

APLICACIONES DE INTERNET

PRÁCTICA 2

Martínez Lona Verónica Montserrat

Para la creación del web service se utilizó NodeJs, MongoDB y Express.

Los pasos más relevantes del desarrollo del servicio son:

1. Creación del proyecto con express
2. Creación de las vistas
3. Creación del esquema de la base de datos
4. Creación de la API
5. Consumo de la API

1. Creación del proyecto con express

La estructura del proyecto es la siguiente:

- **Controlers** //Controlador de los empleados
- **Models** //El modelo a utilizar en la base de datos
- **Public**
 - Images //Imágenes pertenecientes al diseño de las vistas
 - Javascripts // Consumo de la API
 - Stylesheets // Hojas de estilo para el diseño de las vistas
- **Routes** //Rutas a cargar
- **Vistas** //Archivos hbs para el diseño de la interfaz
- **App.js** //Servidor

En las rutas tenemos los procedimientos que se pueden realizar en cada una de ellas:

- **Index** //POST(No implementado)
- **Employee** // GET, PUT, DELETE
- **Employee_add** //POST

App.js

Servidor de la aplicación, no es muy distinto de lo que hemos trabajado en clases., por ello no comentaré todas las líneas de código.

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var exphbs = require('express-handlebars');
var methodOverride = require('method-override');
var mongoose = require('mongoose');
var employeeManager = require('./controllers/employeeManager.js');
```

Importamos las dependencias que utilizaremos en el proyecto. Yo utilice express generator para comenzar el poyecto.

```
//Endpoints
var login = require('./routes/index');
var employee = require('./routes/employee');
var add = require('./routes/add_employee');
```

Estos son los end-points que se utilizaron en todo el proyecto.

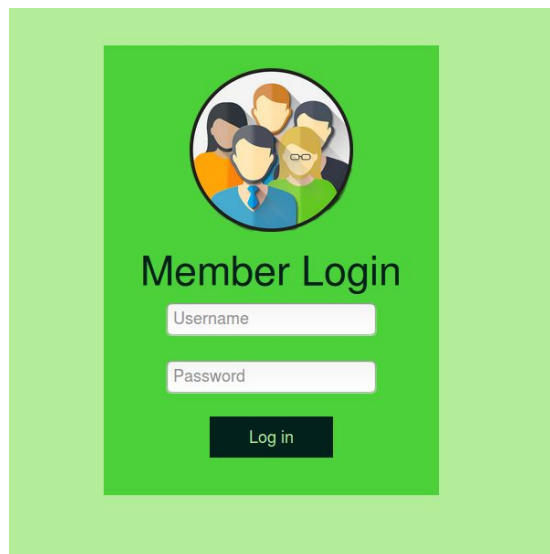
2. Creación de las vistas

Las vistas son:

- Login.hbs
- Employee.hbs
- Edit_employee.hbs
- Add_employee.hbs

Las vistas están hechas con handlebars + css. El diseño es simple, al igual que la funcionalidad.

Login.hbs



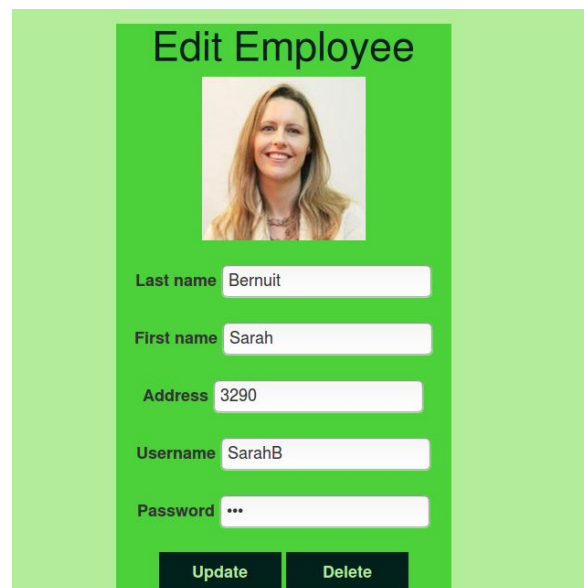
No está implementado el login, sin embargo al pulsar el boton *Log in* redirecciona a la ruta */employee*.

Employee.hbs



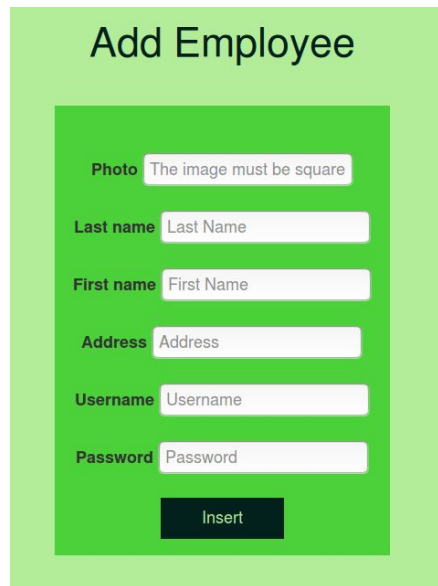
Aquí se muestra una lista de los empleados en la base de datos. Existe un botón de agregar que redirecciona a la página de agregar empleado. Al clickear sobre un empleado debería redireccionarse a su página de empleado específica (esto no funciona correctamente).

edit_employee.hbs

The screenshot shows a web page titled "Edit Employee" with a light green background. At the top center is a circular icon containing a single person's photo. Below the icon, the word "Edit Employee" is written in a large, dark font. Below this, there are several input fields for employee information: Last name (Bernuit), First name (Sarah), Address (3290), Username (SarahB), and Password (***). At the bottom, there are two dark green buttons labeled "Update" and "Delete".

En esta vista se carga la información de un usuario específico, todos los datos pueden ser actualizados, salvo la fotografía. En esta misma vista se elimina el empleado.

add_employee.hbs

A screenshot of a web form titled "Add Employee" with a light green background. The form itself has a darker green background and contains several input fields. At the top is a "Photo" field with a placeholder text "The image must be square". Below it are fields for "Last name", "First name", "Address", "Username", and "Password". At the bottom of the form is a black button with the text "Insert" in white.

Este es el formulario para agregar un empleado. No es necesario que se llenen todos los campos.

3. Creación del esquema de la base de datos

