Curs 2023 - 2024

Desplegament d'una aplicació "clusteritzada" amb Node Express

Guillermo Vidal Frasquet

Desplegament d'Aplicacions Web

Pràctica

IES Jaume II El Just
Tavernes de la Valldigna

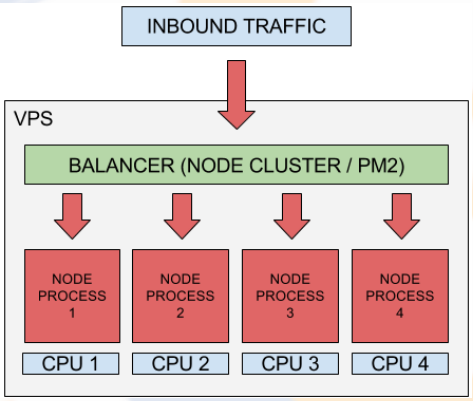
Continguts

1 Desplegament d'una aplicació "clusteritzada" amb Node Express

1.1 Introducció

Quan es construeix una aplicació de producció, normalment es busca la manera d'optimitzar el seu rendiment arribant a una solució de compromís. En esta pràctica donarem una ullada a un enfocament que pot oferir una victòria ràpida quan es tracta de millorar la manera en què les aplicacions `Node.js` manegen la càrrega de treball.

Una instància de `Node.js` s'executa en un sol fil, cosa que significa que en un sistema multinucli (com la majoria dels ordinadors de hui dia), no tots els nuclis seran utilitzats per l'aplicació. Per a aprofitar els altres nuclis disponibles, podem llançar un clúster de processos `Node.js` i distribuir la càrrega entre ells.



Tindre diversos fils per a manejar les peticions millora el rendiment (peticions/segon) del servidor, ja que diversos clients poden ser atesos simultàniament. Veurem com crear processos fills amb el mòdul de clúster de `Node.js` per a, més tard, veure com gestionar el clúster amb el gestor de processos PM2.

1.2 Una ullada ràpida als clústers

El mòdul de clúster de `Node.js` permet la creació de processos secundaris (*workers*) que s'executen simultàniament i comparteixen el mateix port de servidor. Cada fill generat té el seu propi cicle d'esdeveniments i memòria. Els processos secundaris utilitzen IPC (comunicació entre processos) per a comunicar-se amb el procés principal de `Node.js`.

Tindre múltiples processos per a manejar les sol·licituds entrants significa que es poden processar diverses sol·licituds simultàniament i si hi ha una operació de bloqueig/execució prolongada en un *worker*, els altres *workers* poden continuar administrant altres sol·licituds entrants; l'aplicació no es detindrà fins que finalitze l'operació de bloqueig.

L'execució de diversos *workers* també permet actualitzar l'aplicació en producció amb poc o cap temps d'inactivitat. Es poden fer canvis en l'aplicació i reiniciar els *workers* d'un en un, esperant que un procés secundari es genere per complet abans de reiniciar un altre. D'esta manera, sempre hi haurà *workers* executant-se mentre es produeix l'actualització.

Les connexions entrants es distribueixen entre els processos secundaris de dues maneres:

El procés mestre escolta les connexions en un port i les distribueix entre els *workers* de manera rotatòria. Este és l'enfocament per defecte en totes les plataformes, excepte Windows.

El procés mestre crea un `socket` d'escolta i ho envia als *workers* interessats que després podran acceptar connexions entrants directament.

1.3 Usant els clústers

1.3.1 Primer sense clúster

Per a veure els avantatges que ofereix l'agrupació en clústers, començarem amb una aplicació de prova en `Node.js` que no usa clústers i la compararem amb una que sí que els usa, es tracta de la següent:

```
const express = require("express");
const app = express();
const port = 3000;

app.get("/", (req, res) => {
    res.send("Hello World!");
});

app.get("/api/:n", function (req, res) {
    let n = parseInt(req.params.n);
    let count = 0;

    if (n > 5000000000) n = 5000000000;

    for (let i = 0; i <= n; i++) {
    count += i;
    }

    res.send(`Final count is ${count}`);
});

app.listen(port, () => {
    console.log(`App listening on port ${port}`);
});
```

Es tracta d'una aplicació una miqueta prefabricada en el sentit que és una cosa que mai trobaríem en el món real. No obstant això, ens servirà per a il·lustrar el nostre propòsit.

