Node.js

en el desarrollo de aplicaciones web y servicios

7. El framework Express (III)
Plantillas y vistas. Estructura de una aplicación Express

Ignacio Iborra Baeza

Índice de contenidos

1. Servir contenido estático desde Express

Hasta ahora sólo hemos utilizado Express como proveedor de servicios REST, enviando información al cliente en formato JSON, normalmente. En la actualidad que envuelve al desarrollo web, éste es uno de los usos más frecuentes que se le da no sólo a Express, sino al desarrollo *backend* en general.

Sin embargo, el otro uso frecuente consiste en que desde el *backend* se definan las vistas y el código que se va a mostrar en el cliente. Por ello, en esta sesión vamos a aprender a emplear Node y Express para procesar contenido estático y generar vistas con contenido dinámico, complementando así el desarrollo de servicios visto en sesiones previas.

Veremos que con Express la gestión del contenido estático (páginas HTML, hojas de estilo CSS, imágenes, etc) es muy sencilla, a través del correspondiente *middleware* que incorporaremos a nuestros proyectos que lo requieran. Para ello, haremos una prueba en un proyecto llamado "PruebaExpressEstatico" en nuestra carpeta de "ProyectosNode/Pruebas".

1.1. Ubicación del contenido estático

A la hora de ubicar el contenido estático de nuestra aplicación, es habitual dejarlo todo en una única subcarpeta, que típicamente se suele llamar "public". Imaginemos que esta subcarpeta tiene esta estructura:

La imagen "logo_node.png" puede ser cualquiera que queráis. En este ejemplo usaremos un logo de Node buscado en Internet:

La hoja de estilos tiene algunos estilos básicos (tipo de letra y color de fondo):

```
body
{
    background-color:rgb(245, 244, 201);
    font-family: Arial;
```

```
}
```

La página "index.html" es muy simple, incorporando la hoja de estilos, la imagen, un encabezado de primer nivel y un enlace a la otra página:

```
<!doctype html>
<html>
    <head>
        <link href="css/estilos.css" rel="stylesheet">
    </head>
    <body>
        <div align="center">
            <img src="imgs/logo_node.png" />
        </div>
        <h1>Bienvenido a Node.js</h1>
        <p><a href="noticias.html">Noticias de node</a></p>
    </body>
</html>
```

La página de "noticias.html" es similar a la anterior, mostrando un enlace a la página principal y un listado de ejemplo:

```
<!doctype html>
<html>
    <head>
        <link href="css/estilos.css" rel="stylesheet">
    </head>
    <body>
        <div align="center">
            <img src="imgs/logo_node.png" />
        </div>
        <h1>Bienvenido a Node.js</h1>
        <p><a href="index.html">Página de inicio</a></p>
        <ul>
            <li>Concurso Javascript para adictos a Node.js</li>
            <li>LinkedIn migra desde Rails hacia Node</li>
            <li>Conferencia Node.js en Italia</li>
        </ul>
    </body>
</html>
```

1.2. Procesamiento del contenido estático

Si queremos que se sirvan automáticamente estos contenidos al acceder a una URI concreta (que puede ser la propia uri "/public" u otra), empleamos el *middleware* `static`, integrado en Express. Como primer parámetro, le indicamos qué URI queremos utilizar para servir contenido estático (la que nosotros queramos), y en segundo lugar, cargamos e l *middleware* indicando en qué carpeta están realmente dichos contenidos (carpeta "/public" en nuestro caso):

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

let app = express();
```

```
app.use('/public', express.static(__dirname + '/public'));

app.listen(8080);
```

Opcionalmente, también podemos definir una ruta que redirija a la página de inicio ("index.html") si se intenta acceder a la raíz de la aplicación:

```
app.get('/', (req, res) => {
    res.redirect('/public/index.html');
});
```

Sin embargo, esta forma de servir contenido estático se queda muy corta. No hay casi ninguna web que ofrezca un contenido sin cierto dinamismo; el propio listado de noticias podría extraerse de una base de datos y volcarse en la página... pero para eso necesitamos hacer uso de plantillas.

2. Uso de plantillas y vistas

Una plantilla es un documento estático (típicamente HTML, si hablamos de documentos web), en el que se intercalan o embeben ciertas marcas para agregar algo de dinamismo. Por ejemplo, podemos dejar una estructura HTML hecha con un hueco para mostrar un listado de noticias, y que ese listado de noticias se extraiga de una base de datos y se añada a la plantilla dinámicamente, antes de mostrarla.

2.1. Motores de plantillas

Existen varios motores de plantillas que podemos emplear en Express, y que facilitan y automatizan el procesamiento de estos ficheros y el reemplazo de las correspondientes marcas por el contenido dinámico a mostrar. Algunos ejemplos son:

- **Jade**, un motor bastante habitual con una sintaxis específica basada en HAML, una abstracción del propio lenguaje HTML. Recientemente, este motor de plantillas ha cambiado su nombre a **Pug**. En la propia documentación de Express se indica que se puede seguir utilizando Jade, pero que las actualizaciones más recientes afectarán ya a Pug.
- **Mustache**, un motor con una sintaxis que combina HTML con código Javascript embebido con cierta sintaxis especial. A partir de este motor, se han creado otros muy similares, como **HBS**, también conocido como *Handlebars*.
- **EJS**, siglas de *Effective Javascript templating*, un motor de plantillas bastante sencillo de utilizar e integrar con contenido HTML.
- etc.

2.1.1. Instalación del motor de plantillas

Una vez hayamos elegido nuestro motor de plantillas, lo instalaremos en nuestra aplicación como un módulo más de NPM, y lo enlazaremos con Express a través del método `app.set`, como una propiedad de la aplicación. En estos apuntes, haremos uso del motor EJS, por su sencillez de uso. Así que lo descargamos de NPM con el correspondiente comando (desde la carpeta del proyecto Node)...

