

TEMA 3. (Parte 1)

SERVICIOS WEB REST

Pedro Delgado Pérez, David Corral Plaza, Guadalupe Ortiz Bellot

Grado en Ingeniería Informática
Departamento de Ingeniería Informática

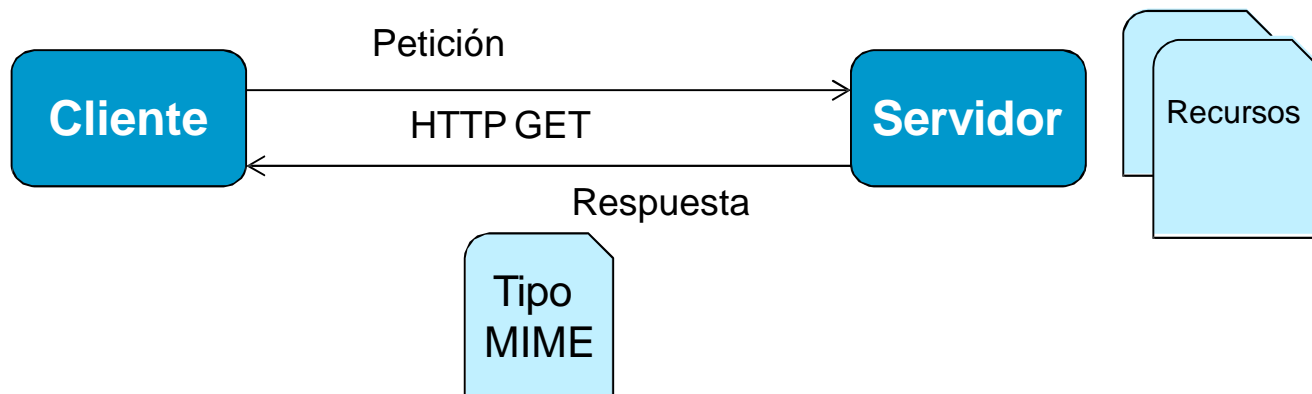
ÍNDICE

- **Introducción**
- Conceptos básicos de REST
- RESTful Hello World
- Creando API de películas
- Mejoras: *routes*, CORS y persistencia
- Conclusiones

REST A VISTA DE PÁJARO

REST: Representational State Transfer.

1. Un usuario hace una petición (por ejemplo, *GET*) a una dirección para solicitar un recurso, por ejemplo un documento HTML a través del navegador.
2. El navegador envía una **petición HTTP** al servidor. Los datos se encapsulan con un formato simple, como XML.
3. El servidor envía una **respuesta HTTP**, en este caso un documento HTML, con un tipo *MIME* (*text/html*, *text/css*, *image/png*,...).



REST vs SOAP

REST: particularmente útiles cuando solo es necesario transmitir y recibir mensajes simples. Los servicios construidos siguiendo esta arquitectura se denominan **RESTful**.

- *Ligero (menos ancho de banda)*
- *Más legible*
- *Fácil de construir*
- *Varios formatos soportados.*
- *Puede usar SOAP*

SOAP	REST
1. Simple Object Access Protocol	1. Representational State Transfer
2. Function-driven (data available as services, e.g.: "getUser")	2. Data-driven (data available as resources, e.g. "user").
3. Message format supported: XML	3. Message format supported: Plain text, HTML, XML, JSON, YAML, etc.
4. Transfer Protocols supported: HTTP, SMTP, UDP, etc.	4. Transfer Protocols supported: HTTP
5. SOAP is recommended for Enterprise apps, financial services, payment gateways	5. REST is recommended for mobile applications and Social networking applications.
6. Highly secure and supports distributed environment.	6. Less secured and not suitable for distributed environment.

Fuente: Dev Community: <https://dev.to/himanshudevghupta/different-restfull-api-and-soap-api-4jn1>

SOAP: se utiliza sobre todo para las aplicaciones empresariales para integrar tipos y aplicaciones más complejos, así como sistemas *legacy*.