Esta aplicació conté dues rutes, una ruta arrel `/` que retorna la cadena `Hello World!` i una altra ruta `/api/n` on es pren n com a paràmetre i va realitzant una operació de suma (el bucle **`for`**) el resultat del qual acumula en la variable `count` que es mostra al final.

Si a este paràmetre `n`, li donem un valor molt alt, ens permetrà simular operacions intensives i d'execució prolongada en el servidor. Li donem com a valor límit 5000000000 per a evitar una operació massa costosa per al nostre ordinador.

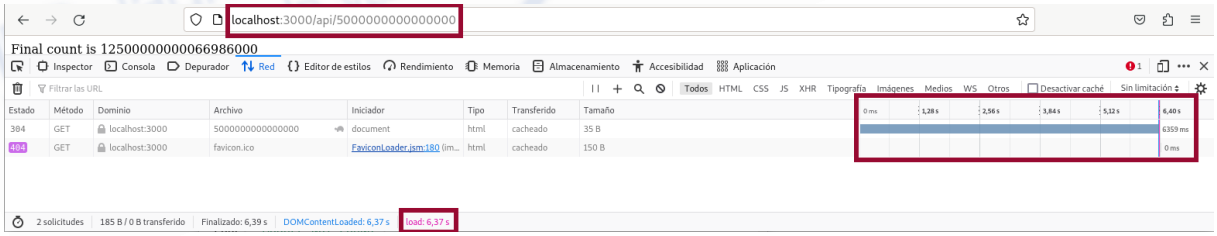
Tasca 1

1. Heu de connectar-vos al servidor Debian mitjançant SSH.
2. Heu de crear un directori per al projecte d'esta aplicació.
3. Dins del directori executareu 2 comandos:
    1. `npm init` per a crear automàticament l'estructura de carpetes i l'arxiu **`package`**`.json` (Amb anar donant-li a a totes les preguntes, hi ha prou).
    2. `npm install express` per a instal·lar `express` per a este projecte.
4. Després d'això, **DINS** del directori, ja podeu iniciar l'aplicació amb: `node nom_aplicacio.js`.

Per a comprovar-ho, podeu accedir a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/50` on `IP-maq-virtual` és la IP de l'adaptador pont de la vostra Debian.

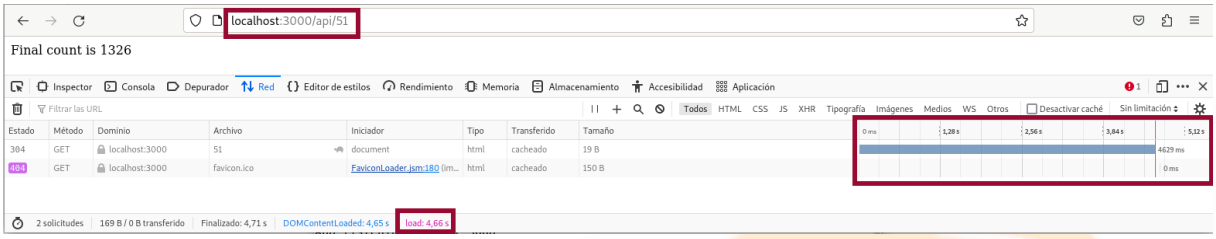
Utilitzeu un valor de `n` relativament xicotet, com el 50 de l'exemple anterior i comprovareu que s'executarà ràpidament, retornant una resposta quasi immediata.

Fem una altra simple comprovació per a valors de `n` més grans. Desplegada i iniciada l'aplicació, accediu a la ruta `http://IP-maq-virtual:3000/api/5000000000`.



Mentre esta sol·licitud que tarda uns segons s'està processant, accediu en una altra pestanya del navegador a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/n`

sent n el valor que li vulgueu donar.



Utilitzant les `devoloper tools`, podem veure el temps que tarden a processar-se les sol·licituds:

1. La primera sol·licitud, en tindre un valor de n gran, ens porta uns quants segons completar-la.
2. La segona sol·licitud, malgrat tindre un valor de n que ja havíem comprovat que oferia una resposta quasi immediata, també es demora uns segons.

