Entrega 10 - Quiz servidor

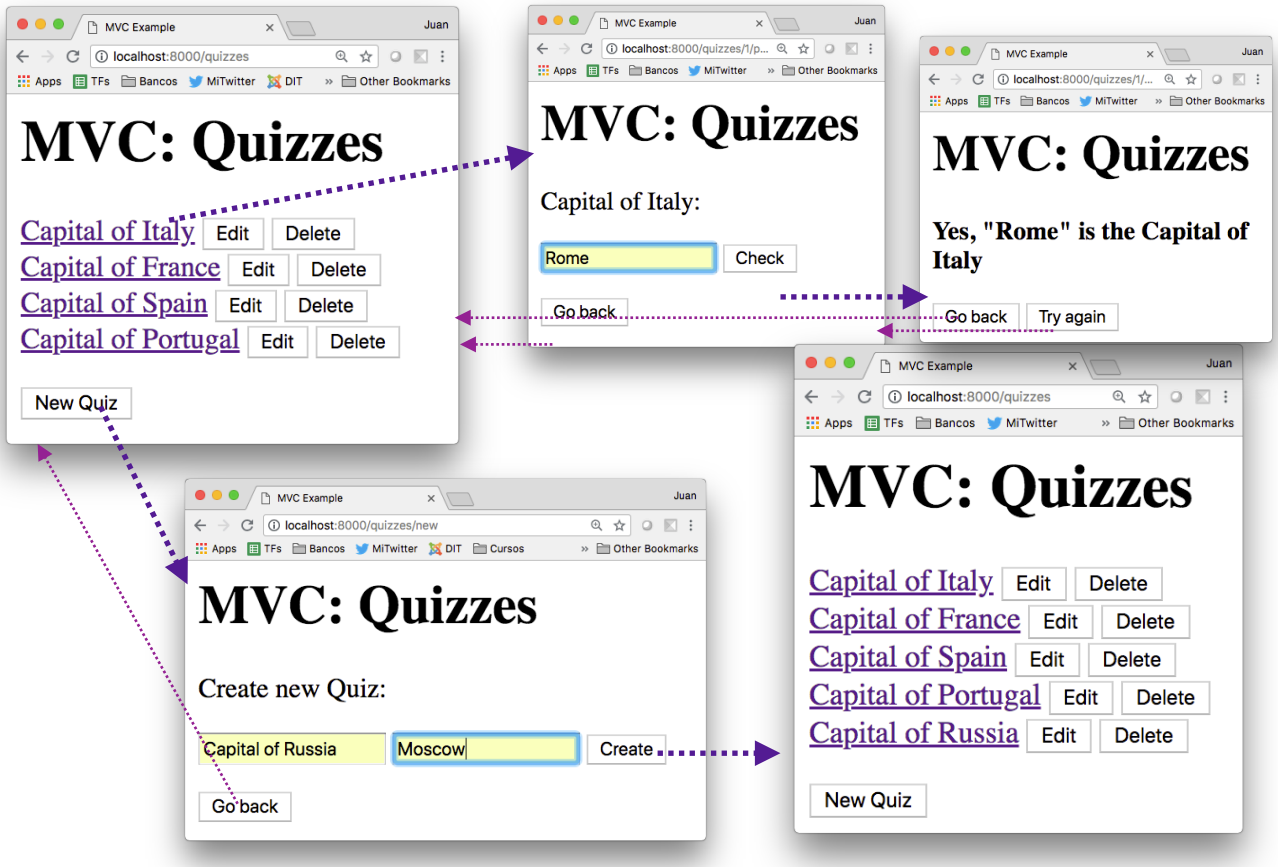
Versión: 12 de Febrero de 2019

Objetivo

Entender y practicar con una aplicación de servidor, estructura con MVC, con un pequeño interfaz REST y que utiliza los paquetes npm express, body-parser, method-override, sequelizejs y sqlite3.