```
npm install ejs
```

… y lo establecemos como motor de plantillas con el comando set:

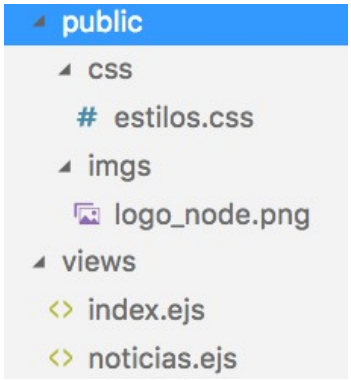
```
let app = express();
...
app.set('view engine', 'ejs');
```

2.1.2. Ubicación de las plantillas

Por defecto, en una aplicación Express las plantillas se almacenan en una subcarpeta llamada "views" dentro del proyecto Node. En nuestro caso, al haber escogido el motor EJS, dichas plantillas tendrán extensión *.ejs*, como por ejemplo "index.ejs".

2.2. El motor de plantillas EJS

Para ver algunas nociones básicas de este motor de plantillas, vamos a trasladar los dos archivos HTML estáticos que hemos definido antes en nuestro proyecto de pruebas "PruebaExpressEstatico" ("index.html" y "noticias.html") a la carpeta "views", renombrándolos a "index.ejs" y "noticias.ejs", respectivamente.



Después, retocamos el archivo "index.ejs" para dejarlo de este modo:

```
<!doctype html>
<html>
    <head>
        <link href="/public/css/estilos.css" rel="stylesheet">
    </head>
    <body>
        <div align="center">
            <img src="/public/imgs/logo_node.png" />
        </div>
        <h1>Bienvenido a Node.js</h1>
        <p><a href="/noticias">Noticias de node</a></p>
    </body>
</html>
```

Lo que hemos hecho es cambiar las rutas a los contenidos estáticos (CSS e imagen) para anteponerles el prefijo "/public", y modificar la ruta a la URL de "/noticias", porque ya no existe "noticias.html".

El archivo de "noticias.ejs", por su parte, queda de esta otra forma:

```
<!doctype html>
<html>
    <head>
        <link href="/public/css/estilos.css" rel="stylesheet">
    </head>
    <body>
        <div align="center">
            <img src="/public/imgs/logo_node.png" />
        </div>
        <h1>Noticias de Node</h1>
        <p><a href="/">Página de inicio</a></p>
        <ul>
            <li>Concurso Javascript para adictos a Node.js</li>
```

```
            <li>LinkedIn migra desde Rails hacia Node</li>
            <li>Conferencia Node.js en Italia</li>
        </ul>
    </body>
</html>
```

También hemos modificado las rutas a los contenidos estáticos y el enlace a la página principal. Finalmente, eliminamos en "app.js" la ruta que habíamos definido para redirigir a "index" desde la carpeta principal:

```
app.get('/', (req, res) => {
    res.redirect('/public/index.html');
});
```

y definimos un par de rutas al servidor para que se muestren las correspondientes vistas:

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

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

Observad cómo utilizamos el método `render` de la respuesta para especificar la vista a cargar en cada caso, y que no hace falta especificar la extensión del archivo (lo que viene bien, si en el futuro cambiamos de motor de plantillas pero mantenemos los nombres de archivo). Si reiniciamos el servidor ahora, la aplicación debería funcionar tal cual la dejamos en el anterior apartado.

Lo que hemos hecho hasta ahora tampoco tiene mucho valor: hemos trasladado un contenido HTML estático de una carpeta a otra, y definido dos servicios para cargar una u otra vista. Ahora vamos a añadir el contenido dinámico a las vistas, para así poder empezar a ver la utilidad real de estas plantillas.

2.2.1. Añadir contenido dinámico simple

Si queremos añadir cualquier contenido dinámico a la plantilla, utilizaremos una sintaxis similar a la de otros lenguajes de servidor como PHP o JSP; en este caso emplearemos los símbolos **<%** y **%>** para englobar el contenido dinámico, y dentro de ellos, pondremos código Javascript. Por ejemplo, podemos modificar la plantilla principal "index" para que muestre la fecha actual:

```
...
<h1>Bienvenido a Node.js</h1>
<p><%= new Date().toString(); %></p>
<p><a href="/noticias">Noticias de node</a></p>
...
```

En este caso, utilizamos **<%= … %>** para directamente mostrar el contenido de la expresión generada, igual que se hace en PHP o JSP.

También podemos emplear otras estructuras de código Javascript dentro de ese contenido dinámico, como por ejemplo condicionales o bucles:

```
<% if (!usuario) { %>
    <p>Usuario no identificado</p>
```

```
<% } %>
```

2.2.2. Pasar contenido dinámico desde los servicios

Vamos ahora a modificar la plantilla de noticias. En lugar de mostrar un listado de noticias estático, vamos a pasarle desde el servicio un vector de noticias para mostrar. Por simplificar, vamos a definir el vector directamente en el código, pero también podemos obtener el listado de noticias de una base de datos o un fichero de texto:

```
app.get('/noticias', (req, res) => {
    let listadoNoticias = [
        {titulo:"LinkedIn se pasa a Node",
         texto: "LinkedIn migra desde Rails hacia Node"},
        {titulo:"Conferencia de Node",
         texto: "El próximo verano habrá una conferencia de Node en Italia"}
    ];
    res.render('noticias', {noticias: listadoNoticias});
});
```

Observad que, para pasar el listado de noticias (obtenido de cualquier fuente) a la vista, utilizamos un segundo parámetro del método `render`, donde especificamos todos los datos que enviamos a la vista como un objeto Javascript.

Ahora, en la vista de noticias, reemplazamos la lista fija de noticias por este otro código:

```
<ul>
    <% noticias.forEach(noticia => { %>
        <li>
            <strong><%=noticia.titulo%></strong>
            : 
            <%=noticia.texto%>
        </li>
    <% }); %>
</ul>
```

Lo que hacemos es recorrer el objeto noticias que nos llega desde el servidor, y para cada noticia mostrar un elemento de lista, con el título en negrita y el texto a continuación.

2.2.3. Incluir unas plantillas en otras

Hasta ahora hemos definido un par de plantillas nada más, pero podemos darnos cuenta de un problema que se nos presenta... ¿qué pasaría si, en un futuro, quisiéramos cambiar el diseño de nuestra web, o añadir otra imagen al encabezado? Tendríamos que editar plantilla a plantilla su código, y cambiar la ruta de la imagen, o añadir nuevos enlaces CSS.

Para evitar esa tarea tediosa, también podemos incluir subvistas dentro de otras, y así, podemos definir la estructura general del encabezado y pie de nuestras páginas, y luego incluirlas en cada vista principal, que sólo deberá encargarse de mostrar su contenido.

Por ejemplo, vamos a definir una vista llamada "cabecera.ejs" en nuestra carpeta de vistas, con el siguiente contenido:

```
<!doctype html>
<html>
    <head>
        <link href="/public/css/estilos.css" rel="stylesheet">
    </head>
    <body>
```

```
        <div align="center">
            <img src="/public/imgs/logo_node.png" />
        </div>
```

Definimos también otra vista llamada "pie.ejs" que simplemente cierre el contenido HTML:

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

Después, reemplazamos ese mismo contenido en las vistas "index" y "noticias" por una llamada a la función `include`, que se encarga de incorporar la vista que le digamos. Por ejemplo, en el caso de "index", el código quedaría así:

```
<%- include('cabecera'); %>
<h1>Bienvenido a Node.js</h1>
<p><%= new Date().toString(); %></p>
<p><a href="/noticias">Noticias de node</a></p>
<%- include('pie'); %>
```

De este modo, cualquier cambio en la cabecera se actualizará automáticamente en todas las vistas que la utilicen, del mismo modo que ocurre con la instrucción `include` en lenguajes como PHP.

3. Estructura de una aplicación Express

Express es un framework que no está orientado hacia ninguna arquitectura en particular, a diferencia de otros que son más específicos para arquitecturas MVC, o SPA. Por lo tanto, existen distintas formas de estructurar los archivos y el código de nuestros proyectos en Express, y en principio ninguna es mejor que otra. De hecho, la propia web de Express lo califica como un framework opuesto a inclinarse por una opinión (*unopinionated*), dando libertad al desarrollador para elegir entre diferentes alternativas para abordar los proyectos software.

Sin embargo, sí puede venir bien tener una guía por la que empezar a desarrollar proyectos complejos, aunque luego la experiencia nos oriente a nuestra propia forma de crear esos proyectos. A lo largo de las sesiones previas ya hemos ido encaminando nuestros proyectos hacia esa estructura, pero es hora de sintetizarlo todo.

3.1. Estructura típica de una aplicación

Recopilando todo lo visto hasta ahora en las diferentes aplicaciones y ejemplos de Express que hemos hecho, podemos intuir una estructura de aplicación que, si bien no es obligatoria (ni siquiera estándar), sí es una de las más utilizadas por la comunidad desarrolladora.

Dicha estructura consta de:

- El programa principal (servidor), en un archivo **`app.js`**
- Una subcarpeta **`public`** con todo el contenido estático que se mostrará tal cual (hojas de estilo, archivos Javascript de la parte cliente, imágenes...)
- Una subcarpeta **`views`** con las vistas y plantillas empleadas para renderizar vistas por parte de Express, combinadas con la parte pública anterior.
- Una subcarpeta **`routes`** con los diferentes enrutadores, en el caso de que decidamos dividir las rutas de nuestra aplicación en diferentes módulos para tratarlas de forma independiente y no aglutinarlas todas en la aplicación principal. Alternativamente, esta carpeta también suele denominarse **`controllers`** en algunos ejemplos que podéis encontrar por Internet.
- Una subcarpeta **`models`** con los modelos de datos, que luego se mapearán en tablas o colecciones (dependiendo de si usamos bases de datos SQL o NoSQL, respectivamente).

3.1.1. Una alternativa: combinar esquemas y rutas

También podemos encontrar en Internet varios ejemplos que combinan un esquema de base de datos (una tabla o colección) con el correspondiente enrutador que gestiona las rutas que acceden a dicha tabla o colección. Así, en lugar de las carpetas "models" y "routes", tendremos una carpeta por cada esquema, con el nombre del mismo, y dentro encontraremos dos archivos: uno llamado "schema.js" con el esquema asociado, y otro llamado "routes.js" con las rutas. Por ejemplo, para nuestra aplicación de contactos, tendríamos las siguientes carpetas (aparte del servidor principal "app.js"):

- Carpetas "public" y "views" (estas suelen ser fijas en casi cualquier estructura de proyecto que encontréis)

- Carpeta "mascota" con los archivos "schema.js" y "routes.js" relativos a las mascotas
- Carpeta "restaurante" con los archivos "schema.js" y "routes.js" relativos a los restaurantes
- Carpeta "contacto" con los archivos "schema.js" y "routes.js" relativos a los contactos

4. Ejemplo completo

Vamos a desarrollar un ejemplo completo de aplicación Express que utilice vistas con contenido dinámico extraído de una base de datos. Nos basaremos en la base de datos de contactos que ya hemos realizado en sesiones previas, y construiremos una aplicación que gestione parcialmente su contenido (ya que hacerlo en su totalidad puede ser algo más largo). En concreto, listaremos, insertaremos y borraremos contactos, y definiremos sus restaurantes y mascotas.

Tenéis una base con la que comenzar, llamada "PruebaContactosExpressEJS_base", y podéis consultar también el ejemplo completo desarrollado en "PruebaContactos ExpressEJS", para contrastarlo con vuestro código en el caso de que sigáis los pasos y algo no funcione.

4.1. Estructura del proyecto

El proyecto está estructurado de una forma similar a la ya vista en los ejemplos anteriores:

- En la carpeta **models** tenemos los modelos de datos para contactos, mascotas y restaurantes, vistos en sesiones anteriores.
- En la carpeta **public** está todo el contenido estático: dos archivos CSS, una imagen, y dos archivos Javascript, que se utilizarán desde las plantillas. Esta carpeta también está completa y no tendremos que tocarla.
- En la carpeta **routes** definiremos los enrutadores. Hay un esqueleto hecho para cada enrutador (mascotas, restaurantes y contactos, además de un "index" que veremos a continuación), pero en este caso no emplearemos los mismos servicios que en sesiones anteriores, ya que en algunos servicios no vamos a devolver datos JSON, sino a renderizar vistas desde estas rutas.
- En la carpeta **views** tenemos las vistas disponibles. La "cabecera.ejs" y el "pie.ejs" ya están completos y definidos con el código que se incorporará en todas las demás vistas, al principio y al final. Además, tenemos también completa la vista principal "index.ejs", y el formulario de alta de "nuevo_contacto.ejs", "nuevo_restaurante.ejs" y "nueva_mascota.ejs", ya que todo el contenido de estas vistas es estático (de momento). Quedan por implementar las vistas "ficha_contacto.ejs" y "lista_contactos.ejs", que completaremos más tarde.
- El servidor principal "app.js" también está implementado: carga las librerías principales (aún por instalar), los enrutadores, conecta con la base de datos, define el *middleware* para contenido estático, el *body-parser*, las rutas, carga el motor de plantillas y pone en marcha el servidor. Todo esto son cosas que ya hemos visto en ejemplos anteriores.

4.1.1. Instalación de módulos necesarios

Antes de empezar, y dado que tenemos archivo "package.json" pero no tenemos la carpeta "node_modules", debemos instalar con `npm install` los módulos que necesitaremos: *mongoose, express, body-parser* y el motor de plantillas *ejs*.

