

# CONTROL DE VERSIONES

## ¿Qué es y para qué sirve?

**Git** es un sistema de control de versiones libre y distribuido diseñado para gestionar proyectos. Su objetivo es controlar y gestionar una enorme cantidad de ficheros de forma fácil y eficiente.

Se basa en repositorios que se inicializan en un directorio concreto y tienen toda la información de los cambios realizados.

## Partes de un repositorio

El repositorio remoto puede ubicarse en GitHub, GitLab, Bitbucket, SourceForge, entre otros. Es una copia del repositorio de git que se encuentra alojado en un servidor o en un sitio distinto al local.

El repositorio local se divide en:

- **Directorio de trabajo** (working directory): donde se almacena el contenido del repositorio.
- **Área de preparación** (staging area o index o cached): donde están indicados los cambios de confirmación.
- **Repositorio local** (local repository): donde están todas las versiones de manera local.

## Estados de un fichero

- **untracked**: Son ficheros que existen en el área de trabajo pero no existen para Git. *No han sido añadidos al seguimiento del repositorio.*
- **staged**: Son ficheros que se han añadido al *staging area* y están listos para ser incluidos en el siguiente commit. *Añadidos con `git add`.*
- **committed**: Son ficheros que se guardaron en el último commit y que no han sido modificados desde entonces. *Están en el repositorio y no tienen cambios*

*pendientes.*

- **modified**: Son ficheros que se han modificado desde el último commit pero que aún no se han añadido al *staging area*. *Han sido editados en el área de trabajo, pero Git aún no los ha preparado para el commit.*
- **ignored**: Son ficheros que Git ignora debido a las reglas definidas en el archivo `.gitignore`.

## Comandos básicos para gestionar ficheros en Git

1. **Añadir un fichero al staging area:**

```
git add fichero
```

2. **Confirmar los cambios del staging area al repositorio:**

```
git commit -m "mensaje"
```

3. **Incluir en el commit los cambios en los ficheros *staged* y también en los *modified*:**

```
git commit -a -m "mensaje"
```

4. **Quitar un fichero del staging area (opuesto a `git add`):**

```
git restore --staged fichero
# Alternativa menos usada:
git reset fichero
```

5. **Quitar un fichero del staging area pero mantenerlo en el área de trabajo:**

```
git rm --cached fichero
```

6. **Borrar un fichero del staging area y del área de trabajo:**

```
git rm fichero
```

7. **Restaurar un fichero modificado en el área de trabajo al estado del último commit:**

```
git restore fichero
# Alternativa menos usada:
git checkout fichero
```

## Comprobación del estado de los ficheros:

Para ver el estado de los ficheros en las diferentes áreas (untracked, staged, modified):

```
git status
```

El comando muestra:

- Archivos en la staging area (cambios a ser confirmados).
- Archivos modificados pero no añadidos al staging area.
- Archivos sin seguimiento (untracked).

Ejemplo de salida:

```
En la rama main
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacarlos del área de stage)
  nuevos archivos: facturas.html

Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git restore <archivo>..." para descartar los cambios en el directorio d
  modificados:      main.js

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlos en el área de stage)
  cliente.html
```

## Notas importantes:

- Si modificas un fichero después de usar `git add` y no vuelves a añadirlo con `git add`, el commit solo incluirá los cambios que estaban en el staging area cuando ejecutaste `git add`.

- Para evitar este problema, se puede usar el comando `git commit -a`, que automáticamente añade los ficheros modificados al staging area antes de confirmar los cambios.

## Sincronización de Git y Ramas

Una misma rama en git puede estar en varios sitios a la vez. El repositorio es el histórico donde están todos los commits que se han hecho.

Explicación de las ramas suponiendo que estamos en `master`:

- Carpeta local o `área de trabajo` o árbol de trabajo: Es donde están los ficheros con lo que trabajamos nosotros directamente mientras programamos.
- `Repositorio local`: Es el repositorio que tenemos en nuestro ordenador.
  - `Rama master`: Es donde se guardan los ficheros al hacer commit
  - `Rama origin/master`: Es una copia que hay del repositorio remoto.
- `Repositorio remoto`: Es un repositorio que está en otra máquina.
  - `Rama master`: Es donde queremos copiar los commits de nuestra rama master. Al hacer referencia a una rama de un repositorio remoto, se usará la expresión `origin master` con un espacio en medio.

Si queremos movernos entre ramas:

```
git switch master
git checkout master # Comando más antiguo, se recomienda switch
```

Si queremos crear una nueva rama:

```
git branch <nombreRama>
```

Si queremos crear una rama y automáticamente movernos a ella:

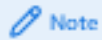
```
git switch -c <nombreRama>
git checkout -b <nombreRama> # Comando antiguo, se recomienda switch
```

Si queremos movernos a un commit en concreto con `switch`:

```
git switch --detach <hashDelCommit>
```

Si queremos movernos a una rama de tipo `origin/rama` debemos hacerlo con:

```
git switch --detach origin/rama
```



**DETACH** nos informa de que no se puede crear commit al lugar al que nos movemos. Es decir, no podemos trabajar en ese lugar

Si queremos publicar los cambios de una rama:

```
git push origin master
git push
```

Si queremos obtener los cambios del remoto:

```
git fetch origin master
git fetch

# FETCH TIENE LA OPCIÓN --PRUNE PARA BORRAR LAS CARPETAS QUE HAN SIDO BORRADAS
git fetch --prune
```

Si queremos obtener y aplicar en nuestra área de trabajo los cambios del remoto:

```
git pull
```



`git pull` hace un fetch y un merge, lo que puede producir conflictos que posteriormente haya que solucionar y puede que el histórico se modifique.

