

Primeros pasos con Express



Autor: P. L. Lucas

Bajo licencia Creative Commons: CC BY-NC-SA

<http://creativecommons.org/licenses/by-nc-sa/4.0/>



Por favor, ahorra papel y no imprimas este documento.

Este documento también usa colores para indicar diferentes conceptos.

Si tienes una copia en PDF de este documento, puedes editarlo usando el programa LibreOffice descargando el documento editable desde:

<https://github.com/selairi/primeros-pasos-con-express>

Con la copia editable podrás copiar y pegar, subrayar, introducir tus comentarios o modificaciones para estudiar mejor los textos y ejemplos.



Sumario

1	Primeros pasos con Express.....	5
1.1	Instalación.....	5
1.2	Nuestro primer proyecto en Express.....	5
1.3	Un poco de teoría: peticiones GET y peticiones POST.....	7
1.4	Añadiendo un formulario con una petición GET.....	7
1.5	Peticiones POST.....	10
2	Funciones middleware.....	11
2.1	Conociendo la URL de la ruta visitada.....	14
2.2	Funciones middleware y paquetes en express.....	15
3	Motores de plantillas.....	16
3.1	Motor de plantillas ejs.....	16
3.1.1	Instalando el motor de plantillas ejs.....	16
3.1.2	Plantillas ejs.....	18
4	Route.....	21
4.1	req.params: Obteniendo parámetros de la URL.....	26
4.2	Route y expresiones regulares.....	27
5	Cookies y sesiones.....	27
5.1	Cookies.....	28
5.1.1	Tiempo de expiración de una cookie.....	29
5.1.2	Cookies firmadas.....	29
5.2	Sesiones.....	30
6	Páginas web estáticas.....	34
7	Express generator.....	35
7.1	Un proyecto de ejemplo usando Express generator.....	37
7.1.1	Creando el proyecto.....	37
7.1.2	Usando una hoja de estilos.....	37
7.1.3	Creando las plantillas ejs.....	38
7.1.4	Archivos de permisos y app.js.....	44
7.1.5	routes los archivos de rutas.....	49
8	Conectando con bases de datos.....	53
8.1	SQLite.....	54
8.1.1	SQLite y express.....	59
8.2	MariaDB / MySQL.....	61
8.2.1	Creando una base de datos MariaDB con Docker.....	61
8.2.2	Conexión desde NodeJS con MariaDB.....	64
8.2.3	MariaDB y Express.....	66
8.3	Sugerencia a la hora de conectar a bases de datos desde Express.....	67
9	Variables de entorno.....	67
10	Código HASH de textos.....	69
10.1	Un ejemplo básico de autenticación con express.....	71
11	Tokens.....	73
11.1	Duración de un token y tipos de cifrado.....	75
11.2	Tokens a través de cookies.....	76
11.3	Envío de tokens a través de cabeceras HTTP.....	77



12	Cifrando conexiones con HTTPS.....	81
12.1	Creando un certificado “auto-firmado”.....	82
12.2	Certificados de “Let’s encrypt”.....	83
13	Websockets.....	84
13.1	Express y websockets.....	84
13.2	Websockets en el lado cliente. Javascript en el navegador.....	86
13.3	Una aplicación de chat de ejemplo.....	87
13.4	Conexiones desde otros lenguajes y routers.....	91
13.5	Routers.....	91
13.6	Cifrando conexiones WebSocket con wss.....	93

1 Primeros pasos con Express

Express es un servidor de páginas web para Node.js. Funciona a base de paquetes que se le pueden ir añadiendo para que aumenten sus prestaciones. Se puede encontrar más información en su sitio web:

<http://expressjs.com/>

Para seguir este texto hay que tener nociones de Javascript en cliente y conocimientos básicos de NodeJS.

Importante: Este texto se basa en “aprender haciendo”, por lo que se recomienda copiar y probar los ejemplos que se van proponiendo. También se recomienda copiar los ejemplos a mano para practicar la memoria neuro-muscular.

1.1 Instalación

Lo primero es instalar Node.js. En sistemas operativos Linux es sencillo, sólo hay que ejecutar el correspondiente comando del sistema de paquetes:

```
sudo apt update
sudo apt install nodejs
```

En sistemas operativos Windows, habrá que seguir las instrucciones dadas por la propia Microsoft: <https://docs.microsoft.com/en-us/windows/dev-environment/javascript/nodejs-on-windows>

Una vez instalado “Node.js”, hay que instalar express siguiendo los pasos del siguiente apartado.

1.2 Nuestro primer proyecto en Express

En la sección anterior, habrás instalado “Node.js” en tu sistema. Lo primero que debes hacer es crear una carpeta en la que se guardará el proyecto. Por ejemplo se crea la carpeta “primero”:

```
mkdir primero
cd primero
```

Ahora se inicializa el proyecto:

```
npm init
```

Se contestan a las preguntas del asistente que habrá creado un archivo “package.json”. Como ejercicio se le pide al lector que abra este archivo y vea sus contenidos.



Ahora se van a instalar las dependencias de nuestro proyecto de “Node.js”. La primera dependencia que se va a instalar es express, para ello se ejecuta:

```
npm install express
```

Si se vuelve a mirar ahora el contenido del archivo “package.json”, se verá que se ha añadido express en el apartado dependencias.

Ahora se va a crear un archivo que se va a llamar “app.js” con el siguiente contenido (se recomienda copiar a mano el código para trabajar la memoria neuro-muscular):

Archivo app.js:

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

app.get('/', (req, res) => {
  res.send('Hola mundo')
})

app.listen(8080)
```

Para ejecutarlo se escribirá en el terminal:

```
node app.js
```

y en un navegador se visitará la dirección:

<http://localhost:8080>

Ya se ha escrito nuestro primer servidor de Express.

Vamos a analizar el código de este primer servidor:

```
// Se carga la biblioteca express:
const express = require('express')
// Se crea el objeto que representa al servidor web
const app = express()

// Se van a escuchar las peticiones HTTP del tipo “get” que van dirigidas a la ruta
// “/”, es decir, las que no se pone ruta en la URL:
app.get('/',
  // Se va a ejecutar la siguiente función por cada petición recibida
  (req, res) => {
    // Se manda como respuesta al cliente una página web con el contenido “Hola
    // mundo”:
    res.send('Hola mundo')
  })

// Se arranca el servidor para que escuche en el puerto 8080
app.listen(8080)
```

1.3 Un poco de teoría: peticiones GET y peticiones POST

El protocolo HTTP permite realizar varios tipos de peticiones GET, POST, PUT,... Vamos a centrarnos en las peticiones GET y POST.

Cuando en un navegador se escribe una URL y se pulsa intro, por ejemplo:

<http://localhost:8080>

Se está realizando una petición del tipo GET al servidor. Estas peticiones tienen la característica que son almacenadas en memoria caché por parte de navegadores web y servidores proxy. En desarrollo web nos sucederá muchas veces que es necesario recargar la página varias veces hasta obtener la última versión del servidor y no la copia almacenada en la caché por el navegador.

Esto puede ser muy útil en algunas situaciones, pues se están evitando hacer peticiones al servidor ahorrando recursos. Pero puede ser desastrosa en otras situaciones, pues el cliente puede no estar recibiendo información actualizada del servidor, por ejemplo, en un servicio de subastas online.

Por otro lado se tienen las peticiones POST. Estas no son almacenadas en la caché del navegador, lo cual las hace más adecuadas para recibir información actualizada.

Existe una diferencia más entre las peticiones GET y POST. Sea el siguiente formulario HTML:

```
<form method='get' action='http://localhost:8080/login'>
Usuario: <input type='text' name='usuario'>
Contraseña: <input type='text' name='password'>
<input type='submit' value='Enviar'>
</form>
```

Se puede ver que es una petición GET (method='get') que se va a enviar a la URL

<http://localhost:8080/login>. Supongamos que es la página de acceso a un servicio en el que el usuario introduce su usuario y contraseña. Supongamos que el usuario es “admin” con contraseña “12345”, al hacer clic en el botón de enviar, aparecerá en la URL del navegador lo siguiente:

<http://localhost:8080/login?usuario=admin&password=12345>

Como se puede ver los datos de las peticiones GET se encadenan a la URL siendo visibles para el resto de usuarios. Para colmo se quedan almacenadas en la caché del navegador y proxies.

Importante: Nunca usar peticiones GET para la solicitud de datos sensibles.
--

Las peticiones POST no envían los datos de los formularios vía URL ni se guardan en la caché, por lo que son los más indicados para el envío de datos sensibles.

1.4 Añadiendo un formulario con una petición GET

Se va a añadir un formulario HTML al servidor recién creado. Se modifica el archivo “app.js” a:

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



```
const formulario = ` // ← Es una comilla invertida (tecla a la derecha de la 'p')
<!doctype html>
<meta charset='utf-8'>
<form method='get' action='/saludo'>
Nombre: <input type='text' name='nombre'>
<input type='submit' value='Enviar'>
</form>
`; // ← Es una comilla invertida (tecla a la derecha de la 'p')

app.get('/', (req, res) => {
  res.send(formulario)
})

app.get('/saludo', (req, res) => {
  const nombre = req.query['nombre'];
  const respuesta = `
    <!doctype html>
    <meta charset='utf-8'>
    <p>Hola ${nombre} </p>
  `;
  res.send(respuesta);
})

app.listen(8080)
```

Importante: Nuestro servidor necesita ser parado, pulsando ctrl+c y arrancado de nuevo con “node app.js”, para que se tengan en cuenta los cambios.

Se puede ver que ahora al visitar:

<http://localhost:8080>

Aparece un formulario en el que se pide un nombre. Este formulario es generado con el código:

```
const formulario = `
<!doctype html>
<meta charset='utf-8'>
<form method='get' action='/saludo'>
Nombre: <input type='text' name='nombre'>
<input type='submit' value='Enviar'>
</form>
`;
```

Este texto es un string de Javascript que ocupa varias líneas. Para crear este tipo de cadenas en Javascript se usan las comillas invertidas, las que están a la derecha de la tecla ‘p’ en un teclado en español. Se debe pulsar la comilla invertida y después espacio, pues se trata como un acento.

Más adelante en el código se envía al cliente usando:

```
app.get('/', (req, res) => {
```




```
    res.send(formulario)
  })
```

En esta página web, la línea del formulario:

```
<form method='get' action='/saludo'>
```

Indica que los datos del formulario se van a enviar al servidor usando un método “get” (method=’get’). Se va a enviar a la dirección <http://localhost:8080/saludo> (action=’saludo’).

Por lo tanto, para procesar esta respuesta en el servidor se añade el siguiente método:

```
app.get('/saludo', (req, res) => {
```

Esta línea significa que se está escuchando en “/saludo” las peticiones ‘get’. Cada vez que se reciba una, se ejecutará la función que se pasa como argumento. Vale la pena analizar este método:

```
  app.get('/saludo', (req, res) => {
    const nombre = req.query['nombre'];
    const respuesta = `
      <!doctype html>
      <meta charset='utf-8'>
      <p>Hola ${nombre} </p>
    `;
    res.send(respuesta);
  })
```

como se puede ver con:

```
    const nombre = req.query['nombre'];
```

se obtiene el parámetro que ha introducido en usuario en el formulario. Hay que recordar que en el formulario se tenía dicho parámetro:

```
Nombre: <input type='text' name='nombre'>
```

Se puede deducir que los parámetros de los formularios se pueden obtener con:

```
const parametro = req.query['nombre parametro'];
```

Después se construye la respuesta que se va a mandar al cliente en el que se incluye el valor del parámetro que se ha pasado (que se ha guardado en la variable **nombre**):

```
const respuesta = `
  <!doctype html>
  <meta charset='utf-8'>
  <p>Hola ${nombre} </p>
`;
res.send(respuesta);
```

Nota: Los string multilínea de Javascript permiten incluir valores de variables definidas previamente. Dichas variables serán convertidas a tipo string. Para incluirlas sólo hay que escribir “\${nombre de la

variable}”. En nuestro caso sería lo mismo que:

```
const respuesta =  
  "<!doctype html>" +  
  "<meta charset='utf-8'>" +  
  "<p>Hola "+ nombre +" </p>"  
;
```

1.5 Peticiones POST

En el ejemplo anterior se ha hecho un formulario que hacía una petición GET, se puede hacer una petición POST cambiando get por post y añadir el siguiente código para que procese la petición:

```
app.use(express.urlencoded({  
  extended: true  
}))
```

Por lo que el código final sería:

```
const express = require('express')  
const app = express()  
  
// El siguiente método se debe poner ANTES de definir el método post:  
app.use(express.urlencoded({  
  extended: true  
}))  
  
const formulario = `  
<!doctype html>  
<meta charset='utf-8'>  
<form method='post' action='/saludo'>  
Nombre: <input type='text' name='nombre'>  
<input type='submit' value='Enviar'>  
</form>  
`;  
  
app.get('/', (req, res) => {  
  res.send(formulario)  
})  
  
app.post('/saludo', (req, res) => {  
  const nombre = req.body['nombre'];  
  const respuesta = `  
    <!doctype html>  
    <meta charset='utf-8'>  
    <p>Hola ${nombre} </p>  
  `;  
  res.send(respuesta);  
})  
  
app.listen(8080)
```

Hay que hacer notar que también se cambiado req.**query**['nombre del parámetro'] por req.**body**['nombre del parámetro'].

2 Funciones middleware

En el ejemplo anterior se ha usado el método “app.use(función)”. Este método sirve para indicar a express que debe ejecutar una función antes de procesar los métodos GET o POST.

Las funciones middleware tienen la forma:

```
(res, req, next) => {  
  // Cuerpo de la función  
}
```

Como se puede ver admiten 3 argumentos. “res” y “req” ya son conocidos, pero “next” es un función que indica a express si se debe ejecutar la siguiente función middleware.

Por ejemplo:

```
const express = require('express')  
const app = express()  
  
app.use((req, res, next) => {  
  console.log("Middleware 1");  
  next();  
})  
  
app.use((req, res, next) => {  
  console.log("Middleware 2");  
  next();  
})  
  
app.get('/', (req, res) => {  
  res.send('Página inicial')  
})  
  
app.get('/hola', (req, res) => {  
  res.send('Hola mundo');  
})  
  
app.listen(8080)
```

Si se mira la consola cuando se visitan las páginas “<http://localhost:8080>” o “<http://localhost:8080/hola>”, se mostrarán los mensajes:

```
Middleware 1  
Middleware 2  
Middleware 1  
Middleware 2
```

Como el lector ya habrá podido deducir las funciones middleware se van ejecutando en el orden en el que han sido introducidas.

En el caso de que se comente la línea con “next()” en la función con el “middleware 1” se verá que no se llega a ejecutar la función con el “middleware 2”.

Si se le pasa un string como argumento al método “next()”, no se seguirá ejecutando las siguientes funciones middleware y se mostrará por el navegador el mensaje introducido en la cadena que se pasa como argumento a next. Por ejemplo:

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

app.use((req, res, next) => {
  console.log("Middleware 1");
  // Se le pasa un argumento a la función next:
  next('Error');
})

// Esta función no llega a ejecutarse
app.use((req, res, next) => {
  console.log("Middleware 2");
  next();
})

app.get('/', (req, res) => {
  res.send('Página inicial')
})

app.get('/hola', (req, res) => {
  res.send('Hola mundo');
})

app.listen(8080)
```

Se ejecutará la función “middleware 1”, pero no se ejecutarán más funciones y se mostrará en el navegador el mensaje “Error”, que es lo que se ha introducido como argumento en la función next.

Esta propiedad de parar la ejecución de las funciones middleware pasando un argumento a la función “next()” se usará, por ejemplo, para verificar si un usuario está autenticado en un servidor o añadir parámetros al argumento req. Por ejemplo, para añadir parámetros al argumento req, se haría:

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

app.use((req, res, next) => {
  console.log("Middleware 1");
  // Se añade un nuevo parámetro a req:
  req.parametro = "Nuevo";
  next();
})

app.use((req, res, next) => {
  console.log("Middleware 2");
  next();
})
```

```

app.get('/', (req, res) => {
  // Se consulta el parámetro que se ha añadido:
  console.log(req.parametro)
  res.send('Página inicial')
})

app.get('/hola', (req, res) => {
  res.send('Hola mundo');
})

app.listen(8080)

```

Como se puede ver cualquier función middleware puede añadir parámetros al argumento req simplemente con:

```
req.nombre_parámetro = valor
```

Se pueden indicar las rutas en las que se ejecutarán las funciones middleware. Para ello se pasará un primer argumento al método app.use() para indicar la ruta:

```
app.use(ruta, función)
```

Por ejemplo:

```

app.use('/hola', (req, res, next) => {
  console.log("Middleware 3");
  next();
})

```

Se ejecutaría cada vez que se visitase “<http://localhost:8080/hola>”.

Se deja como ejercicio al lector averiguar el orden en el que se ejecutan las funciones middleware y cuáles son cuando se corre el siguiente código y se visitan las URL “<http://localhost:8080/>” y “<http://localhost:8080/hola>”:

```

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

app.use((req, res, next) => {
  console.log("Middleware 1");
  next();
})

app.use((req, res, next) => {
  console.log("Middleware 2");
  next();
})

app.use('/hola', (req, res, next) => {
  console.log("Middleware 3");
  next();
})

app.use('/hola', (req, res, next) => {

```

```

    console.log("Middleware 4");
    next();
  })

  app.get('/', (req, res) => {
    res.send('Página inicial')
  })

  app.get('/hola', (req, res) => {
    res.send('Hola mundo');
  })

  app.listen(8080)

```

2.1 Conociendo la URL de la ruta visitada

Con “req.url” se puede saber la ruta que se está visitando. Esto es útil para que una función middleware se comporte de forma diferente cuando se está visitando por una URL determinada. Por ejemplo, tratar de averiguar la salida por consola del siguiente código:

```

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

app.use((req, res, next) => {
  console.log("Middleware 1");
  console.log(req.url)
  if(req.url === '/hola') {
    console.log('Se hacen unas cosas para la ruta "/hola"')
  } else {
    console.log('Se hacen otras cosas cuando no se visita la ruta "/hola"')
  }
  next();
})

app.use((req, res, next) => {
  console.log("Middleware 2");
  next();
})

app.get('/', (req, res) => {
  res.send('Página inicial')
})

app.get('/hola', (req, res) => {
  res.send('Hola mundo');
})

app.listen(8080)

```

2.2 Funciones middleware y paquetes en express

Como ya se ha podido deducir por el título de esta sección, se usarán funciones middleware para cargar bibliotecas en express que cambien su funcionamiento, aumenten su rendimiento, permitan gestionar la seguridad,...

Por ejemplo, la biblioteca “morgan” permite que express muestre por consola mensajes de log que pueden ser útiles para averiguar las peticiones que se están recibiendo.

Para instalar la biblioteca habrá que ir a la carpeta en la que está nuestro proyecto con express y ejecutar:

```
npm install morgan
```

Después se modificará nuestro proyecto para que cargue dicha biblioteca y se le pasará la función middleware que sea usada por express. Por ejemplo:

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

// Se carga la biblioteca morgan:
const logger = require('morgan')

// Se indica a express que use morgan mediante una función middleware:
app.use(logger('common'))

app.get('/', (req, res) => {
  res.send('Página inicial')
})

app.get('/hola', (req, res) => {
  res.send('Hola mundo');
})

app.listen(8080)
```

Si se ejecuta y se visitan las URL “<http://localhost:8080>” y “<http://localhost:8080/hola>”, se mostrarán por la consola mensajes del tipo:

```
::ffff:127.0.0.1 - - [07/Jul/2022:16:34:36 +0000] "GET /hola HTTP/1.1" 304 -
::ffff:127.0.0.1 - - [07/Jul/2022:16:34:42 +0000] "GET / HTTP/1.1" 304 -
```

Habrà que visitar la página web de la biblioteca en cuestión para averiguar cómo configurarla y qué opciones tiene. Por ejemplo, para la biblioteca morgan habrá que leer su documentación en:

<https://github.com/expressjs/morgan>

3 Motores de plantillas

Entre las muchas capacidades de Express está la de poder añadir motores de plantillas. Se pueden crear plantillas para crear nuestras páginas web. Las plantillas se pueden insertar unas dentro de otras para poder construir nuestra página.

