Motores de plantillas: Nunjucks

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.

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:

- **Pug**, un motor bastante habitual en Express, con una sintaxis específica basada en HAML, una abstracción del propio lenguaje HTML.
- **Mustache**, un motor 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.
- **Nunjucks**, un motor de plantillas desarrollado por el equipo de Mozilla, que también se integra perfectamente en Express.
- etc.

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 **Nunjucks**, un motor de plantillas desarrollado por Mozilla, muy similar a *Handlebars* en su sintaxis, y adaptado a su uso con Express. Podéis consultar más información en su web oficial.

Lo primero que haremos será descargarlo e instalarlo en nuestro proyecto con su correspondiente comando (desde la carpeta del proyecto Node):

```
npm install nunjucks
```

Lo deberemos incluir en la aplicación principal, junto con el resto de módulos necesarios...

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

Después, lo establecemos como motor de plantillas en el archivo principal de nuestra aplicación, una vez inicializada la aplicación Express:

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

Finalmente, también es necesario establecer unos parámetros de configuración del motor de plantillas, empleando para ello su método `configure`. En concreto, estableceremos que auto-escape los caracteres que se muestren (para evitar ataques por código embebido, por ejemplo), y le indicaremos qué objeto contiene la aplicación Express. Además, indicaremos que las diferentes plantillas o vistas las vamos a ubicar en la subcarpeta `views` de la aplicación:

```
nunjucks.configure('views', {
    autoescape: true,
    express: app
});
```

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. Así lo hemos configurado con la instrucción anterior `configure` para Nunjucks. En nuestro caso, al haber escogido este motor, dichas plantillas tendrán extensión `.njk`, como por ejemplo `index.njk`.

2. Primeros pasos con Nunjucks

Vamos a definir algunas plantillas con Nunjucks y comprobar cómo muestran la información dinámica, y cómo se les puede pasar dicha información desde los enrutadores.

2.1. Preparando el servidor principal

Para ello, nos basaremos en nuestro ejemplo de contactos que hemos venido desarrollando en sesiones previas. Podemos copiar la carpeta del proyecto "*ContactosREST_v2*" de sesiones anteriores (la que hicimos con toda la estructura de carpetas para modelos y enrutadores), y renombrarla a "*ContactosWeb*". Dentro instalamos Nunjucks y lo dejamos configurado en la aplicación principal como motor de plantillas para la

aplicación. También podemos instalar y configurar Bootstrap si queremos, para poder aplicar sus estilos. El archivo principal `index.js` quedará más o menos así:

```
// Librerías
const express = require('express');
const mongoose = require('mongoose');
const nunjucks = require('nunjucks');

// Enrutadores
const mascotas = require(__dirname + '/routes/mascotas');
const restaurantes = require(__dirname + '/routes/restaurantes');
const contactos = require(__dirname + '/routes/contactos');

// Conexión con la BD
mongoose.connect('mongodb://127.0.0.1:27017/contactos');

// Servidor Express
let app = express();

// Configuramos motor Nunjucks
nunjucks.configure('views', {
    autoescape: true,
    express: app
});

// Asignación del motor de plantillas
app.set('view engine', 'njk');

// Middleware para peticiones POST y PUT
// Middleware para estilos Bootstrap
// Enrutadores para cada grupo de rutas
app.use(express.json());
app.use(express.static(__dirname + '/node_modules/bootstrap/dist'));
app.use('/mascotas', mascotas);
app.use('/restaurantes', restaurantes);
app.use('/contactos', contactos);

// Puesta en marcha del servidor
app.listen(8080);
```

> **IMPORTANTE**: es importante el orden en que aplicamos el *middleware*, como hemos dicho antes. En primer lugar, cargamos `express.json()`, para que se aplique a TODAS las rutas que lo necesiten, después cargamos Bootstrap, y después los enrutadores. Si cargamos primero los enrutadores, por ejemplo, entonces no tendrán disponible ni Bootstrap ni `express.json()`.