Para `mergear` lo que hay en la rama `origin/master` en la rama `master`:

```
git merge origin/master
```

Si queremos eliminar una rama en local:

```
git branch -d <rama>
```

Si queremos eliminar una rama en remoto:

```
git push origin -d <rama>
```

Para fusionar ramas tenemos dos opciones:

```
git merge <nombreRama>
git rebase <nombreRama>
```

## ¿Diferencia entre Merge y Rebase?

**Merge** conserva la historia original de la rama fusionada y **rebase** reescribe la historia de la rama para que parezca que los commits se aplicaron de forma lineal. En otras palabras, **merge** fusiona e incorpora los cambios especificados a la rama actual (*donde está el HEAD*) mientras que **rebase** mueve el HEAD a la rama especificada.



HEAD es donde estamos ubicados.

## Remotes

En Git un "remote" hace referencia a un servidor de Git donde subimos el código. Por defecto cuando hacemos git clone se crea un remote llamado origin.

Si queremos ver la lista de remotes:

```
git remote -v
```

Para añadir un remote:

```
git remote add otroservidor <URL>
```

# Borrado de commits

`GIT RESET` modifica el estado actual de un fichero.

Tienes diferentes opciones:

- `--soft`: Vuelve a un commit y conserva los cambios en el directorio de trabajo y en el stage.

```
git reset --soft <hashCommit>
```


- `--mixed`: Los cambios se conservan en el stage.

```
git reset --mixed <hashCommit>
```

- `--hard`: Los cambios no se conservan.

```
git reset --hard <hashCommit>
```

	Borra los commits siguientes	Se modifica área de trabajo	Modifica staged area
--hard	Si	Si	Si
--mixed	Si	No	Si
--soft	Si	No	No

 **Note**

La opción `HEAD~1` hace referencia al último commit.


# Deshacer commits

`GIT REVERT` deshace un commit y crea uno nuevo volviendo al estado del commit especificado.

```
git revert HEAD~1
git revert <hashCommit>
```

- Prepara un nuevo commit que deshace el último commit pero sin realmente hacer el commit. Lo que hace el modificar los ficheros y añadirlos a la staged area. Después se hace el commit. La opción `--no-commit` es útil porque nos permite revisar que va a hacer el commit y nos permite hacer alguna modificación más. Si no estamos interesados en hacer el commit lo podemos abortar con `git revert --abort` o usar el ya conocido `git restore --staged --worktree fichero`.

```
git revert --no-commit HEAD~1
git commit -am "Mensaje de commit" # -a para añadir a stage & -m para el mensaj
```

 **Note**

`git restore --staged --worktree fichero` restaura el archivo `fichero` tanto en el área de preparación como en el directorio de trabajo a su estado en el último commit, eliminando cualquier cambio realizado.

# Configuración

## `.gitkeep`

Por defecto Git no sube las carpetas vacías, por lo que si queremos que las suba hay que añadir algún fichero. Por convención se suele crear un fichero llamado `.gitkeep` y al ya haber algún fichero, git subirá esa carpeta.

## `.gitignore`

El fichero `.gitignore` permite indicar que carpetas no se deben subir a git. El fichero debe estar en el raíz del proyecto de git.

## Almacenando contraseñas

Normalmente trabajamos siempre en nuestro ordenador y no queremos todas las veces volver a poner la contraseña. En ese caso se puede decir a git que la almacene y no nos la vuelva a pedir:

- No te vuelve a pedir la contraseña en 15 min

```
git config --global credential.helper cache
```

- Almacena la contraseña en el disco en texto plano y nunca mas te la vuelve a pedir.

```
git config --global credential.helper store
```

## Stash

Es un almacén que permite guardar los cambios de los que todavía no se quiere hacer un commit.

- Guardar solo lo modificado (estén o no en el staged area):

```
git stash push
```

- Guarda todos los ficheros aunque NO estén en la stage area. Es decir, también los ficheros nuevos. Y al hacer el pop se restaura el estado en el staged area.

```
git stash push --include-untracked
```

	Por defecto	--include-untracked	--all
Los ficheros modificados (estén o no en el <i>staged area</i> )	✓	✓	✓
Los ficheros nuevos		✓	✓
Los ficheros los ignorados			✓

- Y si luego queremos recuperar los cambios se hace con:

```
git stash pop --index # Si no se añade --index, no se restaurará el staged area
```

## Log

El comando git log permite ver el histórico de commits.

- Ver los commits en una única línea cada commit.

```
git --no-pager log --pretty=oneline
```

- Ver los commits indicando que campos queremos mostrar e indicando el formato de la fecha.

```
git --no-pager log --pretty=tformat:"%h %cn %cd %s" --date=format:"%d/%m/%Y %H:"
```

Pretty format

```
git log --pretty=format:"%H"
```

See the next tables on format variables.

