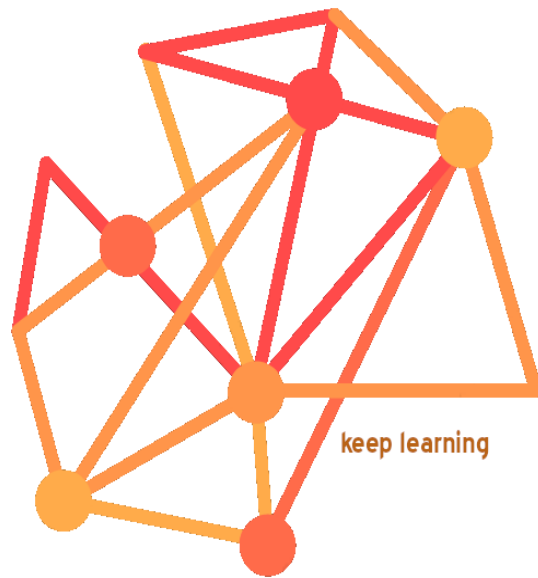


# Node



Juan Carlos Pérez Rodríguez

Juan Carlos Pérez Rodríguez

## Sumario

Introducción.....	5
Primeras apps.....	7
Uso del sistema de ficheros.....	10
Separando la lógica de la aplicación en diferentes ficheros.....	12
Introducción de parámetros por consola en node.....	14
Yargs.....	14
Servidor sin Express.....	16
Typescript.....	18
Anexo: Bases mínimas de Javascript que usaremos.....	19
Desestructuración.....	20
Lambdas: Arrow function.....	21
Asincronía.....	22
callback.....	23
getter y setter.....	23
async await().....	25

Juan Carlos Pérez Rodríguez

# Introducción

Según wikipedia:

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y **basado en el motor V8 de Google**

Posiblemente una definición más entendible es que **Node.js es un entorno de ejecución** para un lenguaje de programación: JavaScript, y permite el uso de frameworks como express. **Haciendo posible ejecutar JavaScript en el lado del servidor**: Permite hacer cosas que javascript no puede como **acceder al sistema de ficheros etc.**

Adicionalmente usando frameworks como express dentro de node se puede gestionar convenientemente peticiones HTTP y tener una aplicación web del lado del servidor plenamente operativa

Por último observar que al estar basado en el motor de google, tenemos el comportamiento de nuestro Javascript similar al que tendríamos en chrome

## Instalación

Descargar desde: <https://nodejs.org/en/> la última LTS para nuestro sistema operativo. Seguir los pasos de la instalación

Como ejemplo se indica una instalación linux ( independiente del sistema de paquetes, ya que no lo usa )

primero descargar un script de instalación:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
```

vemos las versiones disponibles:

```
nvm list-remote
```

vamos a elegir una LTS ( normalmente tomaremos la más reciente )

```
nvm install v14.18.1
```

Si todo sale bien, desde una terminal debe mostrarnos la versión correctamente con el comando:

```
node --version
```

## Actualizar npm

```
npm install -g npm@latest ← instalamos la última versión estable de npm
```

## ¿Qué otro software debemos tener instalado ?

chrome, firefox, visual studio code, git, postman

plugins vscode: bracket pair colorizer 2, typescript importer

## Primeras apps

La ejecución de una aplicación con node es muy sencilla.

Para irnos acostumbrando a la idea de un proyecto, vamos a crear una carpeta que tendrá los ficheros de nuestro proyecto. Por ejemplo: miprimeraapp

Después abrimos esa carpeta con nuestro editor ( por ejemplo visual studio code ) y creamos un fichero: app.js

En el interior lo que ponemos es código javascript. Si ponemos: `console.log()` tendremos el equivalente de los `println()` de Java ( salida de texto en consola )

Así que para hacer un: “hola mundo” basta con que el fichero app.js tenga un `console.log(“hola mundo”);`

Finalmente para ejecutar el programa escribimos en una consola que se haya ubicado en el interior de la carpeta miprimeraapp:

```
node app.js
```

● **Práctica 1:** Crear una aplicación que al ejecutar:  
node app.js  
nos muestre en pantalla nuestro nombre completo

También podemos tener un demonio que se encarga de mostrarnos todos los cambios de nuestro programa cada vez que grabamos nuestro proyecto ( como si nosotros ejecutamos: node app todas las veces que grabamos ) Para ello habría que instalar nodemon:

```
npm install -g nodemon
```

Mediante nodemon podemos tener automáticamente todos los cambios Así que iremos viendo todos los fallos y detalles mientras desarrollamos ( NUNCA lo usamos en producción )

● **Práctica 2:** Hacer uso de nodemon con la app anterior  
nodemon app.js  
y comprobar que mientras editamos el fichero ( agregamos delante de nuestro nombre:  
“nombre del alumno: ” ) y guardamos, automáticamente nos muestra la ejecución de los  
cambios

Vamos a realizar ahora una pequeña aplicación que muestre la tabla de un número 4:



```
Get Started JS app.js ×
JS app.js > getTabla
1  function getTabla( tabla){
2
3      let limite = 10;
4      let respuesta = "";
5      for (let i = 1; i <= limite; i++) {
6          respuesta += `${tabla} * ${i} = ${tabla * i} \n`;
7
8      }
9
10     return respuesta;
11
12 }
13
14
15 console.log(getTabla(4));
```

y lo ejecutamos mediante:

```
node app.js
```

● **Práctica 3:** Crear la aplicación descrita

## Uso del sistema de ficheros

Lo primero que debemos hacer es tener en cuenta donde podemos localizar la información sobre las diferentes librerías que podemos usar:

url de la documentación de node: <https://nodejs.org/dist/latest-v14.x/docs/api/>

La url anterior es para la versión 14. Tomar la que corresponda para la última LTS

En este caso precisamos librerías para el sistema de ficheros. Buscamos File System

En la documentación encontraremos información desde promesas a acciones muy elementales como leer y escribir en fichero

En nuestro caso usaremos:

```
const fs = require('fs');
```

Mediante la instrucción `require` podemos incluir módulos javascript. La instrucción hace que se lea el fichero del módulo solicitado, ejecutándolo y devolviendo lo que el módulo haya establecido como: `exports`

Veamos el código completo:

```
const fs = require('fs');
function getTabla( tabla){

    let limite = 10;
    let respuesta =

    -----
    TABLA DEL ${tabla}
    -----

    ;
    for (let i = 1; i <= limite; i++) {
        respuesta += `${tabla} * ${i} = ${tabla * i} \n`;
    }
    return respuesta;
}
//writeFile() es asíncrono. Así que hay un callback
fs.writeFile(
    'tabla.txt', //nombre del fichero
    getTabla(4), //información a guardar
    (err)=>{ //callback respuesta al finalizar
        if(err)
            throw err;
        else
            console.log("se ha grabado");
    }
);
```

vemos que writeFile() recibe 3 parámetros:

- 1 - nombre del fichero
- 2 – información a guardar
- 3 – callback de respuesta: Aquí pondremos una función que nos permita informar de lo ocurrido. Si error lanzaremos una excepción, si todo bien mostraremos un mensaje de éxito

● **Práctica 4:** Crear la aplicación descrita, pero si hay un error no mostrarlo sino mostrar un mensaje que diga que no se pudo grabar

# Separando la lógica de la aplicación en diferentes ficheros

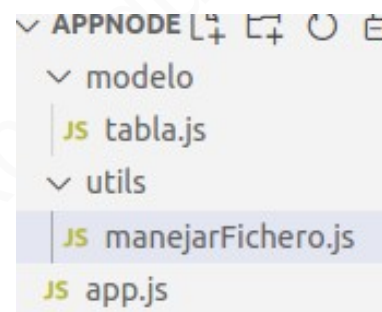
Ya sabemos de la conveniencia de patrones de software como MVC → modelo, vista, controlador. Para poder empezar a conseguir algunas separaciones debemos crear diferentes ficheros y hacer uso del concepto de módulos y la exportación.

Para conseguir lo descrito crearemos diferentes ficheros y lo que queramos que sea accesible desde fuera usaremos: exports.

Vamos a hacer la siguiente separación:

En tabla.js estará la parte de generar la tabla

En manejarFichero.js estará la parte de escribir fichero



app.js nos quedará así:

```
app.js > ...
const {escribir} = require("../utils/manejarFichero");
const {crearTabla} = require("../modelo/tabla");

escribir("tabla.txt",crearTabla(7) )
  .then(console.log("ok grabado"))
  .catch(err=> console.log(err));
```

Observar las llaves en la importación de: {escribir} y en {crearTabla} Se está usando desestructuración. Nos indica que, por ejemplo, en el fichero: modelo/tabla.js se ha exportado un método llamado crearTabla() Si no hubiera un método llamado: crearTabla() exportado en el fichero: tabla.js entonces la desestructuración fallaría

Veamos el  
fichero tabla.js:

```
node > JS tabla.js > [e] crearTabla
1  function getTabla( tabla){
2
3      let limite = 10;
4      let respuesta =
5      -----
6      TABLA DEL ${tabla}
7      -----
8  };
9
10     for (let i = 1; i <= limite; i++) {
11         respuesta += `${tabla} * ${i} = ${tabla * i} \n`;
12     }
13     return respuesta;
14 }
15
16
17
18 exports.crearTabla = getTabla;
```

Observar el uso de: **exports**

Al escribir: `exports.crearTabla` estamos diciendo que puede obtenerse desde otro fichero lo que le hayamos asignado aquí. Y en este caso le hemos asignado el método `getTabla()` por lo tanto desde fuera se puede llamar al método `getTabla()` que tenemos creado aquí mediante: `crearTabla()`

Ahora habría que escribir el código de: `manejarFichero.js` pero para practicar con `async-await` vamos a hacer un cambio. En lugar del método asíncrono: `fs.writeFile()` vamos a usar un método síncrono: `fs.writeFileSync(nombreDelFichero, datosParaGuardar);`

Si usamos el método: `fs.writeFileSync()` no hay una respuesta asíncrona. Eso nos da un problema con lo que está escrito en `app.js`:

```
escribir("tabla.txt",crearTabla(7) )
  .then(console.log("ok grabado"))
  .catch(err=> console.log(err));
```

Vemos que espera un resultado asíncrono del método `escribir` ( espera una promesa ) por eso aparece el método: `then()` y el `catch()` Así que nosotros vamos a tener que “forzar” ese asincronismo haciendo que la función: `escribir()` ( que es la que contendrá `writeFileSync()` ) debe devolver una promesa ( eso se consigue con: `new Promise()` o declarando como `async` el método `escribir()` )

● **Práctica 5:** Crear la aplicación descrita. El fichero: `manejarTabla.js` debe exportar un método: `escribir(nombreDelFichero, textoEscribir)` que tiene que devolver una promesa. En el interior del método `escribir()` se llama a la función: `fs.writeFileSync()`

## Introducción de parámetros por consola en node

`process.argv` es un array con los datos de entrada (parámetros ) de nuestra aplicación ( equivalente a `String[]` de Java por ejemplo )


Así: `node app.js miopcion`

`process.argv[0]` ← ubicación de node

`process.argv[1]` ← ubicación del fichero que estamos ejecutando: `app.js`

`process.argv[2]` ← texto recibido: “miopcion”

Por lo tanto haciendo uso de `process.argv[]` podemos recibir parámetros

 **Práctica 6:** Variar la aplicación que tenemos para grabar una tabla en un fichero para que le pasemos como parámetro el número del que queremos que nos haga la tabla

## Yargs

Como ejemplo de procedimiento de instalación de cualquier paquete vamos a instalar yargs

yargs nos permite el tratamiento de los parámetros cuando tenemos una aplicación node de consola muy cómodo.

Tenemos una lista de todos los paquetes disponibles en:

<https://www.npmjs.com/package/package>

Si allí buscamos yargs y entramos nos muestra la documentación del paquete, etc:

<https://www.npmjs.com/package/yargs>

Instalación: (dentro de la carpeta del proyecto)

```
npm i yargs
```

En la propia documentación nos nombran este ejemplo ( lo incorporamos a nuestra app )

```
const yargs = require('yargs/yargs')
const { hideBin } = require('yargs/helpers')
const argv = yargs(hideBin(process.argv)).argv
```

Para entender mejor el uso de yargs vamos a hacer un console.log() del argv recibido y ejecutamos desde consola:

```
node app parametro1 --parametro2 --parametro3="valor3"
```

Mostrará algo parecido a:

```
{
  _: [ 'parametro1' ],
  parametro2: true,
  parametro3: '"valor3"',
  '$0': 'app'
}
```

Como vemos nos construye las parámetros de una forma elegante. Aquellos que tenían los dos guiones: "--" delante del parámetro los toma como las opciones que le estamos pasando. A estas opciones si no le damos valor nos asimila a que queremos incorporarlo ( boolean: true ) y si le pasamos un valor nos lo asigna al parámetro recibido ( observar que en todos los casos nos ha quitado los dos guiones )

● **Práctica 7:** Variar el ejercicio de la tabla con yargs y nos construya la tabla de un valor ( en el ejemplo vemos el 7 ) mediante:  
node app --tabla=7

## Servidor sin Express

Vamos a crear un servidor web elemental en node sin express:

Comenzaremos creando el package.json de cualquier app node:

`npm init -y`

en `app.js` ponemos:



```
Get Started JS app.js ×
app.js > ...
1
2 const http = require('http');
3 const url = require('url');
4
5 const puerto = 8080;
6 http.createServer( (req, res) =>{
7   res.write(`servidor funcionando en ${puerto} `);
8   const queryObject = url.parse(req.url,true).query;
9   console.log(queryObject);
10  let texto = "";
11  Object.entries(queryObject)
12    .forEach( (par) => texto += (par[0]+ ": " + par[1])
13  res.write(` se recibió como parámetro: ${texto}`);
14  res.end();
15 }).listen(puerto);|
```

y finalmente ejecutamos la app ( se recomienda en todas esta parte el uso de nodemon para ir corrigiendo errores que surjan )

● **Práctica 8:** Crear el ejercicio descrito. Modificar el puerto al 8000 comprobar que arranca correctamente y pasarle diferentes parámetros tomar captura de pantalla de lo obtenido

Observar que response ( la variable `res` ) escribe en el servidor con `write()` y finaliza la salida ( la envía ) con `res.end()`



También vemos que podemos obtener los parámetros que llegan en la request en formato GET ( observar que tomamos la url: req.url )

# Typescript

Por último vamos a ver la api como la haríamos con typescript

Node ha logrado una amplia adopción desde su lanzamiento en 2011. Escribir JavaScript en el servidor puede ser difícil a medida que la base de código crece debido a la naturaleza del lenguaje JavaScript: dinámico y con escritura débil.

Los desarrolladores que llegan a JavaScript desde otros lenguajes a menudo se quejan sobre su falta de escritura estática fuerte, pero aquí es donde entra TypeScript, para cerrar esta brecha.

TypeScript es un superconjunto escrito (opcional) de JavaScript que puede ayudar a la hora de crear y gestionar proyectos JavaScript a gran escala. Puede verse como JavaScript con funciones adicionales como escritura estática fuerte, compilación y programación orientada a objetos.

Lo instalamos mediante:

```
sudo npm install -g typescript
sudo npm install -g eslint
sudo npm install -g eslint@6.4.0 ← esta opción si falla la anterior
```

Ahora en el interior de la carpeta de nuestro proyecto:

```
tsc --init ← para iniciar typescript en el proyecto
npx eslint --init ← para iniciar eslint ( un linter que nos sirve para typescript )
```

## Anexo: Bases mínimas de Javascript que usaremos

### Arrow functions

a nuestros efectos podemos asimilar que es “casi” lo mismo:

```
function saludar(){  
  //aquí el código de la función  
}  
  
const saludar = () => { //aquí el código de la función }
```

Esta última expresión son arrow function y tienen su equivalente en otros lenguajes. Como por ejemplo, en java las lambdas

### Incluir variables dentro de texto (templates):

Para usar formato dentro de salidas/textos generados podemos apoyarnos en el símbolo: “`”

( es la tecla del acento invertido )

```
nombre= “Luis”;  
texto = `Hola amigo ${nombre} es bueno verte por aquí `;
```

Observar que es equivalente a:

```
texto = “Hola amigo “ + nombre + “ es bueno verte por aquí”;
```

Una ventaja es que nos incorpora los saltos de línea:

```
texto = ` primera línea  
esto aparece en la segunda línea  
y finalmente la tercera`;
```

## Declaración de variables

Para declarar variables usamos: let y var. Y para constantes: const

Cuando usamos var estamos en el ámbito de la función que ha sido declarada, esto es, si no ha sido declarada dentro de una función entonces tiene ámbito global

Cuando usamos let tomamos el ámbito de las llaves que lo envuelven

Cuando usamos const pasa similar que con let en cuanto al ámbito

## Desestructuración

Sea el siguiente objeto:

```
const miPersona = {  
  nombre: "Felipe",  
  apellidos: "Perera",  
  getTexto() : {  
    return `${nombre} ${apellidos}`;  
  }  
}
```

En múltiples ocasiones queremos tomar en variables locales parte de la información del objeto.

Por supuesto siempre podemos hacer:

```
let nombre = miPersona.nombre ; let apellidos = miPersona.apellidos;
```

Pero podemos obtener lo mismo mediante **desestructuración**:

```
let { nombre = "Anónimo", apellidos } = miPersona;
```

Observar que adicionalmente, en el caso de que la variable nombre no la encontrara en el objeto le asignaría un valor por defecto a nuestra variable local: `let nombre = "Anónimo";`

Veamos el equivalente en desestructuración con array

```
let miArray = [ "verde", "negro", "gris" ];
```

La forma de tomar los datos sería habitualmente:

```
let dato1 = miArray[1]; let dato2 = miArray[2];
```

Esta última línea sería equivalente a:

```
let [ , dato1, dato2 ] = miArray;
```

Observar que hemos dejado una coma inicial vacía. Eso es porque el dato 0 no queremos tomarlo. Entonces al dejarlo vacío en la desestructuración, el motor sabe que no nos interesa el dato 0 y pasa al siguiente

## Lambdas: Arrow function

Actualmente la forma de trabajar es habitualmente con arrow functions

Así por ejemplo:

```
const concatenar = (a, b = " anónimo" ) => a + b;
```

```
console.log( concatenar( "Armando", "Vettel" );
```

nos mostrará: Armando Vettel

y ya que hemos puesto un valor por defecto llamado anónimo:

```
console.log( concatenar("Armando" ) )
```

mostrará: Armando anónimo

Observamos que `a + b` es el `return` de la función. Cosa que podemos eludir poner cuando únicamente tengamos una única línea en nuestra función. El código anterior es equivalente a:

```
const concatenar = (a, b = " anónimo" ) => {  
    return a + b;  
}
```

Finalmente ¿ cómo hacemos para las funciones que no reciben parámetros ? Pues ponemos paréntesis vacíos

```
const generarAleatorio = ( ) => Math.trunc( 100 * Math.random() );
```

En el ejemplo anterior se genera un número aleatorio de 0 a 99 cuando escribimos:

```
generarAleatorio();
```

## Asincronía

Tomemos el siguiente código:

```
setTimeout(  
    () => {  
        console.log("Este código es el primero en ver node ");  
    }, 3000  
);  
  
setTimeout(  
    () => {  
        console.log("Este código es el segundo en ver node ");  
    }, 0  
);
```

```
console.log("Tercer código que vería node");
```

## callback

Cuando trabajamos con servicios en red muchas veces estamos a expensas de la respuesta de la otra parte. Si ejecutamos de forma síncrona, nos quedamos “bloqueados” esperando a que el otro lado nos responda para poder seguir con nuestro programa. El equivalente más próximo es cuando pedimos al usuario que nos introduzca un dato por teclado. El programa se para y no continúa hasta que el usuario escribe el dato

La asincronía resuelve ese problema. El programa continúa normal y cuando recibe la información que corresponde se desencadena el callback y se ejecuta el código que hayamos previsto para esa situación. Al final lo podemos ver como un equivalente en la programación orientada a eventos con interfaces gráficas. En el momento que el usuario pulsa en un botón se desencadena el evento y se ejecuta el código del método que tenemos asociado. En el caso del callback el evento asociado habitualmente es la respuesta del otro lado de la red

Podemos emularlo mediante: `setTimeout()`

```
setTimeout( ( ) => console.log("estoy en el callback"); 3000 )
```

## getter y setter

Lo siguiente está tomado de la documentación de mozilla:

```
var lenguaje = {  
  set actual(mensaje) {  
    this.log.push(mensaje);  
  },  
};
```

```
    log: []  
}  
lenguaje.actual='ES';  
console.log(lenguaje.log); // ['ES']  
  