Per què ocorre això? Perquè l'únic subprocés estarà ocupat processant l'altra operació d'execució prolongada. L'únic nucli de la CPU ha de completar la primera sol·licitud abans que puga encarregar-se de l'altra.

1.3.2 Ara amb clúster!

Ara usarem el mòdul de clúster en l'aplicació per a generar alguns processos secundaris i veure com això millora les coses.

A continuació es mostra l'aplicació modificada:

```
1  const express = require("express");
2  const port = 3000;
3  const cluster = require("cluster");
4  const totalCPUs = require("os").cpus().length;
5
6  if (cluster.isMaster) {
7      console.log(`Number of CPUs is ${totalCPUs}`);
8      console.log(`Master ${process.pid} is running`);
9
10     // Fork workers.
11     for (let i = 0; i < totalCPUs; i++) {
12         cluster.fork();
13     }
14
15     cluster.on("exit", (worker, code, signal) => {
16         console.log(`worker ${worker.process.pid} died`);
17         console.log("Let's fork another worker!");
18         cluster.fork();
19     });
20 } else {
```

```
    const app = express();
    console.log(`Worker ${process.pid} started`);

    app.get("/", (req, res) => {
        res.send("Hello World!");
    });

    app.get("/api/:n", function (req, res) {
        let n = parseInt(req.params.n);
        let count = 0;

        if (n > 5000000000) n = 5000000000;

        for (let i = 0; i <= n; i++) {
            count += i;
        }

        res.send(`Final count is ${count}`);
    });

    app.listen(port, () => {
        console.log(`App listening on port ${port}`);
    });
}
```

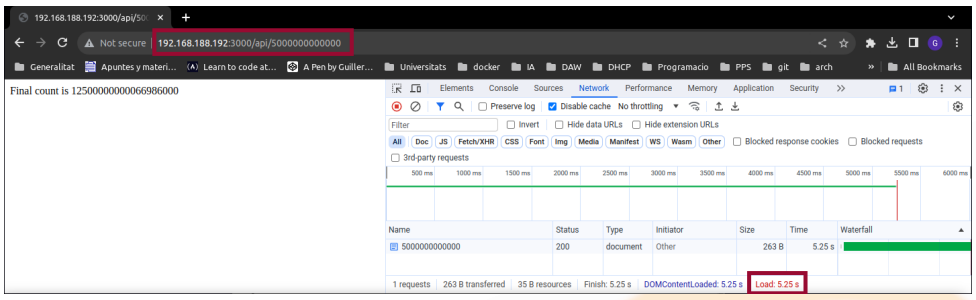
Esta aplicació fa el mateix que abans però esta vegada estem generant diversos processos secundaris que compartiran el port 3000 i que podran manejar les sol·licituds enviades a este port. Els processos de treball es generen utilitzant el mètode `child_process.fork()`. El mètode retorna un objecte `ChildProcess` que té un canal de comunicació incorporat que permet que els missatges es transmeten entre el fill i el seu pare.

Creem tants processos secundaris com nuclis de CPU hi ha en la màquina en la qual s'executa l'aplicació. Es recomana no crear més *workers que nuclis lògics en la computadora, ja que això pot causar una sobrecàrrega en termes de costos de programació. Això succeeix perquè el sistema haurà de programar tots els processos creats perquè es vagen executant per torns en els nuclis.

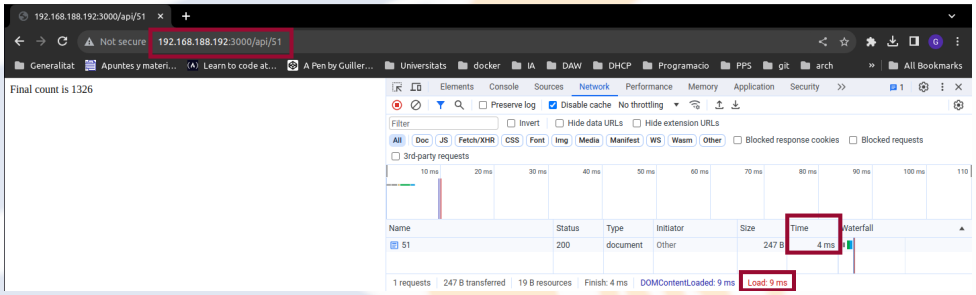
Els *workers* són creats i administrats pel procés mestre. Quan l'aplicació s'executa per primera vegada, verifiquem si és un procés mestre amb `isPrimary`. Això està determinat per la variable `process.env.NODE_UNIQUE_ID`. Si `process.env.NODE_UNIQUE_ANEU` té valor `undefined`, llavors `isPrimary` serà **`true`**.