Commit

%s	commit subject
%F	commit subject, filename style
%b	commit body
%d	ref names
%e	encoding

Author and committer

Author

Name	author
%an	author, respecting mailmap
%aE	author email, respecting mailmap
Date	author date (f1c2882)
%ar	author date (relative)
%at	author date (unix timestamp)
%ai	author date (iso8601)

Committer

Name	committer name
%cn	committer name, respecting mailmap
%ce	committer email
%cE	committer email, respecting mailmap
Date	committer date (f1c2882)
%cr	committer date (relative)
%ct	committer date (unix timestamp)
%ci	committer date (iso8601)

Hash

Commit	
%H	commit hash
%h	(abbrev) commit hash
Tree	
%T	tree hash
%t	(abbrev) tree hash
Parent	
%P	parent hash
%p	(abbrev) parent hash

Presets

Date		%a	Sun	Weekday
%a/%d/%Y	06/05/2013	%A	Sunday	
%A, %B %e, %Y	Sunday, June 5, 2013	%w	6.6 (Sunday is 0)	
%b %e %a	Jun 5 Sun	%y	13	Year
Time		%Y	2013	
%H:%M	23:05	%b	Jan	Month
%I:%M %p	11:05 PM	%B	January	
Used by Ruby, UNIX date, and many more.		%m	01.12	
		%d	01.31	Day
		%e	1.31	

Time

%l	1	Hour
%H	00.23	24h Hour
%I	01.12	12h Hour
%M	00.50	Minute
%S	00.60	Second
%p	AM	AM or PM
%Z	+08	Time zone
%j	001.306	Day of the year
%%	%	Literal % character

## Mensajes de commit

Los mensajes de commit deben seguir el siguiente formato:

```
type(#issue):titulos  
  
Explicación (opcional)
```

Siendo

- **type**
  - **feat**: Si se añade una nueva funcionalidad (feature)
  - **fix**: Si se arregla (fix) un error
  - **docs**: Si solo cambia cosas de la documentación
  - **style**: Si solo se cambia el estilo del código como tabuladores, puntos y comas, formateo, etc.
  - **refactor**: Si se cambia el código para mejorar su calidad pero sin modificar la funcionalidad. Eso se llamada refactorizar.
  - **test**: Si se cambian cosas relacionadas con test automáticos.
  - **chore**: Si se cambian cosas relacionadas con el despliegue.
- **#issue** : Es el N° de la incidencia a la que hace referencia.

Ejemplos:

- Se arregla el fallo 45 que se llama "Falla si la fecha está vacía"

```
fix(#45):Falla si la fecha está vacía
```

- Se añade la nueva funcionalidad llamada "Mostrar el listado de los pacientes" con N° 456

```
feat(#456):Mostrar el listado de los pacientes
```

# CONTROL DE VERSIONES AVANZADO

## Necesidad de ramas

Las ramas con las cuales vamos a trabajar son las siguientes:

- main / master
- hotfix
- release
- develop
- features

Sigue una estrategia muy similar a [GitFlow](#), la cual permite el trabajo en equipo en un proyecto. Su funcionamiento es muy sencillo, cada miembro tiene asignada una feature que hace referencia a un issue, una vez terminada la funcionalidad se une a develop, posteriormente develop y release se unen, al igual que main y release. Esto permite crear un histórico lineal.



Note

Las ramas de funcionalidad una vez terminadas deben ser eliminadas.



Note

Las ramas de funcionalidad deben nombrarse como:

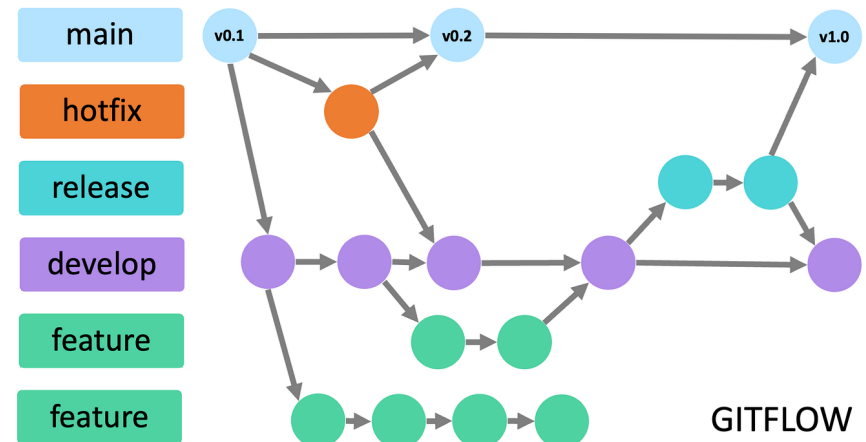
`nombrePersona/númeroIssue`

Un ejemplo sería: pedro/4

Quiere decir que en el issue 4, en esa funcionalidad está trabajando Pedro.

Cada miembro todos los días antes de empezar a trabajar debería obtener los cambios del remoto y ver donde se encuentra `develop` y así ver si es necesario obtener los cambios más recientes de los compañeros, por si hubieran conflictos con su código ir solucionando los poco a poco y no todos de golpe una vez terminada la feature cuando se va a unir a develop. Es decir debería hacer un `git fetch && git merge`.