Hay muchos motores de plantillas como ejs o jade. Se recomienda usar ejs por su sencillez y el alto rendimiento que proporciona.

3.1 Motor de plantillas ejs

Se puede encontrar su documentación en la página:

<https://ejs.co/>

Se va a dar primero una introducción a las plantillas ejs para después integrarlas en nuestro servidor express.

3.1.1 Instalando el motor de plantillas ejs

Para instalar el motor de plantillas, habrá que ir a la carpeta en la que está nuestro proyecto y escribir:

```
npm install ejs
```

Otra cosa que se debe hacer es crear una carpeta en la que se van a guardar las plantillas. Normalmente a esta carpeta se la denominará “views”:

```
mkdir views
```

Una vez instalado nuestro motor de plantillas, se configura nuestro servidor, para ello se añadirán las siguientes líneas:

```
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')
```

Con las líneas:

```
const ejs = require('ejs')
const path = require('path')
```

Simplemente se le está indicando a nodejs que cargue las bibliotecas ejs y path.

Ahora se le indica a express que la carpeta que se va a usar para guardar las plantillas se llama views:

```
app.set('views', path.join(__dirname, 'views'))
```

Se podría poner directamente:

```
app.set('views', 'Ruta absoluta a la carpeta views')
```


Pero si se copia el proyecto a otra carpeta, esta ruta dejará de ser válida, por lo que en lugar de la ruta completa se suele escribir:

```
path.join(__dirname, 'nombre carpeta')
```

que le indica a Express que en la carpeta actual (__dirname) se busque la carpeta 'nombre carpeta'.

Por último queda indicar el motor de plantillas, por ello se escribe:

```
app.set('view engine', 'ejs')
```

A partir de este momento se introducirán las plantillas en la carpeta views y se mostrarán con:

```
res.render('nombre plantilla')
```

Por ejemplo, supongamos que se han creado las plantillas:

Archivo views/pagina1.ejs:

```
<!doctype html>
<html>
<head>
<meta charset='utf-8' >
<title>Página 1</title>
</head>
<body>
Página 1
</body>
</html>
```

Archivo views/pagina2.ejs:

```
<!doctype html>
<html>
<head>
<meta charset='utf-8' >
<title>Página 2</title>
</head>
<body>
Página 2
</body>
</html>
```

Nuestro servidor podría ser ahora:

Archivo app.js:

```
const express = require('express')
const app = express()
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

app.get('/', (req, res) => {
```

```

    res.render('pagina1')
  })
  app.get('/hola', (req, res) => {
    res.render('pagina2');
  })
  app.listen(8080)

```

Ahora cuando se visiten las páginas <http://localhost:8080> y <http://localhost:8080/hola> se mostrarán respectivamente las plantillas “pagina1.ejs” y “pagina2.ejs”

3.1.2 Plantillas ejs

Las plantillas ejs consisten en archivos html en los que con códigos especiales se pueden insertar valores de variables definidas en el javascript de nuestra aplicación, incluir plantillas unas dentro de otras o incluso usar javascript para escribir código de la página.

Las plantillas ejs son archivos que tienen la extensión “.ejs”.

Se comenzará con un ejemplo sencillo:

Se van a definir las siguientes plantillas:

Archivo views/cabecera.ejs:

```

<!doctype html>
<html>
<head>
<meta charset='utf-8' >
<title>Página 1</title>
</head>
<body>

```

Archivo views/pie.ejs:

```

</body>
</html>

```

Se pueden incluir plantillas unas dentro de otras con:

```

<% include('nombre plantilla') %>

```

Por ejemplo se puede modificar:

Archivo views/pagina1.ejs:

```

<%- include('cabecera') -%>
Página 1
<%- include('pie') -%>

```

Se recomienda al lector que pruebe a ver los resultados.

Pero hay un problema, se podría hacer lo mismo con “pagina2.ejs”, pero el título (“<title>Página 1</title>”) no coincidiría. Para ello se hace la siguiente modificación:

Archivo views/cabecera.ejs:

```
<!doctype html>
<html>
<head>
<meta charset='utf-8' >
<title><%= titulo %></title>
</head>
<body>
```

Se pueden definir variables en las plantillas y luego escribir su valor. El código:

```
<%= titulo %>
```

está solicitando escribir el valor de la variable título.

Se pueden definir variables en el “include”:

```
<% include('nombre plantilla', {json con variables}) %>
```

Por ejemplo:

```
<%- include('cabecera', {titulo: 'Página 2'}) -%>
```

Se ha definido la variable “titulo” con el valor “Pagina 2”.

Ahora se pueden modificar las plantillas a:

Archivo views/pagina1.ejs:

```
<%- include('cabecera', {titulo: 'Página 1'}) -%>
Página 1
<%- include('pie') -%>
```

Archivo views/pagina2.ejs:

```
<%- include('cabecera', {titulo: 'Página 2'}) -%>
Página 2
<%- include('pie') -%>
```

y se respetará el título de cada página.

También se pueden definir variables en el código Javascript y pasar sus valores a las plantillas. Por ejemplo:

Se modifica views/pagina1.ejs para incluir un nuevo valor:

```
<%- include('cabecera', {titulo: 'Página 1'}) -%>
Página 1. <br>
<%= mensaje %>
<%- include('pie') -%>
```

Se modifica app.js para pasar un valor a “mensaje”:

```
const express = require('express')
```

```

const app = express()
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

app.get('/', (req, res) => {
  const texto = "Hola mundo";
  res.render('pagina1', {mensaje: texto})
})

app.get('/hola', (req, res) => {
  res.render('pagina2');
})

app.listen(8080)

```

Como se puede ver se pueden pasar los valores con un argumento más en el método render al igual que se hace en include.

Pero, ¿y si se quisiera mostrar un array de elementos en una lista?

Se va a modificar app.js de la siguiente forma:

```

const express = require('express')
const app = express()
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

app.get('/', (req, res) => {
  const texto = "Hola mundo";
  const ciudades = ['Ávila', 'Barcelona', 'Madrid', 'Valencia'];
  res.render('pagina1', {mensaje: texto, listado: ciudades})
})

app.get('/hola', (req, res) => {
  res.render('pagina2');
})

app.listen(8080)

```

Se desea mostrar el array ciudades en una lista. Con <% %> se pueden introducir comandos en Javascript que ayudarán a crear el código HTML. Por ejemplo:

Archivo views/pagina1.ejs:

```

<%- include('cabecera', {titulo: 'Página 1'}) -%>

```



```
Página 1. <br>
<%= mensaje %><br>
<ol>
<%
    let valor = null;
    for(let n = 0; n < listado.length; n++) {
        valor = listado[n];
    %>
        <li><%= valor %></li>
<% } %>
</ol>
<%- include('pie') -%>
```

El código marcado en rojo, ejes lo podría traducir a:

```
<%- include('cabecera', {titulo: 'Página 1'}) -%>
Página 1. <br>
<%= mensaje %><br>
<ol>
<script>
    let valor = null;
    for(let n = 0; n < listado.length; n++) {
        valor = listado[n];
        document.write("<li>" + valor + "</li>");
    }
</script>
</ol>
<%- include('pie') -%>
```

La variable valor no es necesaria en ningún caso, sólo se ha puesto como ejemplo de que se pueden introducir varias líneas de código en un <% %>.

4 Route

Hasta ahora se han usado los métodos `app.use()`, `app.get()` y `app.post()` para hacer que se ejecutasen funciones cuando se reciba una petición en una determinada URL:

```
app.get('/hola', (req, res) => {res.send("Hola mundo")})
```

Como se puede ver en la línea anterior esta función se ejecutará cuando se visite la ruta “/hola”.

Hay otra forma de asociar funciones a rutas, que se suele usar para separar las funciones en distintos archivos de forma que el código quede más legible.

Para practicar se va a crear un proyecto nuevo desde cero, para ello se crea una carpeta con el nombre del proyecto:

```
mkdir rutas
cd rutas
```

Ahora se inicializa el proyecto:

```
npm init
```

Se instalan las dependencias, en este caso sólo se va a instalar ejs:

```
npm install ejs
```

Se crea la carpeta views para introducir las plantillas:

```
mkdir views
```

En la carpeta views se van a introducir las siguientes plantillas:

Archivo views/cabecera.ejs:

```
<!doctype html>
<html>
<head>
<meta charset='utf-8' >
<title><%= titulo %></title>
</head>
<body>
```

Archivo views/pie.ejs:

```
</body>
</html>
```

Archivo views/pagina.ejs:

```
<%- include('cabecera') -%>
<h1><%= titulo %></h1>
<%- include('pie') -%>
```

Por último se crea el servidor:

Archivo app.js:

```
const express = require('express')
const app = express()
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

let routerPagina2 = express.Router()
routerPagina2.get('/', (req, res) => {
  res.render('pagina', {titulo: "Página 2"})
})
app.use('/pagina2', routerPagina2)

app.listen(8080)
```

Cuando se visita la URL <http://localhost:8080/pagina2> se muestra una página web con el título “Página 2”.

En lugar del método `app.get()` se ha usado la clase `express.Router()`:

```
let routerPagina2 = express.Router()
```

Esta clase sirve para crear “rutas” a las que se van a asociar funciones.

En las siguientes líneas se asocia un método `get` a la ruta `‘/’`:

```
routerPagina2.get('/', (req, res) => {  
  res.render('pagina', {titulo: "Página 2"})  
})
```

Por último se dice que use esa ruta recién creada en `“/pagina”`:

```
app.use('/pagina2', routerPagina2)
```

Esto puede resultar chocante, en el objeto `routerPagina2` se ha pasado la ruta `“/”`

(`routerPagina2.get(‘/’, ...)`) y en `app.use()` se ha pasado `“/pagina2”` (`app.use(“/pagina2”, ...)`). Lo que está haciendo `express` es concatenar las rutas: `“/pagina2” + “/”`

Para comprobarlo, se puede hacer el siguiente cambio:

```
const express = require('express')  
const app = express()  
const ejs = require('ejs')  
const path = require('path')  
  
app.set('views', path.join(__dirname, 'views'))  
app.set('view engine', 'ejs')  
  
let routerPagina2 = express.Router()  
routerPagina2.get('/pagina', (req, res) => {  
  res.render('pagina', {titulo: "Página 2"})  
})  
app.use('/pagina2', routerPagina2)  
  
app.listen(8080)
```

Ahora cuando se visita la URL <http://localhost:8080/pagina2> se genera un error. Cuando se visita <http://localhost:8080/pagina2/pagina>, se muestra una página web con el título “Página 2”.

¿Cómo se usan los objetos `express.Route()`? Sirven para dividir de forma coherente nuestro proyecto en varios archivos. En los ejemplos que se están usando en este manual, las funciones `middleware` apenas ocupan unas líneas, pero en un sitio web real serán enormes y se visitarán cientos de rutas.

Lo primero que se va a hacer es separar nuestras rutas en archivos. Habitualmente las rutas se suelen poner en la carpeta `“routes”`, por lo que se va a crear una nueva carpeta en el proyecto:

```
$ mkdir routes
```

Ahora se va a modificar el proyecto de la siguiente forma:

Se va a añadir `routes/pagina2.js`:



```
var express = require('express');

let router = express.Router()
router.get('/', (req, res) => {
  res.render('pagina', {titulo: "Página 2"})
})

module.exports = router;
```

Importante: Hay que darse cuenta de que se ha vuelto a poner como ruta “/”.

Con la línea:

```
module.exports = router;
```

Se le está indicando a Node.js que se está exportando el objeto router de este archivo, para que pueda ser incluido en otro archivo usando require.

Ahora se modifica de nuevo nuestro servidor:

Archivo app.js:

```
const express = require('express')
const app = express()
const ejs = require('ejs')
const path = require('path')

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

let routerPagina2 = require("./routes/pagina2")
app.use('/pagina2', routerPagina2)

app.listen(8080)
```

Como se puede ver se ha incluido el objeto router del archivo “routes/pagina2.ejs”:

```
let routerPagina2 = require("./routes/pagina2")
```

y después se ha añadido a la ruta “/pagina2”:

```
app.use('/pagina2', routerPagina2)
```

A partir de ahora se separarán las diversas rutas en archivos y se incluirán en el archivo app.js.

Además del método GET, se pueden escuchar peticiones del método POST:

Para ello se modifica el archivo:

Archivo routes/pagina2.js:

```
var express = require('express');

let router = express.Router()
router.get('/', (req, res) => {
  res.render('pagina', {titulo: "Página 2"})
})
```



```
router.post('/', (req, res) => {
  res.render('pagina', {titulo: "Página 2 en petición POST"})
})
```

module.exports = router;

También se va a modificar la plantilla para tener un formulario que solicite una petición post a la URL <http://localhost:8080/pagina2>:

Archivo views/pagina.ejs:

```
<%- include('cabecera') -%>
<h1><%= titulo %></h1>
<form method='post' action='/pagina2'>
  <input type='text' name='texto'>
  <input type='submit' value='Enviar'>
</form>
<%- include('pie') -%>
```

Ahora se pueden hacer pruebas de petición POST.

Ejercicio: Se recomienda repasar las peticiones GET y POST que se han explicado en este documento y añadir un campo muestre el valor que pasa el usuario en el campo texto del formulario.

Como pequeña práctica se va a añadir una página de inicio a nuestro servidor:

Se añade una nueva plantilla para esta página:

Archivo views/inicio.ejs:

```
<%- include('cabecera', {titulo: "Inicio"}) -%>
<h1>Página de inicio</h1>
<p>Se puede visitar <a href='/pagina2'>Página 2</a></p>
<%- include('pie') -%>
```

Se añade también un nuevo archivo en routes:

Archivo routes/inicio.js:

```
var express = require('express');

let router = express.Router()
router.get('/', (req, res) => {
  res.render('inicio')
})
```

module.exports = router;

Finalmente se modifica nuestro servidor para que acepte esta nueva ruta:

Archivo app.js:

```
const express = require('express')
const app = express()
const ejs = require('ejs')
const path = require('path')
```

```

app.set('views', path.join(__dirname, 'views'))
app.set('view engine', 'ejs')

let routerPagina2 = require("./routes/pagina2")
let routerInicio = require("./routes/inicio")

app.use('/', routerInicio)
app.use('/pagina2', routerPagina2)

app.listen(8080)

```

Ahora ya se puede visitar la página <http://localhost:8080/>.

4.1 req.params: Obteniendo parámetros de la URL

Se pueden introducir campos al definir la route y obtener los valores que el usuario haya introducido en dichos parámetros. Por ejemplo:

Archivo params1.js:

```

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

app.get('/user/:usuario', (req, res) => {
  res.send('Hola ' + req.params.usuario)
})

app.listen(8080)

```

Si se visita la página:

<http://localhost:8080/user/pepe>

Se obtendrá como respuesta:

Hola pepe

Como se puede ver en la ruta se ha introducido:

`/user/:usuario`

En la ruta hay un campo llamado “:usuario”. El campo se introduce poniendo “:” + “identificador”.

Cuando en la URL se introduce:

<http://localhost:8080/user/juan>

express intenta encontrar el parámetro “:usuario” que en este caso es “juan”.

¿Se pueden capturar varios parámetros? Sí. Por ejemplo:

Archivo params2.js:

```

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

```



```
app.get('/carpeta1/:campo1/carpeta2/:campo2', (req, res) => {  
  res.send(req.params.campo1 + ' ' + req.params.campo2)  
})
```

```
app.listen(8080)
```

Ahora se tienen los parámetros “campo1” y “campo2” a capturar. Si se introduce la URL:

<http://localhost:8080/carpeta1/hola/carpeta2/mundo>

La salida será:

hola mundo

4.2 Route y expresiones regulares

Al indicar las rutas se pueden usar expresiones regulares. Una expresión regular es un lenguaje que permite dar unas reglas para identificar cadenas de texto. Por ejemplo, si se desean localizar todas las cadenas de texto que contengan la palabra user se usará la expresión regular:

```
/user/
```

Así se localizarán todas las URL que contengan la palabra “user”:

<http://localhost:8080/user>

<http://localhost:8080/users>

<http://localhost:8080/alluser>

<http://localhost:8080/userlost>

<http://localhost:8080/sfsdfusersfs>

Esto se puede comprobar con el siguiente código:

Archivo regexp1.js:

```
const express = require('express')  
const app = express()  
  
app.get(/user/, (req, res) => {  
  res.send('Hola mundo')  
})  
  
app.listen(8080)
```

Se puede apreciar que la expresión regular se indica usando la sintaxis (**no se ponen las comillas**):

```
/ expresión /
```

5 Cookies y sesiones

Una cookie es un pequeño texto que el servidor manda al cliente y el cliente almacena. A cada petición que se realice, el cliente devolverá la cookie al servidor hasta que ésta expire.

Desde el punto de vista del programador se guardarán variables con sus valores que luego se podrán consultar.

Nota: Hay que evitar almacenar cualquier información sensible en las cookies, especialmente la relacionada con el usuario (información de identificación personal), como sus credenciales o sus preferencias. En la mayoría de los casos, las cookies sólo se usarán para almacenar una clave única y difícil de adivinar (ID de sesión) que coincida con un valor en el servidor. Eso le permite recuperar la sesión de un usuario en posteriores peticiones.

Express también es capaz de crear sesiones, en las que la información se almacena en el servidor y al cliente se le envía una cookie con un identificador de sesión.

5.1 Cookies

Para usar Cookies lo primero que se necesita es instalar en nuestro proyecto el paquete “cookie-parser”:

```
npm install cookie-parser
```

Ahora se va a crear un servidor de ejemplo:

Archivo cookies-app.js:

```
const express = require('express')
const cookieParser = require('cookie-parser')

const app = express()
app.use(cookieParser())

app.get('/', function (req, res) {
  console.log('Cookies: ', req.cookies)
  res.cookie('prueba', 'Valor de prueba')
  res.send("Prueba " + req.cookies['prueba'])
})

app.listen(8080)
```

Ahora se lanza el servidor:

```
$ node cookies-app.js
```

Al visitar la URL <http://localhost:8080> y recargar la página un par de veces, se puede ver que se crea una cookie llamada “prueba” con el valor “Valor de prueba”.