```
npm install
```

4.2. Rutas para vistas estáticas

En la carpeta "views" existen algunas vistas que sólo ofrecen contenido estático: la página principal "index", y los formularios "nuevo_contacto", "nuevo_restaurante" y "nueva_mascota". Podemos mapear cada una de esas vistas con una ruta en el servidor principal "app.js", pero en este caso vamos a hacer uso de un enrutador especial, ubicado en el archivo "routes/index.js". De esta forma, aislaremos por completo las rutas del servidor principal. Este archivo quedará así por el momento:

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

let router = express.Router();

router.get('/', (req, res) => {
    res.render('index');
});

router.get('/nuevo_contacto', (req, res) => {
    res.render('nuevo_contacto');
});

router.get('/nuevo_restaurante', (req, res) => {
    res.render('nuevo_restaurante');
});

router.get('/nueva_mascota', (req, res) => {
    res.render('nueva_mascota');
});

module.exports = router
```

Si ponemos en marcha el servidor (y la base de datos MongoDB), podremos cargar ya la página principal y navegar entre los formularios y la página principal.

4.3. Inserciones de restaurantes y mascotas

Vamos ahora con otro paso sencillo: procesar las inserciones de restaurantes y mascotas desde sus respectivos formularios. Si observamos el código fuente de esas vistas, al hacer clic en el botón de *Enviar* de cada formulario, se llama respectivamente a una función "nuevoRestaurante()" o "nuevaMascota()", cuyo código está definido en el archivo "public/js/app.js". Sin entrar en mucho detalle, ya que no tocaremos el código de estas funciones, lo que hacen es recoger los datos del formulario y enviarlos por POST en formato JSON a las rutas "/restaurantes" y "/mascotas", respectivamente. Después, recogen el resultado de la operación, y dependiendo de si es exitosa o no, muestran un panel de resultado sobre el formulario. Todo esto está implementado apoyándonos en la librería jQuery para simplificar el proceso de llamadas asíncronas, y no es necesario retocarlo, ya que no forma parte de los contenidos del curso.

Lo que debemos hacer es implementar los servicios POST sobre esas rutas "/restaurantes" y "/mascotas", para devolver el resultado requerido. Comencemos por los restaurantes (archivo "routes/restaurantes.js"). El servicio sería algo así:

```
router.post('/', (req, res) => {
    let nuevoRestaurante = new Restaurante({
        nombre: req.body.nombre,
        direccion: req.body.direccion,
        telefono: req.body.telefono
```

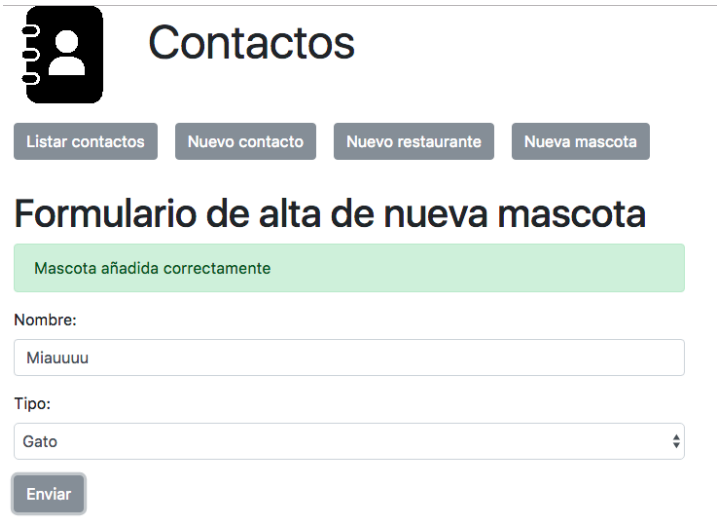
```
    });
    nuevoRestaurante.save().then(resultado => {
        if (resultado)
            res.send({error: false});
        else
            res.send({error: true});
    }).catch(error => {
        res.send({error: true});
    })
});
```

Lo que se hace es recoger los datos del restaurante, hacer la inserción y devolver un booleano en el campo `error` en función de si ha sido exitosa o no.

