Examen

Soluciones febrero 2017

Aplicaciones Web
Año 2016/2017
Grado en Ing. Software
Fdi - UCM

▷ 1. Acceso a BD mediante *Node.js*

En la Figura 1 se muestra el diseño relacional de una base de datos que almacena artículos de revista. Cada artículo tiene asociado un identificador numérico, un título, una fecha de publicación y una lista de palabras clave, que se encuentran en una tabla separada (`palabrasclave`).

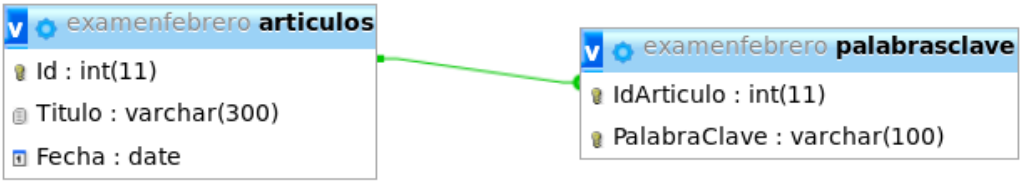

Figura 1: Diseño de la base de datos de artículos y palabras clave

Implementa una función `leerArticulos(callback)` que obtenga todos los artículos de la base de datos. Esta función debe construir un **array de objetos**, cada uno de ellos representando la información de un artículo mediante cuatro atributos: `id` (númerico), `titulo` (cadena de texto), `fecha` (objeto `Date`) y `palabrasClave` (array de cadenas). Por ejemplo:

```
{
  id: 1,
  titulo: "An inference algorithm for guaranteeing Safe destruction",
  fecha: 2008-07-19,  // ⟵ como objeto de la clase Date
  palabrasClave: [ 'formal', 'inference', 'memory' ]
}
```

La función `callback` pasada como parámetro a `leerArticulos` funciona de igual modo que las vistas en clase. Recibe dos parámetros: un objeto con información de error (en caso de producirse), y la lista con los artículos recuperados de la base de datos.

Solución

Supongamos las siguientes instancias de las relaciones `articulos` y `palabrasclave`:

Id	Titulo	Fecha
1	An inference algorithm for guaranteeing Safe destruction	2008-07-20
2	A type system for region management and its proof of correctness	2010-07-21
3	Shape analysis by regular languages	2009-05-30
4	Polymorphic type specifications	2016-03-01
5	Yet to be written	2017-02-01

IdArticulo	PalabraClave
1	memory
1	inference
1	formal
2	type system
2	memory
4	type system
4	success types

Podemos hacer un JOIN entre ambas tablas. Además, dado que algunos de los artículos no tienen palabras clave, hemos de hacer un JOIN asimétrico (LEFT JOIN):

```
SELECT a.Id, a.Titulo, a.Fecha, p.PalabraClave
FROM articulos AS a LEFT JOIN palabrasclave AS p
ON a.Id = p.IdArticulo
```

Sin embargo, para recorrer el resultado de esta consulta nos interesa que todas las filas pertenecientes a un mismo artículo aparezcan juntas. Por ello ordenamos las filas por artículo:

```
SELECT a.Id, a.Titulo, a.Fecha, p.PalabraClave
FROM articulos AS a LEFT JOIN palabrasclave AS p
ON a.Id = p.IdArticulo
ORDER BY a.Id
```

En nuestro ejemplo, obtendríamos:

Id	Titulo	Fecha	PalabraClave
1	An inference algorithm for guaranteeing Safe destruction	2008-07-20	memory
1	An inference algorithm for guaranteeing Safe destruction	2008-07-20	formal
1	An inference algorithm for guaranteeing Safe destruction	2008-07-20	inference
2	A type system for region management and its proof of correctness	2010-07-21	type system
2	A type system for region management and its proof of correctness	2010-07-21	memory
3	Shape analysis by regular languages	2009-05-30	*NULL*
4	Polymorphic type specifications	2016-03-01	type system
4	Polymorphic type specifications	2016-03-01	success types
5	Yet to be written	2017-02-01	*NULL*