- *Fuerte tipado - comprobación de tipos*
- *Cooperación entre sistemas diversos*
- *Más seguridad y mayor regulación del tratamiento de errores.*

ÍNDICE

- Introducción
- **Conceptos básicos de REST**
- RESTful Hello World
- Creando API de películas
- Mejoras: *routes*, CORS y persistencia
- Conclusiones

PRINCIPIOS

REST es una metodología de desarrollo de sistemas hipermedia distribuidos basada en servicios mediante estándares web.

Principios fundamentales:

1. **Arquitectura**: Basada en recursos, cada uno de los cuales puede ser identificado y accedido mediante una **URI** única.
2. **Protocolo**: La comunicación entre el cliente y el servidor es *sin estado* (mediante **HTTP**).
3. **Navegación**: A través de **hiperenlaces** el usuario puede navegar de un recurso a otros.
4. **Comunicación**: Basada en los métodos HTTP estándar: principalmente **GET, POST, PUT** y **DELETE** (operaciones **CRUD**).



¿CÓMO ACCEDER A LOS RECURSOS?

URL: Universal Resource Locator

- URL base: *protocolo* + *dirección* + *ruta al recurso*
`https://www.miservidor.com/MiAplicacion/api/...`
- Para cada tipo de recurso, habitualmente consideramos dos tipos de URL:
 - **Conjunto**: `.../people/`
 - *Operamos sobre la lista de todas las personas*
 - **Individual**: `.../people/pepe`
 - *Operamos sobre una persona en concreto (Pepe)*
- Existe la posibilidad de añadir una *query*:
 - **Búsquedas** (filtrar los resultados con parámetros)
`.../people?age=18&city=cadiz`
 - **Paginación** (paginar el nº de resultados: 10 resultados a partir del registro 100)
`.../people?limit=10&offset=100`
 - **Vistas personalizadas** (recuperar atributos concretos)
`.../people?fields=name,age`

GESTIÓN RESULTADOS

- **Conjunto** - Operaciones CRUD (Create Retrieve Update Delete)

- CREAR UNA PERSONA → POST .../people/
- OBTENER TODAS LAS PERSONAS → GET .../people/
- ~~ACTUALIZAR LAS PERSONAS~~ → PUT .../people/
- BORRAR TODAS LAS PERSONAS → DELETE .../people/

- **Individual** - Operaciones CRUD

- ~~CREAR UNA PERSONA~~ → POST .../people/pepe
- OBTENER UNA PERSONA → GET .../people/pepe
- ACTUALIZAR UNA PERSONA → PUT .../people/pepe
- BORRAR UNA PERSONA → DELETE .../people/pepe

RECURSO	POST	GET	PUT	DELETE
/people	Crea una persona	Lista todas las personas	✗	Elimina todas las personas
/people/pepe	✗	Muestra los datos de “pepe”	Actualiza los datos de “pepe”.	Elimina a “pepe”

CÓDIGOS DE ESTADO

Códigos de estado: nos proporcionan información sobre el resultado de una petición.

- Básicos y comunes:

ÉXITO (2xx)

200 → Ok

201 → Created

ERROR cliente (4xx)

400 → Bad request

401 → Unauthorized

404 → Not Found

405 → Method Not Allowed

409 → Conflict

ERROR servidor (5xx)

500 → Internal
Server Error

- Pero hay muchos más:

<https://www.restapitutorial.com/httpstatuscodes.html>

FORMATO DE DATOS > JSON

- **Express:** se utiliza principalmente JSON como formato de datos para la comunicación entre servidor y cliente.

```
{  
  "name" : " Pepe",  
  "age" : 18,  
  "phones" : [ 987654321, 666777888 ],  
  "emails": [ "pepe@uca.es", "pepe@Gmail.com" ]  
}
```

Validador: <https://jsonformatter.curiousconcept.com/>

- **Ejercicio:** Formatea el JSON de un array de películas (*movies*) en inglés:

Title	Director	Year
Jurassic Park	Steven Spielberg	1993
The Lion King	Rob Minkoff, Roger Allers	1994

¿Cuál sería la ruta base para acceder a todas las películas? ¿Y solo a una por title?