Para el caso de las mascotas ("routes/mascotas.js") el funcionamiento es similar:

```
router.post('/', (req, res) => {
    let nuevaMascota = new Mascota({
        nombre: req.body.nombre,
        tipo: req.body.tipo
    });
    nuevaMascota.save().then(resultado => {
        if (resultado)
            res.send({error: false});
        else
            res.send({error: true});
    }).catch(error => {
        res.send({error: true});
    })
});
```

Ahora ya podremos añadir restaurantes y mascotas desde estos formularios, y comprobar desde esas mismas vistas si la operación ha ido bien o no:



4.4. Listado y ficha de contactos

Vayamos ahora con las rutas y vistas para el listado y la ficha de contactos. Para empezar, el **listado de contactos** renderizará la vista "lista_contactos", pasándole el listado de contactos a visualizar. El servicio puede quedar así (en "routes/contactos.js"):

```
router.get('/', (req, res) => {
    Contacto.find().then(resultado => {
```

```
        res.render('lista_contactos', {contactos: resultado});
    }).catch(error => {
        res.render('lista_contactos', {contactos: []});
    });
});
```

Observad que, a la hora de renderizar la vista, adjuntamos un parámetro llamado "contactos" con el listado de contactos. Este parámetro lo recogemos en la propia vista "lista_contactos" y lo recorremos, generando una lista de contactos en pantalla:

```
<%- include('cabecera'); %>
<h1>Listado de contactos</h1>
<%
if (contactos.length == 0) {
%>
    <p>No se han encontrado contactos que mostrar</p>
<%
} else {
%>
    <ul>
    <%
    contactos.forEach(contacto => {
    %>
        <li><%=contacto.nombre%>
            <a href="/contactos/<%=contacto._id%>">Ver ficha</a>  
            <a href="javascript:borrarContacto('<%=contacto._id%>')">Borrar</a>
        </li>
    <%
    });
    %>
    </ul>
<%
}
%>
<%- include('pie'); %>
```

Para cada contacto, mostramos su nombre, un enlace para ir a su ficha, y otro para eliminar el contacto. Estas dos opciones las implementaremos a continuación.

La **ficha del contacto** tiene asociada la vista "ficha_contacto". El servicio simplemente recibirá el *id* del contacto, lo buscará en la colección y lo devolverá a la vista, para mostrarlo. En caso de no encontrar el contacto, enviaremos un contacto nulo para detectarlo en la vista y mostrar el mensaje correspondiente.

```
router.get('/:id', (req, res) => {
    Contacto.findById(req.params.id)
    .populate('restauranteFavorito')
    .populate('mascotas')
    .then(resultado => {
        if (resultado)
            res.render('ficha_contacto', {contacto: resultado});
        else
            res.render('ficha_contacto', {contacto: null});
    }).catch (error => {
        res.render('ficha_contacto', {contacto: null});
    });
});
```

Aprovechamos también este servicio para poblar (populate) los datos del restaurante y las mascotas del contacto, y así poder mostrar esta información en la vista.

La vista asociada puede quedar así:

```
<%- include('cabecera'); %>
<h1>Ficha de contacto</h1>
<%
if (contacto == null) {
%>
    <p>Contacto incorrecto</p>
<%
} else {
%>
    <ul>
        <li><strong>Nombre:</strong> <%=contacto.nombre%></li>
        <li><strong>Teléfono:</strong> <%=contacto.telefono%></li>
        <li><strong>Edad:</strong> <%=contacto.edad%></li>
        <li><strong>Restaurante favorito:</strong>
            <%=contacto.restauranteFavorito.nombre + "(" +
               contacto.restauranteFavorito.direccion + ")"%>
        </li>
        <li><strong>Mascotas:</strong>
            <ul>
            <%
            contacto.mascotas.forEach(mascota => {
            %>
                <li><%=mascota.nombre%></li>
            <%
            });
            %>
            </ul>
        </li>
    </ul>
<%
}
%>
<%- include('pie'); %>
```

4.5. Borrado de contactos

El borrado de contactos, al que accedemos desde el enlace correspondiente del listado de contactos, llamará una función *borrarContacto* del archivo "public/js/app.js", que básicamente enviará por DELETE el *id* del contacto a eliminar al servidor. Tras recibir la respuesta, lo que haremos será volver a renderizar la lista de contactos.

Por tanto, el servicio de borrado debe limitarse a eliminar el contacto.

```
router.delete('/:id', (req, res) => {
    Contacto.findByIdAndRemove(req.params.id).then(resultado => {
        res.send({error: false});
    }).catch(error => {
        res.render({error: true});
    });
});
```

Como respuesta devolvemos un booleano indicando si ha ido bien o mal la operación, aunque en realidad no hacemos nada con este dato en el cliente en este ejemplo. Podríamos aprovecharlo para mostrar algún mensaje de error o no, pero por simplicidad nos limitaremos a recargar la lista de contactos.

