

Lista de todas las cuentas (y cómo obtenerlas)

Usa los enlaces a cada aplicación para ir al formulario de registro.

- *Correo electrónico* y otras cuentas de comunicación

  Lee communications-chapter para más detalles.

- Bugzilla — coloca tu nick de IRC o alias en tu nombre de usuario de Bugzilla para facilitar las búsquedas. (por ejemplo: `Matthew Riley MacPherson [:tofumatt]`).

- GitHub — una cuenta gratuita es todo lo que se necesita para desarrollar en los proyectos de Mozilla Hispano

  Contacta a `nukeador` en IRC para ser añadido a los grupos/proyectos adecuados.

  En *Git y Github* puedes encontrar más información.

- TeamBox - TODO

CAPÍTULO 2

Git y Github

A menos que tengas un buen motivo, deberías estar utilizando `git` y GitHub para el control de versiones.

Recursos sobre Git

Si no sabes sobre `git` o no lo has usado en equipo, ¡no temas!. Existe una gran cantidad de sitios maravillosos para que te inicies con `git`. Nosotros recomendamos:

- Help.Github te puede ayudar a arrancar con `git` sin importar qué sistema operativo uses. Si no has usado GitHub antes, este es el curso perfecto. También contiene buena información sobre `git` como tal.
- Pro Git es probablemente el mejor recurso sobre `git` que existe. Cubre practicamente todo lo que puedes llegar a necesitar, por lo que es ciertamente extenso. Sin embargo, es una excelente lectura para llegar a conocer los aspectos básicos o para usar como referencia. Pro Git es escrito por uno de los desarrolladores de GitHub.
- Hay una lista de recursos sobre git en StackOverflow que contiene herramientas, tutoriales, guías de referencia, entre otros.

La próxima vez que comiences un proyecto, ¡usa `git`/GitHub!. Trabajar en algo por tu cuenta es diferente a trabajar con otros, pero comenzar con los comandos básicos de `git` (clone, branch, merge) hará que las cosas más avanzadas (múltiples orígenes, rebasing, ...) tengan más sentido.

Prácticas de uso de Git en Mozilla

- Lee sobre el modelo de git-flow. En Mozilla se trabaja de manera similar, exceptuando el uso de `master` como rama de desarrollo, `prod` como rama de producción, y `bug-$BUG_NUMBER` como ramas de funcionalidades. Una vez conoces `git`, entender como manejar las ramas de manera eficiente permitirá mantener correcciones de errores y características diferentes en sus propuas ramas. Esto es **realmente maravilloso**, ¡especialmente en casos de regresiones!.
- Usamos `git submodule` para nuestras librerías. El artículo sobre git submodules explicados te ayudará a entender cómo funcionan.

- Frecuentemente usamos `git rebase` para combinar y corregir commits antes de mezclarlos con los repositorios originales. Esto ayuda a mantener la historia del repositorio limpia y a mejorar las revisiones de código. GitHub tiene un buen artículo sobre rebase.

github.com/mozillahispano

Los nuevos proyectos para Mozilla Hispano deberían iniciarse en la cuenta de Mozilla Hispano.

Contacta a `Nukeador` si deseas añadir un proyecto. Normalmente lo puedes encontrar en el canal **#mozilla-hispano** en IRC.

Trabajando en algún proyecto

Para trabajar en un proyecto existente:

- Haz un fork en tu cuenta
- Crea una rama para tu trabajo
- Envía un pull request para revisión
- Mezcla tu commit con `master`, que debería estar configurada para seguir `origin/master`
- `git push`
- Coloca un enlace al commit en el Bug o Issue relevante

Mensajes de Commit

- Sigue estos lineamientos
- Debería contener un resumen de 50 caracteres, con los detalles necesarios debajo
- Si se trata de un bug o issue, debería contener `bug 1234` en el resumen.

Manteniento master sincronizado

Seguramente querrás mantener tu rama `master` local sincronizada. Típicamente harás un `rebase` de tus ramas con tu `master` para luego enviar (`push`) tus cambios a `origin/master`.

Vamos a asumir que has definido tu remota `origin` de la manera correcta en GitHub. Por ejemplo, para Zamboni.

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

Tu archivo `.gitconfig` debería entonces contener lo siguiente:

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

Haciendo la vida más fácil

Herramientas para Git

shell

Hay un repositorio de herramientas para git llamado git-tools que nos pueden facilitar la vida. Este contiene scripts de shell y Python que hacen cualquier tipo de magia.

Como muestra:

- `git here` te permite saber cuál es la rama actual.
- `git compare` con las opciones apropiadas en `git.config` te dará una URL de comparación en GitHub para tu rama, que permite observar las diferencias de los cambios que ya han sido enviados.
- `git url` con las opciones correctas en `git.config` te devolverá la URL al último commit en GitHub.

Coloca estas herramientas en tu `path` y luego haz un fork y crea tus propias herramientas para compartir.

vim

fugitive.vim puede ser la mejor herramienta para Git y Vim de todos los tiempos.

