**Flex y Grid**

* Modulo Flexbox

[13-flexbox.zip](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a75f16c3-9f12-40dc-b984-f1880c96db69/13-flexbox.zip)

Aprende Flexbox jugando: [@aprendeFlexbox](https://flexboxfroggy.com/#es)

* Modulo Grid

[14-grid.zip](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/14114cce-499c-44d6-bead-9ba07f44171b/14-grid.zip)

Aprende Grid jugando: [@aprendeGrid](http://cssgridgarden.com/#es)

**Git Y GitHub**

* Instalación de git
* Comandos Git
* # Creación
* git init # Crea un repositorio local vacío en el directorio actual.
* # Este se almacena en el directorio .git.
* git status # Este es el más usado de todos. Nos dice la rama en la que
* # nos encontramos y los ficheros que contiene el stage junto a su estado
* # (new/modified/deleted). Además avisa de ficheros sin seguimiento por git
* # (untracked) o ficheros con conflictos.
* # Flujo entre working/stage/repo
* git add <lista de ficheros> # Manda uno o varios ficheros separados por
* # espacios o un directorio al staging area. Recién añadido un nuevo fichero
* # al repositorio, este se encuentra en estado "untracked" y debemos hacer
* # un primer add. Podríamos usar el carácter "." si queremos enviar todos los
* # ficheros del directorio de trabajo que tienen algún cambio.
* # También admite comodines, ej: git add \*.js
* git commit -m "descripción breve" # Registra definitivamente los cambios que
* # previamente estaban en el Stage.
* git commit -am "descripción" # Hace un commit con todos los ficheros con
* # cambios. Por tanto esta opción a es para realizar el add previo y el commit
* # de una sola vez. \*Importante: solo funciona si el fichero ya está en
* # seguimiento por git (tracked).
* # Un commit contiene:
* # id: Identifica el cambio de forma única. Es el hash SHA-1 del resto de
* # campos.
* # mensaje: que habitualmente se trunca a 72 caracteres al visualizarlo en el
* # log o github.
* # autor: El autor del cambio identificado por su name y su email.
* # id del árbol o snapshot en ese momento.
* # id del anterior commit.
* # Histórico de cambios
* git log # Abre un listado que nos va a permitir ver el histórico completo de
* # todos los commit que se han ido realizando en el repositorio a partir de
* # HEAD hacia atrás, incluyendo los que nos traemos del remoto cuando hacemos
* # un fetch. Funciona en modo interactivo y por defecto los últimos cambios
* # aparecen los primeros.
* # Para salir del log usa la tecla q
* git log --oneline # Nos permite ver un commit por linea, mostrando solo el id
* # y el mensaje.
* # Viajando en el tiempo
* git checkout <commit\_id|rama|HEAD> # Podemos usar checkout para visualizar el
* # estado del proyecto en un commit concreto, es decir, en ese momento en el
* # tiempo. Con esta operación lo que internamente estamos haciendo es apuntar a
* # ese commit con el puntero HEAD.
* # Inspeccionar un commit:
* git show <commit\_id> # Nos permite visualizar los cambios en el código que
* # incluye un commit concreto.
* # Recuperar/deshacer cambios
* git reset # Elimina commits. Técnicamente lo que hace es modificar el puntero
* # de la rama activa al valor que le indiquemos.
* git reset --soft <ref> # Elimina los commit que queden por detrás de la nueva
* # posición del puntero y todos los changeset son guardados en el Index/Stage.
* # Borrar n commits en modo soft y seguidamente un commit se suele llamar hacer
* # "squashing", ya que es como aplastar o compactar n commits en uno.
* # NOTA: En caso de compactar varios commits que modifican la misma línea de un
* # fichero, nos estaríamos quedando finalmente con el contenido que tuviese en
* # el commit más reciente en el tiempo.
* git reset --hard <ref> # CUIDADO! Todos los commit que queden por detrás de la
* # nueva posición del puntero son borrados para siempre.
* # Trabajar con ramas
* git branch # Nos muestra la rama activa.
* git branch <nuevarama> # Crea una nueva rama a partir del commit apuntado por
* # HEAD.
* git branch <nuevarama> <commit\_id> # Crea una nueva rama a partir del commit
* # indicado.
* git branch -d <rama> # Borra la rama.
* git branch --list # Listado de las ramas locales.
* git branch -a # Lista de ramas tanto locales como remotas.
* git branch <rama-local> <remoto/rama-remota> # Clonar una rama en concreto.
* # Recordemos que el comando clone solo trae la rama master.
* # NOTA: No nos posiciona en la rama clonada.
* git checkout <rama> # Cambia la rama activa a la indicada.