4.6. Inserción de contactos

Finalmente, para la inserción de contactos, tenemos que basarnos en el formulario de "nuevo_contacto" que ya tenemos desarrollado. Este formulario llama a la función "nuevoContacto", ya implementada en "public/js/app.js", que hace algo similar a las altas de restaurantes y mascotas vistas antes: recoge los datos del contacto del formulario y los envía por POST al servicio correspondiente.

Sin embargo, antes de continuar, vamos a añadir un par de cambios en la ruta que sirve este formulario ("/nuevo_contacto" en el archivo "routes/index.js"). Lo que haremos será enviar a la vista información sobre los restaurantes y mascotas actualmente almacenados, de forma que podamos elegir en el formulario el restaurante y mascotas para el contacto a añadir. Para ello, cargaremos los modelos de restaurantes y mascotas en el enrutador "routes/index.js"...

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

let Restaurante = require(__dirname + '/../models/restaurante.js');
let Mascota = require(__dirname + '/../models/mascota.js');

let router = express.Router();

...
```

… y retocaremos el servicio GET */nuevo_contacto*, para que envíe a la vista los datos de restaurantes y mascotas. En caso de no poder obtener alguno de estos dos listados, enviaremos a la página principal "index".

```
router.get('/nuevo_contacto', (req, res) => {
    Restaurante.find().then(restaurantes => {
        Mascota.find().then(mascotas => {
            res.render('nuevo_contacto', {restaurantes: restaurantes,
                                          mascotas: mascotas});
        }).catch (error => {
            res.render('index');
        });
    }).catch(error => {
        res.render('index');
    });
});
```

La vista "nuevo_contacto" recogerá esa información y con ella añadirá dos nuevos campos en el formulario: un desplegable para elegir el restaurante, y una lista de *checkboxes* para marcar las mascotas del contacto. Esto lo podemos añadir justo antes del botón de enviar:

```
<%- include('cabecera'); %>

<h1>Formulario de alta de nuevo contacto</h1>

...

<div class="form-group">
    <label for="restauranteFavorito">Restaurante favorito:</label>
    <select class="form-control" id="restauranteFavorito"
     name="restauranteFavorito">
        <option value="" selected>--Escoge un restaurante--</option>
        <%
        restaurantes.forEach(restaurante => {
```

```
        %>
            <option value="<%=restaurante._id%>">
                <%=restaurante.nombre%>
            </option>
        <%
        });
        %>
    </select>
</div>
<div class="form-group">
    <label for="mascotas">Mascotas:</label><br />
    <%
    mascotas.forEach(mascota => {
    %>
        <input type="checkbox" name="mascotas" class="mascotas"
         value="<%=mascota._id%>" /><%=mascota.nombre %><br />
    <%
    });
    %>
</div>

<button class="btn btn-secondary" onclick="nuevoContacto()">Enviar</button>
<%- include('pie'); %>
```

Finalmente, nos queda definir el servicio POST para insertar un nuevo contacto, en el archivo "routes/contactos.js", que recibiría los datos de este formulario, desde la función "nuevoContacto" del archivo "public/js/app.js". El servicio podría quedar así:

```
router.post('/', (req, res) => {
    let nuevoContacto = new Contacto({
        nombre: req.body.nombre,
        telefono: req.body.telefono,
        edad: req.body.edad,
        restauranteFavorito: req.body.restauranteFavorito,
        mascotas: req.body.mascotas
    });
    nuevoContacto.save().then(resultado => {
        if (resultado)
            res.send({error: false});
        else
            res.send({error: true});
    }).catch(error => {
        res.send({error: true});
    });
});
```

Con esto, la aplicación ya debería estar completamente operativa, a falta de los servicios que no hemos implementado por aligerar el código de este ejemplo:

- Listado/borrado/ficha de restaurantes y mascotas
- Edición en general (modificación de contactos/restaurantes/mascotas)

5. Ejercicios

Para los ejercicios de esta sesión, deberéis crear la habitual carpeta "Sesion7" dentro de vuestra carpeta de "ProyectosNode/Ejercicios", y copiar dentro el "Ejercicio_7" que se os proporcionará como plantilla.

En esta ocasión, vamos a dedicarnos a una aplicación distinta a la de libros que venimos haciendo en sesiones anteriores, para no tener que depender de soluciones de sesiones previas. Vamos a realizar la gestión de tareas pendientes.

La estructura de la plantilla proporcionada es similar a la que se ha proporcionado para el ejemplo de los contactos:

- La carpeta **models** contiene el modelo para las tareas, de las que almacenaremos su título, fecha y prioridad (un entero de 1 a 3, que las cataloga de mayor a menor prioridad).
- La carpeta **public** con el CSS y archivos Javascript para la parte del cliente, incluyendo funciones jQuery para envío de formularios y borrado ya hechas.
- La carpeta **routes** con los enrutadores "tareas", e "index", que implementaremos a continuación
- La carpeta **views** con las vistas "cabecera", "pie", "index" y "nueva_tarea" ya hechas, a falta de completar las vistas de "lista_tareas" y "ficha_tarea".
- Finalmente, tenemos el servidor principal **app.js**, parcialmente implementado.

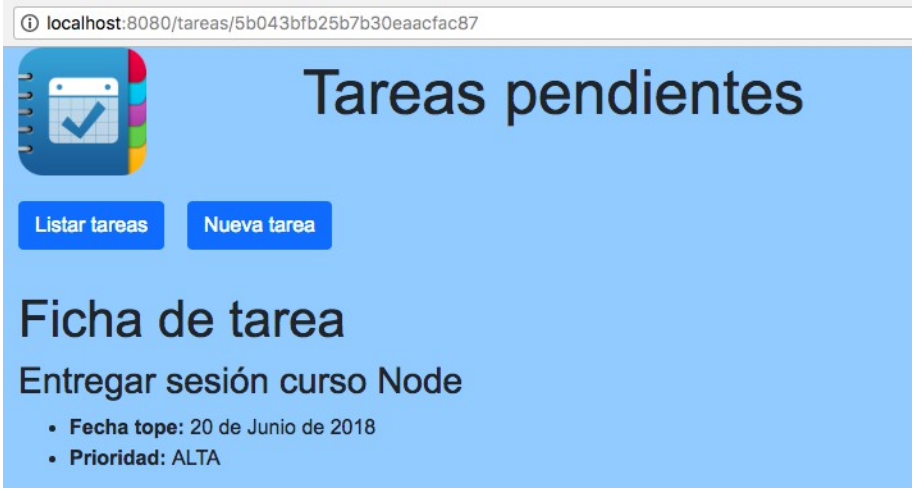
Vamos a completar la aplicación para poder dar de alta tareas, listarlas, ver su ficha resumen y borrarlas. Sigamos estos pasos:

1. Antes de nada, instala los módulos necesarios (*express, mongoose, body-parser* y *ejs*). Ya tienes los módulos incluidos en el archivo "package.json", por lo que bastará con hacer `npm install`.
2. Comencemos por completar el servidor principal "app.js": dicho servidor incorpora los módulos requeridos y los enrutadores, conecta a la base de datos, e inicializa Express. Falta por hacer lo siguiente antes de poner en marcha el servidor:
   - Cargar el motor de plantillas EJS
   - Definir la carpeta de contenido estático "/public"
   - Cargar el *middleware* body-parser, para formato JSON
   - Mapear la ruta "/" con el enrutador "index"
   - Mapear la ruta "/tareas" con el enrutador "tareas"
3. Edita el archivo "routes/index.js" y añade las rutas hacia las vistas de contenido estático:
   - **GET /** renderizará la vista "index"
   - **GET /nueva_tarea** renderizará la vista "nueva_tarea"
4. Edita el archivo "routes/tareas.js" y añade las rutas para la gestión de tareas. Teniendo en cuenta que el prefijo "/tareas" ya viene incorporado desde el servidor principal, tendremos que definir las rutas relativas a dicho prefijo, de este modo:

- **GET** */* renderizará la vista "lista_tareas", pasándole como parámetro el listado de tareas obtenido de la base de datos
- **GET** */:id* renderizará la vista "ficha_tarea", pasándole como parámetro la tarea cuyo *id* se especifica en la URI, o *null* si no se encuentra la tarea
- **POST** */* obtendrá los datos de la tarea del cuerpo de la petición (parámetros llamados "titulo", "fecha" y "prioridad"), dará de alta la nueva tarea y devolverá un parámetro `error` que será `true` o `false` dependiendo de si ha habido un error o no al realizar la inserción.
- **DELETE** */:id* eliminará de la colección la tarea cuyo *id* se especifique. Devolverá también un parámetro `error` con el resultado de la operación (`true` o `false`)

5. Completa la vista "lista_tareas" para mostrar un listado de tareas que recibirá de la correspondiente ruta *GET /tareas*. El formato de la lista es irrelevante, pero deberá mostrar el título, y dos enlaces para ver la ficha de la tarea y eliminarla. Por ejemplo, puede quedarte así:

6. Completa la vista "ficha_tarea" para mostrar la ficha de la tarea que recibe como parámetro. Si es nula, mostrará algún mensaje indicando que no se ha encontrado la tarea. Si no, puedes mostrar la información con el formato que te parezca. Por ejemplo, así:



7. (OPCIONAL) Para finalizar, puedes añadir dos enlaces/botones en la vista de "lista_tareas" que permitan ordenar el listado ascendentemente por prioridad y por fecha, respectivamente. Deberás añadir el/los servicio(s) correspondientes en "routes/tareas.js" para obtener el listado según el criterio de orden elegido.