Vamos a analizar el código:

```
const cookieParser = require('cookie-parser')
...
app.use(cookieParser())
```

Sirven para cargar la biblioteca “cookie-parser” e indicar a express la función middleware que va a procesar las cookies. Esta función puede tener un parámetro que será investigado más adelante.

Para ver las cookies disponibles se tiene el mapa “req.cookies”, se puede consultar el valor de la cookie correspondiente poniendo su nombre:

```
res.send("Prueba " + req.cookies['prueba'])
```

Por último, se puede poner el valor de la cookie con el método cookie de res (**OJO**: de **res** no de req):

```
res.cookie('prueba', 'Valor de prueba')
```

Ejercicio: Se recomienda al lector hacer pruebas con las cookies, por ejemplo, un ejercicio de añadir una cookie que guarde la fecha de la última visita (deberá usar el objeto Date()).

5.1.1 Tiempo de expiración de una cookie

Al método res.cookie() se le pueden pasar más argumentos, por ejemplo, el tiempo de expiración de la cookie en milisegundos. Pasado ese tiempo la cookie será eliminada por el navegador:

```
const express = require('express')
const cookieParser = require('cookie-parser')

const app = express()
app.use(cookieParser())

app.get('/', function (req, res) {
  console.log('Cookies: ', req.cookies)
  res.cookie('prueba', 'Valor de prueba', {maxAge: 10*1000})
  res.send("Prueba " + req.cookies['prueba'])
})

app.listen(8080)
```

En este ejemplo se ha puesto el tiempo de expiración a 10 segundos usando el argumento “maxAge”:

```
res.cookie('prueba', 'Valor de prueba', {maxAge: 10*1000})
```

hay que recordar que este parámetro va en milisegundos.

En este caso si se vuelve a ejecutar el servidor, se puede comprobar que a los 10 segundos la cookie pasa a ser “undefined” y no aparece en la consola del navegador.

Para eliminar una cookie desde el servidor, habrá que poner un tiempo de expiración muy pequeño o cero. Por ejemplo, si el usuario inicia sesión (se autentifica con su usuario y contraseña), se le puede pasar una cookie con el identificador de la sesión:

```
res.cookie('idSesion', '223238')
```

Cuando el usuario pulse en el botón de cerrar la sesión, se le puede volver a pasar la cookie con un tiempo cero:

```
res.cookie('idSesion', '223238', {maxAge: 0})
```

5.1.2 Cookies firmadas

Antes se ha hablado de que a la línea:



```
app.use(cookieParser())
```

se le podían pasar argumentos. Esta función admite como argumento una contraseña que se usará para firmar digitalmente las cookies que así se deseen firmar. El cliente seguirá pudiendo ver el contenido de la cookie, a la cual se le añadirá una firma digital para que el servidor pueda comprobar si el valor de la cookie ha sido modificado de forma mal intencionada por el cliente.

Por ejemplo, se puede modificar nuestro servidor con cookies de la siguiente forma:

```
const express = require('express')
const cookieParser = require('cookie-parser')

const app = express()
app.use(cookieParser("Contraseña"))

app.get('/', function (req, res) {
  console.log('Cookies: ', req.cookies)
  console.log('Cookies Firmadas: ', req.signedCookies)
  res.cookie('prueba', 'Valor de prueba', {maxAge: 10*1000})
  res.cookie('cookieFirmada', 'Valor de prueba firmado', {maxAge: 10*1000,
signed: true})
  res.send(
    "Prueba " + req.cookies['prueba'] +
    "<br> Cookie firmada " + req.signedCookies['cookieFirmada']
  )
})

app.listen(8080)
```

Para indicar la contraseña que se va a usar para firmar, se le añade un argumento:

```
app.use(cookieParser("Contraseña"))
```

Para crear la cookie, se le añadirá el argumento “signed: true” al crear la cookie:

```
res.cookie('cookieFirmada', 'Valor de prueba firmado', {maxAge: 10*1000, signed: true})
```

Para obtener el valor de las cookies firmadas se puede usar el mapa “req.signedCookies”. Por ejemplo:

```
res.send(
  "Prueba " + req.cookies['prueba'] +
  "<br> Cookie firmada " + req.signedCookies['cookieFirmada']
)
```

En este caso si se usan las “Herramientas para el desarrollador” del navegador y se inspecciona el valor de la cookie, se puede ver que tiene la siguiente forma:

cookieFirmada: Valor de prueba firmado.kmn0vYwPGA8iNEkmP/TNBYuXYObJL133YBTMnN4nbiU

Es decir, el cliente puede ver el valor de la cookie y una firma digital añadida.

5.2 Sesiones

Como ya se ha contado Express también es capaz de crear sesiones, en las que la información se almacena en el servidor y al cliente se le envía una cookie con un identificador de sesión.

El mapa con el valor de las cookies esta vez se mantendrá almacenado en el servidor y no llegará al cliente. **Cuando haya información sensible, se recomienda usar sesiones en lugar de cookies.**

Para poder usar sesiones, el proyecto debe tener instalado los paquetes “express-session” y “cookie-parser”:

```
$ npm install express-session cookie-parser
```

Se va a estudiar un ejemplo de sesiones:

Archivo app.js:

```
const express = require('express');
const cookieParser = require('cookie-parser');
const session = require('express-session');
const app = express();

app.use(cookieParser());
app.use(
  session({
    secret: "Contraseña",
    cookie: { maxAge: 30*1000 },
    saveUninitialized: false,
    resave: false
  })
)

app.get('/', function(req, res){
  if(req.session.page_views) {
    req.session.page_views++;
    if(req.session.page_views === 10) {
      res.send('La sesión se ha cerrado')
      req.session.destroy()
    } else
      res.send('La página se ha visitado ' + req.session.page_views + ' veces');
  } else {
    console.log('La sesión está cerrada. Se abre una nueva')
    req.session.page_views = 1;
    res.send('Se abre la sesión');
  }
});

app.listen(8080);
```

Si se ejecuta este proyecto, se puede ver que se inicia una sesión y a las 10 visitas es destruida, o cuando pasen 30 segundos. Notará que debido a que los navegadores guardan las páginas (peticiones GET) en la caché, se necesitan más visitas de las esperadas para que el servidor reciba 10 visitas del cliente.

Vamos a analizar el código:



```
const cookieParser = require('cookie-parser');
const session = require('express-session');
const app = express();
```

```
app.use(cookieParser());
```

En este código se están cargando las bibliotecas de “express-session” y “cookie-parser”. También se están activando las cookies.

```
app.use(
  session({
    secret: "Contraseña",
    cookie: { maxAge: 30*1000 },
    saveUninitialized: false,
    resave: false
  })
)
```

Este código se analizará más a fondo más adelante, sólo indicar que:

- Se le indica a express que active la gestión de las sesiones.
- En el parámetro “secret” se le debe pasar una contraseña que se usará para firmar digitalmente la cookie de la sesión.
- En el parámetro “cookie”, se le indicarán las características de la cookie que se va a enviar al cliente para indicar el ID de la sesión. El parámetro interesante en este caso es “maxAge” que indicará el tiempo de vida de la cookie en milisegundos. Por ejemplo para que la sesión dure tres horas habría que poner: 3*60*1000.
- El resto de parámetros se verán más adelante. Se recomienda dejar los valores que se indican.

¿Cómo funcionan las sesiones?

A partir de ahora es muy sencillo gestionar una sesión. Se tiene el objeto:

req.session

A este objeto se le pueden añadir todos los campos que se necesiten para gestionar la sesión:

```
req.session.page_views = 1;
```

Aquí se ha almacenado un contador de visitas, pero se puede almacenar la variable que se deseé, por ejemplo, un nombre de usuario o si ha activado una determinada opción:

```
req.session.usuario = 'administrador';
req.session.opcion = true;
```

Para comprobar si en la siguiente visita la sesión ha sido iniciada, sólo hay que comprobar que algunos de los parámetros que se han definido para la sesión:

```
if(req.session.page_views) {
  // La sesión se ha iniciado
  ...
} else {
  // La sesión no se ha iniciado
```



```
    ...  
  }
```

En el caso que se está usando de ejemplo:

```
    if(req.session.page_views) {  
      req.session.page_views++;  
      if(req.session.page_views === 10) {  
        res.send('La sesión se ha cerrado')  
        req.session.destroy()  
      } else  
        res.send('La página se ha visitado ' + req.session.page_views + '  
veces');  
    } else {  
      console.log('La sesión está cerrada. Se abre una nueva')  
      req.session.page_views = 1;  
      res.send('Se abre la sesión');  
    }  
  }
```

Es buena idea para comprobar que la sesión se ha creado usar un valor creado al efecto. Por ejemplo, el nombre del usuario:

```
    if(req.session.usuario) {  
      // La sesión se ha iniciado  
      ...  
    } else {  
      // La sesión no se ha iniciado, se inicia la sesión  
      req.session.usuario = 'nombre usuario'  
    }  
  }
```

Nota: Hay que tener cuidado con los valores booleanos, pues por ejemplo:

```
    if(req.session.opcion) {  
      // La sesión se ha iniciado  
      ...  
    } else {  
      // La sesión no se ha iniciado, se inicia la sesión  
      req.session.opcion = false  
    }  
  }
```

En este caso aunque “req.session.opcion” existe, al hacer el if:

```
    if(req.session.opcion) {
```

Javascript entiende que su valor es “false” y no se evalúa.

Por último, para cerrar la sesión se usa:

```
req.session.destroy()
```

Con lo que los parámetros que se hubiesen definido se destruirán.

Se había comentado que el resto de parámetros de configuración de la sesión se iban a ver más adelante. Realmente se le solicita al lector que los consulte en la página oficial:

<https://github.com/expressjs/session>

El sistema de gestión de sesiones de Express es muy potente y permite que las sesiones se guarden en diversos formatos e incluso que se puedan almacenar en bases de datos, quedando lejos del ámbito de este documento.

6 Páginas web estáticas

Hasta ahora todas las páginas web que se han creado con express han sido dinámicas. Es decir se han generado desde express en el momento que han sido solicitadas, por ejemplo, todas las plantillas ejs se generan en el momento en que se solicitan. Pero hay otro tipo de contenido que no se genera siempre de forma dinámica, por ejemplo, una hoja de CSS.

Se puede añadir una carpeta, que normalmente se llamará “public” en la que colocar dicho contenido. En nuestro proyecto se creará la siguiente carpeta:

```
$ mkdir public
```

Se añade las siguientes líneas a nuestro proyecto para indicar que ahí hay contenido estático:

```
const path = require('path')
app.use('/static', express.static(path.join(__dirname, 'public')))
```

Se ha marcado en negrita el parámetro en el que se le indica la carpeta.

Finalmente, se puede poner una página web en dicha carpeta, por ejemplo:

Archivo public/hola.html:

```
<!doctype html>
<meta charset='utf-8'>
Hola mundo
Y visitarla con:
```

<http://localhost:8080/static/hola.html>

Importante: Aunque el archivo “hola.html” se ha introducido en la carpeta “public”, esto no se refleja en la URL. Se accede a la página con:

<http://localhost:8080/static/hola.html>

En lugar de usar:

~~<http://localhost:8080/public/hola.html>~~ ← **Error**

Si dentro de la carpeta “public” se introdujeran otras carpetas, sí habría que reflejarlas en la URL. Por ejemplo si dentro de “public” se tuviese la carpeta “imagenes” y se tiene una imagen dentro de ella:

```
public:
  imagenes:
    imagen.png
```

La URL a usar sería:

<http://localhost:8080/static/imagenes/imagen.png>

Si en lugar de la URL: <http://localhost:8080/static/hola.html>

Se deseara usar la URL: <http://localhost:8080/hola.html>

Se debería cambiar la ruta en nuestro servidor por '/' en lugar de 'static':

```
const path = require('path')
app.use('/', express.static(path.join(__dirname, 'public')))
```

7 Express generator

A la hora de generar un proyecto en express se ha visto que se dan una serie de pasos muy repetitivos, instalar los paquetes de las cookies y las sesiones, definir el motor de plantillas, definir las rutas...

Existe una aplicación que permite crear una plantilla con todos estos parámetros ya instalados y configurados y una serie de cuestiones de seguridad ya habilitadas, esta aplicación es “Express generator”.

Para instalarlo se usará el comando:

```
$ npx express-generator
```

Una vez instalado se tendrá en nuestro sistema un nuevo comando. Si se ejecuta:

```
$ express -h
```

Se obtendrá una ayuda de su uso:

```
Usage: express [options] [dir]

Options:
  --version           output the version number
  -e, --ejs           add ejs engine support
  --pug              add pug engine support
  --hbs              add handlebars engine support
  -H, --hogan         add hogan.js engine support
  -v, --view <engine> add view <engine> support (dust|ejs|hbs|hjs|jade|pug|twig|
vash) (defaults to jade)
  --no-view           use static html instead of view engine
  -c, --css <engine> add stylesheet <engine> support (less|stylus|compass|sass)
(defaults to plain css)
  --git              add .gitignore
  -f, --force         force on non-empty directory
  -h, --help          output usage information
```

Por ejemplo, para crear un proyecto que use ejs:

```
$ express --ejs nombre-proyecto
```

Si se va a crear un proyecto que se llame “segundo”:

```
$ express --ejs segundo
```

```
warning: option '--ejs' has been renamed to '--view=ejs'
```

```
create : segundo/  
create : segundo/public/  
create : segundo/public/javascripts/  
create : segundo/public/images/  
create : segundo/public/stylesheets/  
create : segundo/public/stylesheets/style.css  
create : segundo/routes/  
create : segundo/routes/index.js  
create : segundo/routes/users.js  
create : segundo/views/  
create : segundo/views/error.ejs  
create : segundo/views/index.ejs  
create : segundo/app.js  
create : segundo/package.json  
create : segundo/bin/  
create : segundo/bin/www
```

```
change directory:  
$ cd segundo
```

```
install dependencies:  
$ npm install
```

```
run the app:  
$ DEBUG=segundo:* npm start
```

Nos indica que se cambie a la carpeta del proyecto:

```
$ cd segundo
```

y se ejecute:

```
$ npm install
```

para instalar las dependencias.

Para hacer funcionar el servidor se ejecutará:

```
$ npm start
```

Se puede visitar la página en:

<http://localhost:3000/>

Aunque si vamos a desarrollar, recomienda usar el comando:

```
$ DEBUG=segundo:* npm start
```

Se puede ver que ha creado una carpeta:

- “views” en la que se introducirán las plantillas ejs.
- “routes” para poner las rutas. Por defecto crea “index” y “users” como ejemplos.
- Un archivo “app.js” que es donde se indicarán los paquetes, sesiones o se añadirán las rutas que se definan en “routes”.



- Una carpeta “public” donde se colocarán las hojas de estilo, html o javascript del cliente. Ahí se pueden introducir páginas web de html estático que será visitado por el cliente.

Se es totalmente libre de modificar los archivos incluidos, instalar nuevos paquetes, añadir o borrar lo que se necesite,...

7.1 Un proyecto de ejemplo usando Express generator

A modo de ejemplo se va a crear un proyecto de ejemplo usando express generator. Este proyecto va a constar de una página web en la que se va a crear una página web en la que se solicite un usuario y contraseña. Si el usuario es “admin” con contraseña “admin”, se abrirá una sesión. Cuando el usuario pulse sobre un enlace de cerrar sesión la sesión finalizará.

También tendrá un usuario “user” con contraseña “user”. El usuario “user” no podrá acceder a una página de administración que sólo el usuario “admin” podrá visitar.

También servirá para introducir algunos conceptos básicos sobre seguridad.

7.1.1 Creando el proyecto

Se creará un nuevo proyecto llamado aplicación, para ello se ejecutarán los comandos:

```
$ express --ejs aplicación
$ cd aplicación
$ npm install
```

También se instalarán las express-sessions, para poder manejar las sesiones:

```
$ npm install express-session
```

7.1.2 Usando una hoja de estilos

Es recomendable usar hojas de estilos ya creadas. Algunas son populares como Bootstrap, w3.ccs,... Por su sencillez y documentación se va a usar Materialize, cuya documentación se puede encontrar en:

<https://materializecss.github.io/materialize/getting-started.html>

Es bastante sencilla de usar, conviene echar un pequeño vistazo a su web para hacerse una idea de su funcionamiento. En esa misma página hay plantillas para hacer un comienzo rápido. También es fácil encontrar tutoriales sobre su uso.

Debemos descargarla desde:

<https://github.com/materializecss/materialize/releases/download/1.1.0/materialize-v1.1.0.zip>

Una vez descomprimida, se copiarán el archivos “materialize.css” a la carpeta de nuestro proyecto “aplicacion/public/stylesheets”. También se copiará “materialize.js” a la carpeta de nuestro proyecto “aplicacion/public/javascripts”.

Nota: Hay otros dos archivos llamados “materialize.min.js” y “materialize.min.css”. Son versiones que se usarán en producción. Podemos copiarlos también, para cambiar a una versión más eficiente de los archivos cuando se finalice el proyecto.

Además, se va a bajar un tipo de letra para usar en nuestro proyecto.

En este caso es un conjunto de iconos que respeta las reglas de Material Design de Google. Se puede consultar en:

<https://jossef.github.io/material-design-icons-iconfont/>

Su uso se puede ver en:

<https://materializecss.github.io/materialize/icons.html>

Creamos en **nuestro proyecto** la carpeta “aplicacion/public/stylesheets/fonts”.

Se descarga el tipo de letra:

<https://github.com/jossef/material-design-icons-iconfont/archive/refs/tags/v6.7.0.zip>

Una vez descargado el tipo de letra, se descomprime y se copiarán el archivo “material-design-icons-iconfont-6.7.0/dist/material-design-icons.css” a “aplicacion/public/stylesheets”. También se copiará el archivo “material-design-icons-iconfont-6.7.0/dist/fonts/MaterialIcons-Regular.woff2” a la carpeta “aplicacion/public/stylesheets/fonts”.

7.1.3 Creando las plantillas ejs

Se van a crear diferentes plantillas para usar en la aplicación. Básicamente este sitio va a tener las páginas:

- index: Es la página que se ve por defecto.
- login: Usada para iniciar la sesión.
- logout: Usada para cerrar la sesión.
- inicio: Es la página a la que se redirigen los usuarios cuando inician sesión. No tiene contenido útil, sólo está puesta para demostrar su uso y poder rellenarla en futuras aplicaciones.
- admin: Es una página sin contenido, pero que se usará para demostrar los permisos de usuario.
- Error: Es generada de forma automática por “express-generator” y sirve para mostrar errores en la navegación. Se puede personalizar.