Si el procés és un mestre, cridem a `cluster.fork()` per a generar diversos processos. Registrem els ID de procés mestre i *worker*. Quan un procés secundari mor, generem un nou per a continuar utilitzant els nuclis de CPU disponibles.

Ara repetirem el mateix experiment d'abans, primer realitzem una sol·licitud al servidor amb un valor alt `n`:



I executem ràpidament una altra sol·licitud en una altra pestanya del navegador, mesurant els temps de processament d'ambdues:



Comprovarem que estos es redueixen dràsticament.

Amb diversos *workers* disponibles per a acceptar sol·licituds, es milloren tant la disponibilitat del servidor com el rendiment.

Executar una sol·licitud en una pestanya del navegador i executar ràpidament una altra en una segona pestanya serveix per a mostrar-nos la millora que ofereix l'agrupació en clústers per al nostre exemple d'una forma més o menys ràpida, però és un mètode una miqueta "potiner" i no és una forma adequada o de confiança de determinar les millores de rendiment.

En el següent apartat donarem una ullada a alguns punts de referència que demostraran millor quant ha millorat l'agrupació en clústers la nostra aplicació.

1.3.3 Mètriques de rendiment

Realitzarem una prova de càrrega en les nostres dues aplicacions per a veure com cadascuna maneja una gran quantitat de connexions entrants. Usarem el paquet `loadtest` per a això.

El paquet `loadtest` ens permet simular una gran quantitat de connexions simultànies a la nostra **API** perquè puguem mesurar el seu rendiment.

Per a usar `loadtest`, primer hem d'instal·lar-ho globalment. Després de connectar-vos per **SSH** al servidor Debian:

```
1 npm install -g loadtest
```

Després executem l'aplicació que volem provar (`node nom_aplicacio.js`). Començarem provant la versió que no utilitza l'agrupació en clústers.

Mentre executem l'aplicació, en un altre terminal realitzem la següent prova de càrrega:

```
1 loadtest http://localhost:3000/api/500000 -n 1000 -c 100
```

El comando anterior enviarà 1000 sol·licituds a la URL donada, de les quals 100 són concurrents. El següent és el resultat d'executar el comando anterior:

```
guillermo@tomcat:~$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/500000
Max requests:        1000
Concurrent clients:  100
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          0.865 s
Mean latency:        81.5 ms
Effective rps:       1156

Percentage of requests served within a certain time
  50%      78 ms
  90%      94 ms
  95%      98 ms
  99%      121 ms
 100%      127 ms (longest request)
```

Veiem que amb la mateixa sol·licitud (amb n= 500000) el servidor ha pogut manejar 404 sol·licituds per segon amb una latència mitjana de 232.4 mil·lisegons (el temps mitjà que tarda a completar una sola sol·licitud).

Intentem-ho de nou, però esta vegada amb més sol·licituds (i sense clústers):

```
guillermo@tomcat:~$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100
Requests: 957 (96%), requests per second: 191, mean latency: 494.3 ms

Target URL:          http://localhost:3000/api/5000000
Max requests:        1000
Concurrent clients:  100
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          5.221 s
Mean latency:        495.2 ms
Effective rps:       192

Percentage of requests served within a certain time
  50%      514 ms
  90%      531 ms
  95%      532 ms
  99%      534 ms
 100%      538 ms (longest request)
```

Veiem que les mètriques llancen resultats encara pitjors.