```
var employeeSchema = new Schema({  
  photo: String,  
  lastName: String,  
  firstName: String,  
  address: String,  
  username: String,  
  password: String  
});
```

El esquema es realmente simple.

4. Creación de la API

controllers/employeeManager.js

```
var employee = require('../models/employee.js');
```

Puesto que se va a manejar los documentos de la base de datos, es necesario importar el modelo de *employee*.

```
//GET
exports.findAll = function(req, res) {
  employee.find(function(error, employeeList) {
    if(error) { return res.send(500, error.message); }
    return res.status(200).jsonp(employeeList);
  });
};
```

La primera función que se implementó fue *findAll(GET)*, que permite ver todos los empleados.

```
//GET
exports.findById = function(req, res) {
  var id = req.params.id;
  console.log('Employee: ' + id);
  employee.findById(req.params.id, function(error, found) {
    if(error) { return res.send(500, console.error.message); }
    return res.status(200).json(found);
  });
};
```

La segunda función fue *findById(GET)*, que permite ver los datos específicos de un empleado. El request viene con los datos pertenecientes al empleado seleccionado, lo que necesitamos para filtrar es el id, por ello lo obtenemos de *req.params.id*. Para obtener los datos de ese único id es necesario pasarlo como parámetro de la función *employee.findById*.