Archivo views/cabecera.ejs:

```
<!doctype html>
<html>
<head>
```

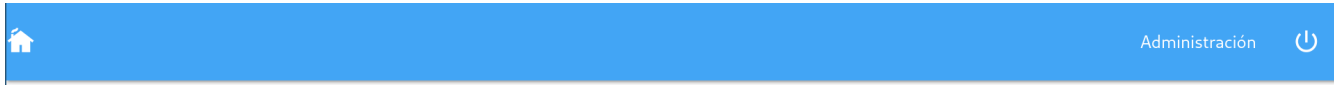


```

<meta charset='utf-8'>
<title><%= titulo %></title>
<!-- Se inserta la hoja de estilo -->
<link rel="stylesheet" href="/stylesheets/material-design-icons.css">
<link rel="stylesheet" href="/stylesheets/materialize.css">
<script src="/javascripts/materialize.js"></script>
</head>
<body>
<!-- Se inserta la cabecera de navegación -->
<nav>
  <div class="nav-wrapper blue lighten-1">
    <a href="/" class="brand-logo left"><i class="material-icons">cottage</i></a>
    <ul id="nav-mobile" class="right">
      <li><a href="/admin">Administración</a></li>
      <li><a href="/logout"><i
class="material-icons">power_settings_new</i></a></li>
    </ul>
  </div>
</nav>

```

Esta plantilla se usará como cabecera. Muestra una barra de navegación en la parte superior de la página. Se le debe pasar una variable “titulo” para que ponga el título de la página:



Archivo views/pie.ejs:

```

<footer class="page-footer blue lighten-1">
  <div class="container">
    <div class="row">
      <div class="col l6 s12">
        <h5 class="white-text">Footer Content</h5>
        <p class="grey-text text-lighten-4">You can use rows and columns here to
organize your footer content.</p>
      </div>
      <div class="col l4 offset-l2 s12">
        <h5 class="white-text">Links</h5>
        <ul>
          <li><a class="grey-text text-lighten-3" href="#">Link 1</a></li>
          <li><a class="grey-text text-lighten-3" href="#">Link 2</a></li>
          <li><a class="grey-text text-lighten-3" href="#">Link 3</a></li>
          <li><a class="grey-text text-lighten-3" href="#">Link 4</a></li>
        </ul>
      </div>
    </div>
  </div>
  <div class="footer-copyright">
    <div class="container">
      © 2014 Copyright Text
      <a class="grey-text text-lighten-4 right" href="#">More Links</a>
    </div>
  </div>
</footer>

```

```
</body>
</html>
```

Este archivo inserta un pie a la página:

Footer Content	Links
You can use rows and columns here to organize your footer content.	Link 1
	Link 2
	Link 3
	Link 4

Todas las plantillas deberán incluir a “cabecera.ejs” y a “pie.ejs”.

Archivo views/admin.ejs:

```
<%- include('cabecera', {titulo: 'Administración'}) -%>
<div class="section">
  <div class="container">
    <br><br>
    <h1 class="header center-align">Administración del sitio</h1>
    <div class="row center-align">
      <h5 class="header col s12">
        Formularios de administración
      </h5>
    </div>
  </div>
</div>
<%- include('pie') -%>
```

Como se puede ver incluye a “cabecera.ejs” y “pie.ejs”. Tiene algo de código HTML para dar formato al mensaje.

Archivo views/inicio.ejs:

```
<%- include('cabecera', {titulo: 'Aplicación'}) -%>
<div class="section">
  <div class="container">
    <br><br>
    <h1 class="header center-align">Sesión iniciada</h1>
    <div class="row center-align">
      <h5 class="header col s12">
        Informaciones de la sesión iniciada
      </h5>
    </div>
  </div>
</div>
<%- include('pie') -%>
```

Esta plantilla es similar a la anterior.

Archivo views/logout.ejs:




```

<%- include('cabecera', {titulo: 'Cerrar sesión'}) -%>
<div class="section">
  <div class="container">
    <br><br>
    <h1 class="header center-align">Sesión cerrada</h1>
  </div>
</div>
<%- include('pie') -%>

```

Archivo views/error.ejs:

```

<h1><%= message %></h1>
<h2><%= error.status %></h2>
<pre><%= error.stack %></pre>

```

La plantilla “error.ejs” se ha dejado tal y como la genera “express-generator”, pero se debería personalizar.

Archivo views/login.ejs:

```

<%- include('cabecera', {titulo: 'Inicio'}) -%>
<div class="section">
  <div class="container">
    <br><br>
    <div class="row">
      <form class="col s12" method='post' action='login'>
        <div class="row">
          <div class="z-depth-5 col s12 red" style='display: <%= login_error %>'>
            <p class="">Error: Usuario o contraseña incorrectos.</p>
          </div>
        </div>
        <div class="row">
          <div class="input-field col s12">
            <input id="usuario" name='usuario' type="text" class="validate">
            <label for="usuario">Usuario</label>
          </div>
        </div>
        <div class="row">
          <div class="input-field col s12">
            <input id="password" name='password' type="password" class="validate">
            <label for="password">Contraseña</label>
          </div>
        </div>
        <div class="row">
          <div class="input-field col s10"></div>
          <div class="input-field col s2">
            <input type="submit" value='Enviar' class="waves-effect waves-light blue
lighten-1 btn">
          </div>
        </div>
      </form>
    </div>
  </div>
<%- include('pie') -%>

```

El formulario de login está un poco más trabajado introduciendo diversos estilos y formatos. Es importante la línea:

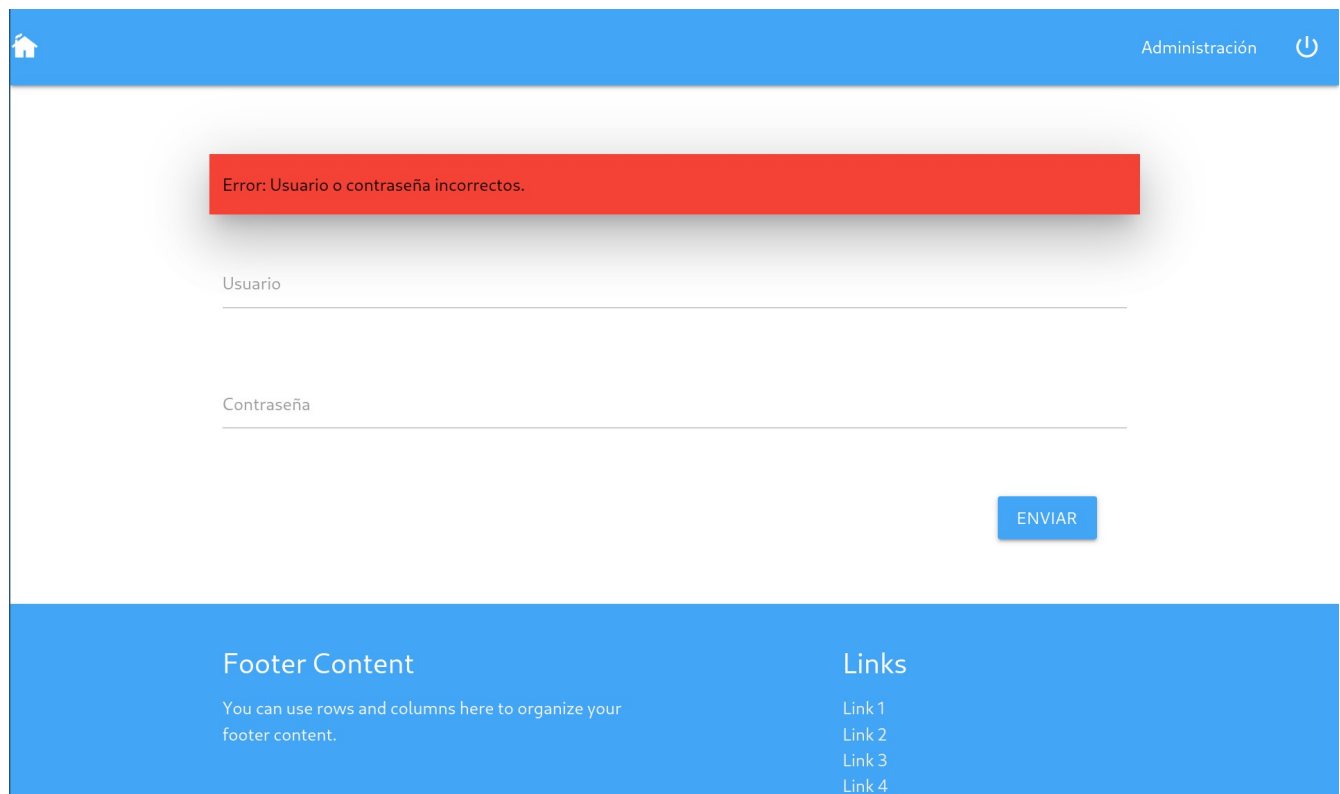
```
<form class="col s12" method='post' action='login'>
```

En ella se indica que el formulario hace una petición del tipo “post” y se envía a la ruta “/login”.

También se le ha añadido un mensaje para cuando el usuario introduce el usuario o contraseña incorrectos:

```
<div class="z-depth-5 col s12 red" style='display: <%= login_error %>'>
  <p class="">Error: Usuario o contraseña incorrectos.</p>
</div>
```

Como se puede ver acepta un parámetro llamado “login_error” que se introduce como valor del atributo CSS “display”. Habitualmente valdrá “none” (“display: none”), haciendo que el mensaje no se muestre. Cuando haya un error “login_error” deberá tomar el valor “inline” (“display: inline”), haciendo el mensaje de error visible:

A screenshot of a web application's login page. At the top is a blue header bar with a home icon on the left, the text "Administración" in the center, and a power icon on the right. Below the header is a white main content area. In the center of this area is a red rectangular box with the text "Error: Usuario o contraseña incorrectos." Below this box are two input fields: the first is labeled "Usuario" and the second is labeled "Contraseña". To the right of these fields is a blue button with the text "ENVIAR". At the bottom of the page is a blue footer bar. On the left side of the footer, under the heading "Footer Content", is the text "You can use rows and columns here to organize your footer content." On the right side of the footer, under the heading "Links", is a list of four links: "Link 1", "Link 2", "Link 3", and "Link 4".

Archivo views/index.ejs:

```
<%= include('cabecera', {titulo: 'Inicio'}) -%>
<div class="section">
  <div class="container">
    <br><br>
    <h1 class="header center-align">Starter Template</h1>
    <div class="row center-align">
```

```

    <h5 class="header col s12">
      A modern responsive front-end framework based on Material Design
    </h5>
  </div>
  <div class="row center-align">
    <a
      href="/login"
      class="btn-large waves-effect waves-light orange"
      target="_blank"
      rel="noopener noreferrer"
      >Iniciar sesión</a>
    </div>
    <br><br>
  </div>
</div>

<div class="container">
  <div class="section">
    <div class="row">
      <div class="col s12 m4">
        <h2 class="center-align">
          <i class="medium material-icons">flash_on</i>
        </h2>
        <h5 class="center-align">Speeds up development</h5>

        <p>
          We did most of the heavy lifting for you to provide a default
          stylings that incorporate our custom components. Additionally, we
          refined animations and transitions to provide a smoother
          experience for developers.
        </p>
      </div>

      <div class="col s12 m4">
        <h2 class="center-align">
          <i class="medium material-icons">group</i>
        </h2>
        <h5 class="center-align">User Experience Focused</h5>

        <p>
          By utilizing elements and principles of Material Design, we were
          able to create a framework that incorporates components and
          animations that provide more feedback to users. Additionally, a
          single underlying responsive system across all platforms allow for
          a more unified user experience.
        </p>
      </div>

      <div class="col s12 m4">
        <h2 class="center-align">
          <i class="medium material-icons">settings</i>
        </h2>
        <h5 class="center-align">Easy to work with</h5>

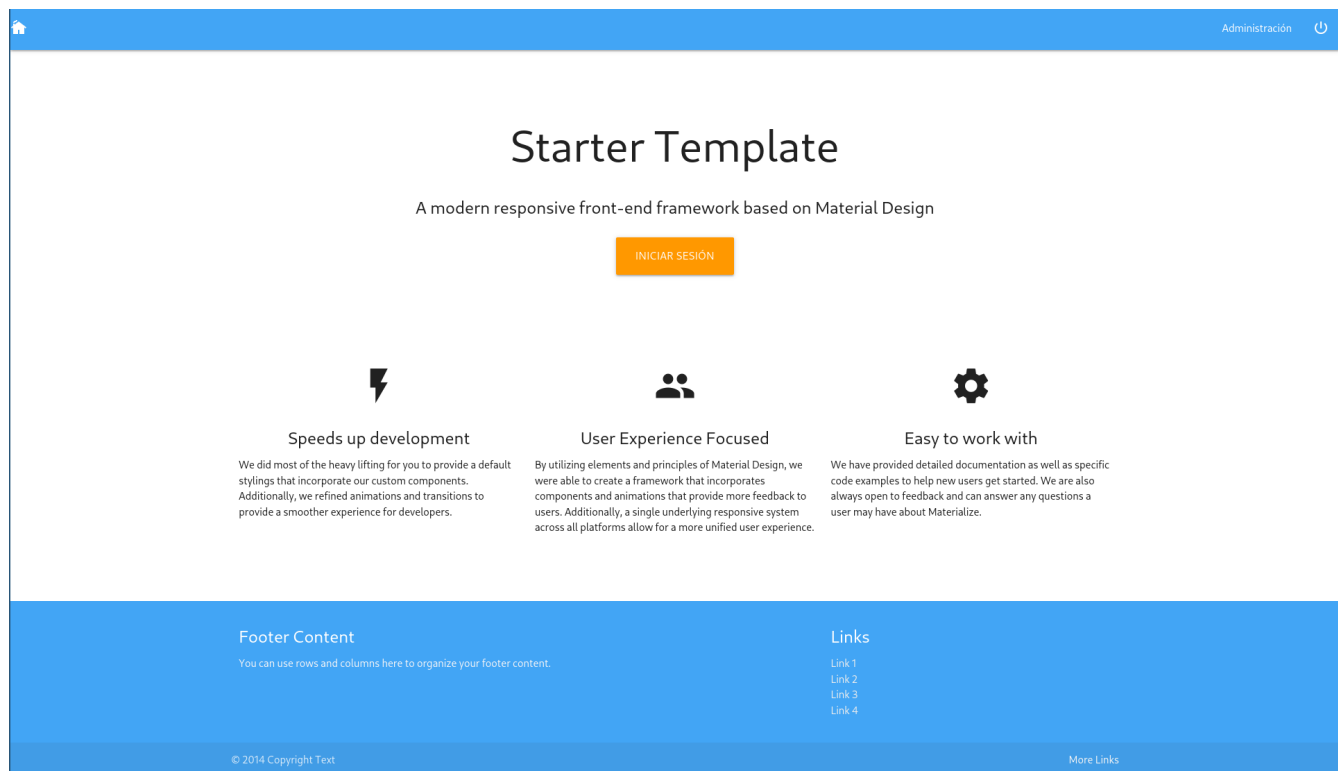
        <p>

```

We have provided detailed documentation as well as specific code examples to help new users get started. We are also always open to feedback and can answer any questions a user may have about Materialize.

```
</p>
</div>
</div>
</div>
<br><br>
</div>
<%- include('pie') -%>
```

Este archivo es una plantilla de “materialize.css” copiada tal cual y con leves modificaciones. Servirá como plantilla a modificar para futuros usos. Sólo se le ha cambiado el enlace al botón “Iniciar sesión” para que redirija a “/login”:



7.1.4 Archivos de permisos y app.js

Se va a definir un archivo que se usará para almacenar los permisos de usuario. Así se tendrán los permisos:

- NONE para indicar que el usuario no tiene permisos.
- USER permisos de usuario normal.
- ADMIN permisos de administrador.



Se pueden añadir más permisos a la lista, tantos como se necesiten. Lo único es que cada permiso tiene que tener un número diferente.

permissions.js:

```
// Lista de permisos.
// IMPORTANTE: A cada permiso se le debe asignar un número diferente.

const Permission = {
  NONE: 1,
  USER: 2,
  ADMIN: 3
}

module.exports = Permission;
```

El archivo app.js es más complejo. Nace de modificar el archivo por defecto que crea express-generator. El archivo sería el siguiente, se va a comentar más adelante:

app.js:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var logger = require('morgan');

// Se cargan las rutas

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var loginRouter = require('./routes/login');
var inicioRouter = require('./routes/inicio');
var logoutRouter = require('./routes/logout');
var adminRouter = require('./routes/admin');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
app.use(
  session({
    secret: "Contraseña",
    cookie: { maxAge: 10*60*1000 },
    saveUninitialized: false,
    resave: false
  })
);
```

```

    })
  )

// Listado de las páginas que se pueden ver sin autenticación
const public_pages = [
  "/",
  "/login"
];

// Listado de permisos
const perm = require("./permissions")

// Listado de páginas que requieren algún tipo de autorización especial
const private_pages = {
  "/inicio": [perm.USER, perm.ADMIN],
  "/logout": [perm.USER, perm.ADMIN],
  "/admin": [perm.ADMIN]
};

// Control de sesión iniciada
app.use((req, res, next) => {
  // Se verifica que el usuario haya iniciado sesión
  if(req.session.usuario) {
    // Se verifica que el usuario tiene permisos para visitar la página
    if(
      (
        req.url in private_pages
        &&
        private_pages[req.url].includes(req.session.permission)
      ) || public_pages.includes(req.url)
    )
      next()
    else
      next(createError(403)) // Forbidden
  } else {
    // Si el usuario no ha iniciado sesión,
    // se verifica que la página es pública:
    if(public_pages.includes(req.url))
      next()
    else if(req.url in private_pages)
      res.redirect('/login')
    else
      next(createError(404)) // Not found
  }
})

// Se asignan las rutas a sus funciones middleware
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/login', loginRouter);
app.use('/inicio', inicioRouter);
app.use('/logout', logoutRouter);
app.use('/admin', adminRouter)

// catch 404 and forward to error handler

```

```

app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;

```

Se van a analizar las partes interesantes del código:

Todavía no se ha hablado de ellas, pero se van a definir una serie de rutas para todas las páginas web que se van a crear. Estas rutas estarán almacenadas en la carpeta “routes”. Primero hay que cargar dichas rutas:

```

// Se cargan las rutas
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var loginRouter = require('./routes/login');
var inicioRouter = require('./routes/inicio');
var logoutRouter = require('./routes/logout');
var adminRouter = require('./routes/admin');

```

Después se asocian las URLs a las rutas:

```

// Se asignan las rutas a sus funciones middleware
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/login', loginRouter);
app.use('/inicio', inicioRouter);
app.use('/logout', logoutRouter);
app.use('/admin', adminRouter)

```

También se configuran los parámetros para almacenar las sesiones:

```

app.use(
  session({
    secret: "Contraseña",
    cookie: { maxAge: 10*60*1000 },
    saveUninitialized: false,
    resave: false
  })
)

```

Evidentemente en “Contraseña” se pondrá la contraseña que más nos interese. Se ha puesto que la duración de la sesión van a ser 10 minutos (10*60*1000).

Para la gestión de los permisos, primero se crearán dos listados, uno con las páginas “públicas” las que se pueden ver sin estar autenticado y otro con las páginas “privadas” las que necesitan estar autenticado y el nivel de permisos requeridos para poder visitarlas. Las páginas que no estén ni en públicas ni privadas, generarán un error al visitarse:

```
// Listado de las páginas que se pueden ver sin autenticación
const public_pages = [
  "/",
  "/login"
];