A partir de ahí ejecutamos esta sentencia mediante la librería `mysql` de Node. Mantenemos una variable `resultado` para almacenar los artículos que vayamos recuperando a medida que recorramos el resultado de la consulta. Cada vez que la fila actual contenga un artículo distinto a la fila anterior, creamos un nuevo artículo.

```
function leerArticulos(callback) {
    connection.connect(function(err) {
        if (err) {
            callback(err);
        } else {
            // La siguiente variable contendrá la sentencia SQL a ejecutar
```

```
var sql = "SELECT␣a.Id,␣a.Titulo,␣a.Fecha,␣p.PalabraClave␣" +
        "FROM␣articulos␣AS␣a␣LEFT␣JOIN␣palabrasclave␣as␣p␣" +
        "ON␣a.Id␣=␣p.IdArticulo␣" +
        "ORDER␣BY␣a.Id";

// Realizamos consulta
connection.query(sql, function(err, rows) {
    connection.end();
    if (err) {
        callback(err);
    } else {
        // Array con los artículos recuperados hasta ahora
        var resultado = [];
        // Ultimo articulo recuperado
        var articuloFilaAnterior = null;

        for (var i = 0; i < rows.length; i++) {
            // Si el articulo de la fila actual es igual al de la
            // fila anterior, no hace falta añadirlo al resultado,
            // pues ya lo hemos hecho previamente.
            //
            // En caso contrario, lo creamos y lo añadimos.
            if (articuloFilaAnterior === null
                    || articuloFilaAnterior.id !== rows[i].Id) {
                var nuevoArticulo = {
                    id: rows[i].Id,
                    titulo: rows[i].Titulo,
                    fecha: rows[i].Fecha,
                    palabrasClave: []
                };
                resultado.push(nuevoArticulo);
                articuloFilaAnterior = nuevoArticulo;
            }

            // Comprobamos que PalabraClave es distinto de NULL,
            // pues puede ser que un articulo no tenga ninguna
            // palabra clave.

            if (rows[i].PalabraClave !== null) {
                nuevoArticulo.palabrasClave.push(rows[i].PalabraClave);
            }
        }
        callback(null, resultado);
    }
});
```

```
        }
    });
}
```

▷ 2. Aplicaciones web mediante *Express.js*

La constante matemática $e \approx 2{,}718281828459\ldots$ es un número irracional que puede aproximarse mediante la siguente expresión (donde $n \geq 0$):

$$e \approx \sum_{i=0}^{n} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

donde la notación $i!$ indica "factorial de i". La aproximación es más precisa cuanto mayor sea el valor de n. Implementa una aplicación web que solicite el valor de n al usuario y calcule una aproximación de e mediante la fórmula anterior (Figura 2).

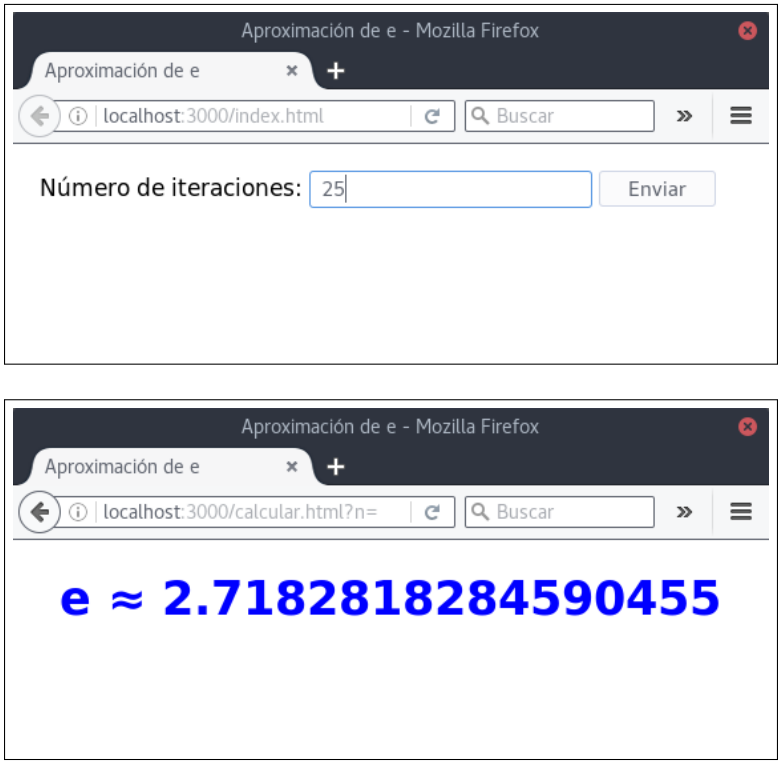

Figura 2: Funcionamiento de la aplicación para el cálculo de e

Si el usuario no ha introducido un número, se volverá a mostrar el formulario junto con un mensaje de error (Figura 3).