Ara detenim l'aplicació sense clústers i executem `node nom_aplicacio_cluster.js` que si que els té. Executarem exactament les mateixes proves amb l'objectiu de realitzar una comparació:

```
guillermo@tomcat:~$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/500000
Max requests:        1000
Concurrent clients:  100
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          0.792 s
Mean latency:        74.7 ms
Effective rps:       1263

Percentage of requests served within a certain time
  50%      74 ms
  90%      85 ms
  95%      101 ms
  99%      135 ms
 100%      158 ms (longest request)
```

```
guillermo@tomcat:~$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100

Target URL:          http://localhost:3000/api/5000000
Max requests:        1000
Concurrent clients:  100
Agent:               none

Completed requests:  1000
Total errors:        0
Total time:          2.964 s
Mean latency:        281.1 ms
Effective rps:       337

Percentage of requests served within a certain time
  50%      291 ms
  90%      304 ms
  95%      305 ms
  99%      307 ms
 100%      312 ms (longest request)
```

És obvi que els clústers permeten manejar una major quantitat de peticions per segon amb una menor latència.

1.3.4 Ús de PM2 per a administrar un clúster de `Node.js`

En la nostra aplicació, hem usat el mòdul clúster de `Node.js` per a crear i administrar manualment els processos.

Primer hem determinat la quantitat de *workers* (usant la quantitat de nuclis de CPU com a referència), després els hem generats i, finalment, escoltem si hi ha *workers* morts per a poder generar nous.

En la nostra aplicació d'exemple molt senzilla, vam haver d'escriure una quantitat considerable de codi només per a administració l'agrupació en clústers. En una aplicació de producció és bastant probable que s'haja d'escriure encara més codi.

Existeix una eina que ens pot ajudar a administrar tot això un poc millor: l'administrador de processos PM2. PM2 és un administrador de processos de producció per a aplicacions Node.js amb un balancejador de càrrega incorporat.

Quan està configurat correctament, PM2 executa automàticament l'aplicació en mode de clúster, generant *workers* i s'encarrega de generar nous *workers* quan un d'ells mor.

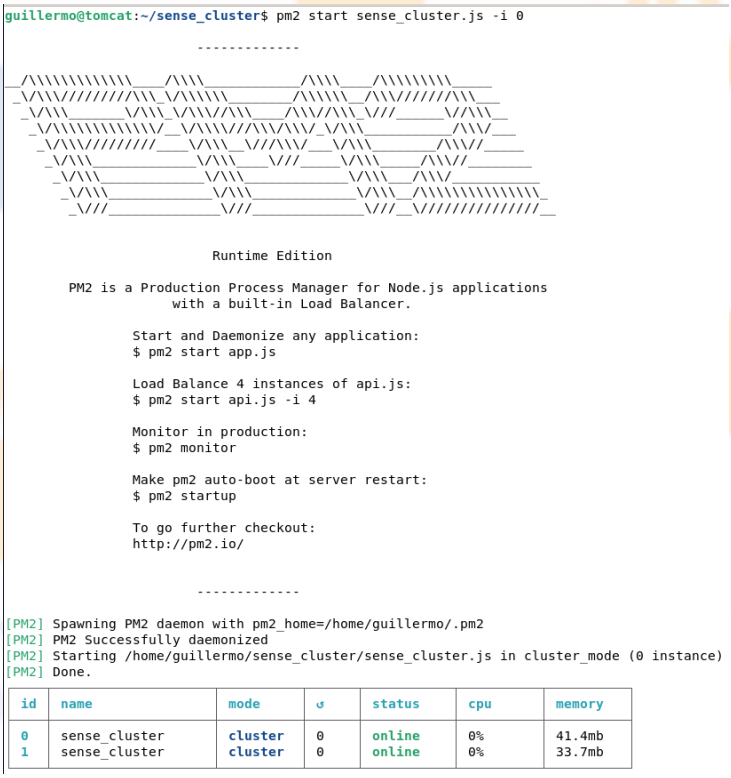
PM2 facilita la parada, eliminació i inici de processos, a més de disposar d'algunes eines de monitoratge que poden ajudar-nos a monitorar i ajustar el rendiment de la seua aplicació.