Representado de forma gráfica un histórico sencillo podría verse de la siguiente forma.



## Definición de las ramas

- **develop**: Lo que se ha desarrollando y el equipo de desarrollo dice que ya funciona. Cuando se quiera se pasa a **release**.
- **release**: Lo que se esta probando en un entorno similar al de producción. Cuando está probado, se copia a **master**.
- **master**: Lo que ya se puede instalar en el producción.
- **rama de funcionalidad (feature)**: Cada vez que alguien quiere hacer algo, crea una rama desde **develop** y cuando acaba la fusiona en **develop**.



Tip

En la rama de funcionalidad se pueden hacer todos los commits que queramos ya que luego se van a unificar en uno solo. Es lo que llamamos "microcommits". Es así ya que son commits *parciales* que se hacen para acabar la tarea pero finalmente en la rama **develop** queremos que sean un solo commit.

Ese único commit debe tener el formato anteriormente explicado (**Control de versiones (Git) > Mensajes de commit**):

`type(#issue):titulos`

# Pasos para hacer una modificación

- Crea la rama `feature` desde `develop`.

```
git branch pedro/4
git switch pedro/4
```

- Realizar la tarea con tantos commits como sea necesario.

```
touch file1.txt
git commit -a -m "cambio 1"

touch file2.txt
git commit -a -m "cambio 2"

touch file3.txt
git commit -a -m "cambio 3"
```

- Mergear la rama `develop` en `feature` ya que así se resuelven los problemas de mergeo en la rama `feature`

```
git switch pedro/4
git merge develop # Resolver conflictos si hay
```

- Ir a la rama `develop` y mergear la rama `feature` con `--squash`

```
git switch develop
git merge --squash pedro/4
git commit -m "feat(#12):Login validation"
```

- Borrar la rama `feature`

```
git branch -d pedro/4 # local
git push origin --delete pedro/4 # remoto
```

- Subir la rama `develop`

```
git push develop
```

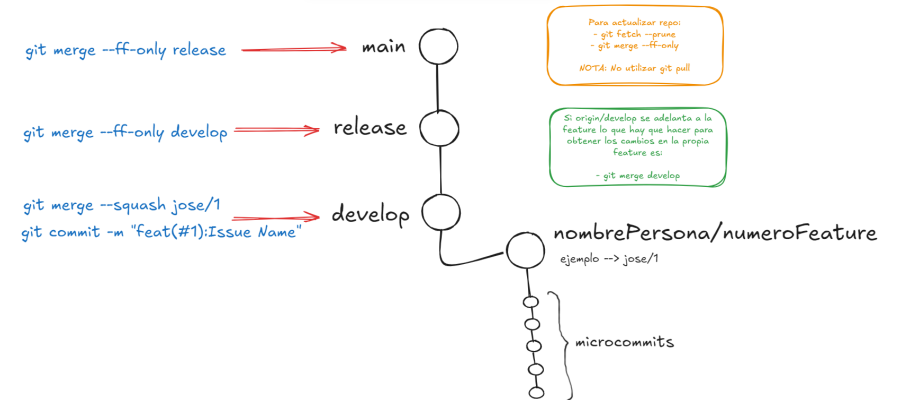
- Mergear la rama `develop` en `release` y subir `release`

```
git switch release
git merge --ff-only develop
```

- Mergear la rama `release` en `master` y subir `master`

```
git switch main
git merge --ff-only release
```

## Estrategia de trabajo Git



### Note

```
git merge --squash pedro/4
```

Se utiliza para juntar todos los microcommits en un solo commit que habrá que indicar. Se usa para pasar los microcommits de una rama `feature` a `develop`

### Note

```
git merge --ff-only develop
```

La opción `--ff-only` es usada para que si la unión no es `Fast-Forward` no se llegue a hacer y falle indicándolo.

La unión puede ser `Fast-Forward` que indica que no hay problemas o en el caso de haber conflictos serían ramas divergentes.



## Cuando usar MERGE y REBASE y de qué tipo utilizarlo

COMANDO	CARACTERÍSTICAS	CUANDO USARLO
<code>git rebase rama</code>	Pone los commits en la nueva rama siempre después de los que ya hay, es decir que los reordena	Para incluir los cambios de origin/develop en develop. Pasos: <code>git fetch --prune</code> <code>git switch develop</code> <code>git rebase origin/ develop</code>
<code>git merge --squash rama</code>	Junta todos los "microcommits" de la rama en uno solo en la rama destino. <i>Es necesario acto seguido el commit</i>	<i>De una rama feature a develop.</i>
<code>git merge --ff-only rama</code>	Nunca crea nuevos commits en la rama destino pero falla si es necesario crear un nuevo commit, por lo que <code>--ff-only</code> ayuda a detectar si hay algún problema	<i>De develop a Release.</i>  De release a master o para actualizar cualquier rama desde GitHub cuando no hay commits nuevos en local
<code>git merge rama</code>	Es el merge normal que puede crear nuevos commits	<i>De una rama develop a feature.</i>  Para resolver los conflictos desde nuestra rama feature ya que nos da igual si hay nuevos commits

- Subir las 3 ramas

```
git switch master && git push && git switch release && git push && git switch d
```

## Mini-Script de automatización

- Bajarse lo último de las 3 ramas.

```
git fetch --prune && git switch develop && git merge --ff-only origin/develop &
```

- Hacer un merge de las 3 ramas

```
git switch release && git merge --ff-only develop && git switch master && git m
```

# INTRODUCCIÓN

## LA WEB

---

La web son páginas (En formato HTML, Imagen, JSON, XML ,etc) que se interconectan entre ellas por enlaces (urls).