Solución

Suponemos el siguiente formulario:

```
<form action="/calcular.html" method="GET">
```

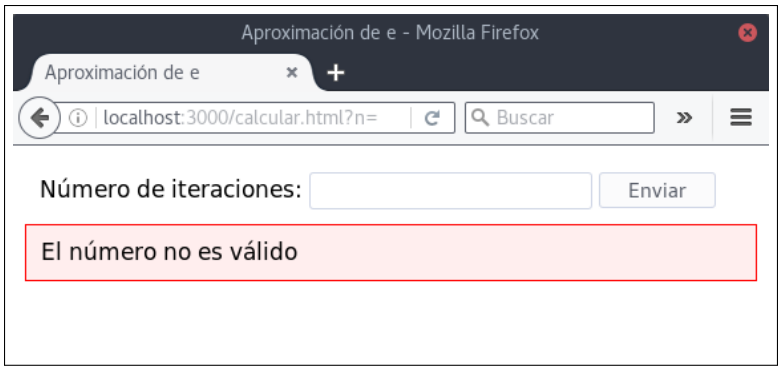

Figura 3: Mensaje de error en el formulario

```
    Número de iteraciones:
    <input type="text" name="n">
    <input type="submit" value="Enviar">
</form>
```

Debemos implementar el manejador de la dirección `/calcular.html` para que calcule la aproximación del número *e*:

```
app.get("/calcular.html", function(request, response) {
    var n = Number(request.query.n);
    if (isNaN(n)) {
        response.render("calculoE", { error: "El numero no es valido" });
    } else {
        var result = 1;
        var factorial = 1;
        for (var i = 1; i <= n; i++) {
            factorial = factorial * i;
            result = result + (1 / factorial);
        }
        response.render("result", { resultado: result } );
    }
});
```

Al finalizar se muestra la vista `result`, definida en un fichero `result.ejs` que se encuentra bajo el directorio de vistas:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Aproximación de e</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="css/calculoE.css">
    </head>
```

```
    <body>
        <h1><%= resultado %></h1>
    </body>
</html>
```

En caso de error, se muestra la vista `calculoE`, que contiene el formulario junto con el mensaje de error:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Aproximación de e</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link rel="stylesheet" href="css/calculoE.css">
    </head>
    <body>
        <div class="formulario">
          <form action="/calcular.html" method="GET">
              Número de iteraciones:
              <input type="text" name="n">
              <input type="submit" value="Enviar">
          </form>
        </div>
        <% if(error !== false) { %>
            <div class="error">
                <%= error %>
            </div>
        <% } %>
    </body>
</html>
```

Al acceder a la página por primera vez, basta con mostrar la vista `calculoE` pasando `false` a la variable de tipo `error`:

```
app.get("/index.html", function(request, response) {
    response.render("calculoE", { error: false });
});
```

▷ 3. Uso de sesiones en *Express.js*

El objetivo de este ejercicio es la realización de una aplicación web en la que el usuario pueda introducir una lista de tareas pendientes y marcar aquellas que ha finalizado (Figura 4). Las tareas han de guardarse **en la sesión del navegador** utilizando el *middleware* `express-session`.

(a). Implementa un manejador de ruta `/index.html` que muestre todas las tareas introducidas por el usuario, junto con un formulario invitando al usuario a introducir una nueva tarea. Implementa otro manejador para la ruta `/anyadirTarea` que recoja la información introducida en el formulario y añada la tarea introducida a la lista. De momento puedes ignorar la línea `Número de tareas`

Figura 4: Aplicación de lista de tareas

pendientes que se muestra en la Figura 4 y el enlace `Marcar como finalizada` mostrado a la derecha de cada enlace.

(b). Modifica la página anterior para que se muestre un enlace `[Marcar como finalizada]` a la derecha de cada tarea. Al hacer clic en ese enlace, la tarea se marcará como finalizada, y se mostrará de nuevo la página con la lista de tareas. Las tareas marcadas como finalizadas aparecerán tachadas y sin el enlace en el lado derecho (Figura 4).

