Práctica 3.1: Instalación de Tomcat

Introducción

Si consultamos el apartado de versiones de Tomcat en su página oficial, nos daremos cuenta de que no vamos a usar la última versión, la 10, para esta práctica, sino la anterior, la 9. La pregunta es casi inevitable:

¿Por qué?

En el enlace anterior vemos como desde su versión 9, Tomcat da soporta a Java 8 y superiores. Sin embargo, a partir de la versión 10.1.x, da soporte a Java 11 y superiores.

¿Qué significa esto?

En Java 9 se introdujeron novedades como un nuevo sistema de módulos (Jigsaw), entre otras.

En Java 11 se dio un paso más al haber renombrado completamente las rutas de paquetes javax. *a jakarta.*. Oracle, a pesar de haber hecho público el desarrollo de Java, no hizo lo mismo con su nombre.

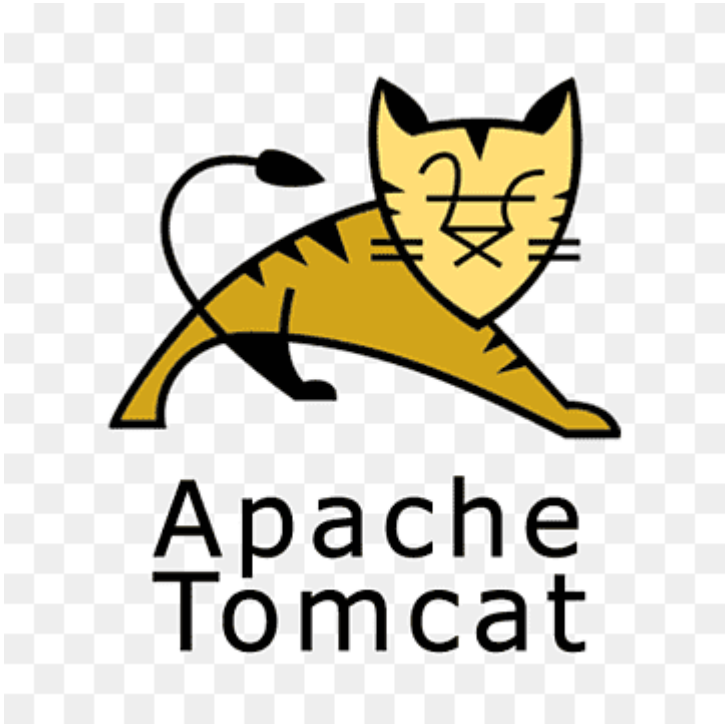
Así las cosas, resulta que Java 8 puede que a día de hoy aún sea la más usada en proyectos reales. Dicho esto, podría realizarse un proceso de migración de un proyecto de Java 8 a Java 11 y utilizarlo en Tomcat 10.

No obstante, para Java 8 su soporte para uso comercial (pagando) acabó en Marzo de 2022, pero para uso no comercial sigue hasta 2030.

En conclusión, no es raro encontrarse en el mundo real un proyecto a desplegar realizado en Java 8. Podría realizarse una migración y los conceptos de despliegue que veremos seguirían aplicando. Así las cosas, por facilidad en la realización de las prácticas utilizaremos Tomcat 9 y el plugin oficial de Maven para Tomcat 7 para el despliegue (luego veremos el motivo).

Instalación de Tomcat

Esta práctica es muy sencilla y va a consistir en realizar la instalación del servidor de aplicaciones Tomcat 9, en una máquina virtual corriendo Debian 11 Bullseye.



Se puede hacer tanto con el administrador de paquetes `apt` como de forma manual. La forma más recomendable por su sencillez es la primera.

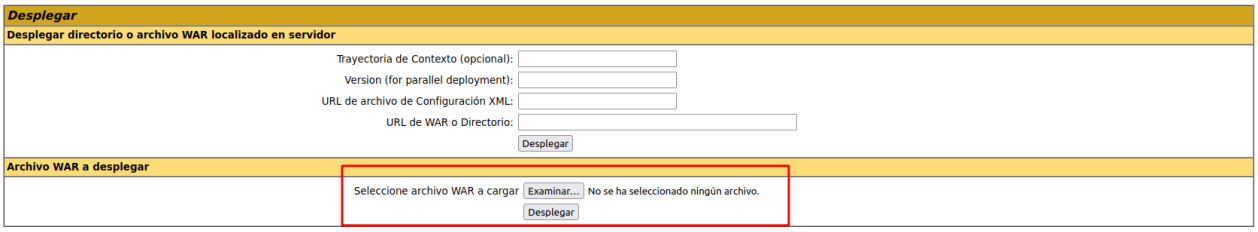
Para ello, y como sugerencia, podéis apoyaros en este tutorial online, aunque sóis libres de consultar tantas fuentes como deseéis.

Obviamente, debéis utilizar vuestro propios usuarios y contraseña.

Despliegue manual mediante la GUI de administración

Realizaremos el despliegue manual de una aplicación ya previamente empaquetada en formato WAR. Para ello:

1. Nos logueamos con el usuario previamente creado.
2. Buscamos la sección que nos permite desplegar un WAR manualmente, seleccionamos nuestro archivo y lo desplegamos.



Tras estos pasos, se nos listará la aplicación ya desplegada como un directorio más y podremos acceder a ella.

Task

Documenta el despliegue manual de la aplicación que os podéis descargar para tal efecto en Aules (archivo .war).

Despliegue con Maven

Instalación de Maven

Para instalar Maven en nuestro Debian tenemos, de nuevo, dos opciones:

- Instalación mediante gestor de paquetes APT
- Instalación manual

La primera, recomendada, es mucho más sencilla y automatizada (establece todos los paths y variables de entorno), aunque con la segunda se podría conseguir un paquete más actualizado.

Ambos métodos vienen explicados aquí

Si decidimos seguir el primer método, el más sencillo, vemos que es tan simple como actualizar los repositorios:

```
sudo apt update
```

E instalar Maven

```
sudo apt install maven
```

Para comprobar que todo ha ido correctamente, podemos ver la versión instalada de Maven:

```
mvn --v
```

Configuración de Maven

Para poder realizar despliegues en nuestro Tomcat previamente instalado, necesitamos realizar la configuración adecuada para Maven. Ya sabemos que esto en Linux significa editar los archivos de configuración adecuados. Vamos a ello.

1. En primer lugar necesitamos asegurarnos de que en el apartado anterior de la práctica hemos añadido todos los usuarios necesarios, así como sus respectivos roles. Debemos añadir el rol de `manager-script` para permitir que Maven se autentique contra Tomcat y pueda realizar el despliegue.

Los roles utilizados por Tomcat vienen detallados en su documentación, que merece ser consultada:

You can find the role names in the `web.xml` file of the Manager web application. The available roles are:

- **manager-gui** — Access to the HTML interface.
- **manager-status** — Access to the "Server Status" page only.
- **manager-script** — Access to the tools-friendly plain text interface that is described in this document, and to the "Server Status" page.
- **manager-jmx** — Access to JMX proxy interface and to the "Server Status" page.

En dicha documentación se nos indica que, por temas de seguridad, es recomendable no otorgar los roles de **manager-script** o **manager-jmx** al mismo usuario que tenga el rol de **manager-gui**.

Info

Tendremos dos usuarios, uno para la GUI y otro exclusivamente para hacer los deploys de Maven.

Así las cosas, modificamos el archivo `/etc/tomcat9/tomcat-users.xml` acorde a nuestras necesidades (los nombres de usuario y contraseña deberán ser los que elijáis para vosotros):

```
  <role rolename="admin"/>
  <role rolename="admin-gui"/>
  <role rolename="manager"/>
  <role rolename="manager-gui"/>
  <role rolename="manager-script"/>
  <user username="raul-profesor" password="[redacted]" roles="admin,admin-gui,manager,manager-gui"/>
  <user username="raul-deploy" password="[redacted]" roles="manager-script"/>

</tomcat-users>
```

2. Editar el archivo `/etc/maven/settings.xml` para indicarle a Maven, un identificador para el servidor sobre el que vamos a desplegar (no es más que un nombre, ponedle el nombre que consideréis), así como las credenciales. Todo esto se hará dentro del bloque *servers* del XML:

```
  <servers>
    <!-- server
     | Specifies the authentication information to use when connecting to a particular server, identified by
     | a unique name within the system (referred to by the 'id' attribute below).
     |
     | NOTE: You should either specify username/password OR privateKey/passphrase, since these pairings are
     |       used together.
     |
    <server>
      <id>deploymentRepo</id>
      <username>repouser</username>
      <password>repopwd</password>
    </server>
    -->

    <!-- Another sample, using keys to authenticate.
    <server>
      <id>siteServer</id>
      <privateKey>/path/to/private/key</privateKey>
      <passphrase>optional; leave empty if not used.</passphrase>
    </server>
    -->
    <server>
    <id>Tomcat.P.3.1</id>
    <username>raul-deploy</username>
    <password[redacted]</password>
    </server>
  </servers>
```

3. Ahora debemos modificar el `POM` del proyecto para que haga referencia a que el despliegue se realice con el plugin de Maven para Tomcat.

Info

No existen plugins **oficiales** para Tomcat más allá de la versión 7 del servidor. No obstante, el plugin para Tomcat 7 sigue funcionando correctamente con Tomcat 9.

Otra opción sería utilizar el plugin Cargo

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>raul</groupId>
  <artifactId>war-deploy</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>war-deploy Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
        <finalName>war-deploy</finalName>
        <plugins>
        <plugin>
         <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
            <url>http://localhost:8080/manager/text</url>
            <server>Tomcat.P.3.1</server>
            <path>/myapp</path>
        </configuration>
        </plugin>
        </plugins>
</build>
</project>
~
```

Donde lo que añadimos es el bloque

```
<build>
        <finalName>war-deploy</finalName> #


        <plugins>
        <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
            <url>http://localhost:8080/manager/text</url> #


            <server>Tomcat.P.3.1</server> #


            <path>/myapp</path> #

    3.
        </configuration>
        </plugin>
        </plugins>
</build>
```

Despliegue

Teniendo ya todo listo para realizar despliegues, ahora crearemos una aplicación Java de prueba para ver si podemos desplegarla sobre la arquitectura que hemos montado. Para ello utilizamos el comando:

```
    mvn archetype:generate -DgroupId=raul -DartifactId=war-deploy -
DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false
```

Podéis sustituir los valores de `groupID` y `artifactId` (este será el nombre de la aplicación) por lo que queráis.

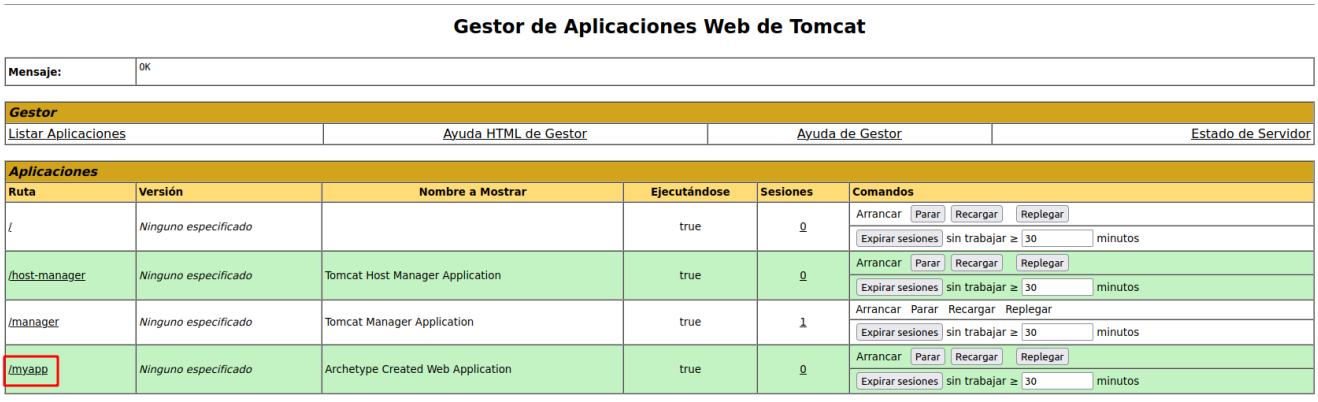
Tras generar esta aplicación, los comandos finales que se utilizan en Maven para desplegar, volver a desplegar o desplegar una aplicación, son:

- `mvn tomcat7:deploy`
- `mvn tomcat7:redeploy`
- `mvn tomcat7:undeploy`

Así pues, tras el despliegue con Maven nos indicará que todo ha ido correctamente con un mensaje de `BUILD SUCCESS`, tal que así:

```
[INFO] --- tomcat7-maven-plugin:2.2:redeploy (default-cli) @ war-deploy ---
[INFO] Deploying war to http://localhost:8080/myapp
Uploading: http://localhost:8080/manager/text/deploy?path=%2Fmyapp&update=true
Uploaded: http://localhost:8080/manager/text/deploy?path=%2Fmyapp&update=true (3 KB at 356.0 KB/sec)

[INFO] tomcatManager status code:200, ReasonPhrase:
[INFO] OK - Desplegada aplicación en trayectoria de contexto [/myapp]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  9.969 s
[INFO] Finished at: 2022-07-20T23:02:08+02:00
```

Y, accediendo a través de la GUI, debemos ver que la aplicación está desplegado y que podemos acceder a ella perfectamente:



Gestor de Aplicaciones Web de Tomcat

Mensaje:	OK

Gestor			
Listar Aplicaciones	Ayuda HTML de Gestor	Ayuda de Gestor	Estado de Servidor

Aplicaciones					
Ruta	Versión	Nombre a Mostrar	Ejecutándose	Sesiones	Comandos
/	Ninguno especificado		true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/host-manager	Ninguno especificado	Tomcat Host Manager Application	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/manager	Ninguno especificado	Tomcat Manager Application	true	1	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/myapp	Ninguno especificado	Archetype Created Web Application	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos

Tarea

Realizar el despliegue con la aplicación de prueba.

Repetir el despliegue pero esta vez con otra aplicación que no es la de prueba. Más adelante ya hablaremos de `git` pero de momento, usaremos los comandos que veremos a continuación.

Nos clonamos el repositorio:

```
git clone https://github.com/cameronmcnz/rock-paper-scissors.git
```

Nos situamos dentro de él:

```
cd rock-paper-scissors
```

Y cambiamos de rama:

```
git checkout patch-1
```

Tras esto debemos proceder exactamente igual que en el caso anterior, con la ventaja de que ya tenemos configurados los usuarios de Tomcat y los parámetros de Maven.

Así pues, sólo habría que añadir el bloque `<plugin>...</plugin>` adecuado para poder hacer nuestro despliegue.

Task

Documenta, incluyendo capturas de pantallas, el proceso que has seguido para realizar el despliegue de esta nueva aplicación, así como el resultado final.

Cuestiones

Habéis visto que los archivos de configuración que hemos tocado contienen contraseñas en texto plano, por lo que cualquiera con acceso a ellos obtendría las credenciales de nuestras herramientas.

En principio esto representa un gran riesgo de seguridad, ¿sabrías razonar o averigüar por qué esto está diseñado de esta forma?

Práctica 3.2: Despliegue de aplicaciones con Node Express

Introducción

En esta práctica vamos a realizar el despliegue de aplicaciones Node.js sobre un servidor Node Express. Lo curioso de este caso es que el despliegue aquí cambia un poco puesto que no se hace sobre el servidor, sino que la aplicación *es* el servidor.

Warning

Comprueba que el servidor Tomcat de prácticas anteriores no está corriendo o nos dará problemas:

```
sudo systemctl status tomcat9
```

Y en caso de salir activo, pararlo:

```
sudo systemctl stop tomcat9
```

Instalación de Node.js, Express y test de la primera aplicación

La primera parte de la práctica es muy sencilla. Consistirá en instalar sobre nuestra Debian 11 tanto Node.js como Express y tras ello crear un archivo `.js` de prueba para comprobar que nuestro primer despliegue funciona correctamente.

Para ello, os podéis apoyar en este sencillo tutorial.

En lugar de acceder a `http://localhost:3000`, debéis acceder desde vuestra máquina local a `http://IP-maq-virtual:3000`, utilizando la IP concreta de vuestra máquina virtual.

Recordad parar el servidor (CTRL+C) al acabar la práctica.

Task

Documenta, incluyendo capturas de pantallas, el proceso que has seguido para realizar el despliegue de esta nueva aplicación, así como el resultado final.

Despliegue de una nueva aplicación

Vamos ahora a realizar el despliegue de una aplicación de terceros para ver cómo es el proceso.

Se trata de un "prototipo" de una especie de CMS que podéis encontrar en este repositorio de Github.

Tal y como indican las instrucciones del propio repositorio, los pasos a seguir son, en primer lugar, clonar el repositorio a nuestra máquina:

```
git clone https://github.com/contentful/the-example-app.nodejs.git
```

Movernos al nuevo directorio:

```
cd the-example-app.nodejs
```

Instalar las librerías necesarias (paciencia, este proceso puede tardar un buen rato):

```
npm install
```

Y, por último, iniciar la aplicación:

```
npm run start:dev
```

Tarea

Documenta, incluyendo capturas de pantallas, el proceso que has seguido para realizar el despliegue de esta nueva aplicación, así como el resultado final.

Cuestiones

Cuando ejecutáis el comando `npm run start:dev`, lo que estáis haciendo es ejecutar un script:

- ¿Dónde podemos ver que script se está ejecutando?
- ¿Qué comando está ejecutando?

Como ayuda, podéis consultar esta información.

Referencias

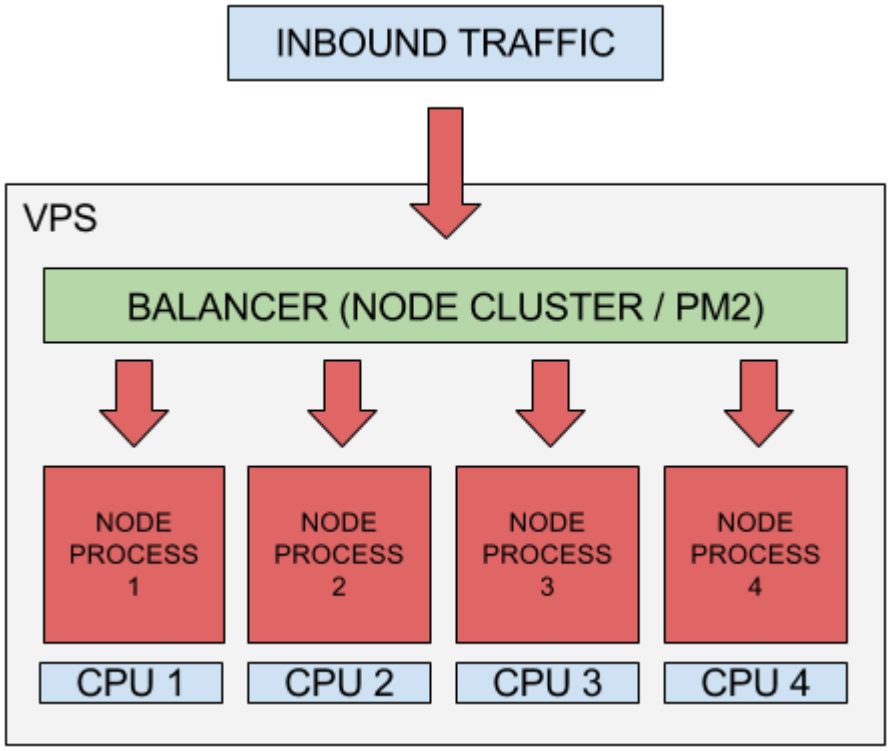
How to install ExpressJS on Debian 11?

Práctica 3.3: Despliegue de una aplicación "clusterizada" con Node Express

Introducción

Cuando se construye una aplicación de producción, normalmente se busca la forma de optimizar su rendimiento llegando a una solución de compromiso. En esta práctica echaremos un vistazo a un enfoque que puede ofrecer una victoria rápida cuando se trata de mejorar la manera en que las aplicaciones Node.js manejan la carga de trabajo.

Una instancia de Node.js se ejecuta en un solo hilo, lo que significa que en un sistema multinúcleo (como la mayoría de los ordenadores de hoy en día), no todos los núcleos serán utilizados por la aplicación. Para aprovechar los otros núcleos disponibles, podemos lanzar un clúster de procesos Node.js y distribuir la carga entre ellos.



Tener varios hilos para manejar las peticiones mejora el rendimiento (peticiones/segundo) del servidor, ya que varios clientes pueden ser atendidos simultáneamente. Veremos cómo crear procesos hijos con el módulo de cluster de Node.js para, más tarde, ver cómo gestionar el cluster con el gestor de procesos PM2.

Un vistazo rápido a los clusters

El módulo de clúster de Node.js permite la creación de procesos secundarios (*workers*) que se ejecutan simultáneamente y comparten el mismo puerto de servidor. Cada hijo generado tiene su propio ciclo de eventos y memoria. Los procesos secundarios utilizan IPC (comunicación entre procesos) para comunicarse con el proceso principal de Node.js.

Tener múltiples procesos para manejar las solicitudes entrantes significa que se pueden procesar varias solicitudes simultáneamente y si hay una operación de bloqueo/ejecución prolongada en un

worker, los otros *workers* pueden continuar administrando otras solicitudes entrantes; la aplicación no se detendrá hasta que finalice la operación de bloqueo.

La ejecución de varios *workers* también permite actualizar la aplicación en producción con poco o ningún tiempo de inactividad. Se pueden realizar cambios en la aplicación y reiniciar los *workers* uno por uno, esperando que un proceso secundario se genere por completo antes de reiniciar otro. De esta manera, siempre habrá *workers* ejecutándose mientras se produce la actualización.

Las conexiones entrantes se distribuyen entre los procesos secundarios de dos maneras:

- El proceso maestro escucha las conexiones en un puerto y las distribuye entre los *workers* de forma rotatoria. Este es el enfoque por defecto en todas las plataformas, excepto Windows.
- El proceso maestro crea un socket de escucha y lo envía a los *workers* interesados que luego podrán aceptar conexiones entrantes directamente.

Usando los clusters

Primero sin clúster

Para ver las ventajas que ofrece la agrupación en clústeres, comenzaremos con una aplicación de prueba en Node.js que no usa clústeres y la compararemos con una que sí los usa, se trata de la siguiente:

```
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}`);
});
```

Se trata de una aplicación un tanto *prefabricada* en el sentido de que es algo que jamás encontraríamos en el mundo real. No obstante, nos servirá para ilustrar nuestro propósito.

Esta aplicación contiene dos rutas, una ruta raíz `/` que devuelve la cadena `Hello World!` y otra ruta `/api/n` donde se toma `n` como parámetro y va realizando una operación de suma (el bucle for) cuyo resultado acumula en la variable `count` que se muestra al final.

Si a este parámetro `n`, le damos un valor muy alto, nos permitirá simular operaciones intensivas y de ejecución prolongada en el servidor. Le damos como valor límite `5000000000` para evitar una operación demasiado costosa para nuestro ordenador.

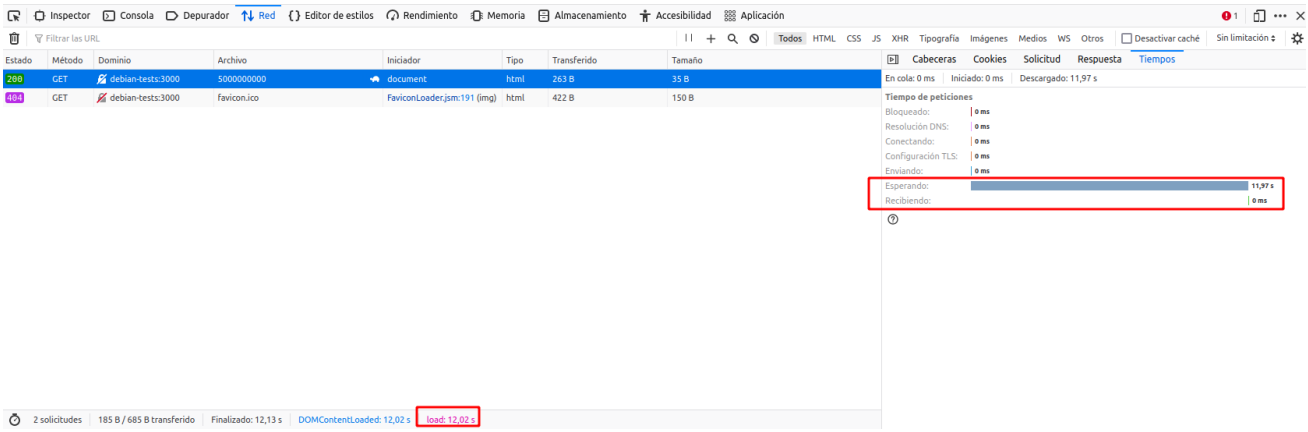
Task

1. Debéis conectaros al servidor Debian mediante SSH
2. Debéis crear un directorio para el proyecto de esta aplicación
3. DENTRO del directorio ejecutaréis 2 comandos:
    1. `npm init` para crear automáticamente la estructura de carpetas y el archivo `package.json` (Con ir dándole a <ENTER> a todas las preguntas, os basta)
    2. `npm install express` para instalar express para este proyecto
4. Tras esto, DENTRO del directorio, ya podéis iniciar la aplicación con: `node nombre_aplicacion.js`

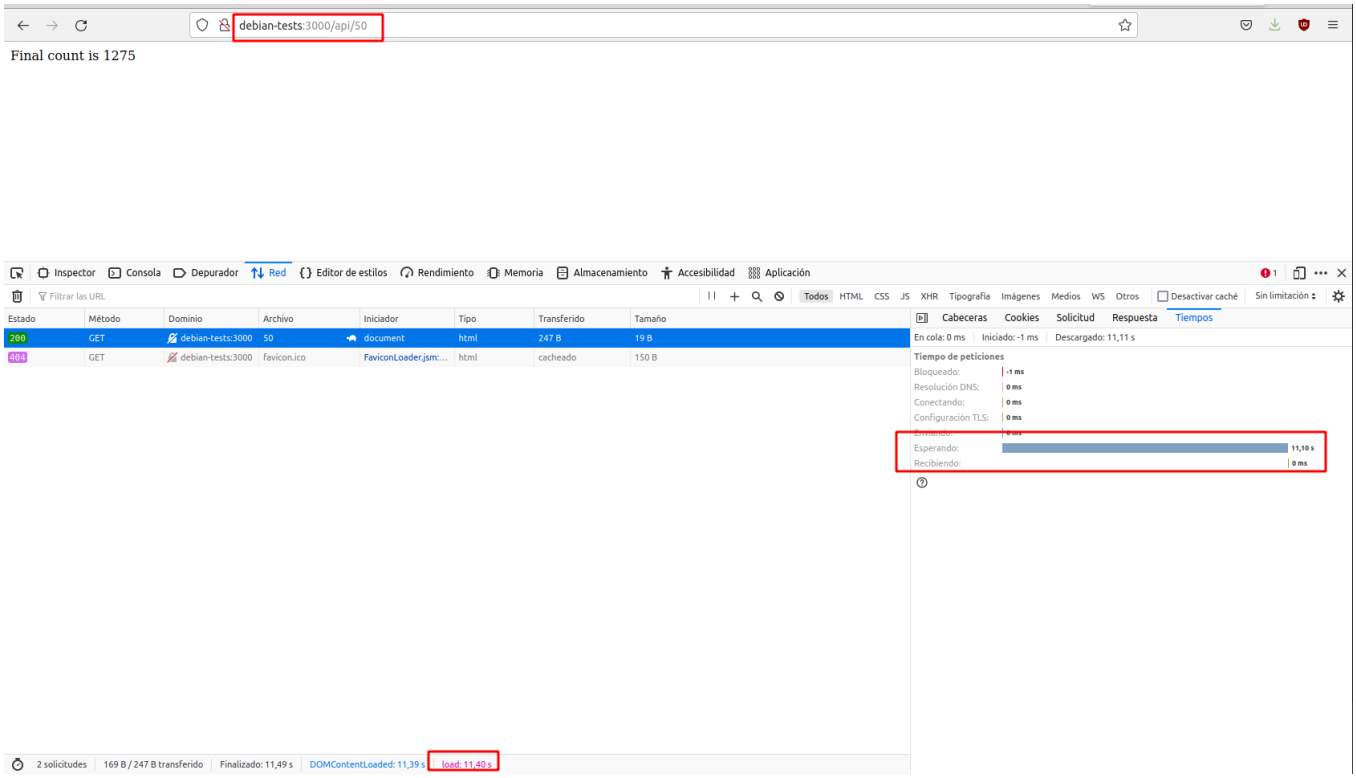
Para comprobarlo, podéis acceder a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/50` donde `IP-maq-virtual` es la IP del adaptador puente de vuestra Debian.

Utilizada un valor de `n` relativamente pequeño, como el 50 del ejemplo anterior y comprobaréis que se ejecutará rápidamente, devolviendo una respuesta casi inmediata.

Hagamos otra simple comprobación para valores de `n` más grandes. Desplegada e iniciada la aplicación, acceded a la ruta `http://IP-maq-virtual:3000/api/5000000000`.



Mientras esta solicitud que tarda unos segundos se está procesando, acceded en otra pestaña del navegador a `http://IP-maq-virtual:3000` o a `http://IP-maq-virtual:3000/api/n` siendo `n` el valor que le queráis dar.



Utilizando las devoloper tools, podemos ver el tiempo que tardan en procesarse las solicitudes:

1. La primera solicitud, al tener un valor de n grande, nos lleva unos cuantos segundos completarla.
2. La segunda solicitud, pese a tener un valor de n que ya habíamos comprobado que ofrecía una respuesta casi inmediata, también se demora unos segundos.

¿Por qué ocurre esto? Porque el único subproceso estará ocupado procesando la otra operación de ejecución prolongada. El único núcleo de la CPU tiene que completar la primera solicitud antes de que pueda encargarse de la otra.

¡Ahora con más clúster!

Ahora usaremos el módulo de clúster en la aplicación para generar algunos procesos secundarios y ver cómo eso mejora las cosas.

A continuación se muestra la aplicación modificada:

```
const express = require("express");
const port = 3000;
const cluster = require("cluster");
const totalCPUs = require("os").cpus().length;

if (cluster.isMaster) {
 console.log(`Number of CPUs is ${totalCPUs}`);
 console.log(`Master ${process.pid} is running`);

 // Fork workers.
 for (let i = 0; i < totalCPUs; i++) {
 cluster.fork();
 }

 cluster.on("exit", (worker, code, signal) => {
 console.log(`worker ${worker.process.pid} died`);
 console.log("Let's fork another worker!");
```