* git checkout -b <rama> # Hace dos pasos en uno: crea la rama a partir de la
* # activa y se cambia a ella.
* git checkout -b <rama-local> <remoto/rama-remota> # Clona la rama en concreto
* # pero además se posiciona en ella.
* git merge <rama> # Se fusiona la rama indicada con la rama activa en ese
* # momento, creando un commit de merge, que tiene dos commit padres, que
* # podremos ver en el log.
* git merge <rama> --squash # Se fusiona la rama indicada en la rama activa,
* # pero sus n commits se juntan en uno solo, de forma que en la historia de la
* # rama activa solo figurará un commit de mergeo.
* # Repositorios remotos
* git clone <url> <nombre\_directorio> # Descarga en local una copia de la rama
* # master (únicamente) de un repositorio. Además configura un repositorio
* # remoto con nombre origin asociándole la url desde la que hemos hecho clone.
* # De este modo ya podríamos hacer pull sin tener que añadirlo manualmente.
* git push <repo-remoto> <rama> # Envía todos los commits de la rama indicada
* # que no existan en el repositorio remoto.
* git pull <repo> # Es el equivalente a realizar un fetch + merge. Si no
* # indicamos repositorio, por defecto usa el repositorio origin.
* git remote add <alias\_repo> <url> # Nos permite añadir un repositorio remoto
* # y asociarlo con un alias. Como decía al principio del artículo, git clone
* # automáticamente crea el alias origin apuntando a la url desde la que
* # clonamos.
* git remote -v # Lista todos los alias a repositorios remotos que tenemos dados
* # de alta en nuestro repositorio local.
* # Fichero .gitignore
* # Este fichero que se encuentra en la raíz del working directory y sirve para
* # hacer que git ignore la existencia de ciertos ficheros o directorios, de
* # forma que no realice ningún seguimiento de estos sin reportarlo en el git
* # status. La idea es incluir aquí los ficheros autogenerados, temporales,
* # compilados etc, que se obtienen a partir de los fuentes. Admite diferentes
* # comodines y ciertas expresiones:
* # comentarios: líneas precedidas por #
* # \*.tmp: Podemos usar el comodín para indicar cero o más caracteres. Ej.
* fichero.tmp, fichero\\prueba.tmp, log.tmp\_
* # fichero?.tmp: La interrogación indica exactamente un carácter. Ej.
* fichero1.tmp, fichero2.tmp
* # \*\*/fichero.txt: EL doble asterisco sustituye a cero o n directorios. Ej:
* src/components/component1/fichero.txt, src/api/index.js, /fichero.txt
* # fichero\_[a-z].tmp: podemos usar conjuntos de caracteres definidos por rangos
* # Ej:
* fichero\\a.tmp, fichero\_b.tmp\_
* # fichero\_[0123].tmp: podemos usar conjuntos específicos. Ej:
* fichero\\1.tmp, fichero\_2.tmp y fichero\_3.tmp\_
* #!fichero.tmp: Podemos incluir excepciones a ficheros ignorados en las líneas
* # previas.
* dir/ # Debe incluirse la barra final para ignorar todo el directorio y su
* # contenido. Si no incluimos la barra entonces ignora directorios con ese
* # nombre pero también ficheros con el mismo. Ej:
* dir/fichero.txt, dir/subdir/fichero.txt, dir/subdir2/fichero.txt
* directorio/subdir/fichero.txt # un fichero en una ruta concreta.
* # Se puede tener un .gitignore global para todos nuestros repositorios,
* # situándolo en el directorio de usuario.
* # Etiquetas | versiones
* git tag # Nos muestra si hay tags creados.
* git tag -a <nombre\_tag> -m "descripción breve" <commit\_id> # Crea un nuevo tag
* # a partir del commit indicado.
* git show-ref --tags # Listado de los tags creados.
* git push origin --tags # Enviar los tags al repositorio.
* git tag -2 <nombre\_tag> # Eliminar un tag.
* git push origin :refs/tags/<nombre\_tag> # Eliminar del repositorio.
* # gitk: Gitk se ha convertido en parte oficial del desarrollo de git, y es un
* # paquete que (creo) que se instala cuando instalamos git. Gitk nos ofrece
* # mucha información, ordenada y herramientas para poder gestionar esa
* # información.
* # Vemos de un vistazo qué commits se han realizado, vemos de una manera
* # gráfica como discurre el historial del repositorio, quien ha realizado los
* # commits y en qué fechas.
* # También nos permite buscar un texto dentro de todo el historial, y comparar
* # commits. Hay herramientas con un aspecto más moderno, pero no publicadas
* # como software libre. Además gitk es estable, y consume pocos recursos.
* Registro en github
* Subiendo al repositorio remoto
* Material

[15-git.zip](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/cfc616ed-c5dd-4dc4-b080-7004f9f1c2fc/15-git.zip)