(c). Añade un *middleware* a la aplicación anterior para que calcule el número de tareas de la lista que no hayan sido finalizadas y lo asigne a la variable `response.locals.numPendientes`. Modifica la página para que también muestre la lista de tareas pendientes.

Solución

(a). Guardaremos en la sesión una lista de objetos, donde cada objeto tiene dos atributos: `descripcion` (donde se indica el texto de la tarea) y `finalizada` (un booleano que indica si la tarea ha sido marcada como finalizada o no). En *Express.js* suponemos que hemos añadido el middleware de sesión, de modo que tenemos la variable `request.session` a nuestro alcance. Almacenaremos la lista de tareas en el atributo `request.session.lista`.

Para evitar distinguir constantemente si el atributo `request.session.lista` está inicializado o no, creamos un *middleware* que lo inicializa, en caso de ser necesario.

```
app.use(function(request, response, next) {
    if (request.session.lista === undefined) {
```

```
        request.session.lista = [];
    }
    next();
});
```

A continuación implementamos el manejador que muestra la lista de tareas:

```
app.get("/index.html", function(request, response) {
    var lista = request.session.lista;
    response.render("tareas", { listaTareas: lista });
});
```

donde la vista de tareas se define del siguiente modo:

```
...
<ul class="listaTareas">
    <% listaTareas.forEach(function(tarea, indice) { %>
    <li>
        <span><%= tarea.descripcion %></span>
    </li>
    <% }); %>
</ul>
...
<div class="nuevaTarea">
    <form action="/anyadirTarea" method="POST">
        <label for="nuevaTarea">Nueva tarea: </label>
        <input type="text" name="nuevaTarea" id="nuevaTarea">
        <input type="submit" value="Enviar">
    </form>
</div>
...
```

El siguiente paso es gestionar la inserción de una nueva tarea. Para ello implementamos el manejador `/anyadirTarea`:

```
app.post("/anyadirTarea", function(request, response) {
    var descripcion = request.body.nuevaTarea;
    if (descripcion !== undefined && descripcion.trim() !== "") {
        request.session.lista.push({
            descripcion: descripcion,
            finalizada: false
        });
    }
    response.redirect("/index.html");
});
```

En este ejercicio suponemos que se ha añadido previamente el middleware `body-parser`.

(b). Modificamos nuestra vista EJS para distinguir si la tarea ha sido finalizada o no:

```
...
<ul class="listaTareas">
    <% listaTareas.forEach(function(tarea, indice) { %>
    <li>
        <% if (tarea.finalizada) { %>
            <span class="finalizada"><%= tarea.descripcion %></span>
        <% } else { %>
            <span><%= tarea.descripcion %></span>
            <a class="botonfinalizada"
               href="/finalizarTarea?ind=<%=indice%>">
                  Marcar como finalizada
            </a>
        <% } %>
    </li>
    <% }); %>
</ul>
...
```

El enlace `[Marcar como finalizada]` llama a la URL `/finalizarTarea`, cuyo manejador se implementa del siguiente modo:

```
app.get("/finalizarTarea", function(request, response) {
    var indice = Number(request.query.ind);
    var lista = request.session.lista;
    if (!isNaN(indice) && lista[indice] !== undefined) {
        lista[indice].finalizada = true;
    }
    response.redirect("/index.html");
});
```

(c). El *middleware* se define del siguiente modo:

```
app.use(function(request, response, next) {
    var numTareasPendientes = request.session.lista.filter(function(tarea) {
        return !tarea.finalizada;
    }).length;
    response.locals.numPendientes = numTareasPendientes;
    next();
});
```

Aplicamos la función `filter` a la lista de tareas, de modo que devuelve una copia del array `request.session.lista` en la que los elementos cuyo atributo `finalizada` sea `true` se descartarán. Calculando la longitud de la lista resultado obtenemos el número de tareas pendientes.

Con ello basta modificar la vista de la lista de tareas para añadir:

```
<div class="numTareas">
    Número de tareas pendientes: <%= numPendientes %>
</div>
```

▷ 4. AJAX y servicios web

Supongamos un juego en línea que almacena un array con las puntuaciones obtenidas por cada jugador, ordenadas de mayor a menor puntuación:

```
var records = [
    { nombre: "Fran", puntos: 955 },
    { nombre: "Rafael", puntos: 865 },
    { nombre: "Carmen", puntos: 563 },
    { nombre: "Rosario", puntos: 534 },
    { nombre: "Juan", puntos: 234 },
    { nombre: "Estela", puntos: 107 },
    ...
];
```

(a). Implementa un servicio web que devuelva un JSON con los cinco primeros objetos (es decir, los de más puntuación) del array `records`. En caso de que el array contenga menos de cinco objetos, se devolverá todo el array.