```
 cluster.fork();
 });
} 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}`);
 });
}
```

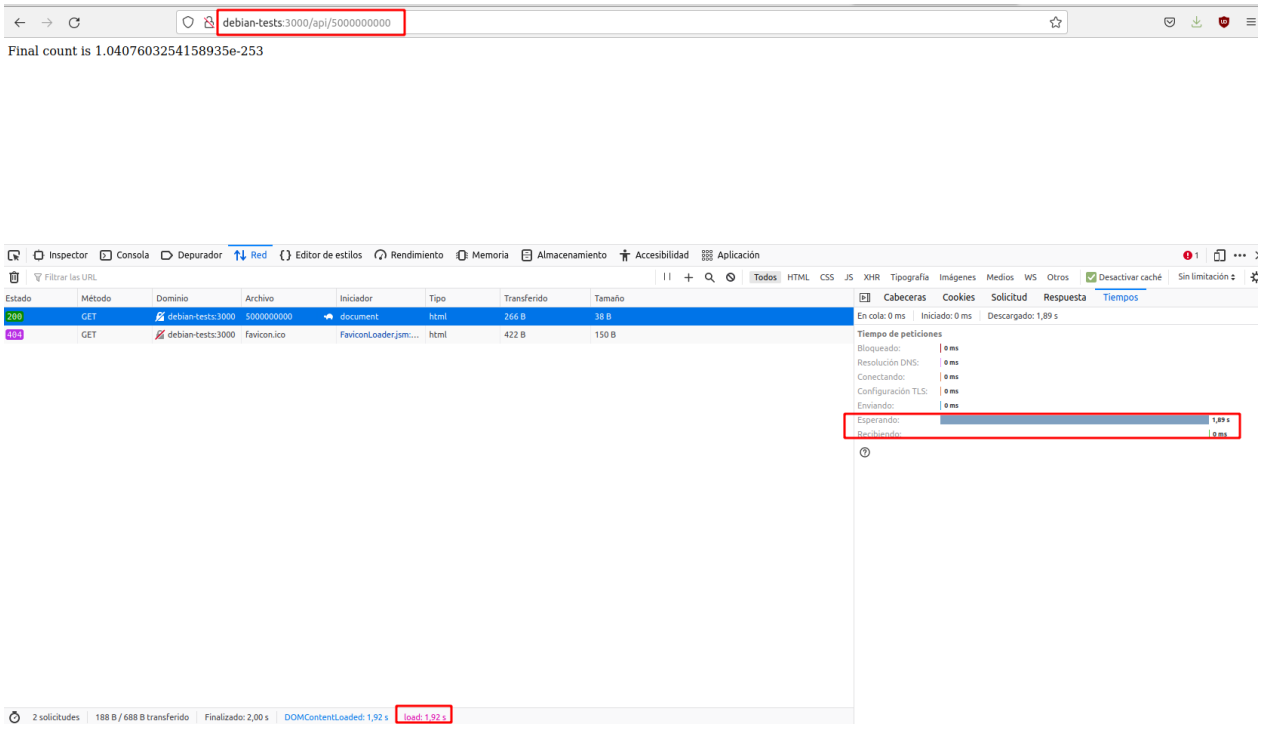
Esta aplicación hace lo mismo que antes, pero esta vez estamos generando varios procesos secundarios que compartirán el puerto 3000 y que podrán manejar las solicitudes enviadas a este puerto. Los procesos de trabajo se generan utilizando el método `child_process.fork()`. El método devuelve un objeto `ChildProcess` que tiene un canal de comunicación incorporado que permite que los mensajes se transmitan entre el hijo y su padre.

Creamos tantos procesos secundarios como núcleos de CPU hay en la máquina en la que se ejecuta la aplicación. Se recomienda no crear más *workers* que núcleos lógicos en la computadora, ya que esto puede causar una sobrecarga en términos de costos de programación. Esto sucede porque el sistema tendrá que programar todos los procesos creados para que se vayan ejecutando por turnos en los núcleos.

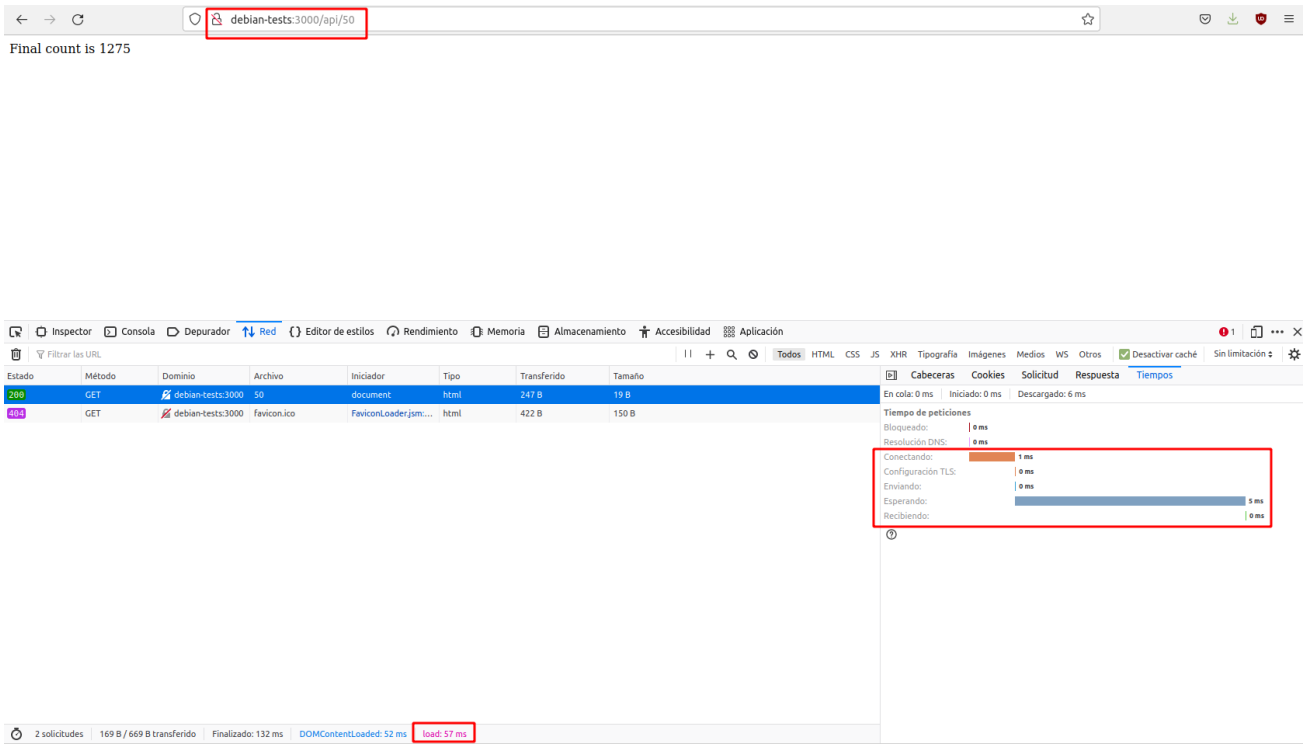
Los *workers* son creados y administrados por el proceso maestro. Cuando la aplicación se ejecuta por primera vez, verificamos si es un proceso maestro con `isMaster`. Esto está determinado por la variable `process.env.NODE_UNIQUE_ID`. Si `process.env.NODE_UNIQUE_ID` tiene valor *undefined*, entonces `isMaster` será *true*.

Si el proceso es un maestro, llamamos a `cluster.fork()` para generar varios procesos. Registramos los ID de proceso maestro y *worker*. Cuando un proceso secundario muere, generamos uno nuevo para seguir utilizando los núcleos de CPU disponibles.

Ahora repetiremos el mismo experimento de antes, primero realizamos una solicitud al servidor con un valor alto `n`:



Y ejecutamos rápidamente otra solicitud en otra pestaña del navegador, midiendo los tiempos de procesamiento de ambas:



Comprobaremos que éstos se reducen drásticamente.

Note

Con varios *workers* disponibles para aceptar solicitudes, se mejoran tanto la disponibilidad del servidor como el rendimiento.

Ejecutar una solicitud en una pestaña del navegador y ejecutar rápidamente otra en una segunda pestaña sirve para mostrarnos la mejora que ofrece la agrupación en clústeres para nuestro ejemplo

de una forma más o menos rápida, pero es un método un tanto "chapucero" y no es una forma adecuada o confiable de determinar las mejoras de rendimiento.

En el siguiente apartado echaremos un vistazo a algunos puntos de referencia que demostrarán mejor cuánto ha mejorado la agrupación en clústeres nuestra aplicación.

Métricas de rendimiento

Realizaremos una prueba de carga en nuestras dos aplicaciones para ver cómo cada una maneja una gran cantidad de conexiones entrantes. Usaremos el paquete `loadtest` para esto.

El paquete `loadtest` nos permite simular una gran cantidad de conexiones simultáneas a nuestra API para que podamos medir su rendimiento.

Para usar `loadtest`, primero debemos instalarlo globalmente. Tras conectaros por SSH al servidor Debian:

```
npm install -g loadtest
```

Luego ejecutamos la aplicación que queremos probar (`node nombre_aplicacion.js`). Comenzaremos probando la versión que no utiliza la agrupación en clústeres.

Mientras ejecutamos la aplicación, en otro terminal realizamos la siguiente prueba de carga:

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

El comando anterior enviará 1000 solicitudes a la URL dada, de las cuales 100 son concurrentes. El siguiente es el resultado de ejecutar el comando anterior:

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100
[Sat Jul 23 2022 13:11:30 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Target URL:          http://localhost:3000/api/500000
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Max requests:        1000
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level:   100
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Agent:               none
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Completed requests:  1000
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Total errors:        0
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Total time:          2.4763393500000004 s
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 404
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Mean latency:        232.4 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO   50%      227 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO   90%      263 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO   95%      268 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO   99%      314 ms
[Sat Jul 23 2022 13:11:32 GMT+0200 (hora de verano de Europa central)] INFO  100%      330 ms (longest request)
```

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo.js
App listening on port 3000
```

Vemos que con la misma solicitud (con n= 500000) el servidor ha podido manejar 404 solicitudes por segundo con una latencia media de 232.4 milisegundos (el tiempo promedio que tarda en completar una sola solicitud).

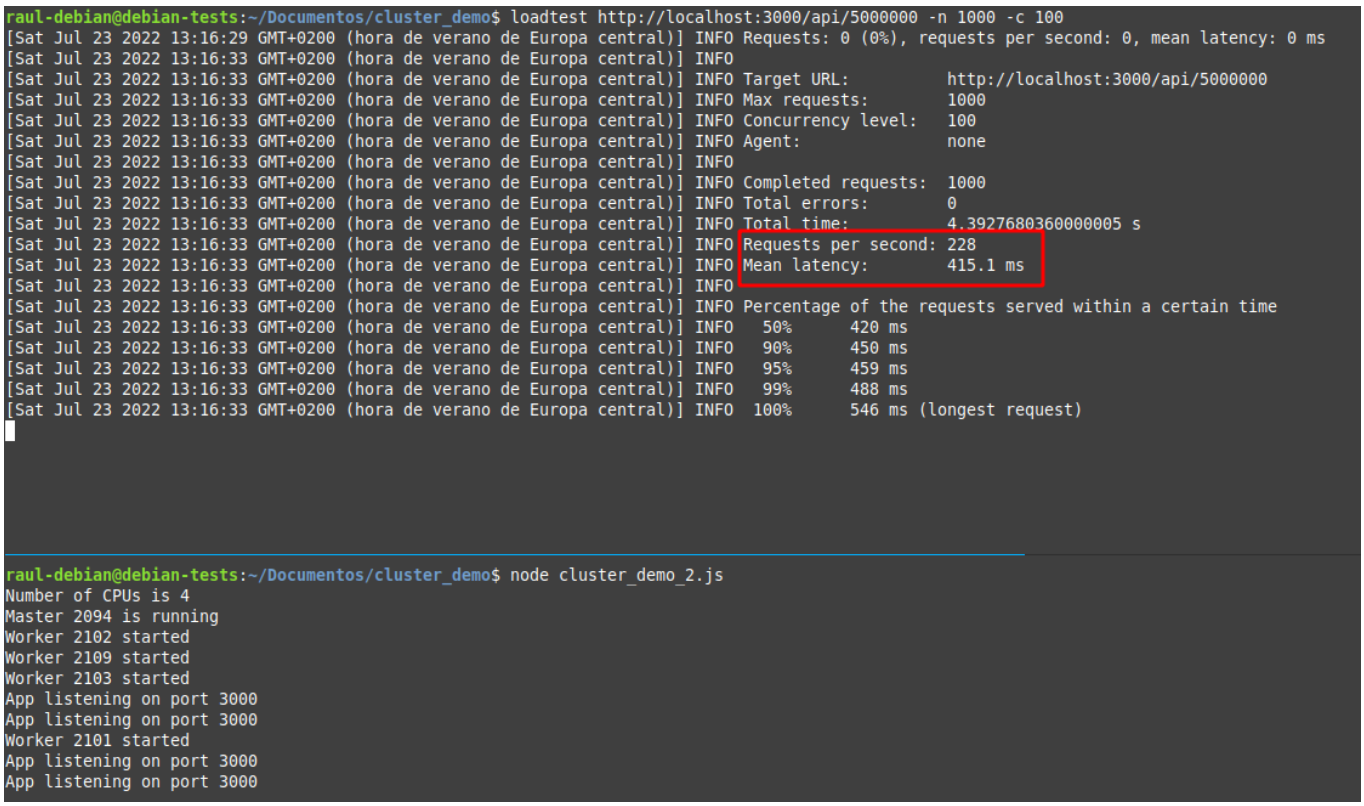
Intentémoslo de nuevo, pero esta vez con más solicitudes (y sin clústeres):

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/5000000 -n 1000 -c 100
[Sat Jul 23 2022 13:15:08 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:15:13 GMT+0200 (hora de verano de Europa central)] INFO Requests: 369 (37%), requests per second: 74, mean latency: 1183.5 ms
[Sat Jul 23 2022 13:15:18 GMT+0200 (hora de verano de Europa central)] INFO Requests: 756 (76%), requests per second: 77, mean latency: 1297.6 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Target URL:          http://localhost:3000/api/5000000
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Max requests:        1000
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level:   100
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Agent:               none
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Completed requests:  1000
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Total errors:        0
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Total time:          13.211158043 s
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 76
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Mean latency:        1253.6 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO   50%      1299 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO   90%      1319 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO   95%      1342 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO   99%      1389 ms
[Sat Jul 23 2022 13:15:22 GMT+0200 (hora de verano de Europa central)] INFO  100%      1451 ms (longest request)

raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo.js
App listening on port 3000
```

Vemos que las métricas arrojan resultados aún peores.

Ahora detenemos nuestra aplicación sin clústers y ejecutamos la que sí los tiene (`node nombre_aplicacion_cluster.js`). Ejecutaremos exactamente las mismas pruebas con el objetivo de realizar una comparación:

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/500000 -n 1000 -c 100
[Sat Jul 23 2022 13:12:52 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Target URL:          http://localhost:3000/api/500000
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Max requests:        1000
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level:   100
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Agent:               none
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Completed requests:  1000
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Total errors:        0
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Total time:          1.529857035 s
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 654
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Mean latency:        141.7 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO   50%      137 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO   90%      189 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO   95%      212 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO   99%      250 ms
[Sat Jul 23 2022 13:12:53 GMT+0200 (hora de verano de Europa central)] INFO  100%      271 ms (longest request)

raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo_2.js
Number of CPUs is 4
Master 2004 is running
Worker 2014 started
Worker 2013 started
App listening on port 3000
Worker 2017 started
App listening on port 3000
App listening on port 3000
Worker 2021 started
App listening on port 3000
```



Es obvio que los clústers permiten manejar una mayor cantidad de peticiones por segundo con una menor latencia.

Uso de PM2 para administrar un clúster de Node.js

En nuestra aplicación, hemos usado el módulo `cluster` de Node.js para crear y administrar manualmente los procesos.

Primero hemos determinado la cantidad de *workers* (usando la cantidad de núcleos de CPU como referencia), luego los hemos generado y, finalmente, escuchamos si hay *workers* muertos para poder generar nuevos.

En nuestra aplicación de ejemplo muy sencilla, tuvimos que escribir una cantidad considerable de código solo para administración la agrupación en clústeres. En una aplicación de producción es bastante probable que se deba escribir aún más código.

Existe una herramienta que nos puede ayudar a administrar todo esto un poco mejor: el administrador de procesos `PM2`. `PM2` es un administrador de procesos de producción para aplicaciones Node.js con un balanceador de carga incorporado.

Cuando está configurado correctamente, `PM2` ejecuta automáticamente la aplicación en modo de clúster, generando *workers* y se encarga de generar nuevos *workers* cuando uno de ellos muera.

`PM2` facilita la parada, eliminación e inicio de procesos, además de disponer de algunas herramientas de monitorización que pueden ayudarnos a monitorizar y ajustar el rendimiento de su aplicación.

Para usar `PM2`, primero instalamos globalmente en nuestra Debian:

```
npm install pm2 -g
```

Vamos a utilizarlo con nuestra primera aplicación, la que no estaba "clusterizada" en el código. Para ello ejecutaremos el siguiente comando:

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

Donde:

- `-i` `PM2` que inicie la aplicación en `cluster_mode` (a diferencia de `fork_mode`).

Si `PM2` generará automáticamente tantos *workers* como núcleos de CPU haya.

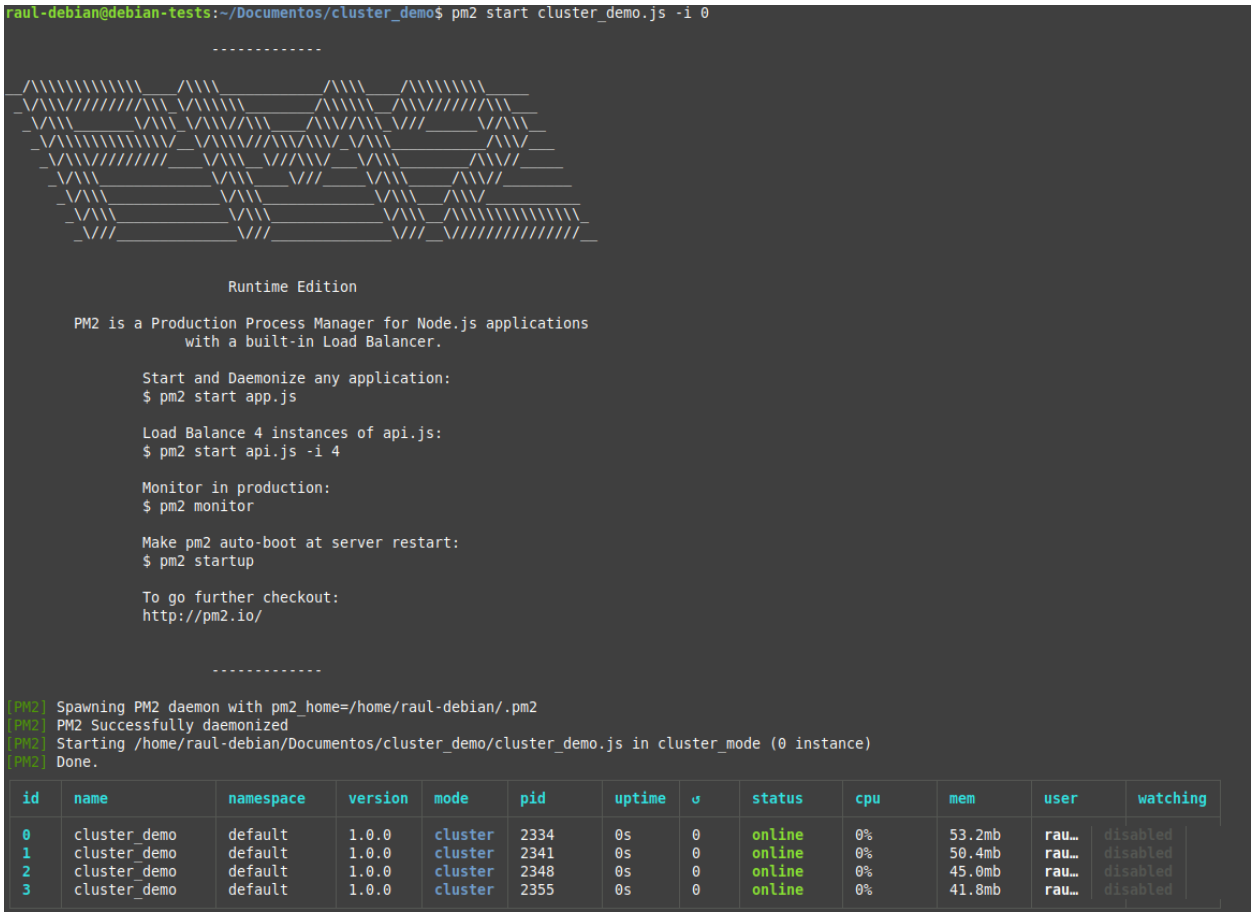
Y así, nuestra aplicación se ejecuta en modo de clúster, sin necesidad de cambios de código.

Task

Ejecuta y documenta con capturas de pantallas, las mismas pruebas que antes pero utilizando PM2 y comprueba si se obtienen los mismos resultados.

Por detrás, `PM2` también utiliza el módulo `cluster` de Node.js, así como otras herramientas que facilitan la gestión de procesos.

En el Terminal, obtendremos una tabla que muestra algunos detalles de los procesos generados:



Podemos detener la aplicación con el siguiente comando:

```
pm2 stop app.js
```

La aplicación se desconectará y la salida por terminal mostrará todos los procesos con un estado `stopped`.

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ pm2 stop cluster_demo.js
[PM2] Applying action stopProcessId on app [cluster_demo.js](ids: [ 0, 1, 2, 3 ])
[PM2] [cluster_demo](0) ✓
[PM2] [cluster_demo](1) ✓
[PM2] [cluster_demo](2) ✓
[PM2] [cluster_demo](3) ✓
```

id	name	namespace	version	mode	pid	uptime	↺	status	cpu	mem	user	watching
0	cluster_demo	default	1.0.0	cluster	0	0	0	stopped	0%	0b	rau…	disabled
1	cluster_demo	default	1.0.0	cluster	0	0	0	stopped	0%	0b	rau…	disabled
2	cluster_demo	default	1.0.0	cluster	0	0	0	stopped	0%	0b	rau…	disabled
3	cluster_demo	default	1.0.0	cluster	0	0	0	stopped	0%	0b	rau…	disabled

En vez de tener que pasar siempre las configuraciones cuando ejecuta la aplicación con `pm2 start app.js -i 0`, podríamos facilitarnos la tarea y guardarlas en un archivo de configuración separado, llamado Ecosystem.