Oh My Zsh

Oh My Zsh es una colección excelente de scripts de zshell que pueden hacer que tu ambiente de *zsh* sea maravilloso. Comprende una colleción de plugins, incluyendo algunos para `git` y GitHub.

Algunos de esos se solapan con `git-tools`. Adicionalmente, al usar Oh My Zsh puedes ver fácilmente la rama actual y su estado en el `prompt`.

Por ejemplo:

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

Donde:

- `bootcamp` es el *virtualenv* activo.
- `±` significa que estoy en un repositorio `git`.
- `master` es la rama actual.
- `!` indica que hay cambios sin enviar en la rama actual.

Viendo código de otras personas

En algunas ocasiones vas a tener que probar código de otras personas localmente. Si tienes un pull request o un commit de la otra persona, esto es lo que debes hacer para ver su código:

```
git remote add otro git@github.com:otro/repo.git
git fetch otro
git co otro/rama
```

Nota:

- `otro` es la otra persona.
- La primera línea define una *remota*. Una *remota* no es más que un alias para un repositorio.
- La segunda línea descarga todos los commits de `otro` que aún no tienes localmente. Normalmente esto son solo commits, pero en teoría puede ser cualquier cosa.
- En la tercera línea se hace un cambio a la rama de `otro`. Si tienes el hash de un commit, puedes hacer `git co $COMMIT_HASH`.

CAPÍTULO 3

Como programar

En general, sigue las mejores prácticas establecidas para cada lenguaje llenando los posibles huecos con sentido común.

Lineamientos Generales

- *El estilo importa*

  La manera como el código se alinea importa, porque el código es revisado, editado y publico. El código difícil de leer no está acorde con el espíritu del código abierto.

- *Consistencia*

  Si haces algo de cierta manera, debes ser capaz de justificarlo. No mezcles *camelCase* con *guiones_bajos* al menos que tengas una buena razón para hacerlo.

- *Sigue el código que te rodea*

  Si no sabes lo que haces intenta seguir lo que los demás han hecho.

Pruebas

En lenguajes y *frameworks* que proveen facilidades para pruebas, **¡escribe pruebas!**.

Trabaja en tener 80 % o más de cobertura, especialmente en:

- *Privacidad*: prueba que las cosas privadas son privadas.
- *Código muy usado*: por ejemplo, páginas iniciales o código en librerías.
- *Bugs re-abiertos*

Codificar pruebas toma más tiempo que hacer el código, debido a que suelen definir la funcionalidad de los productos. Las pruebas son muy valiosas porque nos permiten realizar cambios de manera rápida sin miedo afectar las funcionalidades.

La otra mitad de las pruebas es la integración contínua. Deberíamos estar corriendo nuestras pruebas en cada cambio realizado y ser capaces de decir con certeza que el código es correcto hasta nuestro mejor conocimiento.

Python

Hacemos lo que otros en la comunidad de Python han establecido:

- Seguir el PEP8.
- Hacemos pruebas usando check.py u otra utilidad que combine *pep8.py* y *pyflakes*.
- Seguimos las extensiones al PEP8 de Pocoo.

Sentencias de importación

Extendemos las sugerencias establecidas en el PEP8 para sentencias de importación. Estas mejoran sustancialmente la habilidad de conocer qué está y que no está disponible en un módulo dado.

Importa solo un módulo por sentencia de importación:

```
import os
import sys
```

no:

```
import os, sys
```

Separa los grupos de importes con una línea en blanco: librería estándar; Django (o framework); módulos de terceros; e importes locales:

```
import os
import sys

from django.conf import settings

import pyquery

from myapp import models, views
```

Alfabetiza tus importes, esto hará que tu código sea más fácil de escanear. Observa lo terrible que es esto:

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

Con un orden sencillo:

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

Imports primero, `from`-imports después:

```
import x
import y
import z
```

```
from bears import pandas
from xylophone import bar
from zoos import lions
```

Eso es muchísimo más fácil de leer que:

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

Por último, cuando se importen objetos dentro de tu espacio de nombres desde un paquete, utiliza el orden `CONSTANTES`, `Clases`, `variables` de forma alfabética:

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

Si es posible, sería más fácil importar el paquete completo. Especialmente en el caso de métodos esto ayuda a responder la pregunta "¿de dónde saliste `tu`?"

Malo:

```
from foo import you


def my_code():
    you()  # espera, ¿esto está definido en este archivo?
```

Bueno:

```
import foo


def my_code():
    foo.you()  # ah ok...
```

El espacio en blanco importa

- Usa siempre 4 espacios para indentar, no 2. Esto mejora la legibilidad considerablemente.
- Nunca utilices tabulaciones, la historia ha demostrado que no podemos manejarlas.