Método: `GET`

URL: `/highestRecords`

Parámetros de entrada: Ninguno

Código de respuesta: `200` (OK).

Tipo resultado: JSON con un array de objetos. Cada uno de ellos tiene dos propiedades: `nombre` y `puntos`.

Resultado: Devuelve las cinco personas que han obtenido una puntuación más alta. Para cada una de ellas, se devuelve su nombre y su puntuación.

(b). Supongamos una página como la de la Figura 5, en la que se muestra una tabla con las cinco mejores puntuaciones. Escribe una función `actualizarLista()` para que, mediante `jQuery` y una petición AJAX al servicio implementado en el apartado anterior, actualice dicha tabla con los datos devueltos por la petición AJAX.

Indicación: La sentencia `$(selector).empty();` elimina del DOM todos los hijos que se encuentren debajo del elemento indicado por el `selector` dado.

(c). Suponemos que añadimos a la página anterior un formulario invitando al usuario introducir un nombre, junto con un botón `[Enviar]`. Modifica la aplicación para que, cuando se pulse dicho botón, se añada el usuario a la tabla de records con una puntuación aleatoria entre 1 y 1000 y actualice la tabla de records. Para ello puedes suponer implementado el siguiente servicio:

Método: `POST`

URL: `/newRecord`

Parámetros de entrada: En el cuerpo de la petición POST, un objeto JSON con un único atributo (llamado `"nombre"`) que contenga el nombre introducido en el formulario.

Código de respuesta: `201` (*Created*).

Figura 5: Tabla de puntuaciones más altas

Tipo resultado: Ninguno.

Resultado: Ninguno.

Solución

(a). Añadimos un manejador de tipo GET a nuestra aplicación que obtenga mediante el método `slice` los cinco primeros elementos del array, y devuelva ese mismo segmento en formato JSON:

```
app.get("/highestRecords", function(request, response) {
    response.json(records.slice(0, 5));
});
```

(b). Partimos del siguiente código HTML:

```
<body>
    <h1>Mejores puntuaciones</h1>
    <table id="records">
        <thead>
            <tr>
                <th>Nombre</th>
                <th>Puntuación</th>
            </tr>
        </thead>
        <tbody>
            <!-- Insertar aquí las filas de la tabla -->
```

```
        </tbody>
    </table>
    <div class="formulario">
        <label for="nueva">Introduce tu nombre: </label>
        <input type="text" id="nueva">
        <button id="enviar">Enviar</button>
    </div>
</body>
```

La función pedida realiza una petición AJAX a la URL del apartado anterior para obtener los cinco primeros elementos de la tabla. En caso de éxito, inserta las filas dentro de la sección <tbody> de la tabla:

```
function actualizarTabla() {
    $.ajax({
        type: "GET",
        url: "/highestRecords",
        success: function (data, textStatus, jqXHR) {
            var tabla = $("#records > tbody");
            tabla.empty();
            data.forEach(function(record) {
                var fila = $("<tr>");
                var columna1 = $("<td>").text(record.nombre);
                var columna2 = $("<td>").text(record.puntos);
                fila.append(columna1);
                fila.append(columna2);
                tabla.append(fila);
            });
        },
        error: function (jqXHR, textStatus, errorThrown) {
            alert(errorThrown);
        }
    });
}
```

(c). Basta con realizar una petición AJAX al servicio descrito en el enunciado, y actualizar la tabla:

```
$("#enviar").on("click", function() {
    // Realizar petición AJAX a /newRecord y llamar a
    // actualizarTabla() cuando esta petición finalice
    // con éxito
    $.ajax({
        type: "POST",
        url: "/newRecord",
        contentType: "application/json",
        data: JSON.stringify({ nombre: $("#nueva").val() }),
```

```
        success: function (data, textStatus, jqXHR) {
            actualizarTabla();
        },
        error: function (jqXHR, textStatus, errorThrown) {
            alert(errorThrown);
        }
    });
});
```