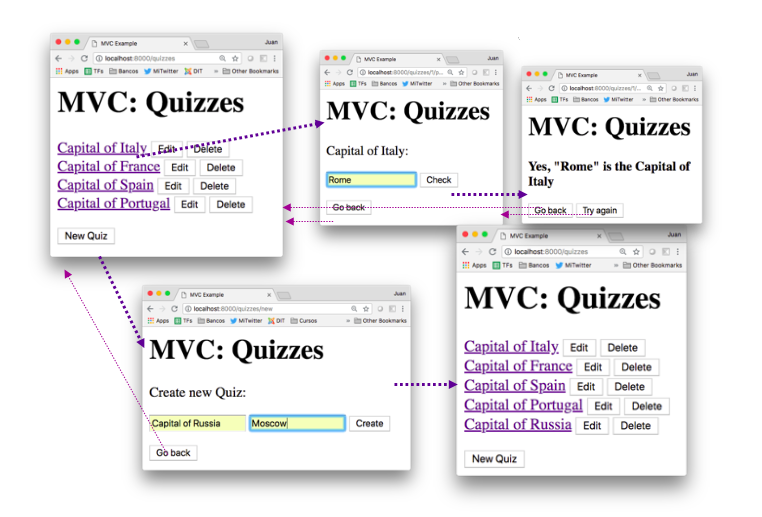
**Módulo 6: Introducción a Express.js y sus Middlewares, MVC (Modelo - Vista - Controlador), AJAX - Entrega P2P: Quiz server**

**Objetivos**

* Entender y practicar con una aplicación de servidor Express
* Comprender y aplicar el patrón MVC
* Conocer los principales paquetes npm que acompañan a Express: body-parser, method-override, sequelize y sqlite3.

**Descripción de la práctica**

El proyecto clonado contiene el fichero main.js: un esqueleto de un servidor web que consiste en un juego de preguntas y respuestas. El esqueleto funciona correctamente solo para la parte del juego relacionada con contestar (play) a los quizzes (hacer click en la pregunta) y crearlos (create), tal y como muestran las siguientes capturas:

[](https://raw.githubusercontent.com/ging-moocs/MOOC_node_mod6_quiz-server_entrega/master/img/diagrama.png)

*Nota*: Estas capturas tienen la misma funcionalidad que el esqueleto dado, pero su aspecto es ligeramente diferente de los que muestra el esqueleto.

El esqueleto envía la lista de preguntas a un navegador que acceda al URL [http://localhost:8000](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 servidor también permite crear nuevos quizzes, tal y como se ve en las capturas.

Mostrar quizzes, jugar y crear quizzes funciona porque las siguientes primitivas de la interfaz REST tienen todos sus elementos de su MVC implementados (router, controlador, modelo y vista):

GET /quizzes -> indexController()

GET /quizzes/:id/play -> playController()

GET /quizzes/:id/check -> checkController()

GET /quizzes/new -> newController()

POST /quizzes -> createController()

Sin embargo, 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 main.js) para que todos los botones funcionen correctamente. En particular el código añadido debe incluir:

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

GET /quizzes/:id/edit

PUT /quizzes/:id/update

DELETE /quizzes/:id

* Añadir la vista *editView(..)* de edición de quizzes que falta.
* Introducir una tabla HTML en la vista *indexView(quizzes)* para que los botones de editar y borrar queden bien alineados en una columna. Opcionalmente, también se puede mejorar el aspecto de las vistas con nuevo CSS.

El esqueleto de esta app está estructurado 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 dando instrucciones sobre los nuevos componentes a realizar en este ejercicio.

### 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.

**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 **indexView (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 del botón New\_Quiz 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 [${quiz.question}](https://github.com/ging-moocs/MOOC_node_mod6_quiz-server_entrega/blob/master/%E2%80%9C/quizzes/$%7Bquiz.id%7D/play%E2%80%9D), 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 **playView (quiz, 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 resultView con Try\_again.

La vista **resultView (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 playView.

La vista **newView (quiz)** se utiliza para renderizar el formulario de creación de quizzes en **newController(...)**.

La vista **editView (quiz)** que hay que crear, debe utilizase para renderizar el formulario de edición de quizzes en editController(). Los atributos del formulario, method y action, deben configurarse 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 de HTML solo permiten los métodos GET y POST.

**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 newView (quiz)). 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 nuevo formulario de edición de un quiz (vista editView (quiz) a desarrollar). Los parámetros question y answer deberán llevar el contenido del quiz en la BBDD para que al renderizar los cajetines del formulario se muestre el quiz que se quiere editar. Por ello se debe buscar primero en la BBDD el quiz asociado.
* **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.

**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/new -> newController()

POST /quizzes -> createController()

DELETE /quizzes/:id -> destroyController()

**Descargar el código del proyecto**

El proyecto debe clonarse en el ordenador desde el que se está trabajando:

$ git clone https://github.com/ging-moocs/MOOC\_node\_mod6\_quiz-server\_entrega

A continuación se debe acceder al directorio de trabajo y abrir el fichero index.html con el editor de la elección del alumno.

$ cd MOOC\_node\_mod6\_quiz-server\_entrega

**Prueba de la práctica**

Para ayudar al desarrollo, se provee una herramienta de autocorrección que prueba las distintas funcionalidades que se piden en el enunciado. Para utilizar esta herramienta debes tener node.js (y npm) (<https://nodejs.org/es/>) y Git instalados.

Para instalar y hacer uso de la [herramienta de autocorrección](https://www.npmjs.com/package/moocauto) en el ordenador local, ejecuta los siguientes comandos en el directorio del proyecto:

$ npm install -g moocauto ## Instala el programa de test

$ moocauto ## Pasa los tests al fichero a entregar

............................ ## en el directorio de trabajo

... (resultado de los tests)

También se puede instalar como paquete local, en el caso de que no se dispongas de permisos en el ordenador desde el que estás trabajando:

$ npm install moocauto ## Instala el programa de test

$ npx moocauto ## Pasa los tests al fichero a entregar

............................ ## en el directorio de trabajo

... (resultado de los tests)

Se puede pasar la herramienta de autocorrección tantas veces como se desee.

**Entrega de la práctica**

El alumno debe subir un fichero comprimido ZIP incluyendo todos los ficheros de la práctica excepto el directorio node\_modules (si existe).

**Evaluación de la práctica**

La evaluación de la práctica se realizará mediante revisión por pares (P2P). Cada alumno tendrá que revisar la práctica de 3 de sus compañeros y otros 3 revisarán la suya. Se puede utilizar la herramienta de autocorrección (moocauto) como ayuda para revisar la práctica de los compañeros.

El objetivo de este curso es sacar el máximo provecho al trabajo que están dedicando, por lo que les recomendamos que utilicen la evaluación para ayudar a sus compañeros enviando comentarios sobre la corrección del código, su claridad, legibilidad, estructuración y documentación.

Dado que es un curso para principiantes, ante la duda les pedimos que sean benevolentes con sus compañeros, porque muchos participantes están empezando y los primeros pasos siempre son difíciles.

**OJO! Una vez enviada la evaluación, está no se puede cambiar.** Piensen bien su evaluación antes de enviarla.

**RÚBRICA:** Se puntuará el ejercicio a corregir sumando el % indicado a la nota total si la parte indicada es correcta:

* **50%:** La funcionalidad de editar un quiz funciona correctamente
* **50%:** La funcionalidad de borrar un quiz funciona correctamente