Este archivo también nos permite establecer configuraciones específicas para diferentes aplicaciones.

Crearemos el archivo *Ecosystem* con el siguiente comando:

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ pm2 ecosystem
File /home/raul-debian/Documentos/cluster_demo/ecosystem.config.js generated
raul-debian@debian-tests:~/Documentos/cluster_demo$
```

Que generará un archivo llamado *ecosystem.config.js*. Para el caso concreto de nuestra aplicación, necesitamos modificarlo como se muestra a continuación:

```
module.exports = {
 apps: [
 {
 name: "nombre_aplicacion",
 script: "nombre_aplicacion_sin_cluster.js",
 instances: 0,
 exec_mode: "cluster",
 },
 ],
};
```

Al configurar `exec_mode` con el valor `cluster`, le indica a `PM2` que balancee la carga entre cada instancia. `instances` está configurado a **0** como antes, lo que generará tantos *workers* como núcleos de CPU.

La opción `-i` o `instances` se puede establecer con los siguientes valores:

- `0` o `max`(en desuso) para "repartir" la aplicación entre todas las CPU
- `-1` para "repartir" la aplicación en todas las CPU - 1
- `número` para difundir la aplicación a través de un número concreto de CPU

Ahora podemos ejecutar la aplicación con:

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

La aplicación se ejecutará en modo clúster, exactamente como antes.

Podremos iniciar, reiniciar, recargar, detener y eliminar una aplicación con los siguientes comandos, respectivamente:

```
$ pm2 start nombre_aplicacion
```

```
$ pm2 restart nombre_aplicacion
$ pm2 reload nombre_aplicacion
$ pm2 stop nombre_aplicacion
$ pm2 delete nombre_aplicacion

# Cuando usemos el archivo Ecosystem:

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

El comando `restart` elimina y reinicia inmediatamente los procesos, mientras que el comando `reload` logra un tiempo de inactividad de 0 segundos donde los *workers* se reinician uno por uno, esperando que aparezca un nuevo *worker* antes de matar al anterior.

También puede verificar el estado, los registros y las métricas de las aplicaciones en ejecución.

Task

Investiga los siguientes comandos y explica que salida por terminal nos ofrecen y para qué se utilizan:

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

Warning

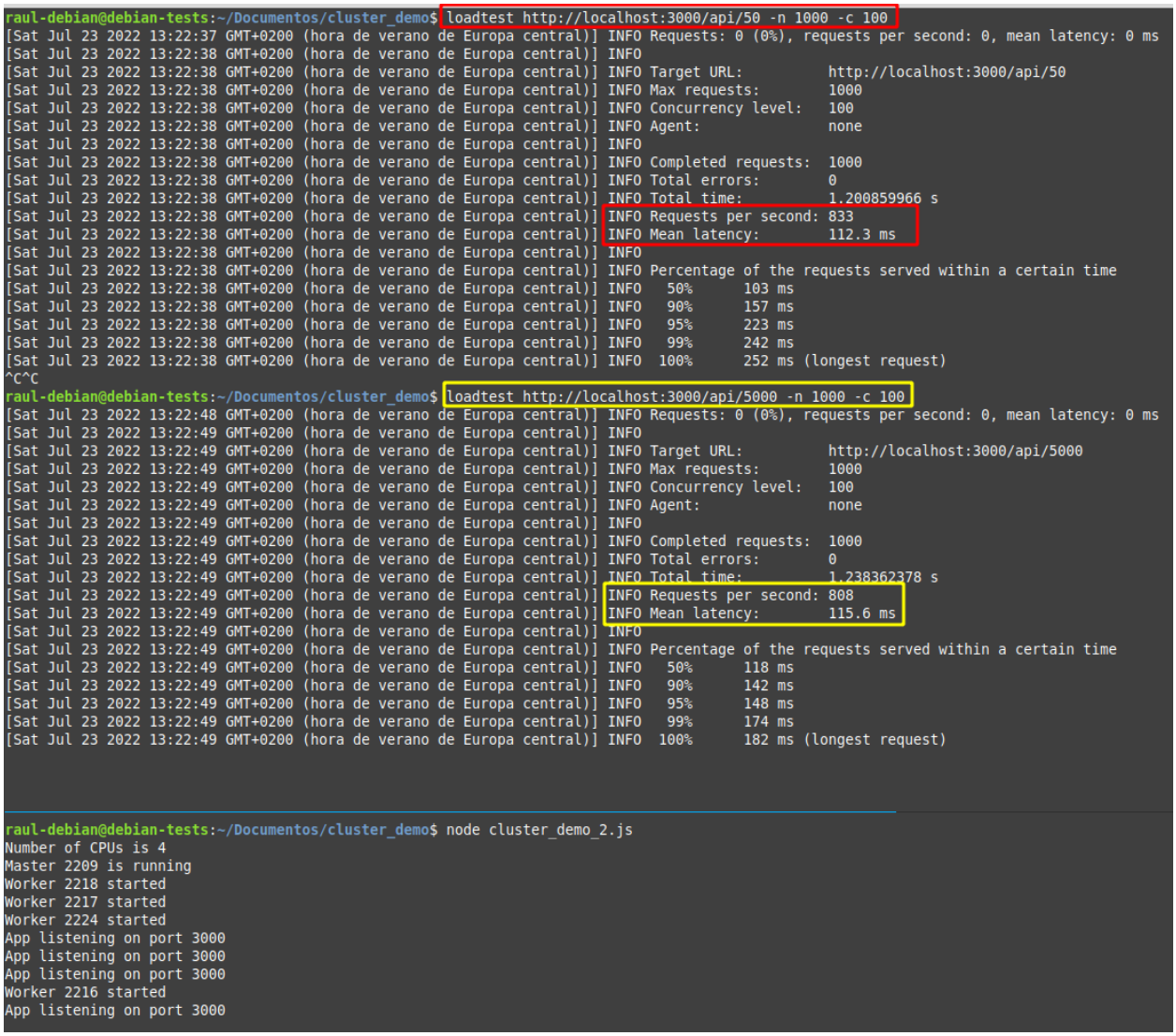
Documenta la realización de toda esta práctica adecuadamente, con las explicaciones y justificaciones necesarias, las respuestas a las preguntas planteadas y las capturas de pantalla pertinentes.

Cuestiones

Fijaos en las siguientes imágenes:

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/50 -n 1000 -c 100
[Sat Jul 23 2022 13:20:18 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Target URL:          http://localhost:3000/api/50
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Max requests:        1000
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level:   100
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Agent:               none
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Completed requests:  1000
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Total errors:        0
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Total time:          1.1908691169999999 s
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 840
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Mean latency:        109.4 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO   50%      103 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO   90%      154 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO   95%      161 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO   99%      183 ms
[Sat Jul 23 2022 13:20:20 GMT+0200 (hora de verano de Europa central)] INFO  100%      192 ms (longest request)

^C
raul-debian@debian-tests:~/Documentos/cluster_demo$ loadtest http://localhost:3000/api/5000 -n 1000 -c 100
[Sat Jul 23 2022 13:20:33 GMT+0200 (hora de verano de Europa central)] INFO Requests: 0 (0%), requests per second: 0, mean latency: 0 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Target URL:          http://localhost:3000/api/5000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Max requests:        1000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Concurrency level:   100
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Agent:               none
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Completed requests:  1000
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Total errors:        0
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Total time:          1.088355608 s
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Requests per second: 919
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Mean latency:        97.6 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO Percentage of the requests served within a certain time
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO   50%      88 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO   90%      124 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO   95%      130 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO   99%      158 ms
[Sat Jul 23 2022 13:20:34 GMT+0200 (hora de verano de Europa central)] INFO  100%      162 ms (longest request)
```

```
raul-debian@debian-tests:~/Documentos/cluster_demo$ node cluster_demo.js
App listening on port 3000
```



La primera imagen ilustra los resultados de unas pruebas de carga sobre la aplicación sin clúster y la segunda sobre la aplicación clusterizada.

¿Sabrías decir por qué en algunos casos concretos, como este, la aplicación sin clusterizar tiene mejores resultados?

Referencias

[How to install ExpressJS on Debian 11?]

[Improving Node.js Application Performance With Clustering]

Práctica 3.4: Despliegue de una aplicación Node.js en Heroku (PaaS) y una aplicación React en Netlify (PaaS)

Introducción

En la práctica anterior hemos visto cómo desplegar una aplicación de Node.js sobre un servidor Express en local (en nuestro propio servidor Debian).

La práctica anterior podría asemejarse a las pruebas que realiza un desarrollador antes de pasar su aplicación al entorno de producción.

Ya sabemos que entendemos el *despliegue o deployment* como el proceso de mover nuestro código típicamente de un sistema de control de versiones a una plataforma de hosting donde se aloja y es servida a los usuarios finales.

A la hora de desplegar la aplicación en producción, podría utilizarse el método de copiar los archivos al servidor concreto vía el vetusto FTP, SSH u otros y desplegarla para dejarla funcionando. No obstante, esta práctica se acerca más a la realidad ya que utilizaremos un repositorio de Github y una plataforma de PaaS (Platform as a Service) como Heroku o Netlify para desplegar adecuadamente nuestra aplicación en producción.

¿Qué es Github?

A pesar de que trataremos un poco más en profundidad Github en el siguiente tema, daremos una breve explicación aquí.

GitHub es un servicio basado en la nube que aloja un sistema de control de versiones (VCS) llamado Git. Éste permite a los desarrolladores colaborar y realizar cambios en proyectos compartidos, a la vez que mantienen un seguimiento detallado de su progreso.

El control de versiones es un sistema que ayuda a rastrear y gestionar los cambios realizados en un archivo o conjunto de archivos. Utilizado principalmente por ingenieros de software para hacer un seguimiento de las modificaciones realizadas en el código fuente, el sistema de control de versiones les permite analizar todos los cambios y revertirlos sin repercusiones si se comete un error.

¿Qué es Heroku?

Heroku es una solución de Plataforma como Servicio (PaaS) basada en la nube para que el cliente solo se preocupe de desarrollar su aplicación mientras Heroku se encarga de la infraestructura que hay detrás.

Para proporcionar este servicio se dispone de unos contenedores virtuales que son los encargados de mantener y ejecutar las aplicaciones. Estos contenedores virtuales son totalmente escalables bajo demanda. Tanto en número como en capacidades.

Una ventaja de elegir Heroku es su capacidad de soportar múltiples lenguajes de programación. Los principales a utilizar son: Node.js, Ruby, Python, Java, PHP, Go, Scala y Clojure. Aunque esta cantidad de lenguajes puede aumentar en el caso de utilizar Heroku Buildpacks, que permiten compilar las aplicaciones en multitud de ellos más.

Note

Tanto **Github**, como **Heroku**, como **Netlify** pueden ser controlados desde el terminal de nuestro Linux, por lo que seguiremos el procedimiento de contectarnos vía SSH a nuestro Debian y realizar las operaciones por terminal.

¿Qué es Netlify?

Netlify es un proveedor de alojamiento en la nube que proporciona servicios de backend sin servidor (*serverless*) para sitios web estáticos. Está diseñado para maximizar la productividad en el sentido de que permite a los desarrolladores (especialmente orientados al frontend), y a los ingenieros construir, probar y desplegar rápidamente sitios web/aplicaciones.

Funciona conectándose a un repositorio de GitHub, de donde extrae el código fuente. A continuación, ejecutará un proceso de construcción para pre-renderizar las páginas de nuestro sitio web/aplicación en archivos estáticos.

Hay numerosas razones a favor de usar Netlify, aquí están algunas de ellas:

- Netlify hace que sea increíblemente sencillo desplegar un sitio web - de hecho, la forma más sencilla de lograrlo es utilizar GitHub, GitLab o Bitbucket para configurar el despliegue continuo.
- Netlify hace que sea súper fácil lanzar un sitio web con su solución de gestión de DNS incorporada.
- Podríamos desplegar fácilmente sólo una rama específica de nuestro proyecto Git - esto es útil para probar nuevas características que pueden o no llegar a la rama maestra/principal, o para determinar rápidamente cómo un PR (Pull Request) afectará a su sitio.
- Netlify te permite previsualizar cualquier despliegue que hagas o quieras hacer - esto te permite a ti y a tu equipo ver cómo se verán los cambios en producción sin tener que desplegarlos en tu sitio existente.
- Netlify proporciona una práctica función de envío de formularios que nos permite recoger información de los usuarios.

Creación de nuestra aplicación para Heroku

Tras loguearnos por SSH en nuestro Debian, nos crearemos un directorio para albergar la aplicacón con el nombre que queramos. En ese directorio, crearemos los 3 archivos (dos `.html` y un `.js`)que conformarán nuestra sencilla aplicación de ejemplo:

```
<!DOCTYPE html>
<html>
<head>
        <title>Hola Mundo</title>
</head>
<body>

        <h1>Esta es la página principal</h1>
```

```
<p><a href="/tailPage">Ir a la siguiente página</a></p>


</body>
```

Ahora, tal y como hacemos siempre a la hora de crear nuestra aplicación **Node.js**, con el fin de crear el archivo `package.json`, utilizaremos en el terminal el comando:

```
npm init
```

Podemos probar que nuestra aplicación funciona perfectamente en local:

```
node aplicacion.js
```

Y tras ello, debemos poder acceder, desde nuestra máquina anfitriona a `http://IP-maq-virtual:8080`

Ya con la aplicación creada y comprobada, podremos desplegarla en múltiples plataformas en la nube, como AWS, GCP, Azure, Digital Ocean, Heroku...

¡Ojo!

Para que nos funcione en Heroku, en el archivo `package.json` que se nos ha creado al hacer el `npm init` debemos hacerle una modificación.

En el bloque scripts, debemos borrar lo que haya dentro y dejar únicamente dentro de él:

```
"start": "node aplicacion.js"
```

De forma que Heroku sepa que comando utilizar para iniciar la aplicación tras desplegarla.

Proceso de despliegue en Heroku

Para trabajar con Heroku desde nuestro terminal, debemos instalar el propio CLI de Heroku. Consultando la documentación, vemos que hemos de ejecutar:

```
curl https://cli-assets.heroku.com/install.sh | sh
```

Y comprobamos que se ha instalado correctamente consultando su versión:

```
heroku -v
```

Lo siguiente será loguearnos en nuestra cuenta de Heroku mediante el terminal, para ello:

```
heroku login
```

Esto en teoría nos abre una pestaña del navegador para loguearnos en nuestra cuenta. Puesto que estamos conectados por SSH a nuestra Debian, no sucederá esto ya que el único puerto por el que nos comunicamos es por el 22. Necesitaríamos un túnel SSH para redirigir los puertos de la máquina Debian remota a la nuestra y que nos abriese el navegador en nuestra máquina.

Puesto que esto escapa de los objetivos del módulo y con el fin de agilizar el proceso, simplemente copiaremos la URL y la pegaremos en nuestro navegador para loguearnos.

Antes de continuar, conviene asegurarnos de que tenemos la última versión de git en nuestra Debian:

```
sudo apt-get update && sudo apt-get install git
```

Ahora, dentro del directorio que habíamos creado previamente para nuestra aplicación, se trata de seguir unos sencillos pasos:

Tip

Aquí aparece explicado con lenguaje *llano* más adelante en el módulo ya hablaremos con mayor propiedad de estas acciones con git

1. Nos aseguramos de que nuestro directorio no es aún un repositorio: `git status`

Y lo iniciamos: `git init`

```
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ git status
fatal: no es un repositorio git (ni ninguno de los directorios superiores): .git
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ git init
ayuda: Usando 'master' como el nombre de la rama inicial. Este nombre de rama predeterminado
ayuda: está sujeto a cambios. Para configurar el nombre de la rama inicial para usar en todos
ayuda: de sus nuevos repositorios, reprimiendo esta advertencia, llama a:
ayuda:
ayuda:  git config --global init.defaultBranch <nombre>
ayuda:
ayuda: Los nombres comúnmente elegidos en lugar de 'master' son 'main', 'trunk' y
ayuda: 'development'. Se puede cambiar el nombre de la rama recién creada mediante este comando:
ayuda:
ayuda:  git branch -m <nombre>
Inicializado repositorio Git vacío en /home/raul-debian/Documentos/ejemplo_heroku/.git/
```

2. Ahora añadimos todos los archivos presentes en el directorio (`.`) para ser enviados al repositorio: `git add .`

Y los preparamos para que sean envidos al repositorio: `git commit -m "Comentario explicativo del commit"`

```
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ git add .
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ git commit -m "test"
[master (commit-raíz) 8a8146d] test
 5 files changed, 184 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 aplicacion.js
 create mode 100644 head.html
 create mode 100644 package.json
 create mode 100644 tail.html
```

3. Creamos nuestra aplicación en Heroku: `heroku create`

```
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ heroku create
Creating app... done, ● boiling-tundra-97116
https://boiling-tundra-97116.herokuapp.com/ | https://git.heroku.com/boiling-tundra-97116.git
```

Esto creará un git remoto que conectará con nuestro repositorio git local

4. Desplegamos nuestra aplicación en el server de Heroku : `git push heroku master`

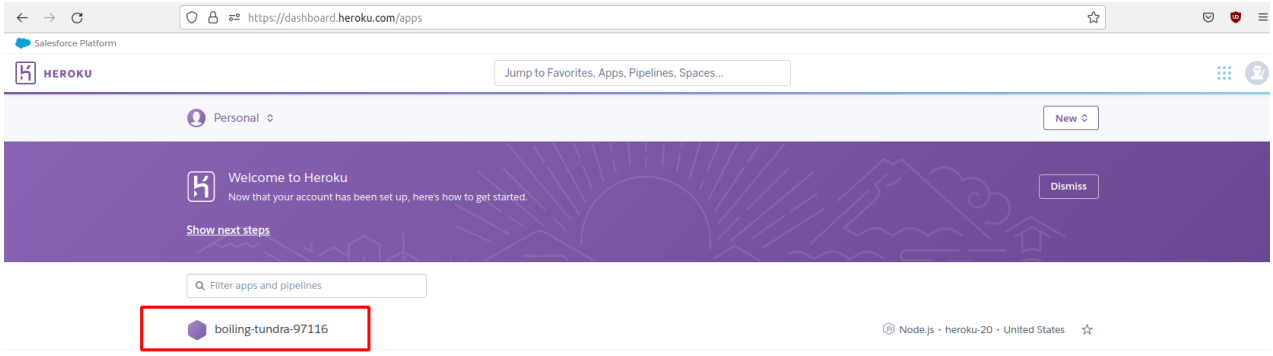
Y comprobamos que la instancia está corriendo: `heroku ps:scale web=1`

```
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ git push heroku master
Enumerando objetos: 7, listo.
Contando objetos: 100% (7/7), listo.
Compresión delta usando hasta 4 hilos
Comprimiendo objetos: 100% (7/7), listo.
Escribiendo objetos: 100% (7/7), 2.04 KiB | 2.04 MiB/s, listo.
Total 7 (delta 0), reusados 0 (delta 0), pack-reusados 0
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Building on the Heroku-20 stack
remote: -----> Determining which buildpack to use for this app
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
remote:        NODE_VERBOSE=false
remote:        NODE_ENV=production
remote:        NODE_MODULES_CACHE=true
remote:
remote: -----> Installing binaries
remote:        engines.node (package.json):  unspecified
remote:        engines.npm (package.json):   unspecified (use default)
remote:
remote:        Resolving node version 16.x...
remote:        Downloading and installing node 16.16.0...
remote:        Using default npm version: 8.11.0
remote:
remote: -----> Installing dependencies
remote:        Installing node modules (package.json)
remote:
remote:        up to date, audited 1 package in 97ms
remote:
remote:        found 0 vulnerabilities
remote:
remote: -----> Build
remote:
remote: -----> Caching build
remote:        - node_modules (nothing to cache)
remote:
remote: -----> Pruning devDependencies
remote:
remote:        up to date, audited 1 package in 90ms
remote:
remote:        found 0 vulnerabilities
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
 0  1h 16m  1 heroku
```

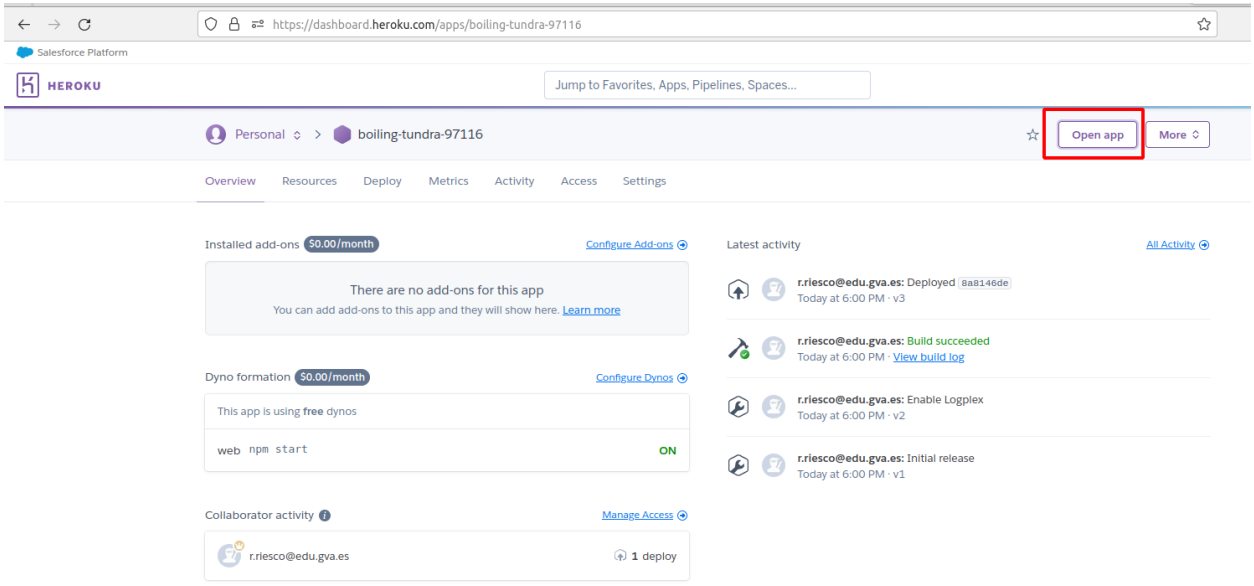
```
remote: -----> Pruning devDependencies
remote:
remote:        up to date, audited 1 package in 90ms
remote:
remote:        found 0 vulnerabilities
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:        Done: 31.6M
remote: -----> Launching...
remote:        Released v3
remote:        https://boiling-tundra-97116.herokuapp.com/ deployed to Heroku
remote:
remote: This app is using the Heroku-20 stack, however a newer stack is available.
remote: To upgrade to Heroku-22, see:
remote: https://devcenter.heroku.com/articles/upgrading-to-the-latest-stack
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/boiling-tundra-97116.git
 * [new branch]      master -> master
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free
raul-debian@debian-tests:~/Documentos/ejemplo_heroku$
```

5. El comando `heroku open` abriría nuestra aplicación en el navegador. Sin embargo, por el problema explicado antes de estar conectados por SSH, esto no ocurrirá. No obstante, podemos acceder a nuestra aplicación de otra forma rápida y sencilla desde nuestro dashboard de Heroku:

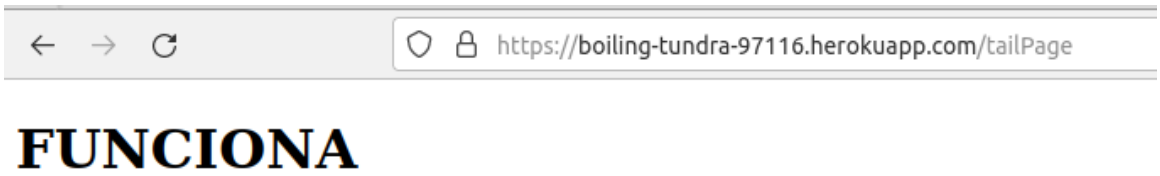
    - Localizamos nuestra aplicación:



    - Y tras hacer click en ella, localizamos el botón que nos permite abrirla y volvemos a hacer click:



- Comprobando que nuestra aplicación, efectivametne se ha desplegado en Heroku y funciona a la perfección:



Aplicación para Netlify

Puesto que el interés en este módulo radica en el proceso de despliegue, suponiendo que la parte de desarrollo ya es abordada en otros módulos, vamos a utilizar una aplicación de ejemplo que nos ahorre tiempo para centrarnos en el despliegue.

Nos clonaremos este repositorio:

```
git clone https://github.com/StackAbuse/color-shades-generator
```

Proceso de despliegue en Netlify

Por mera curiosidad y ambición de aprendizaje, vamos a ver dos métodos de despliegue en Netlify:

- Despliegue manual desde el CLI de Netlify, es decir, desde el terminal, a partir de un directorio local de nuestra máquina.
- Despliegue desde un código publicado en uno de nuestros repositorios de Github

El primero nos permitirá conocer el CLI de Netlify y el segundo nos acercara más a una experiencia real de despliegue.

Task

Vuestra primera tarea será registraros en Netlify con vuestro email (no con vuestra cuenta de Github) y decirle que no cuando os pida enlazar con vuestra cuenta de Github (lo haremos más adelante).

Despliegue mediante CLI

Una vez registrados, debemos instalar el CLI de Netlify para ejecutar sus comandos desde el terminal:

```
sudo npm install netlify-cli -g
```

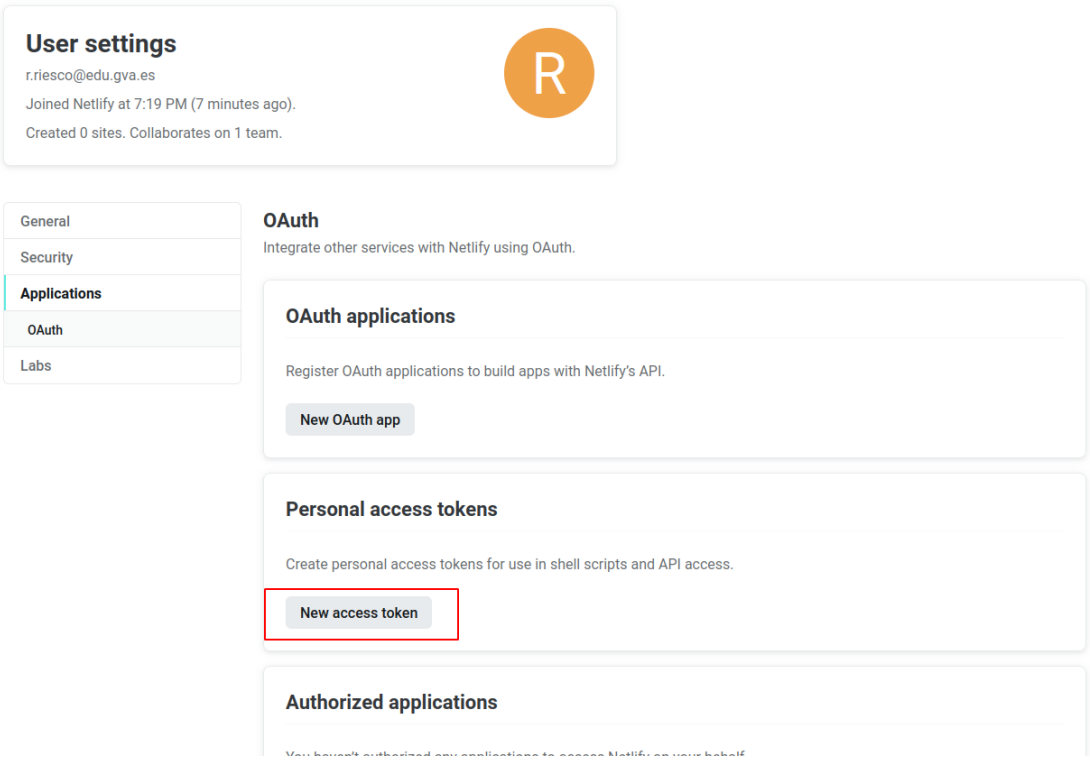
Está claro que para realizar acciones de deploy, Netlify nos solicitará una autenticación, esto se hace mediante el comando:

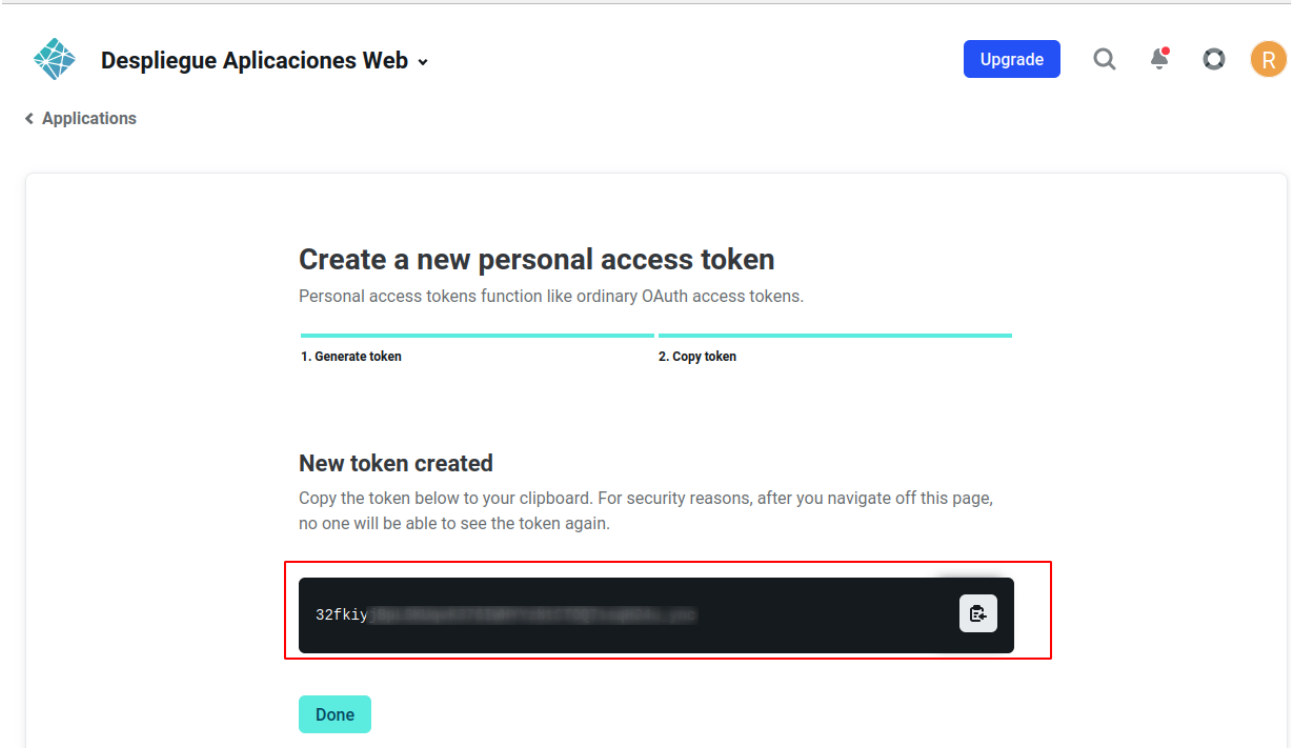
```
netlify login
```

El cual nos muestra una pantalla del navegador para que concedamos la autorización pertinente. Sin embargo, recordemos el problema de que estamos conectados por SSH a nuestro servidor y no tenemos la posibilidad del uso de un entorno gráfico.

En este caso, siguiendo las instrucciones de la documentación:

- Generamos el token de acceso





- Lo establecemos como variable de ambiente:

Y nos logueamos

```
netlify login
```

Bueno, tenemos el código de nuestra aplicación, tenemos nuestra cuenta en Netlify y tenemos el CLI necesario para ejecutar comandos desde el terminal en esa cuenta... ¿Podemos proceder al despliegue sin mayores complicaciones?

La respuesta es **NO**, como buenos desarrolladores y en base a experiencias anteriores, ya sabéis que hay que hacer un *build* de la aplicación para, posteriormente, desplegarla. Vamos a ello.

En primer lugar, como sabemos, debemos instalar todas las dependencias que vienen indicadas en el archivo `package.json`:

```
npm install
```

Y cuando ya las tengamos instaladas podemos proceder a realizar el build:

```
npm run build
```

Esto nos creará una nueva carpeta llamada `build` que contendrá la aplicación que debemos desplegar. Y ya podemos hacer un pre-deploy de la aplicación de la que hemos hecho build antes:

```
netlify deploy
```

Nos hará algunas preguntas para el desplieuge:

- Indicamos que queremos crear y configurar un nuevo site
- El Team lo dejamos por defecto
- Le indicamos el nombre que queremos emplear para la web (`nombre-practica3-4`) y el directorio a utilizar para el deploy (directorio `./build`).

Y si nos indica que todo ha ido bien e incluso podemos ver el "borrador" (Website Draft URL) de la web que nos aporta, podemos pasarla a producción finalmente tal y como nos indica la misma salida del comando:

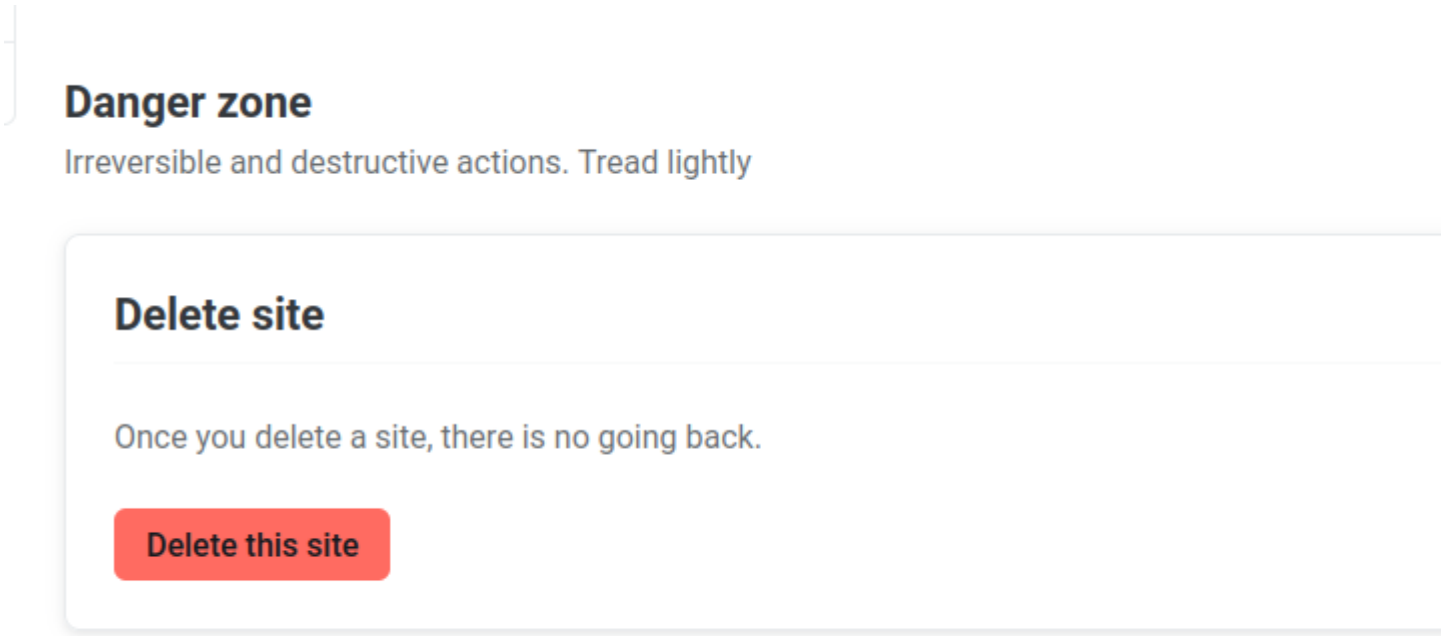
```
If everything looks good on your draft URL, deploy it to your main site URL with
the --prod flag.
netlify deploy --prod
```

Warning

No olvides desplegar finalmente en producción y comprobar que puedes acceder a la URL.

Despliegue mediante conexión con Github

En primer lugar, vamos a eliminar el site que hemos desplegado antes en Netlify para evitarnos cualquier problema y/o conflicto:



En segundo lugar, vamos a borrar el directorio donde se halla el repositorio clonado en el paso anterior para así poder empezar de 0:

```
rm -rf directorio_repositorio
```

Como queremos simular que hemos picado el código a mano en local y lo vamos a subir a Github por primera vez, nos descargaremos los fuentes en formato `.zip` sin que tenga ninguna referencia a Github:

```
wget https://github.com/StackAbuse/color-shades-
generator/archive/refs/heads/main.zip
```

Creamos una carpeta nueva y descomprimimos dentro el zip:

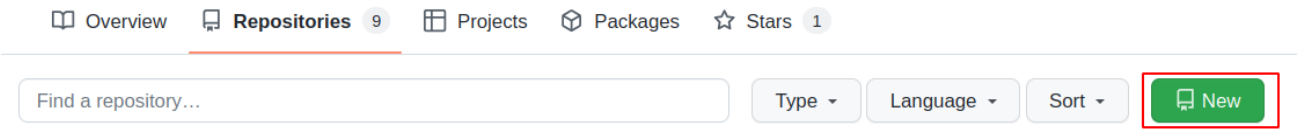
```
mkdir practica3.4
```

```
unzip main.zip -d practica3.4/
```

Entramos en la carpeta donde está el código:

```
cd practica3.4/color-shades-generator-main/
```

Ahora debemos crear un repositorio **completamente vacío** en Github que se llame `practicaTresCuatro`:



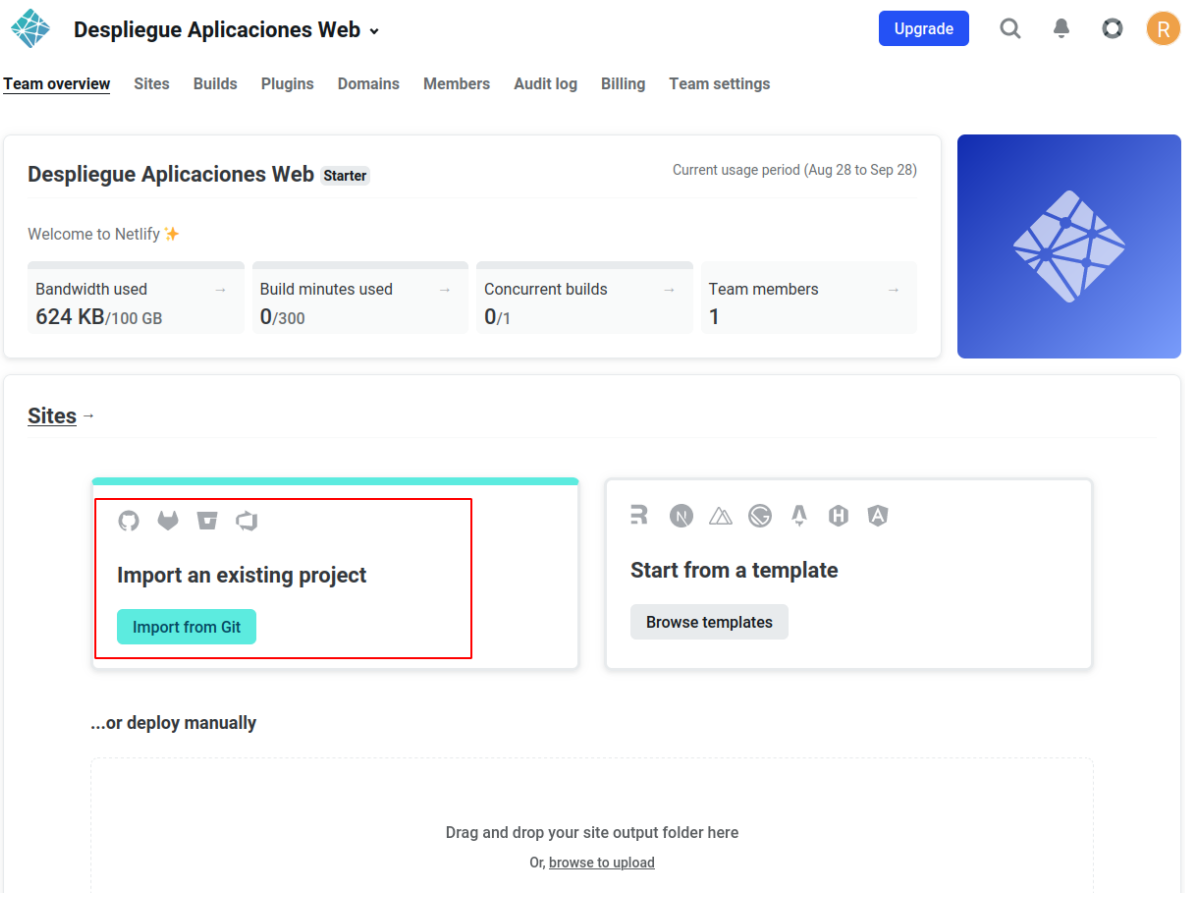
Y tras ello, volviendo al terminal a la carpeta donde estábamos, la iniciamos como repositorio, añadimos todo el contenido de la misma para el commit, hacemos el commit con el mensaje correspondiente y creamos la rama main:

```
$ git init
$ git add .
$ git commit -m "Subiendo el código..."
$ git branch -M main
```

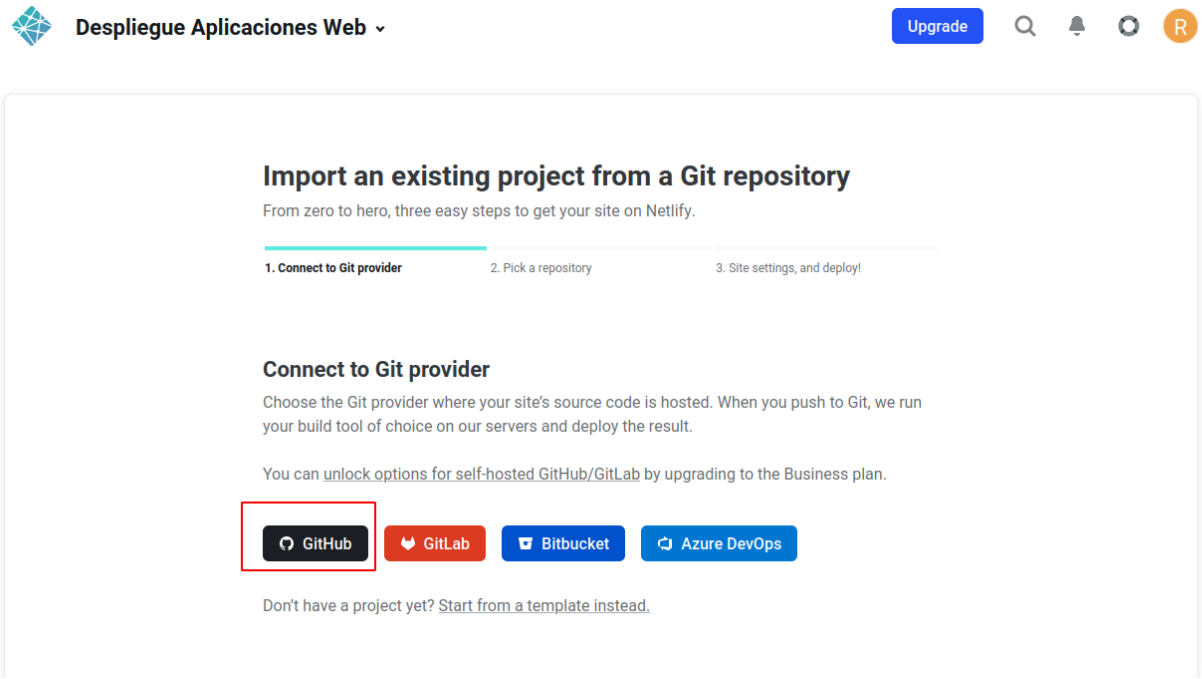
Y ahora sólo queda referenciar nuestra carpeta al repositorio recién creado en Github y hacer un `push` para subir todo el contenido del commit a él:

```
$ git remote add origin https://github.com/username/practicaTresCuatro.git
$ git push -u origin main
```

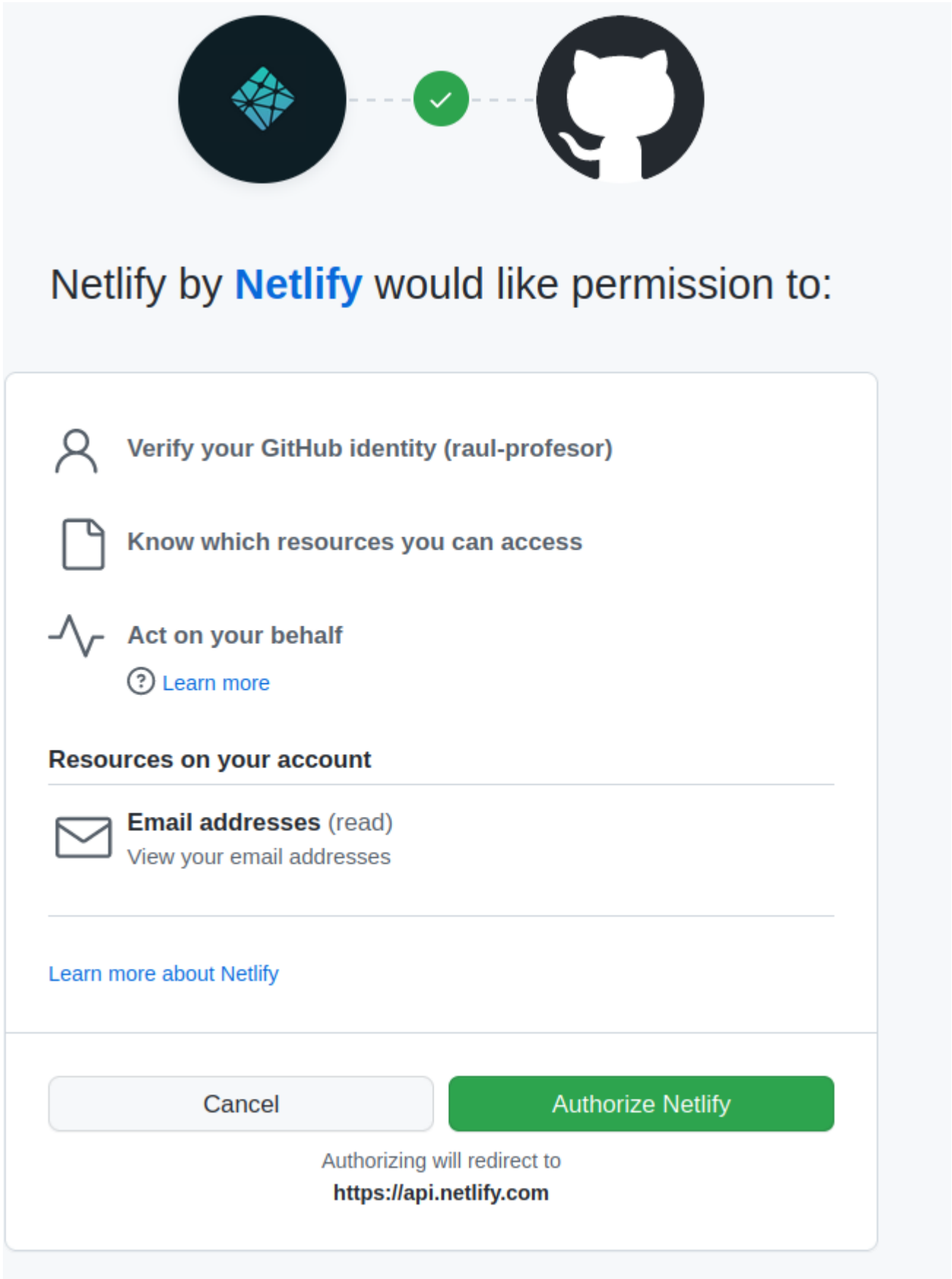
Ahora que ya tenemos subido el código a GitHub, de alguna manera debemos *enganchar* o enlazar nuestra cuenta de Github con la de Netlify para que éste último pueda traerse el código de allí, hacer el build y desplegarlo. Así pues, entramos en nuestro dashboard de Netlify y le damos a importar proyecto existente de `git`:



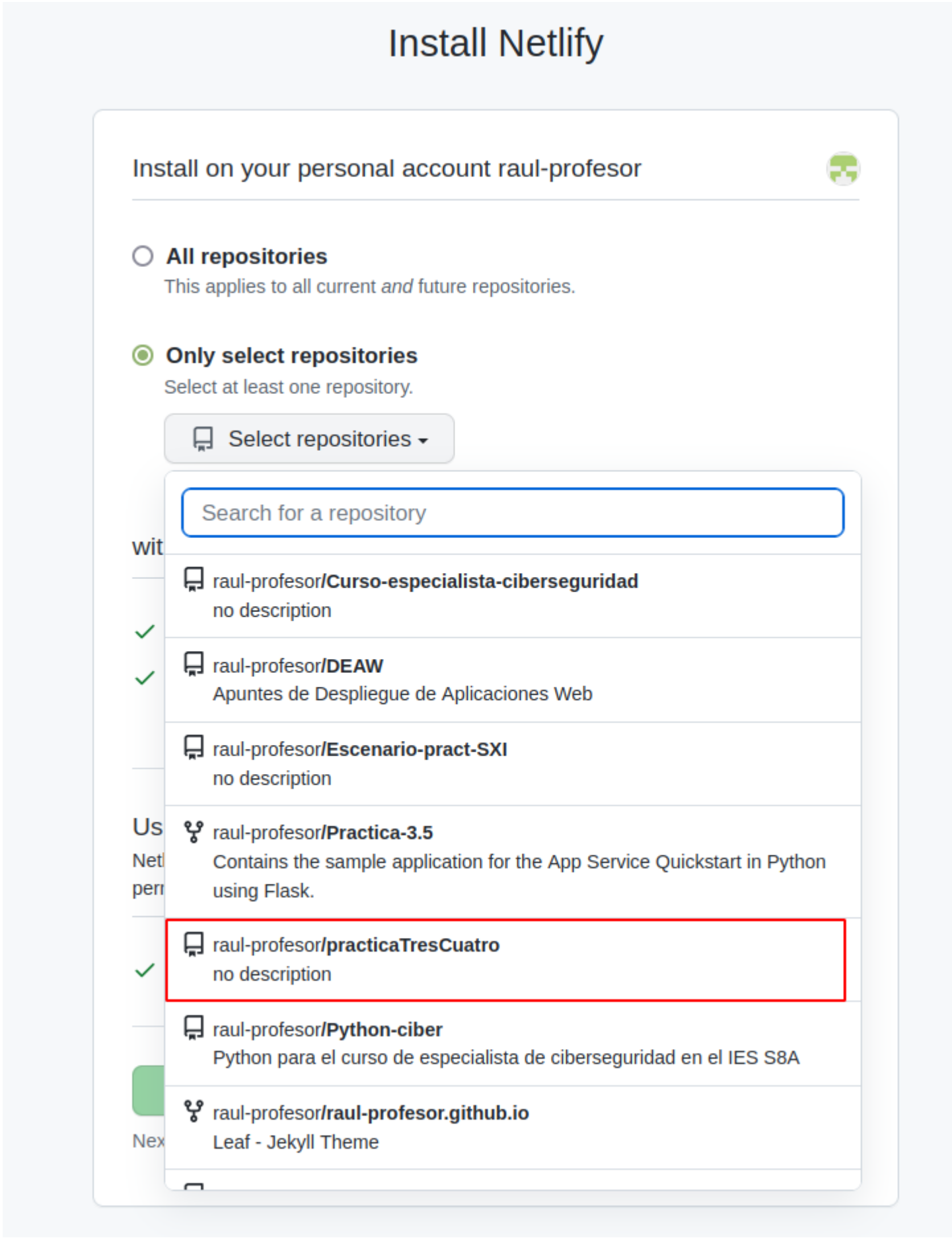
Le indicamos que concretamente de Github:



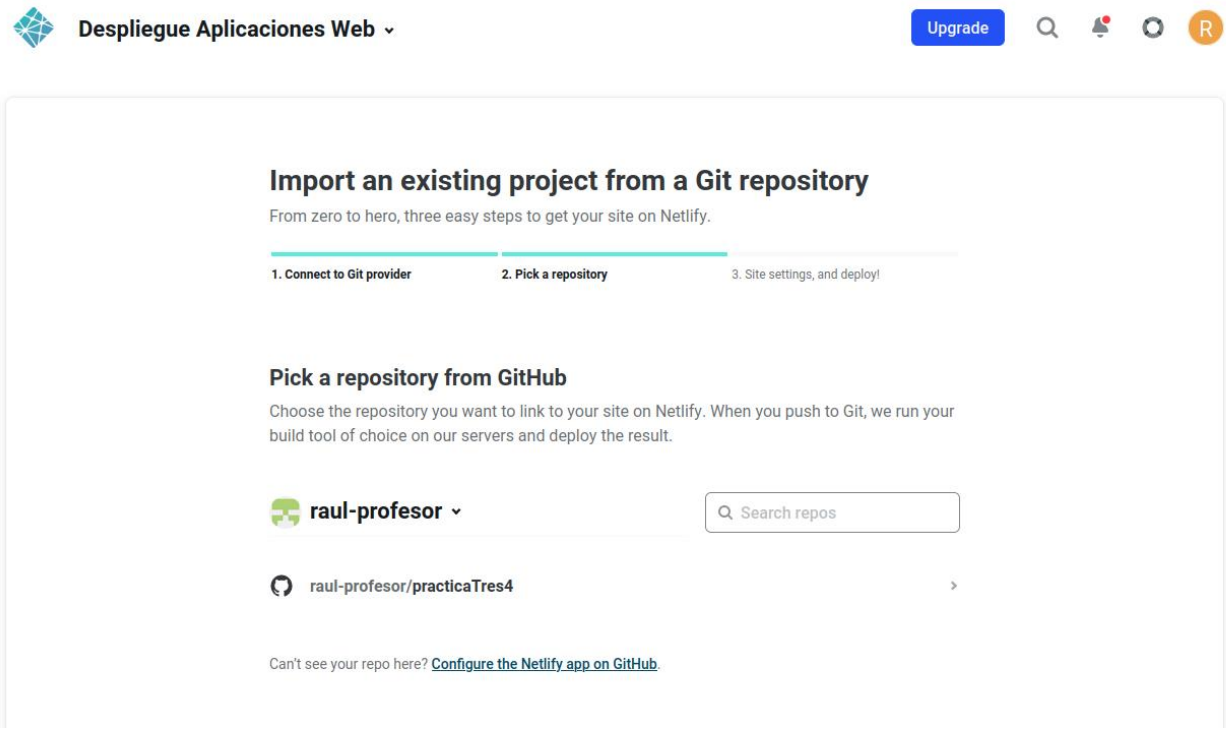
Y nos saltará una ventana pidiendo que autoricemos a Netlify a acceder a nuestros repositorios de Github:



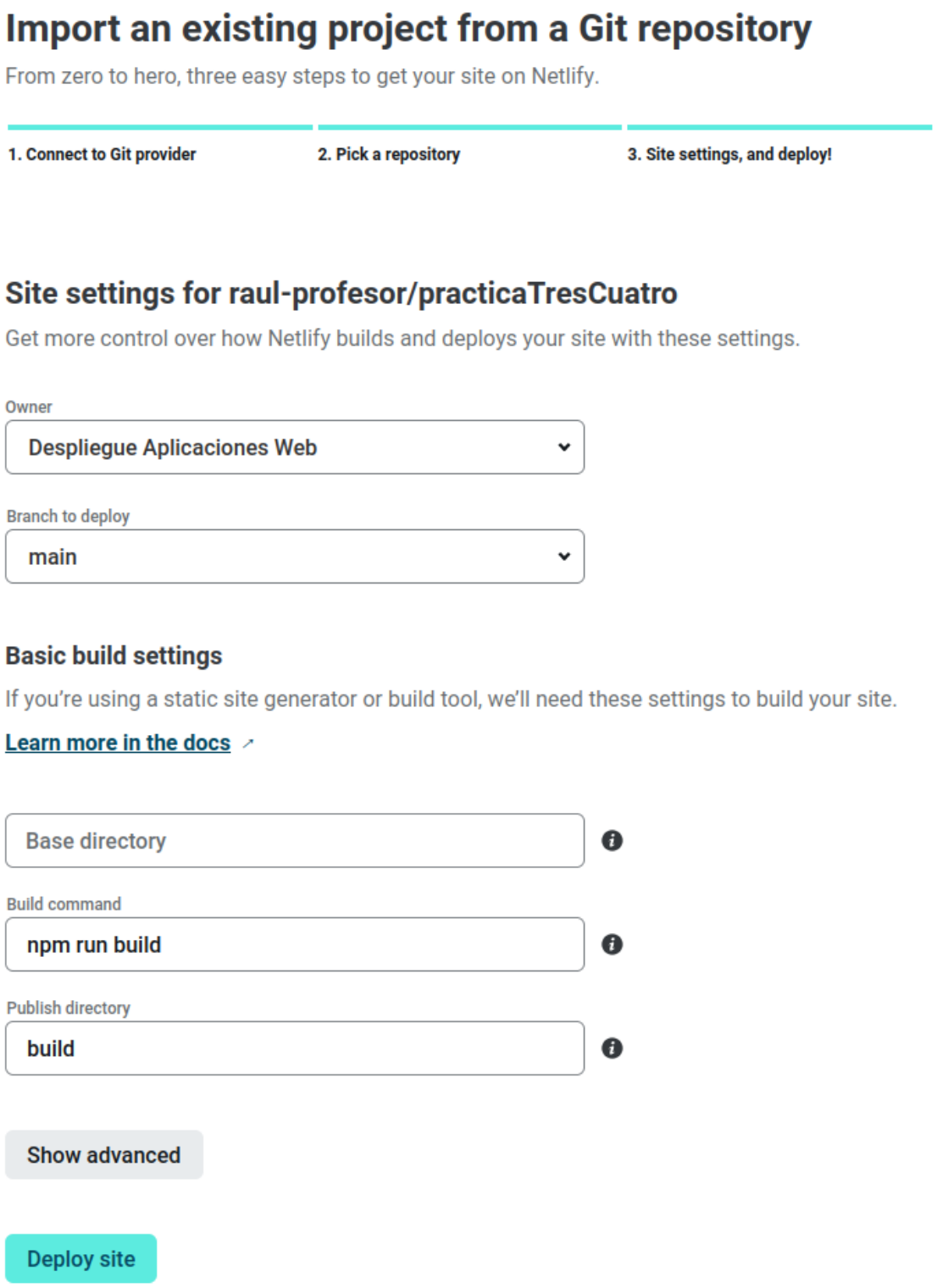
Y luego le indicaremos que no acceda a todos nuestros repositorios sino sólo al repositorio que necesitamos, que es donde tenemos el código de nuestra aplicación:



Y ya quedará todo listo:



Y desplegamos la aplicación:



Netlify se encargará de hacer el `build` de forma automática tal y como hemos visto en la imagen de arriba, con el comando `npm run build`, publicando el contenido del directorio `build`.

Atención

Tras el deploy, en "Site settings" podeís y debéis cambiar el nombre de la aplicación por nombre-practica3-4, donde *nombre* es vuestro nombre.

Lo que hemos conseguido de esta forma es que, cualquier cambio que hagamos en el proyecto y del que hagamos `commit` y `push` en Github, automáticamente genere un nuevo despliegue en Netlify. Es el principio de lo que más adelante veremos como *despliegue continuo*.

Comprobemos que realmente es así:

- Dentro de la carpeta `public` encontramos el archivo `robots.txt`, cuyo cometido es indicar a los rastreadores de los buscadores a qué URLs del sitio pueden acceder. A este archivo se puede acceder a través de la URL del site:

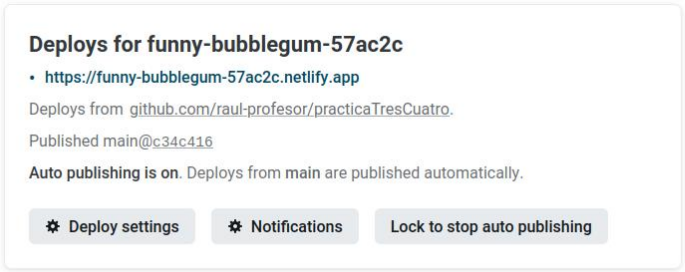

```
# https://www.robotstxt.org/robotstxt.html
User-agent: *
Disallow:
```

- Dentro de la carpeta `public`, utilizando el editor de texto que prefiráis en vuestro terminal, modificad el archivo `robots.txt` para que excluya un directorio que se llame `nombre_apellido`, utilizando obviamente vuestro nombre y apellido.

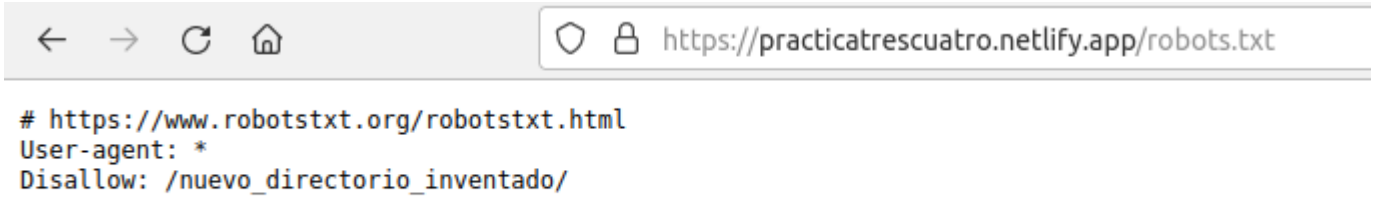
```
User-agent: *
Disallow: /nombre_y_apellido/
```

- Haz un nuevo `commit` y `push` (del caso anterior, recuerda el commando `git` previo para añadir los archivos a hacer commit)
- Comprueba en el dashboard de Netlify que se ha producido un nuevo deploy de la aplicación hace escasos segundos

```
raul@debian-server:~/practicas/practica3.4/color-shades-generator-main$ git push -u origin
Username for 'https://github.com': raul-profesor
Password for 'https://raul-profesor@github.com':
Enumerando objetos: 9, listo.
Contando objetos: 100% (9/9), listo.
Compresión delta usando hasta 2 hilos
Comprimiendo objetos: 100% (5/5), listo.
Escribiendo objetos: 100% (5/5), 454 bytes | 454.00 KiB/s, listo.
Total 5 (delta 4), reusado 0 (delta 0), pack-reusado 0
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To https://github.com/raul-profesor/practicaTresCuatro.git
   3ad464a..c34c416  main -> main
Rama 'main' configurada para hacer seguimiento a la rama remota 'main' de 'origin'.
raul@debian-server:~/practicas/practica3.4/color-shades-generator-main$ date
vie 02 sep 2022 09:09:53 CEST
raul@debian-server:~/practicas/practica3.4/color-shades-generator-main$
```



- Accede a `https://url_de_la_aplicacion/robots.txt` y comprueba que, efectivamente, se ve reflejado el cambio



Cuestiones

1. Investiga y explica que es un Dyno en terminología Heroku.
2. En Heroku no todo es de color de rosa, tiene sus limitaciones y desventajas. Busca, investiga y explica algunas de ellas detalladamente.

Task

Documenta la realización de toda esta práctica adecuadamente, con las explicaciones y justificaciones necesarias y las capturas de pantalla pertinentes.

Referencias

¿Qué es Github?

¿Qué es Heroku?

Deploying Node.js applications

List of all limitations in Heroku platform

How to deploy your website to Netlify for free

A Step-by-Step Guide: Deploying A Static Site or Single-page App

Guide to Deploying a React App to Netlify

Práctica 3.5: Despliegue de una aplicación Flask (Python)

Prerrequisitos

Servidor Debian con los siguientes paquetes instalados:

- Nginx
- Gunicorn
- Pipenv

Introducción

¿Qué es un framework?

Actualmente en el desarrollo moderno de aplicaciones web se utilizan distintos Frameworks que son herramientas que nos dan un esquema de trabajo y una serie de utilidades y funciones que nos facilita y nos abstrae de la construcción de páginas web dinámicas.

En general los Frameworks están asociado a lenguajes de programación (Ruby on Rails (Ruby), Symphony (PHP)), en el mundo de Python el más conocido es Django pero Flask es una opción que quizás no tenga una curva de aprendizaje tan elevada pero nos posibilita la creación de aplicaciones web igual de complejas de las que se pueden crear en Django.

Flask

En la actualidad existen muchas opciones para crear páginas web y muchos lenguajes (PHP, JAVA), y en este caso Flask nos permite crear de una manera muy sencilla aplicaciones web con Python.

Flask es un “micro” Framework escrito en Python y concebido para facilitar el desarrollo de Aplicaciones Web bajo el patrón MVC.

La palabra “micro” no designa a que sea un proyecto pequeño o que nos permita hacer páginas web pequeñas sino que al instalar Flask tenemos las herramientas necesarias para crear una aplicación web funcional pero si se necesita en algún momento una nueva funcionalidad hay un conjunto muy grande extensiones (plugins) que se pueden instalar con Flask que le van dotando de funcionalidad.

De principio en la instalación no se tienen todas las funcionalidades que se pueden necesitar pero de una manera muy sencilla se pueden extender el proyecto con nuevas funcionalidades por medio de plugins.

El patrón MVC es una manera o una forma de trabajar que permite diferenciar y separar lo que es el modelo de datos (los datos que van a tener la App que normalmente están guardados en BD), la vista (página HTML) y el controlador (donde se gestiona las peticiones de la app web).

Gunicorn

Cuando se implementa una aplicación web basada en Python, normalmente se tienen estas tres piezas:

- Servidor web (Nginx, Apache)
- Servidor de aplicaciones WSGI (Gunicorn, uWSGI, mod_wsgi, Waitress)
- Aplicación web (Django, Flask, Pyramid, FastAPI)

Los servidores web procesan y distribuyen las solicitudes de los navegadores y otros clientes y envían respuestas a los mismos.

WSGI (Web Server Gateway Interface) proporciona un conjunto de reglas para estandarizar el comportamiento y la comunicación entre servidores web y aplicaciones web. Mediante el uso de servidores y aplicaciones web compatibles con WSGI, los desarrolladores pueden concentrar su tiempo y energía en el desarrollo de aplicaciones web en lugar de administrar la comunicación entre la aplicación y el servidor web.

Finalmente, Gunicorn, que es la abreviatura de Green Unicorn, es un servidor de aplicaciones WSGI que se encuentra entre el servidor web y su aplicación web, gestionando la comunicación entre los dos. Acepta solicitudes del servidor y las traduce (a través de WSGI) en algo que la aplicación web puede entender antes de pasarla a la aplicación web real. Envía respuestas desde la aplicación web al servidor. También se encarga de ejecutar varias instancias de la aplicación web, reiniciándolas según sea necesario y distribuyendo solicitudes a instancias saludables.

Gestor de paquetes `pip`

`pip` es el comando para instalar paquetes de Python integrados en las fuentes desde la versión 3.4.

Este comando automatiza la conexión al sitio https://pypi.org/, la descarga, la instalación e incluso la compilación del módulo solicitado.

Además, se ocupa de las dependencias de cada paquete.

Entornos virtuales en Python

Un entorno virtual es una forma de tener múltiples instancias paralelas del intérprete de Python, cada una con diferentes conjuntos de paquetes y diferentes configuraciones. Cada entorno virtual contiene una copia independiente del intérprete de Python, incluyendo copias de sus utilidades de soporte.

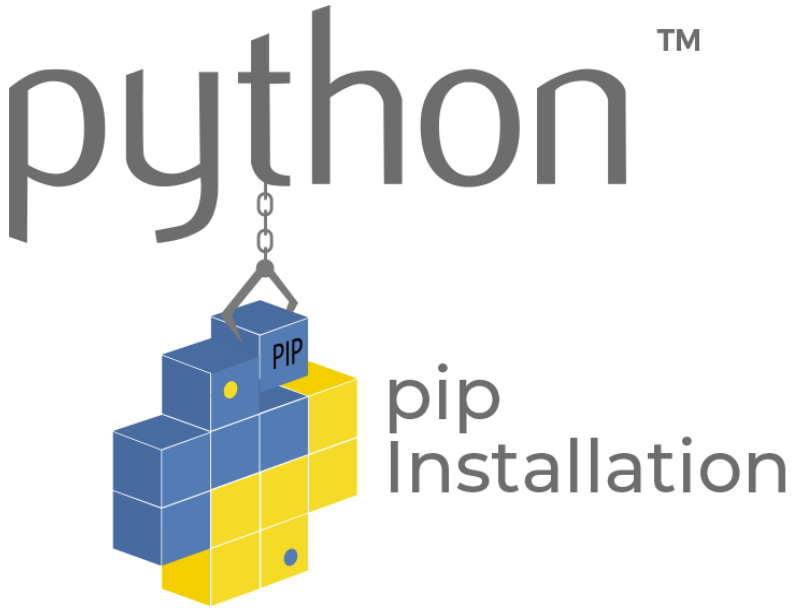
Los paquetes instalados en cada entorno virtual sólo se ven en ese entorno virtual y en ningún otro. Incluso los paquetes grandes y complejos con binarios dependientes de la plataforma pueden ser acorralados entre sí en entornos virtuales.

De esta forma, tendremos entornos independientes entre sí, parecido a como ocurría con los directorios de los proyectos de `Node.js`. De este modo, los entornos virtuales de Python nos permiten instalar un paquete de Python en una ubicación aislada en lugar de instalarlo de manera global.

Pipenv

`Pipenv` es una herramienta que apunta a traer todo lo mejor del mundo de empaquetado (bundler, composer, npm, cargo, yarn, etc.) al mundo de Python.

Automáticamente crea y maneja un entorno virtual para tus proyectos, también permite agregar/eliminar paquetes desde tu Pipfile así como como instalar/desinstalar paquetes. También genera lo más importante , el archivo `Pipfile.lock`, que es usado para producir determinado build.



Procedimiento completo para el despliegue

1. Instalamos el gestor de paquetes de Python pip:

```
sudo apt-get update
```

```
sudo apt-get install python3-pip
```

2. Instalamos el paquete `pipenv` para gestionar los entornos virtuales:

```
pip3 install pipenv
```

3. Y comprobamos que está instalado correctamente mostrando su versión:

```
PATH=$PATH:/home/raul/.local/bin
```

```
pipenv --version
```

4. Creamos el directorio en el que almacenaremos nuestro proyecto:

```
sudo mkdir /var/www/nombre_mi_aplicacion
```

5. Al crearlo con `sudo`, los permisos pertenecen a root:

```
raul-debian@debian-tests:/var/www$ sudo mkdir nuevo_flask
raul-debian@debian-tests:/var/www$ ll | grep nuevo
drwxr-xr-x 2 root        root     4,0K jul 26 21:13 nuevo_flask
raul-debian@debian-tests:/var/www$
```

6. Hay que cambiarlo para que el dueño sea nuestro usuario (`raul-debian` en mi caso) y pertenezca al grupo `www-data`, el usuario usado por defecto por el servidor web para correr:

```
sudo chown -R $USER:www-data /var/www/mi_aplicacion
```

7. Establecemos los permisos adecuados a este directorio, para que pueda ser leído por todo el mundo:

```
chmod -R 775 /var/www/mi_aplicacion
```

Warning

Es **indispensable** asignar estos permisos, de otra forma obtendríamos un error al acceder a la aplicación cuando pongamos en marcha **Nginx**

8. Dentro del directorio de nuestra aplicación, creamos un archivo oculto `.env` que contendrá las variables de entorno necesarias:

```
touch .env
```

9. Editamos el archivo y añadimos las variables, indicando cuál es el archivo `.py` de la aplicación y el entorno, que en nuestro caso será producción:

```
raul-debian@debian-tests:/var/www/ejemplo_flask$ cat .env
       File: .env
   1   FLASK_APP=wsgi.py
   2   FLASK_ENV=production
```

Nota

En el mundo laboral real, se supone que la aplicación previamente ha pasado por los entornos de dev, test y preprod para el desarrollo y prueba de la misma, antes de pasarla a producción.

10. Iniciamos ahora nuestro entorno virtual. `Pipenv` cargará las variables de entorno desde el fichero `.env` de forma automática:

```
pipenv shell
```

Veremos que se nos inicia el entorno virtual, cosa que comprobamos porque aparece su nombre al inicio del prompt del shell:

```
(ejemplo_flask) raul-debian@debian-tests:/var/www/ejemplo_flask$
```

11. Usamos `pipenv` para instalar las dependencias necesarias para nuestro proyecto:

```
pipenv install flask gunicorn
```

12. Vamos ahora a crear la aplicación *Flask* más simple posible, a modo de *PoC* (proof of concept o prueba de concepto). El archivo que contendrá la aplicación propiamente dicha será `application.py` y `wsgi.py` se encargará únicamente de iniciarla y dejarla corriendo:

```
touch application.py wsgi.py
```

Y tras crear los archivos, los editamos para dejarlos así:

```
(ejemplo_flask) raul-debian@debian-tests:/var/www/ejemplo_flask$ cat application.py wsgi.py
```

File: **application.py**

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5
6  @app.route('/')
7  def index():
8      '''Index page route'''
9
10     return '<h1>Aplicacion desplegada</h1>'
```

File: **wsgi.py**

```
1  from application import app
2
3  if __name__ == '__main__':
4      app.run(debug=False)
5
```

13. Corramos ahora nuestra aplicación a modo de comprobación con el servidor web integrado de Flask. Si especificamos la dirección `0.0.0.0` lo que le estamos diciendo al servidor es que escuche en todas sus interfaces, si las tuviera:

```
(ejemplo_flask) raul-debian@debian-tests:/var/www/ejemplo_flask$ flask run --host '0.0.0.0'
 * Tip: There are .env or .flaskenv files present. Do "pip install python-dotenv" to use them.
 * Serving Flask app 'wsgi.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.0.109:5000 (Press CTRL+C to quit)
```

14. Ahora podremos acceder a la aplicación desde nuestro ordenador, nuestra máquina anfitrión, introduciendo en un navegador web: `http://IP-maq-virtual:5000`:

Tras la comprobación, paramos el servidor con `CTRL+C`

15. Comprobemos ahora que Gunicorn funciona correctamente también. Si os ha funcionado el servidor de desarrollo de Flask, podéis usar el siguiente comando para probar que la alicación funciona correctamente usando Gunicorn, accediendo con vuestro navegador de la misma forma que en el paso anterior:

```
gunicorn --workers 4 --bind 0.0.0.0:5000 wsgi:app
```

Donde:

- `--workers N` establece el número de `workers` o hilos que queremos utilizar, como ocurría con Node Express. Dependerá del número de cores que le hayamos dado a la CPU de nuestra máquina virtual.
- `--bind 0.0.0.0:5000` hace que el servidor escuche peticiones por todas sus interfaces de red y en el puerto 5000
- `wsgi:app` es el nombre del archivo con extensión `.py` y `app` es la instancia de la aplicación Flask dentro del archivo.

16. Todavía dentro de nuestro entorno virtual, debemos tomar nota de cual es el path o ruta desde la que se ejecuta `gunicorn` para poder configurar más adelante un servicio del sistema. Podemos averigurarlo así:

```
(ejemplo_flask) raul-debian@debian-tests:/var/www/ejemplo_flask$ which gunicorn
/home/raul-debian/.local/share/virtualenvs/ejemplo_flask-tlSrtaXB/bin/gunicorn
(ejemplo_flask) raul-debian@debian-tests:/var/www/ejemplo_flask$
```

Tip

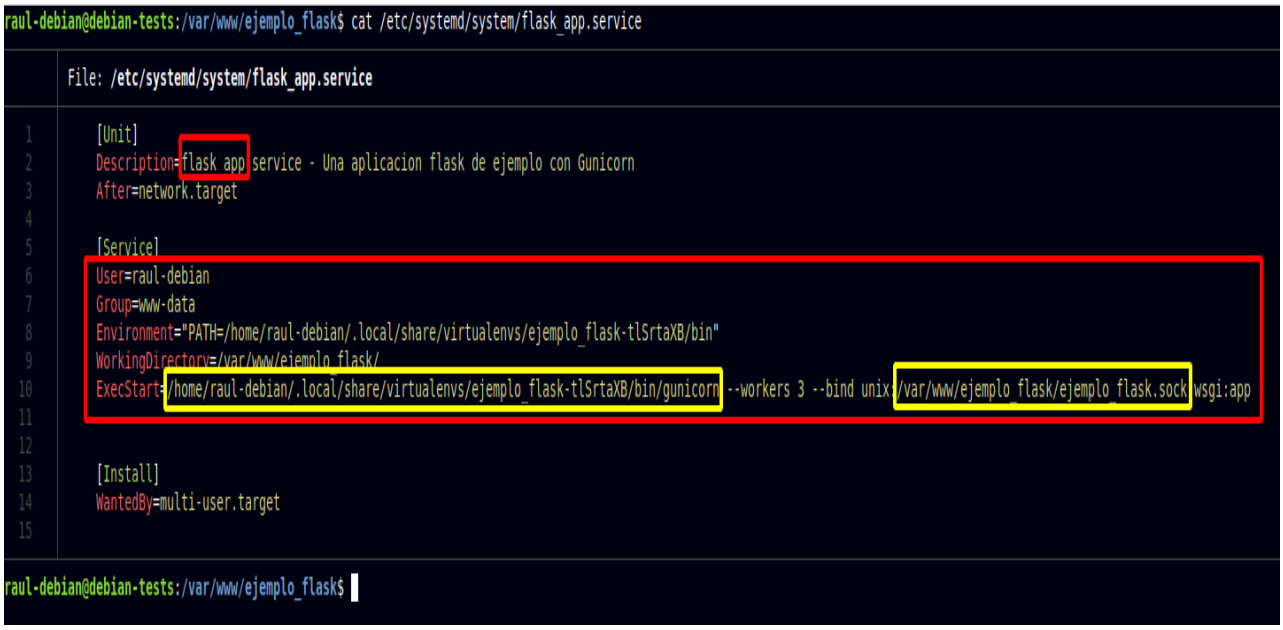
Y tras ello debemos salir de nuestro entorno virtual con el sencillo comando `deactivate`

17. Puesto que ya debemos tener instalado Nginx en nuestro sistema, lo ininciamos y comprobamos que su estado sea activo:

```
sudo systemctl start nginx
```

```
sudo systemctl status nginx
```

18. Ya fuera de nuestro entorno virtual, crearemos un archivo para que systemd corra Gunicorn como un servicio del sistema más:



Donde:

- `User`: Establece el usuario que tiene permisos sobre el directorio del proyecto (el que pusistéis en el paso 5)
- `Group`: Establece el grupo que tiene permisos sobre el directorio del proyecto (el que pusistéis en el paso 5)
- `Environment`: Establece el directorio `bin` (donde se guardan los binarios ejecutables) dentro del entorno virtual (lo vistéis en el paso 14)
- `WorkingDirectory`: Establece el directorio base donde reside nuestro proyecto
- `ExecStart`: Establece el *path* donde se encuentra el ejecutable de `gunicorn` dentro del entorno virtual, así como las opciones y comandos con los que se iniciará

Warning

Debéis cambiar los valores para que coincidan con los de vuestro caso particular.

19. Ahora, como cada vez que se crea un servicio nuevo de `systemd`, se habilita y se inicia:

```
systemctl enable nombre_mi_servicio
```

```
systemctl start nombre_mi_servicio
```

Recordad que el nombre del servicio es el nombre del archivo que creastéis en el paso anterior.

Pasemos ahora a configurar **Nginx**, que es algo que ya deberíamos tener dominado de capítulos anteriores.

20. Creamos un archivo con el nombre de nuestra aplicación y dentro estableceremos la configuración para ese sitio web. El archivo, como recordáis, debe estar en

`/etc/nginx/sites-available/nombre_aplicacion` y tras ello lo editamos para que quede:

```
server {
    listen 80;
    server_name mi_aplicacion www.mi_aplicacion; #



    access_log /var/log/nginx/mi_aplicacion.access.log; #


    error_log /var/log/nginx/mi_aplicacion.error.log;

    location / {
            include proxy_params;
            proxy_pass
http://unix:/var/www/nombre_aplicacion/nombre_aplicacion.sock; #

    20.
    }
}
```

21. Recordemos que ahora debemos crear un link simbólico del archivo de sitios webs disponibles al de sitios web activos:

```
sudo ln -s /etc/nginx/sites-available/nombre_aplicacion /etc/nginx/sites-
enabled/
```

Y nos aseguramos de que se ha creado dicho link simbólico:

```
ls -l /etc/nginx/sites-enabled/ | grep nombre_aplicacion
```

22. Nos aseguramos de que la configuración de Nginx no contiene errores, reiniciamos Nginx y comprobamos que se estado es activo:

```
nginx -t

sudo systemctl restart nginx

sudo systemctl status nginx
```

23. Ya no podremos acceder por IP a nuestra aplicación ya que ahora está siendo servida por Gunicorn y Nginx, necesitamos acceder por su `server_name`. Puesto que aún no hemos tratado con el DNS, vamos a editar el archivo `/etc/hosts` de nuestra máquina anfitriona para que asocie la IP de la máquina virtual, a nuestro `server_name`.

Este archivo, en Linux, está en: `/etc/hosts`

Y en Windows: `C:\Windows\System32\drivers\etc\hosts`

Y deberemos añadirle la línea:

```
192.168.X.X myproject www.myproject
```

donde debéis sustituir la IP por la que tenga vuestra máquina virtual.

24. El último paso es comprobar que todo el desplieuge se ha realizado de forma correcta y está funcionando, para ello accedemos desde nuestra máquina anfitrión a:

```
http://nombre_aplicacion
```

O:

```
http://www.nombre_aplicacion
```

Y debería mostraros la misma página que en el paso 14:

Ejercicio

Repite todo el proceso con la aplicación del siguiente repositorio:
`https://github.com/raul-profesor/Practica-3.5`

Recuerda que deberás clonar el repositorio en tu directorio `/var/www`:

```
git clone https://github.com/raul-profesor/Practica-3.5
```

Y, *tras activar el entorno virtual dentro del directorio del repositorio clonado*, para instalar las dependencias del proyecto de la aplicación deberás hacer:

```
pipenv install -r requirements.txt
```

Y un último detalle, si miráis el código del proyecto, quee es muy sencillo, veréis que *Gunicorn* debe iniciarse ahora así:

```
gunicorn --workers 4 --bind 0.0.0.0:5000 wsgi:app
```

Y el resto sería proceder tal y como hemos hecho en esta práctica.

Warning

Documenta adecuadamente con explicaciones y capturas de pantalla los procesos de despliegue de **ambas** aplicaciones en Flask, así como las respuestas a las cuestiones planteadas.

Cuestiones

Cuestion 1

Busca, lee, entiende y explica qué es y para que sirve un servidor *WSGI*

Tareas de ampliación

Ampliación

Despliega cualquiera de las dos aplicaciones Flask presentadas aquí en Heroku.

Referencias

¿Qué es Flask?

Deploy Flask The Easy Way With Gunicorn and Nginx!

Deploy flask app with Nginx using Gunicorn