El navegador (cliente) solicita por **TCP/IP** el recurso a obtener. El formato de como solicita ese recurso es por el protocolo HTTP.

Y el servidor de TCP/IP responde usando el protocolo HTTP devolviendo los datos. A ese software lo llamaremos servidor Web. El servidor es "personalizable" permitiendo que se ejecute código específico para nuestra aplicación. Ese código específico es el código de servidor que se suele escribir en Java, PHP, NodeJS, etc.

El servidor no solo puede comunicarse con servicios en local, sino también puede acceder a servicios externos.

## Chrome DevTools

---

Los navegadores suelen llevar herramientas para depurar las páginas web. En Chrome , si pulsamos F12 y pinchamos en la opción de "Network" (Menú superior), podemos ver todas las peticiones que hace el navegador al cargar una página.

## Desplegar

---

Desplegar es instalar a aplicación web que hemos desarrollado (HTML,CSS,JS,Código Servidor, etc) en un servidor Web.

## Desafíos y tareas para desplegar

---

### 1. Generación de la aplicación a instalar

1. Obtener el código fuente
2. Compilar el código fuente
3. Probar la aplicación
4. Analizar la calidad del código

### 2. Múltiples programas a instalar

1. Sistema Operativo
2. Servidor Web
3. Base de datos
4. La aplicación
5. Servidor de envío de correos
6. Etc.

### 3. Hosts

1. La base de datos puede estar en un Host y el servidor web en otro Host, etc.
2. Los Hosts pueden ser máquinas virtuales o máquinas físicas, estando cada una de ellas en proveedores distintos (Amazon AWS, Microsoft Azure, Google Cloud, etc).

### 4. Administración

1. Creación de copias de seguridad
2. Logs
3. Seguridad: Recuperación en caso de pérdida de datos por algún fallo

### 5. Rendimiento: Permitir que la aplicación siga funcionando aunque haya un pico de peticiones

1. Añadir nuevas máquinas si hay un pico de peticiones y quitarlas cuando ya no las hay tantas peticiones
2. Balanceo de carga entre todos los servidores de la aplicación
3. Monitorización del rendimiento
4. Alertas de bajo rendimiento

### 6. Fiabilidad: Permitir que la aplicación siga funcionando aunque falle algún Host.

### 7. Microservicios:

1. Una única aplicación se divide en pequeñas micro-aplicaciones, llamadas microservicios, con lo que se multiplican todas las complicaciones anteriores por el número de microservicios que tengamos.

## Herramientas relacionadas con el despliegue

---

Herramientas que vamos a usar en clase:

- Script de Bash
- Git
- Docker
- Git Actions
- NodeJS

Otras herramientas y Servicios:

- Kubernetes
- Vagrant
- AWS
- Ansible
- Jenkins
- etc. Hay muchísimas herramientas distintas y servicios que ayudan al despliegue de aplicaciones.

## Conceptos

- **VPS (Virtual Private Server)**: Una máquina virtual que alquilas a una empresa.
- **Empresa de Hosting**: Empresa que alquila los Host sean tanto un VPSs como una máquina real
- **Balanceador de carga**: Software que le llega una petición y la redirige a otro Host de entre varios para no sobrecargar ningún Host o evitar enviarlo a un host que no funciona
- **Escalabilidad**: Diseñar la aplicación de forma que se alquilen o se desalquilen Hosts (VPS o máquinas reales) en función de la carga del sistema
- **Tolerancia a fallos**: Diseñar la aplicación de forma que aunque un host falle, la aplicación siga funcionando

## IAAS - PAAS - SAAS

- **IAAS (Infraestructura como servicio)**: Si la empresa de Hosting solo se ofrece el Host y nosotros nos tenemos que instalar todo el software, incluyendo el sistema operativo y administrarlo todo.
- **PAAS (Plataforma como servicio)**: Si la empresa de Hosting nos ofrece el Host pero también software genérico ya instalado como el Sistema Operativo, Servidor Web, Servidor de Correo, Balanceador de Carga , etc. En este caso nosotros solo debemos instalar el código específico de nuestra aplicación. En este caso aunque nos ofrecen un host ya que en algún sitio debe estar la app, realmente nos están ofreciendo el servidor web donde instalar nuestra app. En el caso de PAAS, no tenemos que administrar nosotros ni el Sistema Operativo ni el servidor.
- **SAAS (Software como Servicio)**: Como desarrolladores nunca usamos un SASS ya que la empresa de hosting ya ofrece hasta la aplicación instalada. Un ejemplo sería Google con "Google Docs" , Microsoft con su "MS Office 365" , Dropbox , etc. que ya ofrecen todo para el usuario final.

## IAAS vs PAAS

El IAAS es mas versátil ya que solo nos ofrecen el ordenador y nosotros nos montamos todo como queremos. El problema es que es mas complicado todo de hacer y tenemos que administrarlo todo: Sistema operativo, servidor web y aplicación.