// Listado de permisos
const perm = require("./permissions")

// Listado de páginas que requieren algún tipo de autorización especial
const private_pages = {
  "/inicio": [perm.USER, perm.ADMIN],
  "/logout": [perm.USER, perm.ADMIN],
  "/admin": [perm.ADMIN]
};
```

Según esta lógica, “/login” se puede visitar libremente (lógico, pues es la que se usará para solicitar el usuario y contraseña). “/inicio” sólo se puede visitar si se ha iniciado sesión y se tiene el permiso o bien de usuario, o bien de administrador.

Ahora se describe la función middleware que posibilita la gestión de permisos. Hay que fijarse que es la primera función de este tipo que se define, por lo tanto la primera que se ejecuta al visitar cualquier página:

```
// Control de sesión iniciada
app.use((req, res, next) => {
  // Se verifica que el usuario haya iniciado sesión
  if(req.session.usuario) {
    // Se verifica que el usuario tiene permisos para visitar la página
    if(
      (
        req.url in private_pages
        &&
        private_pages[req.url].includes(req.session.permission)
      ) || public_pages.includes(req.url)
    )
      next()
    else
      next(createError(403)) // Forbidden
  } else {
    // Si el usuario no ha iniciado sesión, se verifica que la página es
    pública
  }
});
```



```

    if(public_pages.includes(req.url))
      next()
    else if(req.url in private_pages)
      res.redirect('/login')
    else
      next(createError(404)) // Not found
  }
})

```

Como se puede ver en la sesión del usuario se definen dos campos “req.session.usuario” que es simplemente el nombre del usuario y “req.session.permission” que es el permiso que tiene asignado ese usuario. Si el usuario ha iniciado sesión, se comprueba que está visitando una página privada y tiene el permiso para verla. También puede visitar la página si es pública. Si no tiene permisos, se genera un error 403 (Forbidden).

En el caso de que el usuario no haya iniciado sesión, se comprueba que la página que va a visitar está en el listado de las páginas públicas. Si no lo está y está en el listado de las páginas privadas, se le redirige a la página “/login” para que se autentifique. En el caso de que no esté en página públicas ni privadas, se genera un error 404 (Not found).

Es importante fijarse el orden en el que se ejecutan las funciones middleware. Se ha dicho que se ejecutan en el orden en que se definen en el archivo “app.js”, por ello, primero se define la función que controla los permisos. Después se definen las rutas y por último express-generator define un par de funciones que generan los errores de navegación de página no encontrada (error 404), si no se ha podido ejecutar ninguna ruta.

7.1.5 routes los archivos de rutas

Se van a describir las rutas que se han creado para las diversas páginas a visitar. Estas rutas ya se han asociado en el archivo “app.js”. Básicamente este sitio va a tener las páginas:

- index: Es la página que se ve por defecto
- login: Usada para iniciar la sesión
- logout: Usada para cerrar la sesión
- inicio: Es la página a la que se redirigen los usuarios cuando inician sesión. No tiene contenido útil, sólo está puesta para demostrar su uso y poder rellenarla en futuras aplicaciones
- admin: Es una página sin contenido, pero que se usará para demostrar los permisos de usuario.

Archivo routes/admin.js:

```

var express = require('express');
var router = express.Router();

```

```
router.get('/', function(req, res, next) {
  res.render('admin');
});
```

```
module.exports = router;
```

Este archivo muestra la página “/admin” la cual sólo puede visitar un usuario con permisos de ADMIN. No hay que preocuparse por la gestión de los permisos pues ya se ha hecho en “app.js”, por lo que se limita a mostrar la plantilla “admin” definida en “views/admi.ejs”.

Archivo routes/index.js:

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

```
module.exports = router;
```

Muestra la página “/”, normalmente asociada al archivo “index.html”. En este caso se muestra la plantilla “index” definida en “views/index.ejs”. Se le pasa un parámetro a la plantilla sólo como demostración.

Archivo routes/inicio.js:

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
  res.render('inicio', {titulo: 'Inicio'});
});
```

```
module.exports = router;
```

Ahora viene un archivo importante, que es el que gestiona el login del usuario:

Archivo routes/login.js:

```
// Se solicita usuario y contraseña y se inicia la sesión
// asignando permisos al usuario.
```

```
var express = require('express');
var router = express.Router();
const perm = require('../permissions')
```

```
router.get('/', function(req, res, next) {
  if(req.session.usuario) {
    console.log('Sesión iniciada')
    res.redirect('/inicio')
  }
}
```

```

    res.render('login', {login_error: "none"});
  });

router.post('/', function(req, res, next) {
  const usuario = req.body['usuario']
  const password = req.body['password']
  if(req.session.usuario) {
    res.redirect('/inicio')
  } if(usuario === 'admin' && password === 'admin') {
    // FIXME: Nunca usar contraseñas codificadas directamente
    // Se inicia la sesión
    req.session.usuario = usuario
    req.session.permission = perm.ADMIN
    res.redirect('/inicio')
  } if(usuario === 'user' && password === 'user') {
    // FIXME: Nunca usar contraseñas codificadas directamente
    // Se inicia la sesión
    req.session.usuario = usuario
    req.session.permission = perm.USER
    res.redirect('/inicio')
  } else {
    if(!usuario && !password)
      res.render('login', {login_error: "none"});
    else
      res.render('login', {login_error: "inline"});
  }
});

module.exports = router;

```

Analizando el código, se puede ver que el comportamiento es diferente si la petición es GET o POST. En la petición GET:

```

router.get('/', function(req, res, next) {
  if(req.session.usuario) {
    console.log('Sesión iniciada')
    res.redirect('/inicio')
  }
  res.render('login', {login_error: "none"});
});

```

Simplemente se muestra la plantilla de “login”, si el usuario no ha iniciado sesión o se redirige a “/inicio” si ya ha iniciado sesión.

La plantilla “views/login.ejs” tiene un parámetro “login_error” que ya se ha dicho que puede tener dos valores, “none” si no se debe mostrar el mensaje de error e “inline” si el usuario ha introducido mal el usuario o la contraseña. Como se puede ver en la petición GET se ignora todo lo relacionado con gestión de la sesión. **ÉSTO SE DEBE HACER EN LA PETICIÓN POST.**

En la petición POST, primero se obtienen los parámetros de usuario y contraseña que ha escrito el usuario en el formulario y se guardan en las constantes “usuario” y “password”:

```

router.post('/', function(req, res, next) {
  const usuario = req.body['usuario']
  const password = req.body['password']

```

Seguidamente se comprueba si el usuario ha iniciado la sesión. Si ha iniciado la sesión, se le redirige a “/inicio”:

```
if(req.session.usuario) {  
    res.redirect('/inicio')  
}
```

Ahora toca comprobar si las credenciales que el usuario ha introducido son correctas:

```
} if(usuario === 'admin' && password === 'admin') {  
    // FIXME: Nunca usar contraseñas codificadas directamente  
    // Se inicia la sesión  
    req.session.usuario = usuario  
    req.session.permission = perm.ADMIN  
    res.redirect('/inicio')  
} if(usuario === 'user' && password === 'user') {  
    // FIXME: Nunca usar contraseñas codificadas directamente  
    // Se inicia la sesión  
    req.session.usuario = usuario  
    req.session.permission = perm.USER  
    res.redirect('/inicio')  
}
```

Como se puede ver si las credenciales son correctas se crean los campos “req.session.usuario” y “req.session.permission”. Según el tipo de usuario se le asignan sus permisos y se le redirige a “/inicio”.

Importante: Desde el punto de vista de la seguridad, este código es inseguro. Nunca se deben codificar el usuario y contraseña directamente en el código. Lo habitual es que la contraseña lleve algún tipo de encriptación, por ejemplo, se le aplica un hash MD5 y lo que se comparan son dichos MD5. También se suelen guardar en bases de datos dichas credenciales de usuario. Más adelante se verá la forma correcta de iniciar una sesión de usuario.

La última parte del if se va a ejecutar si el usuario ha cometido algún error al introducir las credenciales:

```
} else {  
    if(!usuario && !password)  
        res.render('login', {login_error: "none"});  
    else  
        res.render('login', {login_error: "inline"});  
}
```

La primera parte del if sirve para comprobar si el usuario ha introducido algún tipo de credencial. Si no ha introducido credenciales, no se muestra el mensaje de error (login_error: “none”). Este comportamiento se dará en la primera visita. Si ha introducido credenciales, se debe indicar que hay un error (login_error: “inline”).

El siguiente archivo se usa para cerrar la sesión. Si la sesión está abierta, se destruye:

Archivo routes/logout.js:

```
// Se cierra la sesión del usuario

var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
  if(req.session.usuario) {
    req.session.destroy()
    res.render('logout')
  }
});

module.exports = router;
```

El último archivo lo genera express-generator y se deja como ejemplo:

Archivo routes/users.js:

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});

module.exports = router;
```

Se recomienda al lector que explore el funcionamiento de esta sencilla aplicación y estudie la forma en la que se han implementado los permisos.

8 Conectando con bases de datos

Para la conexión con bases de datos y la posterior realización de consultas, hay que tener en cuenta que la mayoría de las llamadas que se van a realizar a la base de datos son asíncronas, de forma que se hará la consulta y se exigirá el paso de una función que se ejecutará cuando dicha consulta haya finalizado.

Como ejemplo, las conexiones con la base de datos SQLite se harán de forma asíncrona y las conexiones con la base de datos MariaDB / MySQL se harán de forma síncrona. Se puede ver que la forma síncrona es más fácil de entender y leer.

En este texto sólo se van a estudiar las conexiones con SQLite y MariaDB / MySQL, pero es fácil encontrar la forma de realizar la conexión con otras bases de datos como pueden ser MongoDB o Postgress.

8.1 SQLite

SQLite es una base de datos muy ligera que destaca por almacenar los datos en un archivo. No está orientada a la típica conexión cliente-servidor que hacen con la mayoría de bases de datos.

Para comenzar habrá que instalar la conexión con SQLite en nuestro proyecto. Por lo que se ejecutará:

```
$ npm install sqlite3
```

Una vez instalada se puede ver en el siguiente ejemplo una conexión en el que se crea una tabla, se insertan valores y se hacen consultas:

Archivo `sqlite.js`:

```
const sqlite3 = require('sqlite3');

let db= new sqlite3.Database('./ejemplo.db', sqlite3.OPEN_READWRITE |
sqlite3.OPEN_CREATE, (err) => {
  if (err) {
    console.log("Error: " + err);
    process.exit(1);
  }
  createDatabase(db);
  runQueries(db);
});

function createDatabase(db) {
  db.exec(
    `
    create table if not exists user (
      user text primary key not null,
      password text not null,
      permission int not null
    );

    insert into user (user, password, permission)
      values ('admin', 'admin', 3),
             ('user', 'user', 2),
             ('user2', 'user2', 2);
  `,
    (err) => {
      if(err) {
        console.log("Error: " + err);
        process.exit(1);
      }
    }
  );
};

db.all('insert into user (user, password, permission) values(?,?,?)',
  ["user3", "user3", 2],
  (err) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
  }
);
```

```

    }
  )
}

function runQueries(db) {
  db.all("select * from user", (err, rows) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
    console.log("Consulta con forEach:");
    rows.forEach(row => {
      console.log(row.user + "\t" + row.password + "\t" + row.permission)
    })
  });

  db.all("select * from user", (err, rows) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
    console.log("Consulta con for:")
    let buffer = "";
    for(let n = 0; n < rows.length; n++) {
      const row = rows[n];
      buffer += row.user + "\t" + row.password + "\t" + row.permission + "\n";
    }
    console.log(buffer);
  });

  db.all("select * from user where permission = ? and user = ?", 2, "user", (err,
rows) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
    console.log("Consulta con argumentos:");
    rows.forEach(row => {
      console.log(row.user);
    });
  });
  console.log("Consultas finalizadas");
}

```

Si se ejecuta, una vez, tecleando:

```
$ node sqlite.js
```

Aparecerá una salida similar a:

```
Consultas finalizadas
Consulta con forEach:
admin    admin    3
user     user     2
```

```
user2  user2  2
Consulta con argumentos:
user
Consulta con for:
admin  admin  3
user   user   2
user2  user2  2
```

Son los resultados de las diversas consultas. También se verá que aparece un archivo llamado “ejemplo.db”. En este archivo se encuentra almacenada la base de datos.

Nota: El orden en el que se muestra los resultados puede cambiar al ejecutar el lector el ejemplo, pues al hacer una ejecución paralela usando asincronía, no se puede establecer el orden en el que se van a ejecutar los comandos.

Si se ejecuta una segunda vez, aparece el siguiente mensaje:

```
Consultas finalizadas
Error: Error: SQLITE_CONSTRAINT: UNIQUE constraint failed: user.user
```

Este error se debe a que se crea la siguiente tabla:

```
create table if not exists user (
  user text primary key not null,
  password text not null,
  permission int not null
);
```

Si nos fijamos la clave primaria es el campo “user”. En el código se inserta una serie de valores:

```
insert into user (user, password, permission)
values ('admin', 'admin', 3),
      ('user', 'user', 2),
      ('user2', 'user2', 2);
```

La primera vez funciona sin problemas pues la base de datos está vacía.

La segunda vez se genera el error, pues las claves primarias para los valores que se desean insertar ya existen. Hay que recordar que la clave primaria no se puede repetir y no se pueden insertar dos valores con la misma clave primaria.

Para superar este error, en este caso sólo vamos a borrar el archivo “ejemplo.db” y volverlo a ejecutar. También se puede comentar las líneas en las que se hacen las inserciones, después de la primera ejecución.

Vamos a analizar el código:

```
const sqlite3 = require('sqlite3');
```

Con esta línea simplemente se importa la biblioteca correspondiente a “sqlite3”.

El siguiente paso es abrir la conexión con la base de datos. Para ello se usa una función de la forma:


```
let db= new sqlite3.Database(archivo, sqlite3.OPEN_READWRITE | sqlite3.OPEN_CREATE,
función );
```

Donde “archivo” es el archivo en el que está almacenada la base de datos.

Con `sqlite3.OPEN_READWRITE` se le indica que la base de datos se va a abrir para lectura y escritura.

Con `sqlite3.OPEN_CREATE` la base de datos se va a crear si no existe (hay situaciones en las que se puede eliminar esta opción).

La función es una función que se va a ejecutar cuando se termine el proceso de apertura de la base de datos. Este proceso se lanza de forma asíncrona, por lo que se debe tener en cuenta. La función admite un parámetro que indica si se ha producido un error.

En el ejemplo que nos ocupa:

```
let db= new sqlite3.Database('./ejemplo.db', sqlite3.OPEN_READWRITE |
sqlite3.OPEN_CREATE, (err) => {
  if (err) {
    console.log("Error: " + err);
    process.exit(1);
  }
  createDatabase(db);
  runQueries(db);
});
```

Se puede ver que se abre el archivo “./ejemplo.db” y que la función que se pasa como argumento, muestra el mensaje de error en caso de hacer errores y para la ejecución del programa. En caso de no haber errores, ejecuta las funciones “createDatabase()” y “runQueries()”. Estas dos funciones están puestas como ejemplo.

Otra cosa importante es que se devuelve un objeto “db” que posteriormente se usará para hacer consultas o ejecutar comando SQL.

Con los métodos:

- “db.exec(comandoSQL, función)” Se puede ejecutar un comando SQL en la base de datos que no devuelva valores.
- “db.all(comandoSQL, argumentos, función)” Se puede ejecutar un comando SQL que devuelve valores o se pueden pasar argumentos a los comandos SQL para evitar los ataques de inyección de SQL.

Por ejemplo para crear una base de datos se podría ejecutar:

```
db.exec(
  create table if not exists user (
    user text primary key not null,
    password text not null,
    permission int not null
  );
```

```
, (err) => {
  if(err) {
    console.log("Error: " + err);
    process.exit(1);
  }
});
```

Como se puede ver la **función** admite un argumento que indica si se ha producido un error. En este caso se muestra el error y se para la ejecución del script.

Un ejemplo de uso de “db.all()” sería:

```
db.all('insert into user (user, password, permission) values(?,?,?)',
  "user3", "user3", 2,
  (err) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
  }
)
```

Como se puede ver en el **comando SQL** hay 3 interrogaciones, estas 3 interrogaciones se sustituyen por los valores:

```
"user3", "user3", 2,
```

Después está la **función** que toma un argumento que representa si se ha producido un error.

Con estos dos métodos “db.exec()” y “db.all()” se crea la función de nuestro ejemplo “createDatabase()” que crea la base de datos e inserta valores en ella:

```
function createDatabase(db) {
  db.exec(
    `
    create table if not exists user (
      user text primary key not null,
      password text not null,
      permission int not null
    );

    insert into user (user, password, permission)
      values ('admin', 'admin', 3),
             ('user', 'user', 2),
             ('user2', 'user2', 2);
    `
  , (err) => {
    if(err) {
      console.log("Error: " + err);
      process.exit(1);
    }
  });

  db.all('insert into user (user, password, permission) values(?,?,?)',
    "user3", "user3", 2,
    (err) => {
      if(err) {
        console.log("Error: " + err);
      }
    }
  )
}
```

```

        process.exit(1);
    }
}
)
}

```

El método “db.all()” también es capaz de recibir los valores que se producen de una consulta. En este caso la función que se le pasa como argumento, va a admitir un segundo valor que representa las filas que se devuelven en la consulta (err, rows):

```

db.all("select * from user where permission = ? and user = ?",
    2, "user",
    (err, rows) => {
        if(err) {
            console.log("Error: " + err);
            process.exit(1);
        }
        console.log("Consulta con argumentos:");
        rows.forEach(row => {
            console.log(row.user);
        });
    });

```

El argumento rows guarda los resultados de la consulta, se pueden recorrer dichos argumentos usando un bucle for o el método forEach como se muestra en el ejemplo anterior. Se pueden consultar las columnas de cada fila de la consulta poniendo su nombre (en este caso la base de datos tiene las columnas row.user, row.password y row.permission).

En lugar del método forEach, se podría usar un simple bucle for:

```

db.all("select * from user", (err, rows) => {
    if(err) {
        console.log("Error: " + err);
        process.exit(1);
    }
    console.log("Consulta con for:")
    let buffer = "";
    for(let n = 0; n < rows.length; n++) {
        const row = rows[n];
        buffer += row.user + "\t" + row.password + "\t" + row.permission + "\n";
    }
    console.log(buffer);
});

```

Esto se puede ver en el método “runQueries()” del ejemplo.

Nota para el lector: Se recomienda practicar creando una base de datos en SQLite, haciendo inserciones y consultas antes de continuar.

8.1.1 SQLite y express



Teniendo en cuenta que los comandos de SQLite se van a ejecutar de forma asíncrona, es muy fácil añadir operaciones SQL a express usando SQLite. Lo más importante es que las respuestas que se van a enviar al cliente se van a hacer desde las funciones asíncronas que ejecutan los comandos de SQLite.

Por ejemplo, el siguiente código genera un servidor express que hace operaciones en la base de datos. Muy importante fijarse en qué sitio se han puesto las respuestas `res.send()` (o `res.render()` si se diera el caso):

Archivo `sqlite3.js`:

```
const express = require('express')
const app = express()
const sqlite3 = require('sqlite3');

// Se abre la base de datos
let db= new sqlite3.Database('./datos.db', sqlite3.OPEN_READWRITE |
sqlite3.OPEN_CREATE, (err) => {
  if (err) {
    console.log("Error: " + err);
    process.exit(1);
  }
});

// Consulta que crea una tabla y la llena de contenido
app.get('/crear', (req, res) => {
  db.exec(
    `
    create table if not exists user (
      user text primary key not null,
      password text not null,
      permission int not null
    );

    insert into user (user, password, permission)
      values ('admin', 'admin', 3),
             ('user', 'user', 2),
             ('user2', 'user2', 2);
  `,
    (err) => {
      if(err) {
        res.send("Error: " + err);
        return console.error(err.message);
      }
      res.send('Base de datos creada')
    }
  );
});

// Se hace una consulta y se muestra el resultado
app.get('/', (req, res) => {
  db.all("select * from user", (err, rows) => {
    if(err) {
      res.send("Error: " + err);
      return console.error(err.message);
    }
  })
});
```

```

        let buffer = "";
        for(let n = 0; n < rows.length; n++) {
            const row = rows[n];
            buffer += row.user + "\t" + row.password + "\t" + row.permission + "<br>\n";
        }
        res.send(buffer);
    });
})
app.listen(8080)

```

En el ejemplo anterior, si se visita la página <http://localhost:8080/crear> , se ejecuta la operación de crear la base de datos e insertar datos. Es importante fijarse en el código marcado en rojo.

Si se visita la página <http://localhost:8080/> , se hace una consulta y se muestra el resultado de la consulta.

Nota: Cuando se produce un error en la base de datos se ha añadido la siguiente línea:

```
return console.error(err.message);
```

Esta línea hace que no se genere una excepción que pare el funcionamiento de express. También muestra el resultado por consola.

8.2 MariaDB / MySQL

MariaDB y MySQL son unas bases de datos muy usadas para las realización de páginas web.

Inicialmente MySQL era una base de datos de software libre. Esta base de datos fue comprada por Oracle. Oracle comenzó a introducir cambios en MySQL que no gustaron a sus desarrolladores, por lo que decidieron bifurcar el proyecto y hacer una nueva base de datos que fue compatible con MySQL pero sin los cambios que había introducido Oracle. Así nació MariaDB.

Al ser compatibles la forma de conectarse con ellas es muy similar.

Se suponen una serie de conocimientos básicos para el manejo de MariaDB.

8.2.1 Creando una base de datos MariaDB con Docker

Para hacer ejemplos con MariaDB se puede usar el siguiente archivo de Docker:

Archivo docker-compose.yaml:

```

version: "3.7"

services:
  contenedor_mariadb:
    image: mariadb:latest
    ports:

```



```
- 3306:3306
volumes:
- ./datos:/var/lib/mysql
environment:
  MARIADB_USER: admin
  MARIADB_PASSWORD: admin-password
  MARIADB_ROOT_PASSWORD: root-password

contenedor_phpmyadmin:
  image: phpmyadmin
  ports:
  - 8000:80
  environment:
    PMA_HOST: contenedor_mariadb
```

Para arrancarlo se ejecutará:

```
$ docker-compose up
```

Evidentemente se deberá instalar Docker y Docker-compose en nuestro equipo para que funcione.

Se tendrán en ejecución dos contenedores Docker. Uno con la base de datos MariaDB y otro con PHPMyAdmin que es un entorno gráfico para administrar MariaDB.

Ahora se va a crear una base de datos llamada “pruebas”, un usuario “pruebas” con la contraseña “pruebas”.

Nos conectamos a PHPMyAdmin usando la URL:

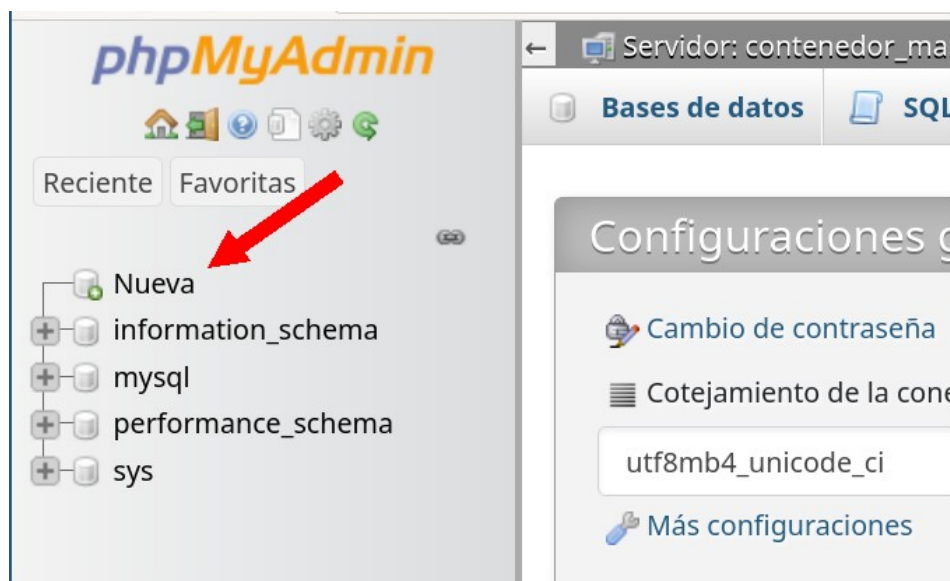
<http://localhost:8000>

Nos autentificamos usando las credenciales de administrador de la base de datos:

Usuario: root

Contraseña: root-password

Ahora se va a crear la base de datos “pruebas” y se le va a asignar el usuario “pruebas”. Para ello se hace clic en “Nueva”:



Se le va a poner nombre a la base de datos (“pruebas”) y se hace clic en “Crear”:

Crearé la base de datos, pero está vacía, sin tablas ni datos. Ahora se creará el usuario “pruebas” con privilegios para operar sobre dicha base de datos. Se hace clic en la pestaña “Privilegios” y se hace clic en “Agregar cuenta de usuario”:

	Nombre de usuario	Nombre del servidor	Tipo	Privilegios	Conceder	Acción
<input type="checkbox"/>	root	%	global	ALL PRIVILEGES	Sí	Editar privilegios Exportar
<input type="checkbox"/>	root	localhost	global	ALL PRIVILEGES	Sí	Editar privilegios Exportar

☐ Seleccionar todo
 Para los elementos que están marcados: Exportar

Nuevo
[Agregar cuenta de usuario](#)

Se añadirá el usuario “pruebas” con contraseña “pruebas”. No olvidar activar las casillas de los permisos en “DATOS” y “ESTRUCTURA”:

En la parte final se pulsará el botón “Continuar” y nos indicará que el usuario ha sido creado.

8.2.2 Conexión desde NodeJS con MariaDB

Lo primero es instalar la biblioteca “mariadb” en el proyecto en el que estemos trabajando:

```
$ npm install mariadb
```

El proceso de conexión con MariaDB es sencillo:

1. Se carga la biblioteca

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

2. Se prepara la conexión con la base de datos, se necesitará la IP o el dominio en el que se encuentra la base de datos funcionando, el usuario y la contraseña:

```
const pool = mariadb.createPool({host: 'localhost', user: 'pruebas', password: 'pruebas', connectionLimit: 5});
```

3. Se abre la conexión con la base de datos:

```
conn = await pool.getConnection();
```

4. Se hacen las consultas, inserciones,...:

```
const res = await conn.query("INSERT INTO nombres value (?, ?)", [1, "síncrono"]);
const rows = await conn.query("SELECT * from nombres");
rows.forEach((row) => {console.log(row.id, row.nombre)})
```

5. Se cierra la conexión:




```
conn.release();
```

Un ejemplo completo:

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

// Se prepara la conexión a la base de datos:
const pool = mariadb.createPool({host: 'localhost', user: 'pruebas', password:
'pruebas', connectionLimit: 5});

async function asyncFunction() {
  let conn;
  try {

    // Se abre una conexión a la base de datos
    conn = await pool.getConnection();

    // Se usa la base de datos "pruebas"
    const res1 = await conn.query('use pruebas')

    const res2 = await conn.query('create table if not exists nombres(id
integer primary key not null, nombre text not null)')

    // Se borran los contenidos de la tabla sólo como ejemplo
    const res3 = await conn.query('delete from nombres')

    const res = await conn.query("INSERT INTO nombres value (?, ?)", [1,
"síncrono"]);
    console.log(res)
    // res: { affectedRows: 1, insertId: 1, warningStatus: 0 }

    console.log('Select ejecutado de forma síncrona:')
    const rows = await conn.query("SELECT * from nombres");
    rows.forEach((row) => {console.log(row.id, row.nombre)})

    // Select con argumentos
    const rows2 = await conn.query("SELECT * from nombres where id = ?", [1]);
    rows2.forEach((row) => {console.log(row.id, row.nombre)})

  } finally {
    // Se cierra la conexión a la base de datos
    if (conn) conn.release();
    console.log('Finalizado')
  }
}

asyncFunction()
```

En este ejemplo se usa “await” para transformar las llamadas asíncronas en síncronas.

8.2.3 MariaDB y Express

La conexión de MariaDB en Express es bastante directa y sencilla de entender. Por ejemplo:

```
const express = require('express')
const app = express()
const mariadb = require('mariadb');

// Se prepara la conexión a la base de datos:
const pool = mariadb.createPool({host: 'localhost', user: 'pruebas', password:
'pruebas', connectionLimit: 5});

async function crear(res) {
  let conn;
  try {

    // Se abre una conexión a la base de datos
    conn = await pool.getConnection();

    // Se usa la base de datos "pruebas"
    const r1 = await conn.query('use pruebas')

    const r2 = await conn.query('create table if not exists nombres(id integer
primary key not null, nombre text not null)')

    // Se borran los contenidos de la tabla sólo como ejemplo
    const r3 = await conn.query('delete from nombres')

    const r4 = await conn.query("INSERT INTO nombres value (?, ?)", [1,
"síncrono"]);

  } finally {
    // Se cierra la conexión a la base de datos
    if (conn) conn.release();
  }
  res.send('Base de datos creada')
}

app.get('/crear', (req, res) => {
  crear(res)
})

async function consulta(res) {
  let conn;
  try {

    // Se abre una conexión a la base de datos
    conn = await pool.getConnection();

    // Se usa la base de datos "pruebas"
    const res1 = await conn.query('use pruebas')

    const rows = await conn.query("SELECT * from nombres where id = ?", [1]);
    let buffer = ''
    rows.forEach((row) => {buffer += row.id + ' ' + row.nombre})
  }
```

```

        res.send(buffer)

    } finally {
        // Se cierra la conexión a la base de datos
        if (conn) conn.release();
        console.log('Finalizado')
    }
}

app.get('/consulta', (req, res) => {
    consulta(res)
})

app.listen(8080)

```

8.3 Sugerencia a la hora de conectar a bases de datos desde Express

En los ejemplos anteriores se hacían las conexiones desde Express desde el mismo archivo. El objetivo era simplificar el código para hacerlo más legible.

En proyectos más grandes se recomienda crear una clase base que represente las operaciones que se van a realizar en la base de datos. Cuando se quiera conectar a una base de datos, por ejemplo MySQL, se creará una clase que herede de esa clase base de datos con las operaciones y llamadas que se van a realizar en MySQL.

Si en un futuro se desea cambiar de base de datos, por ejemplo SQLite, se volverá a extender desde dicha clase base y se harán las operaciones y llamadas para SQLite.

Según la configuración, se podrá seleccionar entre una u otra base de datos. Para ello se debe crear una función que lea alguna variable de configuración, o archivo de configuración, de nuestra aplicación y devuelva la conexión a la base de datos que sea más adecuada.

Desde la aplicación se llamará a dicha función y se trabajará con independencia de la base de datos.

9 Variables de entorno

En el código de conexión a la base de datos MariaDB, se ha tenido que introducir en el código y en texto plano el usuario y la contraseña de acceso a la base de datos. Este tipo de información se suele poner en archivos de configuración o en variables de entorno más que directamente en el código.

Esto permite que el código sea fácilmente reutilizable y transportable. En el caso de usar variables de entorno, nuestra aplicación se podrá empaquetar fácilmente en una imagen de Docker.

Las variables de entorno son pares de valores de la forma nombre – valor que el sistema operativo define y que las aplicaciones pueden consultar. El sistema operativo pasa una copia de estas variables de entorno a las aplicaciones en el momento de lanzar su ejecución.

En un sistema operativo Unix, estas variables de entorno se pueden consultar con el comando `env`:

```
$ env
SHELL=/bin/bash
WINDOWID=0
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc::/usr/share
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session1
XDG_MENU_PREFIX=lxqt-
LANGUAGE=es
LC_ADDRESS=es_ES.UTF-8
LC_NAME=es_ES.UTF-8
LC_MONETARY=es_ES.UTF-8
XCURSOR_SIZE=32
...
```

Como se puede ver el sistema operativo pasa mucha información del tipo, idioma que está usando el usuario (`LANGUAGE=es`), tamaño del cursor (`XCURSOR_SIZE=32`),...

Desde NodeJS las variables de entorno se pueden leer usando:

```
process.env.nombre
```

Por ejemplo, para leer la variable de entorno `PUERTO`:

```
const puerto = process.env.PUERTO || 8080
console.log(puerto)
```

Puede ser que esta variable de entorno no se haya definido, por lo que habría que poner un `if` comprobando si su valor es `undefined`. Para ahorrar código se pone la sintaxis:

```
const nombre = process.env.nombre || valor
```

de forma que si la variable de entorno no está definida, toma “valor” como valor por defecto.

En sistemas operativos Unix se pueden definir variables de entorno para que sean usadas por nuestras aplicaciones de dos formas:

- Usando `export`
- Definiendo las variables antes de ejecutar nuestra aplicación

Usando `export` se necesita la siguiente sintaxis:

```
$ export nombre=valor
```

Por ejemplo:

```
$ export PUERTO=3000
```

La variable de entorno queda definida en la consola que se está usando y es pasada de forma automática a los comandos que se ejecuten en dicha consola. Si la consola se cierra, la variable de entorno es destruida. Las variables de entorno “viven” la consola en la que se definen.

También se pueden definir antes de ejecutar nuestra aplicación:

```
$ nombre=valor comando
```

Por ejemplo:

```
$ PUERTO=3000 node app.js
```

10 Código HASH de textos

Cuando se almacena una contraseña en una base de datos, esta contraseña debe almacenarse cifrada. Lo habitual es aplicar una función HASH a la contraseña y guardar el resultado en la base de datos. Si la base de datos es robada, será muy difícil recuperar las contraseñas originales por los delincuentes.

Una función HASH hace una operación matemática muy compleja sobre un texto (realmente es un conjunto de bits) que se le pasa como entrada y la transforma en otro conjunto de bits. La gracia de la función HASH es que es muy fácil de generar, pero muy difícil volver a obtener el texto original.

Cuando se va a realizar la autenticación de un usuario, se procede a aplicar la función HASH sobre la contraseña que envía y se compara con la almacenada en la base de datos. Si son iguales, significa que el usuario ha introducido unas credenciales de acceso correctas.

Algunas funciones HASH requieren el paso de un código adicional para realizar su operación correctamente. Para aumentar la seguridad hay algoritmos que generan este código de forma automática por cada contraseña que de la que se obtiene su código HASH, así es más difícil obtener las contraseñas. Este código se va a denominar **salt**.

Así en nuestra base de datos de credenciales además del código HASH de la contraseña, se debe almacenar su **salt**. Cuando un usuario introduzca sus credenciales de acceso, se le aplicará la función HASH a la contraseña que introduzca usando el salt almacenado en la base de datos. Si el código HASH resultante coincide con el código HASH almacenado en la base de datos para la contraseña, las credenciales son válidas.

Para hacer esta operación se usará la biblioteca 'pbkdf2-password'.

Se deberá instalar en nuestro proyecto ejecutando:

```
$ npm install pbkdf2-password
```

Esta biblioteca nos provee de una función llamada hash() que funciona de la siguiente forma:

A la función hash() se le van a pasar dos argumentos. El primer argumento es un mapa con la password a cifrar y el **salt**. Si no se pone el salt, se genera uno de forma automática.

El segundo argumento es una función que se va a ejecutar de forma asíncrona que se va a ejecutar cuando se termine la generación del código HASH.

Por ejemplo:

```
let password_salt = ''
let password_hash = ''

hash({ password: 'Contraseña a cifrar' }, function (err, pass, salt, hash) {
```

```

    if (err) throw err;
    password_salt = salt;
    password_hash = hash;
  });

```

En este caso se le pasa como argumento ‘Contraseña a cifrar’. Como no se le pasa un salt, se genera uno automáticamente. Ambos valores se deberían almacenar en una base de datos, en este caso en las variables password_salt y password_halt. Hay que fijarse que el salt y el hash generados está disponibles en las variables “salt” y “hash”.

Supongamos que un usuario introduce una contraseña y se desea comparar con el HASH de la contraseña almacenada. En este caso sí se pasará como argumento el código salt generado en el paso anterior (password_salt):

```

hash({ password: 'Contraseña a cifrar', salt: password_salt }, function (err, pass, salt, hash) {
  if (err) throw err;
  console.log('Probando la contraseña: ' + password_correcta)
  if( password_hash === hash)
    console.log('La contraseña es correcta')
  else
    console.log('La contraseña es incorrecta')
});

```

Como se puede comprobar en el if se compara el hash almacenado con el que se acaba de generar.

Un ejemplo completo sería:

Archivo password.js:

```

const hash = require('pbkdf2-password')()

// Contraseña que se va a cifrar
const password = 'Contraseña'

// Variables en las que se va almacenar el hash y el salt
let password_salt = ''
let password_hash = ''

// Se genera el hash y el salt y se guardan
hash({ password: password }, function (err, pass, salt, hash) {
  if (err) throw err;
  password_salt = salt;
  password_hash = hash;
  verificar()
});

function verificar() {
  const password_correcta = 'Contraseña'
  const password_incorrecta = '12345'

  // Se verifica la password_correcta
  hash({ password: password_correcta, salt: password_salt }, function (err, pass, salt, hash) {

```

```

    if (err) throw err;
    console.log('Probando la contraseña: ' + password_correcta)
    if( password_hash === hash)
        console.log('La contraseña es correcta')
    else
        console.log('La contraseña es incorrecta')
  });

  // Se verifica la password_incorrecta
  hash({ password: password_incorrecta, salt: password_salt }, function (err,
pass, salt, hash) {
    if (err) throw err;
    console.log('Probando la contraseña: ' + password_incorrecta)
    if( password_hash === hash)
        console.log('La contraseña es correcta')
    else
        console.log('La contraseña es incorrecta')
  });
}

```

Si se ejecuta este código, por consola se puede tener una salida similar a:

```

Probando la contraseña: 12345
La contraseña es incorrecta
Probando la contraseña: Contraseña
La contraseña es correcta

```

Como la generación del código HASH se hace de forma asíncrona, es posible que cambie el orden de la salida.

