Programación Web 2 Servidor Validators, Middlewares y Ejercicio (put y delete)

U-TAD

Validators

Deberíamos hacer validación de los datos recibidos antes de enviarlos al modelo para escritura (createItem).

npm i express-validator

Con express-validator realizamos las comprobaciones justo antes del controller.

Creamos el directorio *validators/* y dentro comenzamos con *tracks.js*

```
const { check } = require("express-validator")
const validateResults = require("../utils/handleValidator")
const validatorCreateItem = [
    check("name").exists().notEmpty(), //.isLength(min:5, max:90)
    check("album").exists().notEmpty(),
    check("cover").exists().notEmpty(),
    check("artist").exists().notEmpty(),
    check("artist.name").exists().notEmpty(),
    check("artist.nickname").exists().notEmpty(),
    check("artist.nationality").exists().notEmpty(),
    check("duration.start").exists().notEmpty().isInt(),
    check("duration.end").exists().notEmpty().isInt(),
    check("mediaId").exists().notEmpty().isMongoId(),
    (req, res, next) => validateResults(req, res, next)
]
module.exports = { validatorCreateItem }
```

Validators

En utils/ creamos handleValidator.js

```
const { validationResult } = require("express-validator")

const validateResults = (req, res, next) => {
    try {
        validationResult(req).throw()
        return next()
    } catch (err) {
        res.status(403)
        res.send({ errors: err.array() })
    }
}

module.exports = validateResults
```

Y en routers/tracks.js

```
…
const { validatorCreateItem } = require("../validators/tracks")
…
router.post("/", validatorCreateItem, createItem)
```

Middleware

Actúa entre la ruta y el controlador

```
router.post("/", validatorCreateItem, customHeader, createItem)
```

Puede revisar autentificación y autorización entre otros.

Por ejemplo, podemos buscar una cabecera “custom” tipo api-key

En el middleware recibimos los parámetros (req, res, **next**)

En el cliente:

```
POST http://localhost:3000/api/tracks HTTP/1.1
Content-Type: application/json
api_key: Api-publica-123
```

Middleware

Creamos el directorio middleware/ y dentro el fichero customHeader.js

```
const customHeader = (req, res, next) => {
    try {
        const apiKey = req.headers.api_key;
        if(apiKey === 'Api-publica-123') { //Probar con otra para ver el error
            next()
        }else {
            res.status(403).send("api key no es correcto")
        }
    }catch(err) {
        res.status(403).send(err)
    }
}
module.exports = customHeader
```

Recuerda incluirlo en routes/tracks.js

```
const customHeader = require("../middleware/customHeader")

...
router.post("/", validatorCreateItem, customHeader, createItem)
```

Más adelante, veremos cómo gestionar tokens de sesión con los middlewares.

Manejo de errores

En utils/, creamos handleError.js

```
const handleHttpError = (res, message, code = 403) => {
    res.status(code).send(message)
}
module.exports = { handleHttpError }
```

En controllers/tracks.js

```
const getItems = async (req, res) => {
    try{
        const data = await tracksModel.find({})
        res.send(data)
    }catch(err){
//Si nos sirve el de por defecto que hemos establecido, no es necesario pasar el 403
        handleHttpError(res, 'ERROR_GET_ITEMS', 403)
}
```

Manejo de errores (matchedData)

En controllers/tracks.js

```
…
const { matchedData } = require('express-validator')
const { handleHttpError } = require('../utils/handleError')
…
const createItem = async (req, res) => {
    try {
      const body = matchedData(req) //El dato filtrado por el modelo (probar con body=req)
      const data = await tracksModel.create(body)
      res.send(data)
    }catch(err){
        handleHttpError(res, 'ERROR_CREATE_ITEMS')
    }
}
```

En index.http añadir algún campo “fake” en el post a tracks.

Ejercicio

Para **tracks.js** completa el resto de controladores:

- controllers/tracks.js: getItem, updateItem y deleteItem

Asegúrate de que todas las funciones sean “async” (porque usaremos “await”)

Asegúrate de que tengas *router.get(“/:id, getItem)* en routes/tracks.js

A getItem, añádele validación en validators/tracks.js e incluyela en su ruta.

```
const validatorGetItem = [
    check("id").exists().notEmpty().isMongoId(),
    (req, res, next) => {
        return validateResults(req, res, next)
    }
]
module.exports = { validatorCreateItem, validatorGetItem }
```

Úsala en el controlador getItem, tal que

```
const {id} = matchedData(req)
```

```
const data = tracksModel.findById(id)
```

Para probarlo desde el index.http, primero llama a “/api/tracks” y toma el valor de cualquier _id, y después “/api/tracks/<valor_id>”

Ejercicio *continuación

Crea la rutas .put y .delete para updateItem y deleteItem respectivamente en el controlador:

updateItem

```
…
const {id, ...body} = matchedData(req) //Extrae el id y el resto lo asigna a la constante body
const data = await tracksModel.findOneAndUpdate(id, body);
…
```

deleteItem

```
…
const {id} = matchedData(req)
const data = await tracksModel.deleteOne({_id:id});
…
```

Soft Delete

Estrategia de borrado lógico: no se borra realmente de la base de datos, te permite recuperar los datos.

Mira en Mongo Atlas: Database -> Browse Collections

Las colecciones son las mismas que las que definimos en el modelo.

Borramos todos los registros e instalamos un paquete en node:

npm i mongoose-delete

Soft Delete

En models/tracks.js (aplicar lo mismo al resto):

```
const mongooseDelete = require("mongoose-delete")
…
TracksScheme.plugin(mongooseDelete, {overrideMethods: "all"})
module.exports = mongoose.model("tracks", TracksSchema)
```

Subir con petición POST un track. Comprueba en la colección Tracks que ahora, en el nuevo registro insertado, tenemos una propiedad “deleted”: false.

En controllers(tracks), modifica el “deleteOne” por “delete”

```
const data = await tracksModel.delete({_id:id});
```

Realiza la petición DELETE al último registro desde index.http (busca el id realizando un GET previo):

```
DELETE http://localhost:3000/api/tracks/63fba226c391fd40d29c4dd7
```

Comprueba en la colección tracks que aún tienes el dato, pero con el parámetro *“deleted”: true*

Soft Delete

Si ahora consulto el registro con una petición GET, no aparece.

Es decir, se ha realizado un borrado lógico, pero el dato sigue existiendo en la BD y es fácilmente recuperable.

Con permisos de administrador sobre la BD puedes cambiar de nuevo el “deleted” a false y volvería a verse. Puedes hacerlo tú mismo actualizando el registro en la base de datos MongoDB.

Haz lo mismo en los otros dos modelos, storage y users.

Ejercicio

1.-Completa el CRUD en controllers/storage y controllers/users tal y como hemos hecho en controllers/tracks.

2.-Completa también las rutas.

3.-Duplica los validadores, tal que tengas uno por esquema.

4.- En deleteItem de storage, necesitaremos borrar el fichero físico.

```
const fs = require("fs")
const MEDIA_PATH = __dirname + "/../storage"
const deleteItem = async (req, res) => {
        const {id} = matchedData(req)
        const dataFile = await storageModel.findById(id)
        await storageModel.deleteOne(id)
        const filePath = MEDIA_PATH + "/" + dataFile.filename
        fs.unlinkSync(filePath)
}
```

Nota: En storage, no vamos a usar en este caso updateItem()