2.2. Vista para listado general

Vamos ahora a crear la carpeta `views`, y dentro definimos una vista llamada `contactos_listado.njk`. Dentro de esta vista definiremos el código HTML que va a tener, dejando un hueco para mostrar el listado de contactos:

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" href="/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <h1>Listado de contactos</h1>
            <div>
                <!-- Aquí mostraremos el listado -->
            </div>
        </div>
    </body>
</html>
```

Para poder mostrar el listado de contactos, necesitamos proporcionar dicho listado a la vista. Esto lo haremos desde la ruta de consulta de contactos. En sesiones anteriores habíamos establecido esta ruta en el archivo `routes/contactos.js`, bajo la URI `GET /contactos`. En esa sesión devolvía los contactos en formato JSON, pero ahora vamos a decirle simplemente que muestre (renderice) la vista del listado de contactos. Así nos quedará ahora este enrutador:

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

Notar que, para mostrar una vista, basta con que indiquemos el nombre del archivo, sin la extensión. Nunjucks ya se encarga de localizar el archivo, procesar el contenido dinámico que tenga y enviar el resultado.

Sin embargo, nos falta algo en el enrutador anterior. Necesitamos poder facilitarle a la vista el listado de contactos. Para ello, utilizaremos Mongoose para obtener dicho listado, y una vez obtenido, renderizaremos la vista con `render`, pasando como segundo parámetro los datos que necesita la vista para trabajar (el listado de contactos, en este caso).

```
router.get('/', (req, res) => {
    Contacto.find().then(resultado => {
        res.render('contactos_listado', {contactos: resultado});
    }).catch(error => {
        // Aquí podríamos renderizar una página de error
    });
});
```

Finalmente, en la plantilla, podemos reemplazar el comentario que hemos dejado de "Aquí mostraremos el listado" con el listado efectivo, con este código:

```
<ul>
    {% for contacto in contactos %}
        <li>{{ contacto.nombre }}</li>
    {% endfor %}
</ul>
```

Hemos empleado la cláusula `for` de Nunjucks para iterar sobre una colección de elementos (la colección `contactos` que recibimos del enrutador). Para cada elemento, mostramos un item de lista, y el nombre de cada contacto en él. Observad la notación de la doble llave `{{ ... }}` para mostrar información de los objetos con los que estamos trabajando (en este caso, cada contacto de la lista que recibimos del enrutador).

2.3. Algunas cuestiones adicionales

La cláusula `for` que hemos empleado antes dispone de algunas utilidades más. Dentro del elemento `loop`, disponemos de algunas propiedades que podemos consultar en cada iteración, como por ejemplo:

- `index`: que obtiene la posición en la colección del elemento que se está explorando actualmente, comenzando por 1.
- `index0`: similar al anterior, pero comenzando a contar por 0.
- `first`: una propiedad booleana que es cierta cuando estamos en el primer elemento de la colección
- `last`: una propiedad booleana que es cierta cuando estamos en el último elemento de la colección
- `length`: que obtiene el número total de elementos de la colección

Se tiene disponible también una cláusula `if` para comprobar condiciones, y los operadores `and` y `or` para enlazar condiciones. Además, se tiene una clásula `elif` para enlazar con `if` y comprobar otras condiciones, y también una cláusula `else` que sirve tanto para mostrar un último camino en secuencias *if..elif..*, como para mostrar qué hacer en un `for` si no hay elementos.

Por ejemplo, así podríamos mostrar el listado de contactos con estilos distintos para los items pares e impares, y con un mensaje personalizado si no hay elementos que mostrar.

```
<ul>
    {% for contacto in contactos %}
        {% if loop.index % 2 == 0 %}
            <li class="par">
        {% else %}
            <li class="impar">
        {% endif %}
        {{ contacto.nombre }}</li>
    {% else %}
        <li>No hay contactos que mostrar.</li>
    {% endfor %}
</ul>
```

3. Definición de vistas jerárquicas e inclusiones

En el ejemplo realizado antes definimos una vista `contactos_listado.njk` utilizando el motor de plantillas Nunjucks para mostrar el listado de contactos de la base de datos. A medida que la aplicación crece y necesitamos ir definiendo más y más vistas, podemos deducir que la forma en que lo hemos hecho en el ejemplo anterior tiene algunas desventajas importantes. Por ejemplo, y sobre todo, su modularidad. Si, por ejemplo, tenemos 20 vistas definidas como la del listado de contactos anterior, y decidimos cambiar el menú de enlaces, tendríamos que editarlo en las 20 vistas. Lo mismo ocurriría si queremos cambiar la información del encabezado o el pie, entre otras cosas, que suele ser común a todas las páginas.

Para evitar este inconveniente Nunjucks (y muchos otros motores de plantillas) permiten realizar un **diseño jerárquico** de las mismas. Es decir, podemos crear una o varias plantillas base con el contenido general que van a tener un conjunto de vistas, y hacer que estas vistas "hereden" de estas plantillas para definir únicamente aquel contenido que les es propio, incluyendo automáticamente el contenido heredado.

3.1. Herencia de plantillas

Vamos a hacer un ejemplo con la aplicación de contactos "*ContactosWeb*" que venimos desarrollando en esta sesión. Creamos en la carpeta `views` una plantilla llamada `base.njk`, que va a tener el contenido general de cualquier vista de la aplicación: el encabezado (*head*) con los estilos y archivos JavaScript para la parte cliente, el menú de la aplicación, y el pie de página, si lo hay. Cada vista va a cambiar el título de página, y el contenido principal de dicha vista. Para dejar estas dos secciones abiertas y que se puedan modificar en cada vista se definen *bloques*, asociando a cada bloque un nombre. Así, nuestra plantilla `base.njk` puede quedar así:

```
<!DOCTYPE html>
<html>
    <head>
        <title>{% block titulo %} {% endblock %}</title>
        <link rel="stylesheet" href="/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            {% block contenido %}

            {% endblock %}
        </div>
    </body>
</html>
```

Como podemos ver, con la estructura `block` definimos bloques de contenidos, asociados a un nombre, de forma que todo lo que queda fuera de esos bloques es fijo para todas las vistas que hereden de la plantilla.

Ahora nuestra página de `contactos_listado.njk` sólo debe limitarse a heredar de esta plantilla, y definir el contenido de los dos bloques. Puede quedar así:

```
{% extends "base.njk" %}

{% block titulo %}Contactos | Listado{% endblock %}

{% block contenido %}

    <h1>Listado de contactos</h1>

    <ul>
        {% for contacto in contactos %}
            <li>{{ contacto.nombre }}</li>
        {% endfor %}
    </ul>

{% endblock %}
```

Del mismo modo, podríamos definir otras vistas, como por ejemplo la ficha de un contacto (`contacto_ficha.njk`):

```
{% extends "base.njk" %}

{% block titulo %}Contactos | Ficha{% endblock %}

{% block contenido %}

    <h1>Ficha de un contacto</h1>

    <p><strong>Nombre:</strong> {{ contacto.nombre }}</p>
    <p><strong>Edad:</strong> {{ contacto.edad }}</p>
    <p><strong>Teléfono:</strong> {{ contacto.telefono }}</p>

{% endblock %}
```

> **NOTA**: en el caso de la ficha del contacto, habría que modificar el contenido del enrutador (método *GET /contactos/:id*) para que renderice la vista y le pase el objeto `contacto` encontrado.

3.2. Inclusión de plantillas

Otra funcionalidad realmente útil que proporcionan muchos motores de plantillas es la posibilidad de incluir el contenido de una plantilla directamente en otra, como si pudiéramos hacer un "copia-pega" directamente de una en otra. Esto evita tener que duplicar el código HTML en las plantillas y, nuevamente, facilitar la posibilidad de posteriores cambios.

Para incluir una vista o plantilla dentro de otra, emplearemos la instrucción `include`. Por ejemplo, si queremos incluir una vista con el menú de navegación de la web, podemos hacer esto, justo en el lugar donde queremos ubicar el menú:

```
{% include "menu.njk" %}
```

3.3. Páginas de error

En algunos casos nos puede interesar mostrar alguna página de error. Por ejemplo, cuando los datos de un contacto no se hayan encontrado, o cuando no haya sido posible realizar alguna operación (un borrado, o una inserción, por ejemplo). Algunos frameworks automatizan estos procesos permitiendo crear páginas con un código de error determinado pero, en el caso de Express, esta gestión se deja más abierta.

Podemos, por ejemplo, crear una vista `error.njk` en nuestra carpeta `views` con una estructura general para mostrar errores en la aplicación:

```
<html>
    <head>
        <title>Error</title>
        <link rel="stylesheet" href="/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <h1>Error</h1>
            <div class="alert alert-danger">
                {% if error %}
                    {{ error }}
                {% else %}
                    Error en la aplicación
                {% endif %}
            </div>
        </div>
    </body>
</html>
```

Nos quedaría renderizar esta vista desde los apartados donde se detecte un error. Por ejemplo, en el listado de contactos o la ficha de un contacto:

```
// Servicio de listado general
router.get('/', (req, res) => {
    Contacto.find().then(resultado => {
        res.render('contactos_listado', {contactos: resultado});
    }).catch(error => {
        res.render('error', {error: 'Error listando contactos'});
    });
});