lenguaje.actual='FR';  
console.log(lenguaje.log); // ['ES', 'FR']
```

Como vemos podemos establecer una función set en algo que aparenta ser una propiedad desde fuera del objeto. El comportamiento es similar con get



## async await()

Supongamos que necesitamos en una aplicación enviar al cliente los servicios que podemos ofertarle para su dirección. Eso son tres tareas consecutivas:

1. Enviamos el dni del cliente → servidor devuelve datos del cliente ( si existe )
2. con la dirección del cliente obtenida consultamos → servidor responde servicios que podemos dar para esa dirección
3. si hay algún servicio disponible → enviamos un correo publicitario para el cliente de los servicios que puede hacer con nuestra empresa

El proceso anterior requiere de **eventos encadenados** ( tiene que terminar correctamente el primero para pasar al segundo):

- evento servidor responde con datos de usuario para el dni dado
- evento servidor responde con servicios disponibles para la dirección que le damos

Observamos que tiene que tener lugar el primero para poder ejecutar el segundo ( un evento está encadenado a que el anterior haya resuelto )

Adicionalmente, no es positivo dejar bloqueado nuestro programa. Simplemente cuando se provoque el evento de recepción de la información del cliente por parte del servidor, entonces solicitaremos los datos de los servicios disponibles para la dirección del cliente ( asíncronismo )

Nosotros ya sabemos que si hemos puesto un listener, el sistema nos informa en el momento que se desencadena un evento y ejecuta el código que hayamos vinculado a esa situación. Por ejemplo al hacer la consulta del dni, habrá un listener que nos informe cuando el servidor nos haya respondido

Pero la cosa se complica cuando son eventos relacionados. Observar que no tiene ningún sentido consultar por los servicios para una dirección de un cliente si no sabemos a qué cliente nos queremos dirigir.

El sistema nos informa de los eventos de forma independiente y por eso no tenemos una solución para que se den todos los eventos encadenados uno detrás de otro.

Siguiendo el ejemplo, lo que nosotros queremos es tener un control y que si hay éxito en un evento ( y solo si hay éxito ), entonces se desencadene el siguiente:

\*Evento de respuesta a: Solicitar los datos de un cliente por un dni

Si OK => preparar y lanzar:

\*Evento de respuesta a: Solicitar los servicios para la dirección del cliente

Si OK => preparar y lanzar:

Enviar correo con la información

El problema es que queremos encadenar los eventos. QUE OCURRA UNO DETRÁS DE OTRO Y HASTA QUE NO SE RESUELVA EL PRIMERO NO INTENTAR ACOMETER EL SEGUNDO

Adicionalmente al problema anterior está el del aburrimiento del usuario y aprovechar los tiempos ociosos de nuestra aplicación. Mientras está esperando la respuesta del servidor se debiera poder hacer otras cosas en nuestro programa ( es aquí donde entra la asincronía )

¿ Qué significa todo lo anterior ? Pues que esas solicitudes deben ser **asíncronas y encadenadas**

Porque queremos que:

- NO BLOQUEEN: QUE SE SIGAN HACIENDO OTRAS COSAS MIENTRAS ESPERA → **ASÍNCRONO**

- LOS **EVENTOS TENGAN COMPORTAMIENTO SERIAL**: TIENE QUE ACABAR CORRECTAMENTE UNO PARA PODER EJECUTAR EL SIGUIENTE

Aquí es donde entra la combinación de **async-await**.

Con la función **await()** estamos obligados, por su construcción, a que se ejecute en un entorno asíncrono ( dentro de una función que se haya declarado: **async** )

Veamos un ejemplo para poder entender:

```
async function queue(dni){
  try{
    let persona = await getPersona(dni);
    let servicios = await getServicios(persona.direccion);
    let correoEnviado = await enviarCorreo(servicios.toText());
    return correoEnviado;
  } catch(err){
    throw err;
  }
}

queue("42137078")
  .then( msg => console.log( msg ) )
  .catch( err => console.log(err) ) ;
```

`await getPersona()` es una consulta al servidor para un dni dado. Javascript no hará nada más de esta función `queue()` hasta que se dispare un evento que avise cuando el servidor responda. En ese momento se almacena la información devuelta en la variable: `persona`.

Acto seguido se hace otra consulta al servidor: `getServicios()` con los datos de dirección recibidos de la consulta anterior. ES AHORA cuando se está ejecutando este método. NO tiene lugar hasta que se ha obtenido respuesta de la anterior consulta que preguntaba por un dni.

Otra vez al tener puesto un: `await getServicios()` , se va a quedar a la expectativa de que se desencadene un evento que informe que el servidor ya tiene respuesta y en ese caso se rellenará la información en la variable `servicios`

Finalmente tiene lugar la tercera acción: `enviarCorreo()`

Observar del código anterior, que nuestra función `async await` devuelven “promesas” ( por eso se ve la función `.then()` y `.catch()` ) , que se desencadenan si salió bien: `.then()` y si salió mal: `.catch()`