Por otro lado en el PAAS, solo nos tenemos que preocupar de nuestra aplicación lo que hace que sea mas sencillo. El problema es que ya no hay tanta versatilidad, ya que debemos ceñirnos al entorno que nos ofrece la empresa.

## NodeJS

NodeJS (o simplemente node) es un lenguaje de programación basado en JavaScript. Al ser un lenguaje interpretado, su forma de trabajar es mas similar a BASH que a Java. Además, incluye un gestor de paquetes: **npm (Node package manager )**

## Proyectos con node

- Inicializar un proyecto en Node. Se crea el fichero package.json que es fundamental en node ya que contiene toda la información del proyecto

```
npm init
```

SHELL

- Instalar un paquete nuestro proyecto. Se guarda en la carpeta node\_modules:
  - Instalar la librería de JavaScript llamada "jQuery"

```
npm install jquery
```

SHELL

- Instalar un paquete de forma global. Se guarda en "/usr/bin"

```
npm install typescript -g
```

SHELL

- Instalar todo de nuevo si no está la carpeta node\_modules

```
npm install
```

SHELL

## Programar en NodeJS

---

Ahora vamos a ver como ejecutar código node.

El programa mas sencillo es hacer el "Hola Mundo". Para ello creamos un fichero llamado "index.js" con el contenido siguiente:

```
#!/usr/bin/env node

console.log(`'Hola Mundo'`);
```

Para ejecutarlo hay que lanzar la orden:

```
node index.js
```

SHELL

*Y mostrará por consola el mensaje "Hola mundo"*

*Como en NodeJS se usa JavaScript, podemos usar todo lo que sabemos de JavaScript en un programa de node.*

## Unicode y UTF-8

---

### Unicode

---

Unicode es un estándar de codificación que tiene como objetivo permitir la representación de todos los caracteres utilizados en los sistemas de escritura del mundo, además de símbolos y emojis. A diferencia del ASCII (7 bits) y ANSI (8 bits), Unicode asigna a cada carácter un número único, denominado "punto de código" y los puntos de código son desde U+0000 hasta U+10FFFF (2.097.152 de caracteres).

Además de unicode están:

- **ASCII**: ASCII es un código de 7 bits que permite representar 128 caracteres diferentes, incluyendo las letras mayúsculas y minúsculas del alfabeto inglés, los números, algunos símbolos de puntuación y caracteres de control (como el salto de línea o el tabulador). Fue desarrollado en los años 60 y está diseñado principalmente para manejar textos en inglés, ya que no incluye acentos ni caracteres especiales de otros idiomas.
- **ANSI**: Amplían el estándar ASCII a 8 bits. Sin embargo hay distintas codificaciones ANSI en los nuevos 128 caracteres para distintos idiomas como ruso o griego. Es decir es como que hay distintos estándares ANSI llamados ISO-8859-x. Por ejemplo

hay un estándar ANSI para el griego (ISO 8859-7), otro para el cirílico (ISO 8859-7), etc. Y son distintos en los 128 caracteres últimos. Cada uno de estos estándares se llama Pagina de código.

- **Windows**: Windows tiene su propio estándar que es muy similar al ANSI

### UTF-8

---

UTF-8 (**Unicode Transformation Format - 8 bits**) es un esquema de codificación variable que utiliza entre 1 y 4 bytes para representar cada carácter Unicode. Es altamente eficiente para textos en idiomas que utilizan caracteres latinos comunes, ya que estos se representan en un solo byte. UTF-8 se ha convertido en la codificación más utilizada en la web debido a su compatibilidad con sistemas más antiguos que utilizan ASCII y su capacidad para codificar cualquier carácter Unicode. Además, es muy eficiente en términos de almacenamiento para la mayoría de los lenguajes occidentales.

## TCP/IP Y UDP

Tanto TCP/IP como UDP son protocolos que forman parte de la suite de protocolos **Internet Protocol (IP)**, pero cada uno tiene un comportamiento diferente.

**TCP/IP** antes de transferir establece una conexión entre el servidor y el receptor, garantizando que ambos están listos para intercambiar información. Además, **garantiza la entrega correcta de datos**, ya que divide los datos en segmentos y asegura que lleguen en el orden correcto. La desventaja es que al ser más fiable es más lento.

*Algunas aplicaciones comunes son navegación web (HTTP/HTTPS), correo electrónico (SMTP), transferencia de archivos (FTP).*

Mientras que **UDP** no establece una conexión previa para enviar datos. Simplemente envía los datos sin asegurarse de que lleguen correctamente, lo cual no garantiza fiabilidad a pesar de ser más rápido.

*Algunas aplicaciones comunes son streaming de video y audio, juegos en línea, transmisiones en tiempo real, VoIP, donde la velocidad es más importante que la fiabilidad.*

### Comparación resumida:

---

- **TCP**: Fiable, orientado a la conexión, control de flujo, reenvía datos perdidos, más lento.
- **UDP**: No fiable, sin conexión, más rápido, pero sin garantía de entrega o de orden.