Descripción de la práctica

El fichero **mooc_node-mod10_quiz_mvc_server.js** contiene el esqueleto de una aplicación Web de servidor estructurada con el modelo MVC (Model-Vista-Controller). Este se descarga del proyecto de prueba en https://github.com/practicas-ging/mooc_node-mod10_quiz_mvc_server). El esqueleto se muestra en las siguientes páginas. Funciona correctamente solo para la parte de la aplicación relacionada con jugar a los quizzes (hacer click en la pregunta) y crearlos, tal y como muestran estas capturas:



Este esqueleto de aplicación node necesita los paquetes npm express, body-parser, method-override, sequelize y sqlite3 para poder ejecutarse. Estos deben instalarse con "npm install …" antes de arrancar. sqlite3 debe instalarse con la opción -g. Este servidor envía la lista de preguntas a un navegador que acceda al URL http://localhost:8000. Desde la vista obtenida (lista de quizzes) podemos jugar y crear quizzes. Al hacer click en el texto de una pregunta (quiz), aparece el formulario para enviar la respuesta. Al enviar la respuesta con el formulario nos indica si es correcta o no. El esqueleto de app también permite crear nuevos quizzes, tal y como se ve

en las capturas. Pero esta aplicación no responde a los botones de editar o borrar quizzes, solo responde a los botones en los que las capturas anteriores muestran con una flecha que llevan a otras vistas.

En este ejercicio se debe completar el esqueleto (fichero **mod10-quiz_MVC_server.js**) para que todos los botones funcionen correctamente. En particular:

- Editar y borrar quizzes añadiendo soporte en router y controladores a las primitivas:

```
GET     /quizzes/:id/edit
PUT     /quizzes/:id/update
DELETE  /quizzes/:id
```

- Que la respuesta a jugar con los quizzes se compruebe con AJAX, de forma que:

  1) Se elimine la vista *check(id, response, answer)* y se modifique la vista *play(id, question, answer)* para insertar el resultado de comprobar la respuesta justo debajo del cajetín de pregunta.
  2) Se modifique la primitiva *GET /quizzes/:id/check* para que esta se invoque por AJAX y solo devuelva el mensaje a insertar en la vista *play(id, question, answer)* y no la vista completa.