Las comillas

Usa comillas sencillas en vez de dobles o triples si puede representar una mejora:

```
'esto es bueno'

'esto \'es\' malo'

"esto 'es' bueno"

"esto es inconsistente, pero se acepta"
```

```
"""a veces 'esto' es "necesario"."""

'''nadie hace esto realmente'''
```

Django

Sigue las recomendaciones para *Python*. Hay algunas cosas en Django que harán tu vida más fácil:

Usa `resolve('myurl')` y `{{ url('myurl') }}` cuando hagas enlaces a URLs internas. Esto manejará automáticamente hosts, nombres relativos y rutas cambiadas por ti. Igualmente, usar estos métodos te dará mensajes de error descriptivos si hay un enlace roto.

La indentación en las plantillas debería manejarse de la siguiente manera:

```
{% if indenting %}
  <p>Así es que se hace</p>
{% endif %}
```

Playdoh

Las nuevas aplicaciones Web deberían estar basadas en Playdoh y las existentes deberían seguir el mismo espíritu de Playdoh. Playdoh reune lecciones que varios proyectos hechos en Django en Mozilla han aprendido y los envuelve en una plantilla para tus proyectos.

En el futuro, la mayor parte de los componentes de Playdoh serán movidos a librerías separadas de manera que esas características no se pierdan.

Javascript

Lee *Guía de Estilo para JavaScript*.

HTML

- Usa HTML5
- Asegúrate de que tu código valida
- No coloques CSS o JS en el HTML
- Se semántico
- Usa comillas dobles para los atributos:

```
<a href="#">Bueno</a>
<a href='#'>Menos bueno</a>
```

CAPÍTULO 4

Guía de Estilo para JavaScript

Primero que nada

Usa **SIEMPRE** JSHint en tu código.

Nota: Hay algunas excepciones para las cuales JSHint se queja de cosas en node que puedes ignorar, como por ejemplo no saber qué es 'const' y 'require'. Para estos casos puedes añadir palabras claves a ignorar en un archivo *.jshintrc*.

Formato de Nombres de Variables

```
// Clases: PalabrasCapitalizadas
var MyClass = ...

// Variables y funciones: camelCase
var myVariable = ...

// Constantes: MAYUSCULAS_CON_GUIONES_BAJOS
// Backend
const MY_CONST = ...

// Del lado del cliente
var MY_CONST = ...
```

Indentación

Se debe indentar con 4 espacios (no tabulaciones).

Para nuestros proyectos, siempre debes asignar variables en una nueva línea, no separado por comas:

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

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

Usa `[]` para asignar un nuevo arreglo, no `new Array()`.

Igualmente, usa `{}` para crear nuevos objetos.

A continuación se presentan dos escenarios para el uso de `[]`, uno puede estar en una sola línea, el otro no tanto:

```
// Bien en una sola línea
var stuff = [1, 2, 3];

// Nunca en una sola línea
var longerStuff = [
    'some longer stuff',
    'other longer stuff'
];
```

Nunca asignes multiples variables en la misma línea

Malo, malo:

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

NO alinees los nombres de variable

Malo:

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

Puntos y comas

Úsalos.

No porque la inserción automática de puntos y comas (ASI) sea magia negra, hazlo por consistencia.

Condicionales y bucles

```
// Malo
if (something) doStuff()

// Bueno
```

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

Espacios después de una palabra clave y antes del corchete

```
// Malo
if(bad){

}

// Bueno
if (something) {

}
```

Funciones

Funciones con nombre

Siempre usa funciones con nombre, inclusive si la estás asignando a otra variable o propiedad. Esto mejora las pilas de error cuando se hace depuración.

No coloques espacios entre el nombre de la función y el paréntesis inicial, pero si entre el paréntesis de cierre y el corchete inicial:

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

}
```

Funciones anónimas

Lo estás haciendo mal, lee sobre las funciones con nombre más arriba.

Operadores

Siempre debes usar `===` con la única excepción de comparaciones con `null` y `undefined`.

Ejemplo:

```
if (value != null) {

}
```

Comillas

Siempre usa comillas simples: `'no dobles'`

La única excepción: `"no escapes comillas simples en las cadenas así: \'. Usa comillas dobles."`

Comentarios

Para funciones en `node`, provee siempre un comentario claro con el siguiente formato:

```
/* Explicación breve de lo que se hace
 * Expects: cualquier parámetro aceptado
 * Returns: cualquier cosa retornada
 */
```

Si los comentarios son realmente largos, utiliza el formato `/* ... */`. De lo contrario, usa comentarios cortos como:

```
// Este es un comentario corto y termina con un punto.
```

Ternarios

Trata de no utilizarlos.

Si un ternario usa varias líneas, no lo uses:

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

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

Buenas prácticas generales

Si te das cuenta que estás repitiendo algo que puede ser una constante, usa una sola definición de constante al comienzo del archivo.

Define las expresiones regulares como constantes siempre.

Siempre debes probar la certidumbre:

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

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

Si el código es demasiado largo, trata de romperlo en varias líneas o refactoriza. Trata de mantenerte dentro del límite de 80 columnas por línea, pero si te pasas un poco no es un gran problema. Cuando rompas una línea, indenta las subsiguientes un nivel (2 espacios)

Si el código luce demasiado inteligente, probablemente lo sea, así que solo manténlo sencillo.

CAPÍTULO 5

Índices

- genindex

Índice