## ADMINISTRACIÓN DE SERVIDORES WEB

Hay dos tipos de servidores web:

- **Servidores Web Externos**: Son programas completos que hacen de servidor Web. Una vez instalados/ejecutados se añade el código específico de la aplicación en la carpeta del servidor que indique la documentación. Algunas herramientas son: Apache2 y Nginx.
- **Servidores Web Integrados** (o librerías de servidores web): Se hace una aplicación (por ejemplo en Java) y se añade como una librería (un JAR), el código del Servidor Web. Se puede hacer con NodeJS y la librería Express.

### DIFERENCIAS ENTRE AMBOS

---

**Servidor Externo**: gasta menos RAM pero cada proyecto no es independiente, todos comparten el mismo servidor.

**Servidor Integrado**: gasta mas RAM, pero es más versátil, se utiliza virtualización (docker).

## servidores web externos

### Apache HTTP Server Project

---

Este servidor Web es un proyecto de una fundación sin ánimo de lucro llamada Apache Software Foundation. El proyecto del servidor web se llama Apache HTTP Server Project por lo que a veces se confunde el nombre del servidor con el nombre de la fundación.

- **Instalación:** Para usar el servidor Apache solo hay que descargárselo y ejecutar el programa `"/bin/httpd.exe"`
- **Despliegue:** Para desplegar páginas web en Apache solo hay que copiarlas en la carpeta `"htdocs"`
- **Configuración/administración:** Para configurar/administrar el servidor hay que modificar los ficheros de la carpeta `"conf"`

## servidores web integrados

Pasemos ahora a ver como es un servidor web integrado. Para entenderlo vamos a usar el lenguaje NodeJS como ejemplo y la librería que tendrá el servidor será Express.

- Instalar paquete express:

```
npm install express
```

- Ejemplo básico: Ejemplo Hello world. Crear el fichero `"index.js"` con el siguiente contenido.

```
#!/usr/bin/env node

const express = require('express')
const app = express()
const port = 8080

app.get('/', (request, response) => {
  response.send('Hello from Express!')
})

app.listen(port, (err) => {
  console.log(`server is listening on ${port}`)
})
```

Ahora debemos ejecutar el fichero index.js

```
node index.js
```

Y si navegamos a `http://localhost:8080` veremos el texto `"Hello from Express!"`

- Servir páginas estáticas: Deberemos crear dentro del proyecto la carpeta `"HTML"` y ahí crear el fichero `"index.html"`. Crear el fichero `"index.js"` con el siguiente contenido.

```
#!/usr/bin/env node

const express = require('express')
const app = express()
const port = 8080
```

```
app.get('/', (request, response) => {
  response.send('Hello from Express!')
})
app.use('/html', express.static(__dirname + '/html'));

app.listen(port, (err) => {
  console.log(`server is listening on ${port}`)
})
```

Ahora debemos ejecutar el fichero index.js

```
node index.js
```

Y si navegamos a <http://localhost:8080/html/index.html> veremos la página que hemos creado.

# EL PROTOCOLO HTTP

*Es usado para enviar y recibir datos de la web.*

## Características

- **Sencillo**: es en modo texto y fácil de usar por una persona.
- **Extensible**: se pueden enviar más metadatos que los que están por defecto (Ej. nº de página).
- **Sin estado**: cada petición es independiente. Eso es un problema en sitios como por ejemplo un carrito de la compra.

## Ventajas

- **Cache**: mejora la velocidad al controlar la cache de las páginas.
- **Autenticación**: permite identificar usuarios.
- **Proxys**: permite de forma transparente el uso de proxys.
- **Sesiones**: Gracias a las cookies podemos mantener el estado entre peticiones.
- **Formatos**: Permite indicar el formato de lo que se envía, se pide y se retorna.

## Formato

Una petición HTTP tiene la siguiente forma:

```
GET /index.html HTTP/1.1
Host: www.fpmislata.com
Accept-Language: fr
```

- Donde cada parámetro nos da cierta información:
  - **GET** : Es el método por el que se piden los datos. Entre sus valores está: GET, PUT, POST, DELETE.
  - **/index.html** : Es la ruta dentro del servidor del documento que estamos

pidiendo

- **HTTP/1.1** : La versión del protocolo. Prácticamente siempre es 1.1
- **Host: www.fpmislata.com** : Indica el nombre del host al que va dirigida la petición.
- **Accept-Language: fr** : Indica en que idioma queremos que nos retorne los datos. En este caso es en francés.

La respuesta del servidor es la siguiente:

```
HTTP/1.1 200 OK
Content-Length: 29769
Content-Type: text/html; charset=utf-8

<!DOCTYPE html... (los 29769 bytes de la página)
```

- Donde cada parámetro nos da cierta información:
  - **HTTP/1.1** : La versión del protocolo con la que responde. Prácticamente siempre es 1.1
  - **200 OK** : Si ha sido exitosa o no la petición.
  - **Content-Length: 29769** : Indica las bytes que ocupan los datos que se retornan
  - **Content-Type: text/html; charset=utf-8** : Indica el formato MIME type de los datos que retornan y su codificación. En este caso es en HTML y en formato UTF-8.
  - **<!DOCTYPE html...** : Son finalmente los datos que se han pedido.

## Cabeceras HTTP

Las cabeceras se dividen entre las que se envían en la petición y las que se retorna en la respuesta.