10.1 Un ejemplo básico de autenticación con express

En el siguiente ejemplo se va a crear una base de datos SQLite en la que se van almacenar el usuario, el código HASH de la contraseña y el salt de la contraseña.

El usuario que se va a crear es “usuario” con contraseña “12345”.

Analice el código antes de probarlo. Vea también la forma en la que se usan las funciones asíncronas.

También compruebe que en la base de datos se almacena el código HASH de la contraseña y el salt de la contraseña. Vea también cómo se obtiene el salt de la contraseña de la base de datos para verificar las credenciales.

Archivo autenticación.js:

```

const express = require('express')
const app = express()
const sqlite3 = require('sqlite3');
const hash = require('pbkdf2-password')()

// Se activan las peticiones POST
app.use(express.urlencoded({
  extended: true
}))

```



```

// Se conecta con la base de datos
let db= new sqlite3.Database('./usuarios.db', sqlite3.OPEN_READWRITE |
sqlite3.OPEN_CREATE, (err) => {
  if (err) {
    console.log("Error: " + err);
    process.exit(1);
  }
});

app.get('/crear', (req, res) => {
  // Se crea la tabla de usuarios y
  // se almacena el usuario "usuario" con contraseña "12345"
  hash({ password: '12345' }, function (err, pass, salt, hash) {
    if (err) throw err;
    db.exec(
      `
      create table if not exists user (
        user text primary key not null,
        password_hash text not null,
        password_salt text not null
      );
    `,
      (err) => {
        if(err) {
          res.send("Error: " + err);
          return console.error(err.message);
        }
        db.all('insert into user (user, password_hash, password_salt)
values(?,?,?)',
          ["usuario", hash, salt,
            (err) => {
              if(err) {
                res.send("Error: " + err);
                return console.error(err.message)
              } else {
                res.send('Base de datos creada')
              }
            }
          ],
          (err) => {
            if(err) {
              res.send("Error: " + err);
              return console.error(err.message)
            } else {
              res.send('Base de datos creada')
            }
          }
        );
      }
    );
  });
});

const html_login = `
<!doctype html>
<meta charset='utf-8'>
<form method='post' action='/login'>
Usuario: <input type='text' name='usuario'><br>
Contraseña: <input type='password' name='password'><br>
<input type='submit' value='Acceder'>
</form>
`;

app.get('/', (req, res) => {
  res.send(html_login)
});

```



```

}))
app.post('/login', (req, res) => {
  const usuario = req.body['usuario'];
  const password = req.body['password'];
  db.all("select password_hash, password_salt from user where user = ?",
    usuario,
    (err, rows) => {
      if(err) {
        res.send("Error: " + err);
        return console.error(err.message);
      } else if(0 < rows.length) {
        const row = rows[0];
        hash({ password: password, salt: row.password_salt }, function
(err, pass, salt, hash) {
          if (err) {
            res.send("Error: " + err)
            return console.error(err.message)
          } else if( row.password_hash === hash) {
            // Las credenciales son correctas
            // Se debería iniciar la sesión
            res.send('La contraseña es correcta')
          } else
            res.send('La contraseña es incorrecta')
          });
        } else {
          res.send('Usuario no encontrado')
        }
      });
    });
  app.listen(8080)

```

Para crear la base de datos e insertar el usuario se usa la URL:

<http://localhost:8080/crear>

Para iniciar la sesión, hay que visitar:

<http://localhost:8080/>

Este código muestra cómo comprobar las credenciales de forma correcta. Una vez verificadas las credenciales, se debería iniciar la sesión del usuario (Ver el apartado [5.2.Sesiones](#)).

11 Tokens

Una forma de almacenar información en el cliente de forma que no sea necesario el uso de bases de datos es el uso de tokens. La idea básica es cifrar la información y enviarla al cliente, por ejemplo, a través de una cookie. Es así el cliente el que almacena la información, pero no puede leerla pues está cifrada. Cuando el cliente desea realizar alguna operación en el servidor, deben enviar la información cifrada que le ha enviado el servidor, el servidor la descifra y trabaja con la información.

Se usa, por ejemplo, para almacenar el inicio de sesión de un cliente. El cliente se autentifica en el servidor y el servidor le manda un token en el que almacena el nombre del usuario que ha iniciado la sesión y la hora de expiración de la sesión. También puede almacenar otras informaciones, como ciertas preferencias. Cada vez que el usuario se conecta al servidor, le manda el token y el servidor sólo tiene que descifrar el token para ver si el usuario ha iniciado sesión y si ha expirado la sesión, sin necesidad de hacer trabajar a la base de datos.

Esto mejora mucho la escalabilidad y permite que sea un servidor diferente al que autoriza la sesión el que responda la petición de los clientes.

Para crear tokens se puede usar la biblioteca “jsonwebtoken”. Para instalarla en nuestro proyecto sólo hay que teclear:

```
$ npm install jsonwebtoken
```

Se va a estudiar su funcionamiento con un ejemplo sencillo:

Archivo jwt3.js:

```
const jwt = require('jsonwebtoken');

const JWT_PASSWORD = 'Contraseña'
const token = jwt.sign({ valor1: 'a', valor2: 2 }, JWT_PASSWORD, {algorithm: 'HS256'});
console.log(token)

const datos = jwt.verify(token, JWT_PASSWORD, {algorithm: 'HS256'})
console.log(datos.valor1)
console.log(datos.valor2)
```

Si se ejecuta se obtiene una salida similar a la siguiente:

```
$ node jwt3.js
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ2YWxvcjEiOiJhIiwidmFsb3IyIjoyLCJpYXQiOiJE2NTg5NDAXNTJ9.jdMtMiPxQqBkehmLGf3tg8RpD94X0X24fMNWWH9Cmn4
a
2
```

Como se puede ver en el ejemplo, primero se carga la biblioteca y se selecciona una contraseña con la que se van a cifrar los tokens:

```
const jwt = require('jsonwebtoken');

const JWT_PASSWORD = 'Contraseña'
```

Es muy recomendable que la contraseña se compleja.

Ahora se crea el token usando la función “jwt.sign()”:

```
const token = jwt.sign({ valor1: 'a', valor2: 2 }, JWT_PASSWORD, {algorithm: 'HS256'});
```

En el token se han almacenado los siguientes datos:

```
{ valor1: 'a', valor2: 2 }
```

Se le deben pasar como argumentos la clave de cifrado (en este caso la variable `JWT_PASSWORD`) y el algoritmo de cifrado, que en este caso es un algoritmo de cifrado simétrico (se usa la misma contraseña para cifrar que para descifrar) el “HS256”.

Como se puede ver en la salida por consola, el token no es legible:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ2YWxvcjEiOiJhIiwidmFsb3IyIjoyLCJpYXQiOi0jE2NTg5NDExNTJ9.jdMtMiPxQqBkehmLGf3tg8RpD94X0X24fMNWWH9Cmn4
```

Para volver a descifrar el token y obtener los datos que estaban en él almacenados, se usará la función `jwt.verify()`:

```
const datos = jwt.verify(token, JWT_PASSWORD, {algorithm: 'HS256'})
```

Se le pasa como argumento el token, la contraseña y el algoritmo de cifrado. Devolverá un objeto desde el que se pueden acceder a los datos almacenados en el token.

En este caso en el token se había almacenado:

```
{ valor1: 'a', valor2: 2 }
```

Por lo que se puede acceder a los valores con sólo poner su nombre:

```
console.log(datos.valor1) // Escribe una 'a'
console.log(datos.valor2) // Escribe un 2
```

El token, dentro de un servidor creado con express, se puede enviar al cliente usando una cookie y comprobando el valor de la cookie cada vez que se deseé. La forma más habitual es enviarlo a través de una cabecera HTTP.

11.1 Duración de un token y tipos de cifrado

En el ejemplo anterior se ha creado un token con cifrado simétrico de duración infinita, pues no se le ha puesto tiempo de expiración. Con la opción “`expiresIn`” se puede indicar en segundos el tiempo de expiración del token. Por ejemplo, para una duración de 3 horas:

```
const token = jwt.sign({ valor1: 'a', valor2: 2 }, JWT_PASSWORD, {algorithm: 'HS256', expiresIn: 60*60*3});
```

En el ejemplo anterior se ha hecho haciendo un cifrado simétrico, se usa la misma contraseña para cifrar y descifrar.

Existen otras formas de cifrado, como el cifrado asimétrico, en el que se usa una clave para cifrar y otra para descifrar. En el cifrado asimétrico existen dos claves, la clave pública y la clave privada. Los mensajes cifrados con la clave pública sólo se pueden descifrar con la privada. Los mensajes cifrados con la clave privada sólo se pueden descifrar con la pública.

Supongamos que se tiene la clave privada almacenada en el archivo “`private.key`”. Se podría usar el algoritmo “RS256” para hacer el cifrado:



```
var privateKey = fs.readFileSync('private.key');
var token = jwt.sign({ foo: 'bar' }, privateKey, { algorithm: 'RS256' });
```

11.2 Tokens a través de cookies

Se ha comentado que se podría enviar un token a través de una cookie. Un ejemplo sencillo sería el siguiente:

```
const express = require('express')
const cookieParser = require('cookie-parser')
const jwt = require('jsonwebtoken');
const app = express()

const JWT_PASSWORD = 'Contraseña'

// Se activan las cookies
app.use(cookieParser())
// Se procesan las peticiones POST
app.use(express.urlencoded({
  extended: true
}))

// Usuario y contraseña: DEBERÍA ESTAR EN UNA BASE DE DATOS
// NUNCA EN EL CÓDIGO
const usuario = {usuario: 'pepe', password: 'pepe'}

const formulario = `
<!doctype html>
<meta charset='utf-8'>
<form method='post' action='/login'>
Nombre: <input type='text' name='usuario'><br>
Contraseña: <input type='password' name='password'><br>
<input type='submit' value='Enviar'>
</form>
`;

app.get('/login', (req, res) => {
  res.send(formulario)
})

app.post('/login', (req, res) => {
  const user = req.body['usuario']
  const password = req.body['password']
  if(usuario.usuario === user && usuario.password === password ) {
    // Las credenciales son correctas se crea y envía el token en una cookie
    const respuesta = `
      <!doctype html>
      <meta charset='utf-8'>
      <p>Hola ${user}</p>
    `;
    const token = jwt.sign({ usuario: user }, JWT_PASSWORD, {algorithm:
'HS256', expiresIn: 60});
    res.cookie('token', token)
    res.send(respuesta);
  }
})
```

```

    } else
        res.send(formulario)
})

// Página para probar el acceso
app.get('/', function (req, res) {
    if(req.cookies) {
        // Si hay cookies, se verifica que hay un token:
        const token = req.cookies['token']
        if(token) {
            // Se comprueba que el token es válido
            const datos = jwt.verify(token, JWT_PASSWORD, {algorithm: 'HS256'})
            if(datos) {
                // Sesión iniciada
                res.send('Acceso permitido ' + datos.usuario)
            } else {
                res.send(formulario)
            }
        }
    } else {
        res.send(formulario)
    }
})

app.listen(8080)

```

En la URL <http://localhost:8080/login> se puede hacer la autenticación del usuario ('pepe' con contraseña 'pepe'). Una vez autenticado se le envía una cookie con el token, que posee una duración de 1 minuto:

```

const token = jwt.sign({ usuario: user }, JWT_PASSWORD, {algorithm:
'HS256', expiresIn: 60});
res.cookie('token', token)

```

Una vez autenticado, si se visita <http://localhost:8080> se comprueba si el token es válido y se permite el acceso:

```

const datos = jwt.verify(token, JWT_PASSWORD, {algorithm: 'HS256'})
if(datos) {
    // Sesión iniciada
    res.send('Acceso permitido ' + datos.usuario)
}

```

Si se deja pasar un minuto, el token caduca y se lanza una excepción indicando que no se puede acceder al recurso. Realmente dicha excepción se debería capturar con un try – catch que muestre un mensaje al usuario. Se deja como ejercicio al lector implementar dicho try – catch.

11.3 Envío de tokens a través de cabeceras HTTP

La forma más habitual de enviar tokens es a través de cabeceras HTTP. Se usa, por ejemplo, para APIs a las que el usuario se conecta a través de aplicaciones en lugar de usar un navegador web.

En el siguiente ejemplo se describe dicha técnica. Para ello se usará la biblioteca “express-jwt”. Esta biblioteca comprueba si hay un token en la cabecera HTTP y nos copia los valores almacenados en la variable “req.auth”.

Para instalarla se ejecuta en nuestro proyecto:

```
$ npm install express-jwt
```

Una vez instalada se puede importar ejecutando:

```
const {expressjwt: expressJwt} =require('express-jwt');
```

Esta biblioteca pondrá a nuestra disposición un middleware que se puede añadir a nuestro proyecto.

Con el método “unless” se puede indicar con una lista las páginas que no van a requerir token de acceso. Por ejemplo, para indicar que se va a requerir token de acceso a todas las páginas salvo “/login”, se usará el siguiente middleware:

```
app.use(expressJwt({secret: JWT_PASSWORD, algorithms: ["HS256"]})).unless({path: ["/login"]}));
```

Una vez que se permita el acceso a una página protegida con un token, se tiene en la variable “req.auth” un mapa con los valores almacenados en el token:

```
app.get('/pagina', (req, res) => {  
  console.log(req.auth);  
  res.json({usuario: req.auth.usuario, permisos: req.auth.permisos});  
});
```

El ejemplo completo sería:

Archivo jwt2.js:

```
const express = require('express');  
const bodyParser=require('body-parser');  
const jwt=require('jsonwebtoken');  
const {expressjwt: expressJwt} =require('express-jwt');  
const app = express();  
  
const JWT_PASSWORD = "Contraseña";  
  
app.use(express.static('public'));  
app.use(bodyParser.json());  
app.use(expressJwt({secret: JWT_PASSWORD, algorithms: ["HS256"]})).unless({path: ["/login"]}));  
  
// Estos valores se debían consultar en una base de datos  
const usuario = { usuario: "pepe", password: "pepe" }  
  
app.get('/pagina', (req, res) => {  
  console.log(req.auth);  
  res.json({usuario: req.auth.usuario, permisos: req.auth.permisos});  
});  
  
app.post("/login", (req, res) => {  
  // Se comparan los valores enviados por el usuario con los almacenados  
  // en la base de datos:
```

```

    if (req.body.usuario === usuario.usuario || req.body.password ===
usuario.password) {
        const token = jwt.sign({ usuario: "pepe", permisos: "lector"},
JWT_PASSWORD, {algorithm: "HS256"});
        res.send(token);
    } else {
        res.status(401).end("usuario incorrecto")
    }
});
app.listen(8080);

```

Como se puede ver en el ejemplo, una vez autenticado el usuario, se crea el token y se envía directamente al usuario:

```

const token = jwt.sign({ usuario: "pepe", permisos: "lector"}, JWT_PASSWORD,
{algorithm: "HS256"});
res.send(token);

```

En el lado cliente se recibirá el token y se enviará al servidor mediante peticiones AJAX. Es muy habitual usarlo junto con bibliotecas como React y jQuery. En el ejemplo, se usa jQuery para ayudar con las peticiones AJAX:

Archivo public/jwt2.html:

```

<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>JWT</title>
<script src='/javascripts/jquery.js'></script>
</head>
<body>
<div>
    <input type="text" id="usuario" /><br>
    <input type="password" id="password" />
</div>
<div>
    <input type="button" id="login" value="Login" />
    <input type="button" id="pagina" value="Pagina" />
</div>
<div id='salida'>
</div>
<script>
    $(document).ready(function() {
        var token = "";
        $("#login").click(function() {
            var usuario = { usuario: $("#usuario").val(), password: $
("#password").val() };
            $.ajax({
                url: 'login',
                type: 'post',
                contentType: 'application/json; charset=utf-8',

```

```

        data: JSON.stringify(usuario),
        success: function(data) { token = data; },
        error: function(error) { console.log(error); }
    });
});
$("#pagina").click(function() {
    $.ajax({
        url: 'pagina',
        type: 'get',
        contentType: 'application/json; charset=utf-8',
        dataType: 'json',
        success: function(data) {
            console.log(data);
            $('#salida').html('Usuario ' + data.usuario);
        },
        beforeSend: function(xhr) {
            console.log(token);
            xhr.setRequestHeader("Accept", "application/json");
            xhr.setRequestHeader("Authorization", "Bearer " +
token);
        },
    });
});
});
</script>
</body>
</html>

```

Como se puede ver en el ejemplo, se usa jQuery para hacer las peticiones AJAX. Para iniciar la sesión se usa:

```

var usuario = { usuario: $("#usuario").val(), password: $("#password").val() };
$.ajax({
    url: 'login',
    type: 'post',
    contentType: 'application/json; charset=utf-8',
    data: JSON.stringify(usuario),
    success: function(data) { token = data; },
    error: function(error) { console.log(error); }
});
});

```

El token se almacena en la variable “token”.

En peticiones posteriores se envía el token también por AJAX en la cabecera HTTP:

```

$.ajax({
    url: 'pagina',
    type: 'get',
    contentType: 'application/json; charset=utf-8',
    dataType: 'json',
    success: function(data) {
        console.log(data);
        $('#salida').html('Usuario ' + data.usuario);
    },
    beforeSend: function(xhr) {

```



```

        console.log(token);
        xhr.setRequestHeader("Accept", "application/json");
        xhr.setRequestHeader("Authorization", "Bearer " + token);
    },
    });
});

```

Nota: En el ejemplo se puede comprobar que al recargar la página se pierde el acceso y hay que volver a introducir las credenciales. Esto se debe a que en el cliente se está guardando el token en una variable. Al recargar la página se pierde dicha variable. Lo más correcto es almacenar el token usando el localStorage de HTML5 (consultar en Google) o en una cookie.

12 Cifrando conexiones con HTTPS

Por seguridad todas las conexiones deberían estar cifradas usando el protocolo HTTPS. Para cifrar una conexión se debería tener un certificado. Supongamos que ya se tiene dicho certificado. Serán dos archivos “server.key” y “server.cert”. Se puede cifrar una conexión mediante HTTPS de la siguiente forma:

Archivo https1.js:

```

const express = require("express");
//const http = require("http");
const https = require("https");
const fs = require("fs");
const app = express();

const httpsKeys = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.cert")
};

app.get('/', (req, res) => {
  res.send('Hola mundo')
})

//http.createServer(app).listen(3000);
https.createServer(httpsKeys, app).listen(3030);

```

Como se puede ver se deben leer los certificados desde los archivos y pasarlos a la hora de crear el servidor https. Se puede comprobar que se está sirviendo en la dirección:

<https://localhost:3030>

Queda pendiente otra pregunta: ¿Dónde obtener un certificado?

Hay dos formas, crear un certificado “auto-firmado” u obtener uno de una entidad certificadora.

12.1 Creando un certificado “auto-firmado”

Para crear un certificado auto-firmado se debe tener instalado openssl en nuestro equipo. En equipos con Ubuntu instalado es tan fácil como ejecutar:

```
$ sudo apt install openssl
```

Una vez instalado se ejecutará el comando:

```
$ openssl req -nodes -new -x509 -keyout server.key -out server.cert
```

Este comando realizará una serie de preguntas y finalmente generará los dos archivos “server.key” y “server.cert” que se necesitan.

Importante: Al no ser generado por una entidad certificadora, este certificado se considera inseguro y los navegadores muestran un mensaje de advertencia al visitar la página. Se debe pulsar en “Avanzado...”:



Advertencia: riesgo potencial de seguridad a continuación

Firefox ha detectado una posible amenaza de seguridad y no ha cargado localhost. Si visita este sitio, los atacantes podrían intentar robar información como sus contraseñas, correos electrónicos o detalles de su tarjeta de crédito.

[Más información...](#)

Retroceder (recomendado)

Avanzado...

y aceptar el certificado pulsando en “Aceptar el riesgo y continuar”:



Advertencia: riesgo potencial de seguridad a continuación

Firefox ha detectado una posible amenaza de seguridad y no ha cargado localhost. Si visita este sitio, los atacantes podrían intentar robar información como sus contraseñas, correos electrónicos o detalles de su tarjeta de crédito.

[Más información...](#)

Retroceder (recomendado)

Avanzado...

localhost:3000 usa un certificado de seguridad no válido.

No se confía en el certificado porque está autofirmado.

Código de error: [MOZILLA_PKIX_ERROR_SELF_SIGNED_CERT](#)

[Ver certificado](#)

Retroceder (recomendado)

Aceptar el riesgo y continuar

12.2 Certificados de “Let’s encrypt”

Las autoridades certificadoras suelen cobrar por generar los certificados. Existe un proyecto desde el cual se puede obtener un certificado de forma gratuita. Este proyecto se denomina “Let’s encrypt”:

<https://letsencrypt.org>

Los certificados generados por “Let’s encrypt” se consideran certificados válidos y no es necesario seguir los pasos para aceptar el certificado del apartado anterior.

Desde la página de “Let’s encrypt” se instalará un asistente llamado “certbot”. Una vez instalado “certbot”, se ejecutará el comando:

```
certbot certonly --manual
```

y se seguirán las instrucciones. Puede que nos solicite colocar un archivo en el dominio que se desee servir para que “Let’s encrypt” verifique nuestro dominio.

13 Websockets

En una conexión clásica entre un cliente y un servidor HTTP el cliente abre una conexión con el servidor. A través de esa conexión solicita una página, el servidor le devuelve la página al cliente y cierra la conexión.

En esta forma de proceder se abre una conexión y se cierra cada vez que se solicita una página.

Pero imagine que se desea escribir una aplicación de chat en el que los usuarios escriben sus mensajes y reciben los mensajes de los otros usuarios. Una forma de hacer esto según el esquema de conexión clásico de HTTP es que cada cliente consulte cada poco tiempo al servidor solicitando los nuevos mensajes que se hayan recibido.

Esta forma de operar hará que el servidor reciba muchas peticiones de los clientes, de forma innecesaria, sólo para comprobar si se han recibido nuevos mensajes del chat.

Para resolver esta situación, y muchas otras, se crearon los websockets.

En un websocket se abre una conexión bidireccional con el servidor, de forma que tanto el cliente y el servidor pueden enviar información en cualquier momento. La conexión no se cierra al finalizar la petición. Permanece abierto para recibir las peticiones sucesivas.

Las conexiones con websockets se nombran con “ws://” o “wss://” si son encriptadas. Así se establecerán conexiones en URLs de la forma:

ws://localhost:8080/chat

wss://localhost:8080/

En los siguientes ejemplos se van a establecer conexiones usando el protocolo ws, **pero se recomienda usar conexiones cifradas en formato wss.**

13.1 Express y websockets

Para usar websockets en express se usará la biblioteca “express-ws”. Se instalará en nuestro proyecto usando:

```
$ npm install express-ws
```

Para cargar esta biblioteca en nuestro proyecto se usará:

```
var express = require('express');  
var app = express();  
var expressws = require('express-ws')(app);
```

Al igual que se tiene “app.get()” o “app.post()”, ahora se tiene un nuevo método para establecer la rutas sobre conexiones websocket:

```
app.ws(ruta, función)
```

Por ejemplo:



```
app.ws('/', function(ws, req) {
  ws.on('message', function(msg) {
    console.log(msg);
    ws.send(msg)
  });
  ws.on('close', () => {
    console.log('Socket cerrado')
  })
  console.log('socket', req.testing);
});
```

Como argumento de la función se le pasan dos argumentos:

```
app.ws('/', function(ws, req) {
```

req es el parámetro que se venía usando en express para representar la petición que hace el cliente.

ws es un objetivo que representa la conexión (socket) establecida con el cliente. Este objeto tiene métodos para enviar información al cliente:

```
ws.send(texto)
```

Este objeto también tiene eventos que se pueden capturar, como los eventos “message” que contiene un mensaje enviado por el cliente y “close” que se dispara cuando se cierra la conexión con el cliente.

El ejemplo completo, tomado de la documentación de “express-ws”.

Archivo ws1.js:

```
var express = require('express');
var app = express();
var expressWs = require('express-ws')(app);

app.use(express.static('public'));

app.use(function (req, res, next) {
  console.log('middleware');
  req.testing = 'testing';
  return next();
});

app.get('/', function(req, res, next){
  console.log('get route', req.testing);
  res.end();
});

app.ws('/', function(ws, req) {
  ws.on('message', function(msg) {
    console.log(msg);
    ws.send(msg)
  });
  ws.on('close', () => {
    console.log('Socket cerrado')
  })
  console.log('socket', req.testing);
});

app.listen(8080);
```

Este ejemplo sólo toma el mensaje que le ha enviado el cliente, lo muestra por consola y lo vuelve a enviar al cliente.

13.2 Websockets en el lado cliente. Javascript en el navegador

Para completar el ejemplo anterior se necesitaría un cliente que enviase mensajes al servidor. El siguiente ejemplo muestra cómo enviar mensajes desde el cliente al servidor:

Archivo public/ws1.html:

```
<!doctype html>
<meta charset='utf-8'>
<script>
let socket = new WebSocket("ws://localhost:8080");

socket.onopen = function(e) {
  alert("[onopen] Conexión abierta. Mandando datos al servidor");
  socket.send("Hola mundo");
};

socket.onmessage = function(event) {
  const mensaje = event.data
  alert(`[onmessage] Mensaje recibido del servidor: ${mensaje}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[onclose] Conexión cerrada limpiamente, code=${event.code} reason=${event.reason}`);
  } else {
    // Servidor apagado o sin red
    alert(`[onclose] La conexión se ha perdido`);
  }
};

socket.onerror = function(error) {
  alert(`[onerror] ${error.message}`);
};
</script>
```

Como se puede ver en el ejemplo, se establece la conexión con:

```
let socket = new WebSocket("ws://localhost:8080");
```

Como se puede ver se le pasa como argumento la URL del servidor con el websocket.

Con este objeto se pueden mandar textos al servidor usando el método send:

```
socket.send("Hola mundo");
```

Como se ve en el ejemplo, el objeto socket tiene diversos eventos que se deben capturar:

- Evento “onopen” que se dispara cuando se conecta por primera vez con el servidor:

```
socket.onopen = function(e) {
  alert("[onopen] Conexión abierta. Mandando datos al servidor");
  socket.send("Hola mundo");
};
```

- Evento “onmessage” que se dispara cada vez que llega un mensaje del cliente:

```
socket.onmessage = function(event) {
  const mensaje = event.data
  alert(`[onmessage] Mensaje recibido del servidor: ${mensaje}`);
};
```

Como se puede ver los datos que envía el cliente estarán en el objeto “event.data”.

- Evento “onclose” que se dispara cuando se cierra la conexión con el servidor:

```
socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[onclose] Conexión cerrada limpiamente, code=${event.code} reason=${event.reason}`);
  } else {
    // Servidor apagado o sin red
    alert('[onclose] La conexión se ha perdido');
  }
};
```

- Evento “onerror” cuando hay un error en la transmisión:

```
socket.onerror = function(error) {
  alert(`[onerror] ${error.message}`);
};
```

13.3 Una aplicación de chat de ejemplo

Para aclarar conceptos se va a hacer una aplicación que simula un pequeño chat. Los clientes escribirán mensajes y el servidor los reenviará a los clientes.

Como curiosidad los datos se serializarán usando JSON.

Con el método:

```
let objeto = JSON.parse(texto)
```

se transformará un texto con información en formato json a un objeto Javascript desde el que se puede consultar la información almacenada en el JSON.

Para hacer el proceso inverso, de objeto a texto en formato JSON, se usará el método:

```
let texto = JSON.stringify(objeto)
```

Comenzando con el cliente, que en este caso es más simple que el servidor:

Archivo public/ws2.html:

```
<!doctype html>
<meta charset='utf-8'>

<div id='mensajes'></div>
```

```

<div>
  Nombre usuario: <input type="text" id="usuario"><br>
  Mensaje: <input type="text" id="mensaje" value="..."><br>
  <button onclick='enviar()'>Enviar</button>
</div>

<script>
  let socket = new WebSocket("ws://localhost:8080/chat");

  function enviar() {
    const usuario = document.getElementById('usuario').value
    const mensaje = document.getElementById('mensaje').value
    const msg = {usuario: usuario, mensaje: mensaje}
    socket.send(JSON.stringify(msg))
  }

  socket.onopen = function(e) {
    // En este caso no se hace nada
  };

  socket.onmessage = function(event) {
    const mensajes = JSON.parse(event.data)
    let html = ''
    for(let n = 0; n < mensajes.length; n++) {
      const mensaje = mensajes[n]
      html += "<p><b>" + mensaje.usuario + "</b>: " + mensaje.mensaje
+ "</p>";
    }
    document.getElementById('mensajes').innerHTML = html;
  };

  socket.onclose = function(event) {
    if (event.wasClean) {
      alert(`[onclose] Conexión cerrada limpiamente, code=${
{event.code} reason=${event.reason}`);
    } else {
      // Servidor apagado o sin red
      alert(`[onclose] La conexión se ha perdido`);
    }
  };

  socket.onerror = function(error) {
    alert(`[onerror] ${error.message}`);
  };
</script>

```

Este HTML es una simple modificación del anterior ejemplo.

Destacar que ahora la conexión se realiza con la URL “ws://localhost:8080/chat”:

```
let socket = new WebSocket("ws://localhost:8080/chat");
```

Se añade un formulario para coger el usuario y el mensaje que se desea enviar y cuando se hace clic en el botón “Enviar”, se ejecuta la función “enviar()” que envía el mensaje al servidor:


```
function enviar() {
    const usuario = document.getElementById('usuario').value
    const mensaje = document.getElementById('mensaje').value
    const msg = {usuario: usuario, mensaje: mensaje}
    socket.send(JSON.stringify(msg))
}
```

Se puede ver que sólo toma la información del formulario, la introduce en el objeto msg y se convierte en texto usando el formato JSON.

Cuando se recibe un mensaje del servidor:

```
socket.onmessage = function(event) {
    const mensajes = JSON.parse(event.data)
    let html = ''
    for(let n = 0; n < mensajes.length; n++) {
        const mensaje = mensajes[n]
        html += "<p><b>" + mensaje.usuario + "</b>: " + mensaje.mensaje + "</p>";
    }
    document.getElementById('mensajes').innerHTML = html;
};
```

El servidor envía un JSON con todos los mensajes que se han enviado hasta el momento en una lista de la forma:

```
[
    {usuario: 'Pepe', mensaje: 'Hola'},
    {usuario: 'Lucho', mensaje: 'Adios'},
]
```

Se recorre con un bucle for la lista de los mensajes construyendo el listado de mensajes. Finalmente, se muestran los mensajes en el div con id “mensajes”.

El servidor que se ha escrito en este caso es:

Archivo ws2.js:

```
var express = require('express');
var app = express();
var expressWs = require('express-ws')(app);

app.use(express.static('public'));

app.get('/', function(req, res){
    res.send('<a href="ws2.html">Chat</a>');
});

const mensajes = []
const sockets = []

app.ws('/chat', function(ws, req) {
    sockets.push(ws)
    ws.on('message', function(msg) {
        console.log(msg);
        const json = JSON.parse(msg)
        console.log(json.usuario, json.mensaje)
        mensajes.push(json)
        for(let n = 0; n < sockets.length; n++)
```

```

        sockets[n].send(JSON.stringify(mensajes))
    });
    ws.on('close', () => {
        console.log('Socket cerrado')
        for(let n = 0; n < sockets.length; n++) {
            if(sockets[n] == ws) {
                console.log('Borrando el socket de la lista')
                sockets.splice(n, 1)
            }
        }
    })
});
app.listen(8080);

```

Este ejemplo es muy parecido al ejemplo que se puso anteriormente sobre servidores websocket. En este caso se han añadido una serie de modificaciones importantes:

Existen dos listas en las que se van a guardar el listado de mensajes que se han recibido y los sockets con los clientes que se han conectado al servidor:

```

const mensajes = []
const sockets = []

```

Cada vez que se crea un nuevo websocket, se añade a la lista “sockets”:

```

app.ws('/chat', function(ws, req) {
    sockets.push(ws)

```

Cada vez que se recibe un mensaje, se añade a la lista de mensajes:

```

ws.on('message', function(msg) {
    ...
    const json = JSON.parse(msg)
    ...
    mensajes.push(json)

```

Una vez añadido, se reenvía el listado de mensajes a cada cliente usando el listado de sockets:

```

        for(let n = 0; n < sockets.length; n++)
            sockets[n].send(JSON.stringify(mensajes))

```

También se añade un código para borrar el socket del listado de sockets cuando se cierre:

```

ws.on('close', () => {
    console.log('Socket cerrado')
    for(let n = 0; n < sockets.length; n++) {
        if(sockets[n] == ws) {
            console.log('Borrando el socket de la lista')
            sockets.splice(n, 1)
        }
    }
})

```

Este ejemplo es muy simple, aunque tiene numerosos errores como, por ejemplo, que se pueden enviar mensajes hasta llenar la memoria RAM del servidor y clientes, o se pueden hacer suplantaciones entre los clientes.

13.4 Conexiones desde otros lenguajes y routers

En los ejemplos anteriores se han hecho conexiones desde un cliente y servidor usando Javascript. Desde NodeJS también se pueden hacer conexiones como cliente a un servidor con websockets. Existen bibliotecas que permiten realizar websockets desde Java, Python,... de forma que se puede crear un servidor con express y enviar informaciones a clientes en Android, aplicaciones de escritorio o aplicaciones escritas en Electron.

13.5 Routers

Se pueden usar routers al igual que se hace con las peticiones GET o POST, por ejemplo si el código estuviese en nuestro código principal:

```
var router = express.Router();

router.ws('/echo', function(ws, req) {
  ws.on('message', function(msg) {
    ws.send(msg);
  });
});

app.use("/conexion", router);
```

En el caso de estar en un archivo aparte, el código completo sería:

app.js:

```
const express = require('express')
const app = express()
const expressWs = require('express-ws')(app);

// Se consulta la ruta de la petición
app.use(function(req, res, next) {
  console.log('Conectando con ' + req.url)
  next()
})

// Se cargan las páginas web estáticas
const path = require('path')
app.use('/', express.static(path.join(__dirname, 'public')))

/*
// Si se descomenta esta código sólo se permitirá el acceso al websocket
app.use(function(req, res, next) {
  if(req.url === '/ruta/.websocket') next()
  else next(404)
})
*/
```

```

})
*/

// Se cargan las rutas
var rutaRouter = require('./routes/ruta');
app.use('/ruta', rutaRouter)

app.listen(8080)

```

router/ruta.js:

```

var express = require('express');
var router = express.Router();
var expressWs = require('express-ws')(router)

router.get('/', function(req, res) {
  console.log('Petición GET')
  res.send('Petición GET')
})

router.ws('/', function (ws, req) {
  console.log("Socket creado")
  ws.on('message', function (msg) {
    console.log("Recibido " + msg)
    ws.send(msg)
  });
  ws.on('close', () => {
    console.log('Socket cerrado')
  })
});

module.exports = router;

```

public/ejemplo.html:

```

<!doctype html>
<meta charset='utf-8'>
<script>
let socket = new WebSocket("ws://localhost:8080/ruta");

socket.onopen = function(e) {
  alert("[onopen] Conexión abierta. Mandando datos al servidor");
  socket.send("Hola mundo");
};

socket.onmessage = function(event) {
  const mensaje = event.data
  alert(`[onmessage] Mensaje recibido del servidor: ${mensaje}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {

```

```

    alert(`[onclose] Conexión cerrada limpiamente, code=${event.code} reason=${
{event.reason}}`);
  } else {
    // Servidor apagado o sin red
    alert(`[onclose] La conexión se ha perdido`);
  }
};

socket.onerror = function(error) {
  alert(`[onerror] ${error.message}`);
};
</script>

```

Importante: En el caso de querer hacer control de acceso con los websockets, hay que tener en cuenta que para express las rutas de los websockets son de la forma “/ruta/.websocket”, por ejemplo, para permitir sólo la conexión al websockets con url “ws://localhost:8080/ruta” se usaría la ruta en express “/ruta/.websocket”:

```

app.use(function(req, res, next) {
  if(req.url === '/ruta/.websocket') next()
  else next(404)
})

```

13.6 Cifrando conexiones WebSocket con wss

Las conexiones WebSocket se pueden cifrar de usando la clave privada y el certificado que se usa para cifrar una conexión https:

Archivo wss.js:

```

const express = require("express");
//const http = require ("http");
const https = require ("https");
const fs = require("fs");
const app = express();

const httpsKeys = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.cert")
};
server = https.createServer(httpsKeys, app);
server.listen(3000);

const expressWs = require('express-ws')(app, server);

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.send('Hola mundo')
})

app.ws('/', function(ws, req) {

```

```

ws.on('message', function(msg) {
  console.log(msg);
  ws.send(msg)
});
ws.on('close', () => {
  console.log('Socket cerrado')
})
console.log('socket', req.testing);
});

//http.createServer(app).listen(3000);

```

Como se puede ver en el código, antes de crear la conexión WebSocket, se debe crear el servidor https:

```

const https = require("https");
...
const httpsKeys = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.cert")
};
server = https.createServer(httpsKeys, app);
server.listen(3000);

```

Una vez creado el servidor, se le pasa como argumento a “express-ws”:

```

const expressWs = require('express-ws')(app, server);

```

En la parte cliente sólo hay que cambiar la URL de la conexión WebSocket de ws a wss:

Archivo public/wss.html:

```

<!doctype html>
<meta charset='utf-8'>
<script>
let socket = new WebSocket("wss://localhost:3000");

socket.onopen = function(e) {
  alert("[onopen] Conexión abierta. Mandando datos al servidor");
  socket.send("Hola mundo");
};

socket.onmessage = function(event) {
  const mensaje = event.data
  alert(`[onmessage] Mensaje recibido del servidor: ${mensaje}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[onclose] Conexión cerrada limpiamente, code=${event.code} reason=${event.reason}`);
  } else {
    // Servidor apagado o sin red
    alert(`[onclose] La conexión se ha perdido`);
  }
};

socket.onerror = function(error) {

```

```
    alert(`[onerror] ${error.message}`);  
};  
</script>
```