// Servicio de listado por id
router.get('/:id', (req, res) => {
    Contacto.findById(req.params['id']).then(resultado => {
        if(resultado)
            res.render('contactos_ficha', {contacto: resultado});
        else
            res.render('error', {error: 'Contacto no encontrado'});
    }).catch(error => {
        res.render('error', {error: 'Error buscando contacto'});
    });
});
```

En el ejemplo de este documento puedes consultar el proyecto *ContactosWeb* con estos elementos ya implementados para poderlos probar.

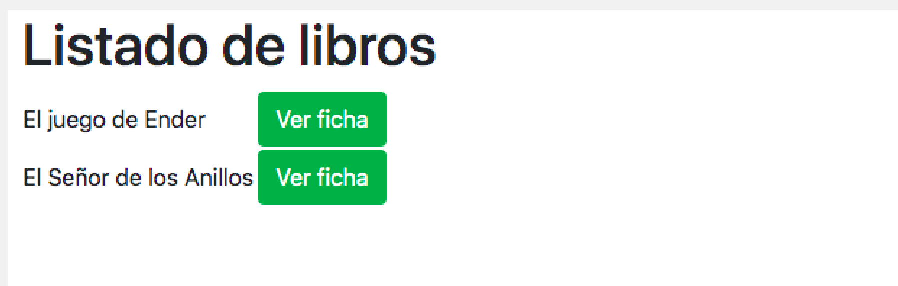
Ejercicio 1:

Crea un proyecto **LibrosWeb** que sea una copia del ejercicio *LibrosREST_v2* de sesiones anteriores. Instala Nunjucks y Bootstrap en el nuevo proyecto, y deja un archivo principal `index.js` similar al del ejemplo de contactos, cargando las librerías, los enrutadores que haya, etc, y configurando Nunjucks como el motor de plantillas de la aplicación.

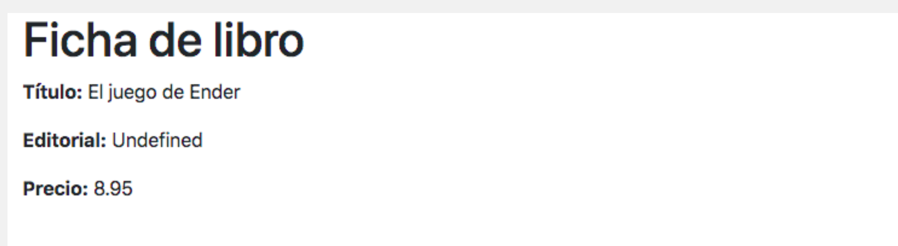
Crea una plantilla llamada `base.njk` que incorpore los elementos comunes a las vistas que vamos a realizar: un encabezado *head* con la inclusión de estilos Bootstrap, pie de página, y un menú de navegación. Deja dos bloques (*block*) editables llamados "titulo" y "contenido", como en el ejemplo de contactos.

Vamos a definir dos vistas, asociadas a los dos enrutadores de listado de libros y ficha de libros, que vamos a modificar. Recuerda que deberás crear los archivos de las vistas en la carpeta `views` del proyecto:

- `libros_listado.njk` : heredará de *base.njk* y recibirá del enrutador GET para `/libros` el listado completo de libros, y mostrará el listado de libros por pantalla. Se deberá mostrar el título del libro, y a su lado, un enlace para ir a la ficha detallada del libro. Esta vista puede quedarte más o menos así. Puedes usar `class="btn btn-success"` en el enlace para mostrarlo con esa apariencia de botón en verde, gracias a Bootstrap. También puedes, opcionalmente, hacer que cada fila (par e impar) se muestre con colores alternos.



- `libros_ficha.njk` : también heredará de *base.njk* y recibirá del enrutador GET para `/libros/:id` los datos del libro a mostrar. Se mostrarán sus datos con un formato como el que se indica a continuación



- `menu.njk` : mostrará un menú de navegación con dos enlaces: uno para ir al listado de libros (URL `/libros`), y otro para ir al formulario de inserción de libros que implementaremos más adelante (URL `/libros/nuevo`). Incluye con `include` esta vista en la plantilla base, justo antes del bloque de "contenido". Esta es la apariencia que puede tener más o menos la vista de listado ahora:

Listado de libros Nuevo libro

Listado de libros

El juego de Ender Ver ficha

El Señor de los Anillos Ver ficha

- Además de las vistas anteriores, crea una nueva vista llamada `error.njk` en la carpeta de vistas, que recibirá como parámetro un mensaje de error y lo mostrará por pantalla, como el ejemplo que se ha hecho antes para los contactos, mostrando un mensaje genérico "Error en la aplicación" si no recibe un mensaje de error concreto. Modifica los enrutadores GET de libros para que, en caso de error, se renderice esta vista con el mensaje adecuado.

NOTA: recuerda que los antiguos enrutadores que devolvían datos JSON deberán modificarse para renderizar las vistas correspondientes en este ejercicio. El resto de enrutadores que aún no hemos tocado (inserción o borrado de libros, por ejemplo), déjalos como estaban de la sesión anterior, y ya iremos modificándolos después.