- Introducir una tabla HTML en la vista *index(quizzes)* para que los botones de editar y borrar queden bien alineados en una columna.

```
========== mod10-quiz_MVC_server: Comienzo del esqueleto de la aplic. de servidor ============

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



   // Import MW for parsing POST params in BODY

const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));



   // Import MW to support Method-Override with express

const methodOverride = require('method-override');
app.use(methodOverride('_method', {methods: ["POST", "GET"]}));


   // MODEL

const Sequelize = require('sequelize');

const options = { logging: false, operatorsAliases: false};
const sequelize = new Sequelize("sqlite:db.sqlite", options);

const quizzes = sequelize.define(  // define table quizzes
    'quizzes',
    {   question: Sequelize.STRING,
        answer: Sequelize.STRING
    }
);

sequelize.sync() // Syncronize DB and seed if needed
.then(() => quizzes.count())
.then((count) => {
    if (count===0) {
```

```
        return (
            quizzes.bulkCreate([
                { id: 1, question: "Capital of Italy",    answer: "Rome" },
                { id: 2, question: "Capital of France",   answer: "Paris" },
                { id: 3, question: "Capital of Spain",    answer: "Madrid" },
                { id: 4, question: "Capital of Portugal", answer: "Lisbon" }
            ])
            .then( c => console.log(`  DB created with ${c.length} elems`))
        )
    } else {  return console.log(`  DB exists & has ${count} elems`); }
})
.then( return app.listen(8000);)               // Server started at port 8000
.catch( err => console.log(`   ${err}`));


   // VIEWs

const index = (quizzes) => `<!-- HTML view -->
<html>
    <head><title>MVC Example</title><meta charset="utf-8"></head>
    <body>
        <h1>MVC: Quizzes</h1>`
+ quizzes.reduce(
    (ac, quiz) => ac +=
`       <a href="/quizzes/${quiz.id}/play">${quiz.question}</a>
        <a href="/quizzes/${quiz.id}/edit"><button>Edit</button></a>
        <a href="/quizzes/${quiz.id}?_method=DELETE"
           onClick="return confirm('Delete: ${quiz.question}')">
           <button>Delete</button></a>
        <br>\n`,
    ""
)
+ `     <p/>
        <a href="/quizzes/new"><button>New Quiz</button></a>
    </body>
</html>`;

const play = (id, question, response) => `<!-- HTML view -->
<html>
    <head><title>MVC Example</title><meta charset="utf-8"></head>
    <body>
        <h1>MVC: Quizzes</h1>
        <form   method="get"   action="/quizzes/${id}/check">
            ${question}: <p>
            <input type="text" name="response" value="${response}" placeholder="Answer" />
            <input type="submit" value="Check"/> <br>
        </form>
        </p>
        <a href="/quizzes"><button>Go back</button></a>
    </body>
</html>`;

const check = (id, msg, response) => `<!-- HTML view -->
<html>
    <head><title>MVC Example</title><meta charset="utf-8"></head>
    <body>
        <h1>MVC: Quizzes</h1>
        <strong><div id="msg">${msg}</div></strong>
        <p>
        <a href="/quizzes"><button>Go back</button></a>
        <a href="/quizzes/${id}/play?response=${response}"><button>Try again</button></a>
    </body>
</html>`;
```

```
const quizForm =(msg, method, action, question, answer) => `<!-- HTML view -->
<html>
    <head><title>MVC Example</title><meta charset="utf-8"></head>
    <body>
        <h1>MVC: Quizzes</h1>
        <form   method="${method}"   action="${action}">
            ${msg}: <p>
            <input  type="text"  name="question" value="${question}" placeholder="Question" />
            <input  type="text"  name="answer"   value="${answer}"   placeholder="Answer" />
            <input  type="submit" value="Create"/> <br>
        </form>
        </p>
        <a href="/quizzes"><button>Go back</button></a>
    </body>
</html>`;


   // CONTROLLER

// GET /, GET /quizzes
const indexController = (req, res, next) => {

     quizzes.findAll()
    .then((quizzes) => res.send(index(quizzes)))
    .catch((error) => `Quizzes not found: \n${error}`);
}

//   GET  /quizzes/1/play
const playController = (req, res, next) => {
    let id = Number(req.params.id);
    let answer = req.query.answer || "";

    quizzes.findById(id)
    .then((quiz) => res.send(play(id, quiz.question, answer)))
    .catch((error) => `Quiz not found - play: \n${error}`);
};

//   GET  /quizzes/1/check
const checkController = (req, res, next) => {
    let answer = req.query.answer, response;
    let id = Number(req.params.id);

    quizzes.findById(id)
    .then((quiz) => {
        response = (quiz.answer===answer) ?
              `Yes, "${answer}" is the ${quiz.question}`
            : `No, "${answer}" is not the ${quiz.question}`;
        return res.send(check(id, response, answer));
    })
    .catch((error) => `Quiz not found - check: \n${error}`);
};

//   GET  /quizzes/1/edit
const editController = (req, res, next) => {

    // ..... introducir código
};

//  PUT  /quizzes/1
const updateController = (req, res, next) => {

    // ..... introducir código
};
```

```
//  GET  /quizzes/new
const newController = (req, res, next) => {
    res.send(quizForm("Create new Quiz", "post", "/quizzes", "", ""));
};

//  POST  /quizzes
const createController = (req, res, next) => {
    let {question, answer} = req.body;

    quizzes.build({question, answer})
    .save()
    .then((quiz) => res.redirect('/quizzes'))
    .catch((error) => `Quiz not created:\n${error}`);
 };

//  DELETE  /quizzes/1
const destroyController = (req, res, next) => {

    // ..... introducir código
};


   // ROUTER

app.get(['/', '/quizzes'],    indexController);
app.get('/quizzes/:id/play',  playController);
app.get('/quizzes/:id/check', checkController);
app.get('/quizzes/new',       newController);
app.post('/quizzes',          createController);

    // ..... introducir rutas de
    //   GET  /quizzes/:id/edit,   PUT  /quizzes/:id/update  y  DELETE  /quizzes/:id


app.all('*', (req, res) =>
    res.send("Error: resource not found or method not supported")
);

========== Final del código ============
```

El esqueleto de esta app está estructurada con el patrón MVC (Model-Vista-Controller). MVC se utiliza mucho en aplicaciones Web de cliente y de servidor, porque estructura de forma clara y concisa una aplicación. A continuación se describen los componentes de la estructura MVC con más detalle y se da también mayor detalle sobre los nuevos componentes a realizar en este ejercicio.

El Modelo.

El modelo define el almacén y el formato de datos para la lista de quizzes. Se define una BBDD gestionada con SQLite3 con una sola tabla donde guardar los quizzes.