```
//PUT
exports.updateEmployee = function(req, res) {
  var id = req.params.employee_id;
  employee.findById(id, function(error, selectedEmployee) {
    if(error) { return res.send(500, console.error.message); }
    //update info
    selectedEmployee.photo = req.body.photo;
    selectedEmployee.lastName = req.body.lastName;
    selectedEmployee.firstName = req.body.firstName;
    selectedEmployee.address = req.body.address;
    selectedEmployee.username = req.body.username;
    selectedEmployee.password = req.body.password;
    //save the employee
    selectedEmployee.save(function(error) {
      if(error) { return res.send(500, console.error.message); }
      employee.find(function(error, employeeList) {
        if(error) { return res.send(500, error.message); }
        return res.status(200).jsonp(employeeList);
      });
    });
  });
};
```

La tercera función es *updateEmployee (PUT)*. Similar al anterior (de hecho, se utiliza aquí). Después de que se llena la información de un usuario específico, al presionar el botón de

Update se prosigue a tomar los datos del formulario y actualizar con ellos la información de la base de datos. Para conseguir esos datos se hace por medio de *selectedEmployee.campo*. Finalmente guardamos los valores.

```
//POST
exports.addEmployee = function(req, res) {
  console.log(req.body);

  //Create a new instance of the employee model
  var newEmployee = new employee({
    employeeId: req.body.employeeId,
    photo: req.body.photo,
    lastName: req.body.lastName,
    firstName: req.body.firstName,
    address: req.body.address,
    username: req.body.username,
    password: req.body.password
  });

  //Save the employee and check for errors
  newEmployee.save(function (error) {
    if(error) { return res.send(500, error.message); }
    employee.find(function(error, employeeList) {
      if(error) { return res.send(500, error.message); }
      return res.status(200).jsonp(employeeList);
    });
  });
};
```

La siguiente función es *addEmployee*, que, como su nombre dice, sirve para añadir un empleado a la base de datos. Se abre un formulario dónde se rellenan libremente los campos del formulario, que después de obtenerse del cuerpo del request y finalmente se guardan en la base de datos.

```
//DELETE
exports.deleteEmployee = function(req, res) {
  employee.remove({ _id: req.params.employee_id }, function(error, selectedEmployee) {
    if(error) { return res.send(500, error.message); }
    employee.find(function(error, employeeList) {
      if(error) { return res.send(500, error.message); }
      return res.status(200).jsonp(employeeList);
    });
  });
};
```

Por último se definió la función *deleteEmployee*, que nos permite eliminar un empleado específico. Para poder obtenerlo específicamente sólo es necesario guardar y enviar el valor del id del empleado seleccionado.

5. Consumo de la API

Los procedimientos se asignan en el app.js, cada uno a la ruta respectiva en la que deben correr.

APP.JS

```
app.get('/api/employee', employeesController.findAll);

app.get('/api/employees/:id', employeesController.findById);
app.put('/api/employees/:id', employeesController.updateEmployee);
app.delete('/api/employees/:id', employeesController.deleteEmployee);

app.post('/api/employee/add', employeeManager.addEmployee);
```

Se tienen 2 archivos .js on jQuery que nos ayudan a consumir la API. Estos archivos son:

- Main.js
- employeeEdit.js

MAIN.JS

Main.js maneja dos métodos http: *GET* y *POST*

```
//View all employees
$(function() {
    var results = $('.results');
    $(document).ready(function() {
        $.ajax({
            url: 'http://localhost:3000/api/employee/',
            type: 'get',
            dataType: 'jsonp',
            success: function(data) {
                for (var i=0;i<data.length;i++) {
                    var row = '<tr><td>'
                        + '<a href="http://localhost:3000/employee/" + data[i].id + ">"'
                        + data[i].lastName + ' ' + data[i].firstName + '</a>'
                        + '</td></tr>';
                    results.append(row);
                }
            },
            error: function(xhr, textStatus, errorThrown) {
                console.log('Error in Operation');
            }
        });
    });
});
```

El primero (*GET*) se utiliza para obtener un listado de todos los empleados en la base de datos. La inyección en el .hbs se hace por medio de la clase *.results*. Éste método corre en la url específica <http://localhost:300/api/employee/> que es la vista dónde se muestran los empleados. Por cada empleado se crea un link, que, al darle click redirecciona a la página de editar a ese empleado en específico.