Per a usar PM2, primer instal·lem globalment en la nostra **Debian**:

```
npm install pm2 -g
```

Ho utilitzarem amb la nostra primera aplicació, la que no estava "clusteritzada" en el codi. Per a això executarem el següent comando:

```
pm2 start nom_aplicacio_sense_cluster.js -i 0
```



On:

- `i` li indicarà a PM2 que inicie l'aplicació en `cluster_mode` (a diferència de `fork_mode`). Si s'estableix a 0, PM2 generarà automàticament tants *workers* com a nuclis de CPU hi haja.

I així, la nostra aplicació s'executa en mode de clúster, sense necessitat de canvis de codi.

Tasca 3
Executa i documenta amb captures de pantalles, les mateixes proves que abans però utilitzant PM2 i comprova si s'obtenen els mateixos resultats.

Per darrere, PM2 també utilitza el mòdul clúster de Node.js, així com altres eines que faciliten la gestió de processos.

En el Terminal, obtindrem una taula que mostra alguns detalls dels processos generats:

Podem detindre l'aplicació amb el següent comando:

```
pm2 stop app.js
```

L'aplicació es desconnectarà i mostrarà per terminal tots els processos amb un estat stopped.

```
guillermo@tomcat:~/sense_cluster$ pm2 stop sense_cluster.js
[PM2] Applying action stopProcessId on app [sense_cluster.js](ids: [ 0, 1 ])
[PM2] [sense_cluster](0) ✓
[PM2] [sense_cluster](1) ✓
```

id	name	mode	↺	status	cpu	memory
0	sense_cluster	cluster	0	stopped	0%	0b
1	sense_cluster	cluster	0	stopped	0%	0b

En comptes de tindre passar sempre les configuracions quan executa l'aplicació amb `pm2 start app.js -i 0`, podríem facilitar-nos la tasca i guardar-les en un arxiu de configuració separat, anomenat Ecosystem.

Este arxiu també ens permet establir configuracions específiques per a diferents aplicacions.

Crearem l'arxiu `ecosystem` amb el següent comando:

```
guillermo@tomcat:~/sense_cluster$ pm2 ecosystem
File /home/guillermo/sense_cluster/ecosystem.config.js generated
```

Que generarà un arxiu anomenat `ecosystem.config.js`. Per al cas concret de la nostra aplicació, necessitem modificar-lo com es mostra a continuació:

```
1  module.exports = {
2      apps: [
3      {
4          name: "nom_aplicacio",
5          script: "nom_aplicacio_sense_cluster.js",
6          instances: 0,
7          exec_mode: "cluster",
8      },
9      ],
10 };
```

En configurar `exec_mode` amb el valor `cluster`, li indica a PM2 que balancege la càrrega entre cada instància. `instances` com abans configurat a 0, el que generarà tants *workers* com a nuclis de CPU.

L'opció `-i` o `instances` es pot establir amb els següents valors:

- `0` o `max` (en desús) per a "repartir" l'aplicació entre totes les CPU.
- `-1` per a "repartir" l'aplicació en totes les CPU - 1.
- `número` per a difondre l'aplicació a través d'un número concret de CPU.

Ara podem executar l'aplicació amb:

```
pm2 start ecosystem.config.js
```

L'aplicació s'executarà en mode clúster, exactament com abans.

Podrem iniciar, reiniciar, recarregar, detindre i eliminar una aplicació amb els següents comandos, respectivament:

```
$ pm2 start nom_aplicacio
$ pm2 restart nom_aplicacio
$ pm2 reload nom_aplicacio
$ pm2 stop nom_aplicacio
$ pm2 delete nom_aplicacio

# Quan usem l'arxiu Ecosystem:

$ pm2 [start|restart|reload|stop|delete] ecosystem.config.js
```

El comando `restart` elimina i reinicia immediatament els processos, mentre que el comando `reload` aconsegueix un temps d'inactivitat de 0 segons on els *workers* es reinicien d'un en un, esperant que aparega un nou *worker* abans de matar a l'anterior.

També pot verificar l'estat, els registres i les mètriques de les aplicacions en execució.

Tasca 4

Investiga els següents comandos i explica que eixida per terminal ens ofereixen i per a què s'utilitzen:

```
pm2 ls
pm2 logs
pm2 monit
```

Documenta la realització de tota esta pràctica adequadament, amb les explicacions i justificacions necessàries, les respostes a les preguntes plantejades i les captures de pantalla pertinents.