### Petición:

Cabeceras que se pueden enviar en la petición

- **Accept** : El formato MIME type en la que queremos que se retornen los datos. Ej: En **text/html**, en **text/xml**, **application/json**, **application/pdf**, etc. Luego el



servidor los retornará en el formato que quiera/pueda

- **Accept-Language** : El idioma en el que queremos que nos retorne los datos. Luego el servidor los retornará en el idioma que quiera/pueda.
- **Host** : El dominio al que se está enviando la petición. Esta cabecera es muy útil ya que permite en un mismo servidor tener alojados varios dominios.
- **Content-Type** : El formato de los datos que envían al servidor. Ej: En `text/html`, en `text/xml`, `application/json`, `application/pdf`, etc. Y como están codificado. Normalmente los formatos son `utf-8` o `ISO-8859-1`.

## Respuesta

---

Cabeceras que se pueden enviar en la respuesta

- **Content-Type** : El formato de los datos que se retorna. Ej: En `text/html`, en `text/xml`, `application/json`, `application/pdf`, etc. Y como están codificado. Normalmente los formatos son `utf-8` o `ISO-8859-1`. No tiene porque coincidir con `Accept`.
- **Content-Language** : El idioma de los datos que se retorna.
- **Content-Length** : Tamaño en bytes de los datos
- **Cache-Control** : Cuanto tiempo pueden estar cacheado los datos.



### Note

La cabecera **Content-Type** es importante para el programador ya que el servidor puede no saber exactamente el formato de los datos y es necesario que lo indiquemos nosotros. Muchas veces hay además problemas con la codificación si es `utf-8` o `ISO-8859-1` por lo que también se debe indicar.

Por otro lado notar que **Content-Type** se puede usar tanto en la petición como en la respuesta. Se usa en la petición si se envían datos en la petición

## Estados HTTP

---

Es estado es lo que indica si una petición HTTP ha tenido éxito o no. Sus principales valores son:

- **200-299**: La petición ha tenido éxito
- **300-399**: Redirección de los datos.
- **400-499**: Los datos que ha enviado el cliente no son correctos
- **500-599**: Se ha producido un error en el servidor.

De entre todos los códigos están algunos que solemos ver a menudo:

- **200**: Todo ha ido bien.
- **201**: Se ha creado el recurso (Suele ser en un INSERT).
- **204**: La petición no retorna datos. (Suele ser en un DELETE).
- **400**: Los datos que ha enviado el cliente no son correctos.
- **401**: Hay que estar logueado.
- **403**: El usuario está logueado pero tiene Prohibido el acceso al documento.
- **404**: No encuentra el documento.
- **500**: Error del servidor.

## Métodos

---

Los métodos ( o verbos) HTTP indican que acción queremos hacer con los datos. Al navegar normalmente se usa siempre el `GET`.

- **GET** : Queremos obtener los datos
- **POST** : Queremos añadir los datos.
- **PUT** : Queremos actualizar nuevos datos.
- **DELETE** : Queremos borrar los datos.

## REST

---

**REST** es una interfaz para conectar varios sistemas basados en el protocolo HTTP (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON (*actualmente el más usado*).

## Las operaciones a realizar

Vamos a ver 4 método HTTP que coinciden con los 4 métodos de un CRUD o con operaciones de SQL:

Método HTTP	Descripción	Metodo CRUD	Metodo SQL
GET	Este método HTTP lo usaremos para cuando queremos leer datos del servidor	Read	SELECT
POST	Este método HTTP lo usaremos para añadir datos al servidor	Create	INSERT
PUT	Este método HTTP lo usaremos para actualizar datos del servidor	Update	UPDATE
DELETE	Este método HTTP lo usaremos para borrar datos del servidor	Delete	DELETE

## La Estructura de la URL

Veamos la estructura de la URL de las peticiones en un supuesto ejemplo de una base de datos de usuarios:

Descripción	URL	Método HTTP	JSON Enviado	JSON Retornado
Obtener un libro	/libro/{idLibro}	GET	Ninguno	Libro leído
Listado de libros	/libro	GET	Ninguno	Array de Libros
Añadir un libro	/libro	POST	Libro a insertar	Libro insertado
Actualizar un libro	/libro/{idLibro}	PUT	Libro a actualizar	Libro actualizado
Borrar un libro	/libro/{idLibro}	DELETE	Ninguno	Ninguno

## Servidor REST en NodeJS

```
const port = 80

app.get('/', (request, response) => {
  response.set('Content-Type', 'text/plain');
  response.status(200);

  if (request.header('Accept-Language').startsWith("ca-ES")) {
    response.send("Hola mon");
  } else if (request.header('Accept-Language').startsWith("en-EN")) {
    response.send("Hello World");
  } else {
    response.send("Hola mundo");
  }
});

app.post('/', (request, response) => {
  response.status(200);
  response.send('Hello from post!');
});

app.delete('/', (request, response) => {
  response.status(200);
  response.send('Hello from delete!');
});

app.delete('/libro/38', (request, response) => {
  response.status(200);
  response.send('Borrado libro 38');
});

app.delete('/libro/39', (request, response) => {
  response.status(404);
  response.send('El libro 39 no existe');
});

app.listen(port, (err) => {
  console.log(`server is listening on ${port}`)
})
```

```
const app = express()
```