Cualquier cambio realizado por un cliente debe guardarse en la BBDD del servidor. De esta forma todos los demás clientes tendrán acceso a dicho cambio. Por ejemplo, si un cliente añade un nuevo quiz, este se mostrará a todos los clientes que accedan al servidor después de ser creado. Lo mismo ocurrirá cuando los quizzes se modifiquen o se borren. Esto es totalmente diferente de la aplicación Quiz de cliente, donde el modelo se guarda en localStorage en el navegador, y el modelo es privado a cada navegador. En este caso los cambios solo los ve el cliente que se ejecuta sobre ese navegador, pero nadie más.

La compartición de datos entre clientes se realiza siempre guardando los cambios en el servidor, normalmente en una BBDD.

Las Vistas.

Las vistas generan las páginas Web en código HTML que se envían al cliente. Las vistas son aquí funciones que insertan los parámetros en un string, generando el código HTML devuelto al cliente.

La vista ***index(quizzes)*** es la más compleja y la renderiza *indexController()*. Esta recibe como parámetro el array con todos los quizzes y genera una página Web con la lista HTML de las preguntas de los quizzes con los botones asociados de Edit y Delete, además de los botones New_Quiz y Rest_Quizzes al final.

Los elementos HTML (texto, botón Edit y botón Delete) de cada quiz llevan el índice *id* del quiz en la BBDD en la ruta asociada a cada transacción HTTP, por ejemplo al clicar en la pregunta de un quiz en *${quiz.question}*, en la ruta */quizzes/:id/play*, *:id* identifica en la tabla el quiz al que se quiere jugar.

Es en esta vista donde hay que añadir el código HTML para que las preguntas de los quizzes, así como los botones de edit y de delete están cada uno en una columna de la tabla, de forma que queden alineados verticalmente.

La vista ***play(id, question, response)*** es el formulario que renderiza *playController(…)* y que permite enviar la respuesta al clicar en Check. Esta vista utiliza el parámetro oculto *response* para no introducir nada en el contenido del cajetín cuando se renderiza la primera vez desde index, pero, en cambio, inicializa el cajetín con la respuesta anterior cuando se vuelve a intentar acertar el quiz al renderizarlo desde la vista *check* con Try_again.

La vista ***check(id, msg, response)*** la renderiza *checkController(…)* y muestra si la respuesta es correcta o no. El botón Try_again lleva un parámetro oculto en el query (la respuesta enviada), para inicializar el cajetín con la respuesta anterior en la vista play.

La vista ***quizForm(msg, method, action, question, answer)*** se utiliza para renderizar el formulario de edición de quizzes en *newController(…)*. Este formulario debe reutilizarse para renderizar el nuevo formulario de creación de quizzes en edit*Controller()*.

Las acciones del Controlador

indexController(…) renderiza la lista de los quizzes, que obtiene de la BBDD, junto con los botones para editar, borrar y crear.

playController(…) renderiza el formulario para jugar con el quiz, identificado por la ruta recibida en la primitiva HTTP. El cajetín se inicializa con el parámetro oculto (vuelta desde *check* con Try_again) o con el string vacío (primer intento).

checkController(…) renderiza la página que indica si la respuesta enviada es correcta o no.

newController(…) renderiza el formulario de creación de un quiz (vista *quizForm(msg, method, action, question, answer)*). *method* y *action* se configuran para que envíe la transacción HTTP: "POST /quizzes". Los parámetros *question y answer* deberán llevar el string vacío ("") para que al renderizar los cajetines del formulario se muestren vacíos.

createController(…) actualiza el modelo guardando el nuevo quiz en la BBDD. Al finalizar redirecciona a "GET /quizzes", para enviar la lista de quizzes al cliente.

Las nuevas acciones del controlador a diseñar deberán hacer lo siguiente:

editController(…) deberá renderizar el formulario de edición de un quiz, inicializando los cajetines del formulario con la pregunta y la respuesta del quiz identificado por su *id* en la ruta. Para ello se debe buscar primero en la BBDD el quiz asociado. *method* y *action* se deben configurar para que

envíe la transacción HTTP: "PUT /quizzes/:id". **Ojo!** El formulario debe enviar el método PUT con method override como un parámetro oculto en la ruta (*action*), ya que los formularios solo permite los métodos GET y POST.

updateController(…) debe actualizar el modelo en la BBDD con las nuevas pregunta y respuesta que se reciben en la transacción HTTP recibida. Al finalizar debe redireccionar a "GET /quizzes", para enviar la lista de quizzes actualizada al cliente.

destroyController(…) debe actualizar el modelo eliminando el quiz identificado por su *id* en la ruta. Antes de eliminarlo debe pedir confirmación con el método *confirm()* de JavaScript de cliente, que genera un pop-up. La vista *index(quizzes)* incluye la invocación a confirm() necesaria. Al finalizar debe redireccionar a "GET /quizzes", para enviar la lista de quizzes al cliente.