```
//Add employee
$('#add').on('click', function(event) {
    event.preventDefault();

    var employee = jQuery(".newEmployee").serialize();
    $.ajax({
        url: "http://localhost:3000/api/employee/add",
        type: "POST",
        data: employee,
        success: function(data, status, jqXHR) {
            alert("Employee Added!");
        },
        error: function(jqXHR, status) {
            alert("An error occurs!");
        }
    });
});
```

El segundo método (*POST*) se utiliza para agregar un empleado. Aquí la inyección se hace por medio de *.newEmployee*. El método se activa al dar click en el botón de “add”, y corre en la url específica `http://localhost:3000/api/employee/add`.

Por otro lado, *employeeEdit.js* maneja 3 métodos: *GET*, *PUT* y *DELETE*.

```
//View specific employee
$(function() {
    var pathname = window.location.pathname;
    var id;
    var detail = $('#detail');
    $(document).ready(function() {
        $.ajax({
            url: 'http://localhost:3000/api' + pathname,
            type: 'get',
            dataType: 'json',
            success: function(data) {
                var form = ''
                    + '<br><br>'
                    + '<label for="lastName">Last name</label>'
                    + '<input id="lastName" name="lastName" type="text" value="' + data.lastName + '>'
                    + '<br><br>'
                    + '<label for="firstName">First name</label>'
                    + '<input id="firstName" name="firstName" type="text" value="' + data.firstName + '>'
                    + '<br><br>'
                    + '<label for="address">Address</label>'
                    + '<input id="address" name="address" type="text" value="' + data.address + '>'
                    + '<br><br>'
                    + '<label for="username">Username</label>'
                    + '<input id="username" name="username" type="text" value="' + data.username + '>'
                    + '<br><br>'
                    + '<label for="password">Password</label>'
                    + '<input id="password" name="password" type="password" value="' + data.password + '>'
                    + '<br><br>';
                detail.append(form);
                id = data._id;
            },
            error: function(xhr, textStatus, errorThrown) {
                alert("It works");
                console.log('Error in Operation');
            }
        });
    });
});
```

El primero (*GET*) es para listar un empleado específico, filtrado por la id. La url se construye dinámicamente, para poder reutilizar esta función con cada una de las url de los empleados. En

esta función se construye todo un formulario con los datos del empleado seleccionado y se inyecta en la clase *.details*.

```
//Update an employee
$('#Update').on('click', function(event) {
    event.preventDefault();
    var edit = 'id=' + id + '&' + jQuery("#.editEmployee").serialize();
    $.ajax({
        url: 'http://localhost:3000/api' + pathname,
        type: "PUT",
        data: edit,
        success: function (data, status, jqXHR) {
            alert("Employee Updated");
        },
        error: function (jqXHR, status) {
            alert("An error occurs");
        }
    });
});
```

El segundo método (*PUT*), se utiliza cuando se actualiza la información de un empleado en específico. Al igual que en el anterior, la url se construye dinámicamente para poder reutilizar la función.

```
//Delete an employee
$('#Delete').on('click', function(event) {
    event.preventDefault();
    var edit = 'id=' + id + '&' + jQuery("#.deleteEmployee").serialize();
    $.ajax({
        url: 'http://localhost:3000/api' + pathname,
        type: "DELETE",
        data: JSON.stringify(id),
        dataType: "json",
        success: function (data, status, jqXHR) {
            alert("Employee deleted");
        },
        error: function (jqXHR, status) {
            alert("An error occurs");
        }
    });
});
```

Finalmente el *DELETE* se utiliza para borrar un empleado específico. La url se construye dinámicamente, igual que los anteriores. Después de que se elimina un empleado se redirecciona la vista de la lista principal de empleados (esto se maneja en el *.js* específico de cada vista en */routes*).