El Router de Eventos.

El router recibe transacciones HTTP a las que responde invocando la acción del controlador asociada al método y ruta recibidos, tal y como es habitual en MVC de servidor. La lista de transacciones HTTP, junto con los middlewares controladores que se deben instalar en la app express una vez completada con la nueva funcionalidad solicitada, es:

```
GET     /quizzes              ->  indexController()
GET     /quizzes/:id/play     ->  playController()
GET     /quizzes/:id/check    ->  checkController()
GET     /quizzes/:id/edit     ->  editController()
PUT     /quizzes/:id          ->  updateController()
GET     /quizzes/:id/new      ->  newController()
POST    /quizzes/:id          ->  createController()
DELETE  /quizzes/:id          ->  destroyController()
```

Prueba de la práctica

Para comprobar que la práctica ha sido realizada correctamente hay que utilizar el validador de este repositorio

https://github.com/practicas-ging/mooc_node-mod10_quiz_mvc_server

Recuerde que para utilizar el validador se debe tener node.js (y npm) (https://nodejs.org/es/) y Git instalados. El proyecto se descarga, instala y ejecuta en el ordenador local con estos comandos:

```
$                                ## El proyecto debe clonarse en el ordenador local
$ git clone https://github.com/practicas-ging/mooc_node-mod10_quiz_mvc_server
$
$ cd mooc_mooc_node-mod10_quiz_mvc_server    ## Entrar en el directorio de trabajo
$
$ npm install                                ## Instala el programa de test
$
$                             ##  ->  Incluir la solución en el esqueleto clonado
$
$ npm run checks                             ## Pasa los tests al fichero solicitado
.....................................        ## en el directorio de trabajo
.....................................
... (resultado de los tests)
$
```

Una vez descargado, se debe entrar en el directorio raiz **mooc_node-mod10_quiz_mvc_server**. El fichero **mooc_node-mod10_quiz_mvc_server.js** se ha incluido incompleto en el directorio raíz del proyecto descargado. Este debe completarse con el editor o sustituirse por otro del mismo nombre que contenga la solución.

Los tests pueden pasarse las veces que sea necesario. Si se pasan nada más descargar el proyecto, indicarán que no se ha realizado ninguno de los tres comandos. También pueden utilizarse para probar el programa de otro compañero sustituyendo los ficheros que se desee probar. El programa de test incluye además un comando para generar el fichero ZIP

```
$
$ npm run zip         ##  Comprime los ficheros del directorio en un fichero .zip
$
```

Este genera el fichero **mooc_node-mod10-quiz_MVC_server_entregable.zip** con el directorio de la practica comprimido. Este fichero ZIP debe subirse a la plataforma para su evaluación.

Instrucciones para la Entrega y Evaluación.

Se debe entregar el fichero **mooc_node-mod10-quiz_MVC_server_entregable.zip** con los ficheros comprimidos de la entrega.

El evaluador debe descargar el fichero entregado y comprobar que funciona correctamente. El fichero descargado es un paquete npm, que puede instalarse con todas sus dependencias con npm. Una vez instalado puede ejecutarse o pueden pasarse los test.

RUBRICA. Se puntuará el ejercicio a corregir sumando el % indicado a la nota total si la parte indicada es correcta:
- 25%: Si el controlador **editController** esta bien implementado
- 25%: Si el controlador **updateController** esta bien implementado
- 25%: Si el controlador **destroyController** esta bien implementado
- 25%: Si la instalación de los 3 controladores en el router está bien realizada

Si pasa todos los tests se deberá dar la máxima puntuación.

El objetivo de este curso es sacar el máximo provecho al trabajo dedicado y para ello lo mejor es utilizar las evaluaciones para ayudar al evaluado, especialmente a los principiantes. Al evaluar se debe dar comentarios sobre la corrección del código, su claridad, legibilidad, estructuración y documentación, siempre que puedan ayudar al evaluado.

¡Cuidado! Una vez enviadas, tanto la entrega, como la evaluación, no se pueden cambiar. Esperar a tener completa y revisada, tanto la entrega, como la evaluación antes de enviarlas.