MEDIOS PARA CREAR UNA API REST

Java:

- JAX-RS
- Spring, Spark

Python:

- Django, Flask

PHP:

- Slim, Laravel

NodeJs:

- Ampliamente extendido a día de hoy en multitud de portales webs.
- **Desarrollo modular**: cada módulo tiene un espacio de nombres independiente

Express:

- Biblioteca sencilla de usar y flexible para crear APIs RESTful en Node.js.
- **Capa intermedia**: recibe las peticiones y las procesa para acceder al recurso.



ÍNDICE

- Introducción
- Conceptos básicos de REST
- **RESTful Hello World**
- Creando API de películas
- Mejoras: *routes*, CORS y persistencia
- Conclusiones

Paso 1 – Iniciar proyecto Node

Prerrequisitos: Node + editor de texto

- Creamos una carpeta (*HelloWorld*) en una ruta representativa.
- Abrimos una consola de comandos en dicha carpeta, ejecutamos el comando y rellenamos:

`npm init`

- Esto creará nuestro *package.json*

```
pedro@pedro-espada:~/Documentos/PNET/HelloWorld$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (helloworld)
version: (1.0.0)
description: primera API con NodeJS
entry point: (index.js)
test command:
git repository:
keywords:
author: Pedro Delgado
license: (ISC)
About to write to /home/pedro/Documentos/PNET/HelloWorld/package.json:

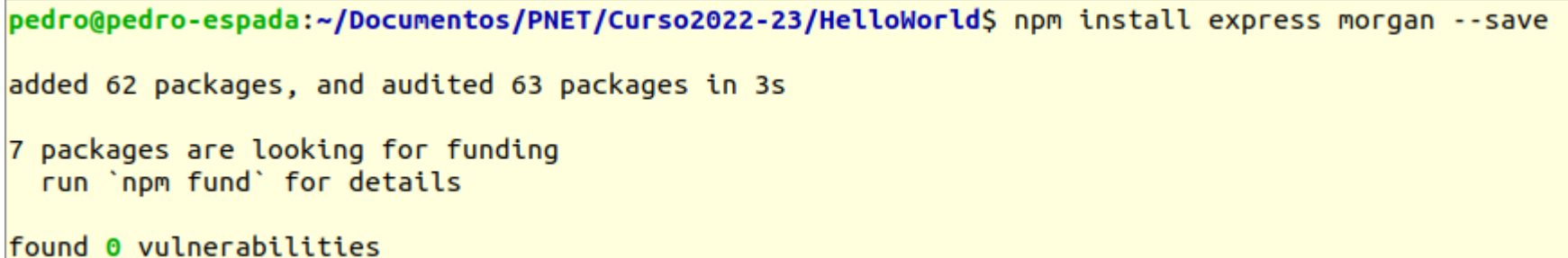
{
  "name": "helloworld",
  "version": "1.0.0",
  "description": "primera API con NodeJS",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Pedro Delgado",
  "license": "ISC"
}

Is this OK? (yes) yes
```

Paso 2 – Instalar dependencias

- El siguiente comando instalará las dependencias necesarias:

```
npm install express morgan --save
```



A terminal window showing the command `npm install express morgan --save` being executed. The output indicates that 62 packages were added and 63 packages were audited in 3 seconds. It also mentions that 7 packages are looking for funding and that no vulnerabilities were found.

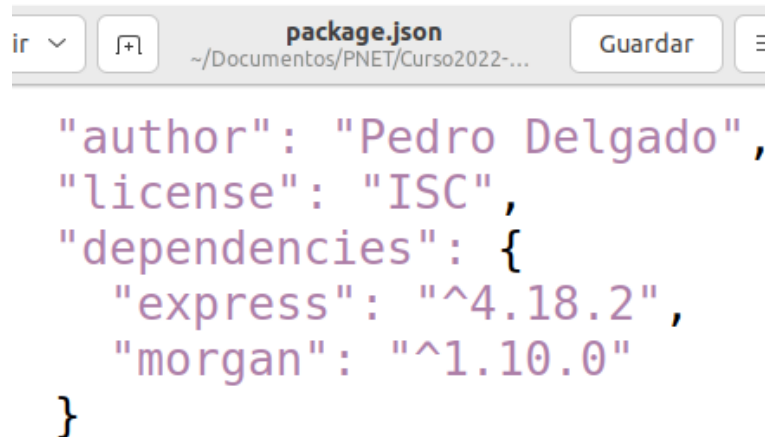
```
pedro@pedro-espada:~/Documentos/PNET/Curso2022-23/HelloWorld$ npm install express morgan --save
added 62 packages, and audited 63 packages in 3s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

- Se descargarán las dependencias y con `--save` se añadirán a nuestro *package.json*.

Nota: Las versiones de estas dependencias pueden variar



A screenshot of a code editor showing the `package.json` file. The file contains the following JSON structure:

```
{
  "author": "Pedro Delgado",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "morgan": "^1.10.0"
  }
}
```

Paso 3 – Programar API REST

- Creamos un fichero “*index.js*” (o nombre especificado como *entry point* en el *package.json*) en el directorio raíz del proyecto (*HelloWorld* en nuestro caso).
- Añadimos dependencias y creamos las **constantes**:

```
const express = require('express');  
const app = express();  
const logger = require('morgan');  
const http = require('http');  
const path = require('path');  
const PORT = process.env.PORT || 8080;  
const baseAPI = '/api/v1'; (opcional)
```

Paquete [express](#)

Paquete [morgan](#):
Función de logger

Biblioteca [http](#):
*Nos permitirá crear un
servidor HTTP*

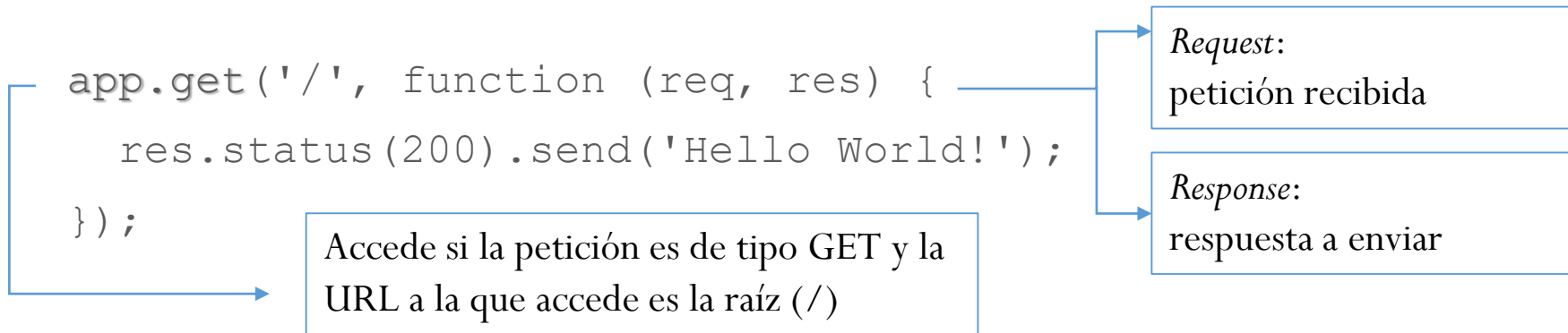
Biblioteca [path](#):
*Nos será de utilidad más
adelante*

- Añadimos **configuraciones** para la app:

```
app.use(express.json());  
app.use(express.urlencoded({extended: true}));  
app.use(logger('dev'));
```

Paso 3 – Programar API REST

- Creación de un Middleware (MW): Añadimos uno para *método GET* :



Nota: los middleware de tipo `app.use()` reciben cualquier tipo de petición.

- Instanciamos el *servidor* y queda a la escucha de peticiones:

```
const server = http.createServer(app);  
  
server.listen(PORT, function () {  
  console.log('Server up and running on localhost:' + PORT);  
});
```

Por defecto, queda a la escucha en el *puerto* 8080

Paso 4 – Iniciar API REST

- Volvemos a la consola de comandos/terminal, en la raíz de nuestro proyecto, y ejecutamos la siguiente orden:

```
node index.js
```

- Si todo ha ido bien, nos saldrá:

“Server up and running on localhost:8080”

- Accedemos a <http://localhost:8080/> y veremos: “Hello World!”
- Finalmente, paramos el servidor NodeJs con:

```
Control + C
```

Adicional: Para no estar parando/arrancando el servidor cada vez:

```
npm install -g nodemon
```

A partir de ese momento, ejecutar *nodemon* en vez de *node*: [+info](#)

```
nodemon index.js
```

ÍNDICE

- Introducción
- Conceptos básicos de REST
- RESTful Hello World
- **Creando API de películas**
- Mejoras: *routes*, CORS y persistencia
- Conclusiones

Paso 1 - Crear recurso *movies*

- A continuación vamos a crear la API de películas que antes hemos diseñado. Continuaremos con el código que acabamos de utilizar, pero haremos una copia en una nueva carpeta.
- Vamos a almacenar nuestras películas en un array (simulando una BBDD), para ello añadimos en *index.js*:

```
let movies = [  
  {  
    "title": "Jurassic Park",  
    "director": "Steven Spielberg",  
    "year": 1993  
  },  
  {  
    "title": "The Lion King",  
    "director": ["Rob Minkoff", "Roger Allers"],  
    "year": 1994  
  }  
];
```

Paso 2 – Método GET /movies y POST /movies

- Añadimos el **método GET** para recuperar todas las películas:

```
app.get('/movies', function (req, res) {  
  res.status(200).send(movies);  
});
```

Se envían las películas y el estado 200 de éxito

- Volvemos a iniciar el servidor NodeJS **en el nuevo directorio de trabajo**:

```
node index.js
```

- Accedemos a <http://localhost:8080/movies> y verificamos.

- Añadimos el **método POST** para crear una nueva película:

```
app.post('/movies', function (req, res) {  
  movies.push(req.body);  
  res.status(201).send("Film created!");  
});
```

Obtenemos la película enviada del cuerpo de la petición

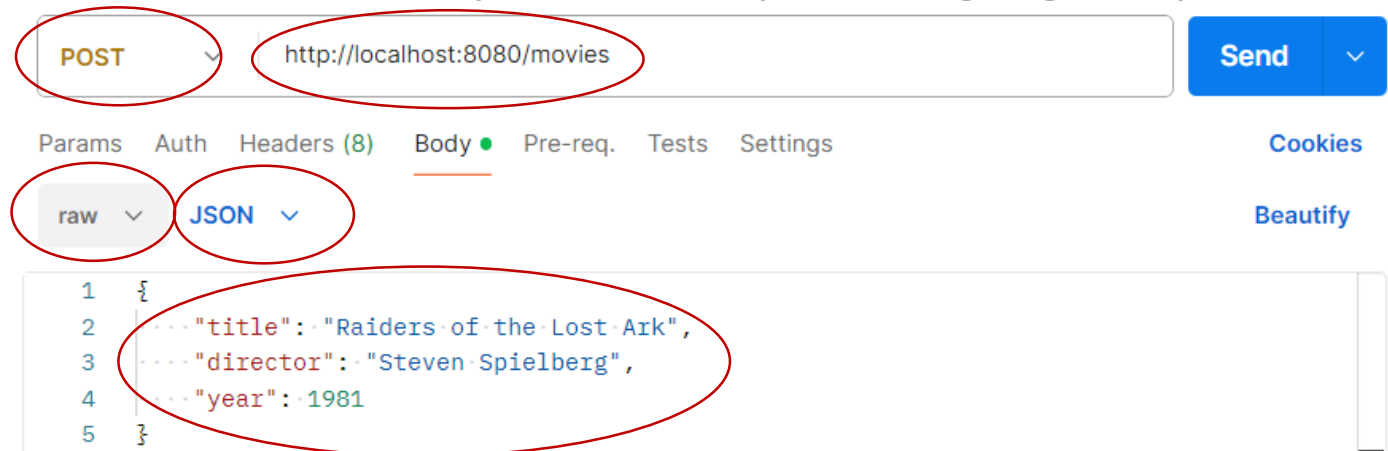
Estado 201: recurso creado con éxito

Paso 2 – Probar método POST /movies

- Abrimos **Postman** y configuramos un envío de petición POST:
 - Creamos una nueva colección de nombre “movies” (panel derecho).
 - Método: POST - - - URL: <http://localhost:8080/movies>
 - Pestaña “**Body**”: (marcar ‘raw’) y seleccionar ‘JSON (application/json)’:

```
{  
  "title": "Raiders of the Lost Ark",  
  "director": "Steven Spielberg",  
  "year": 1981  
}
```

- Reiniciar servidor Node / Pulsar botón azul “*Send*” en Postman.
- Acceder de nuevo a la URL y verificar que se agregó la película.

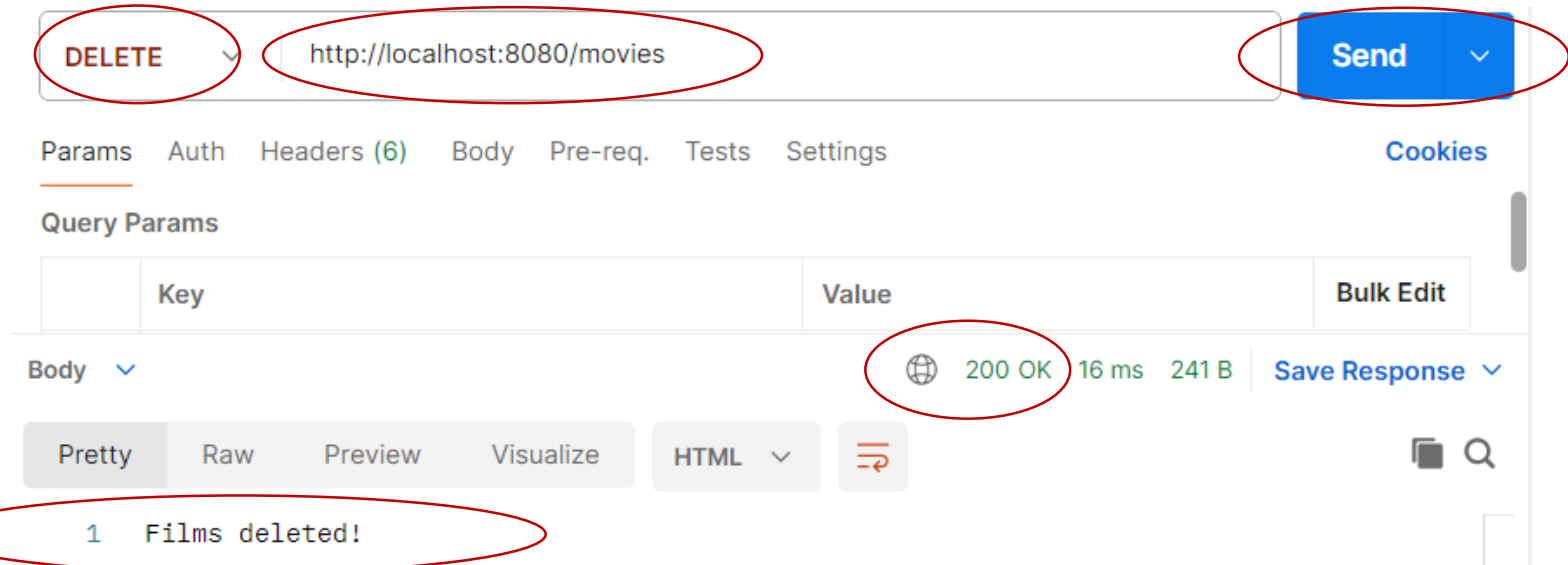


Paso 3 – Método DELETE /movies

- Añadimos el método DELETE para eliminar **todas** las películas:

```
app.delete('/movies', function (req, res) {  
  movies = [];  
  res.status(200).send("Films deleted!");  
});
```

- Abrimos **Postman** y configuramos un envío de petición DELETE:
 - Método: DELETE
 - URL: <http://localhost:8080/movies>



Paso 4 – Métodos GET y PUT /movies/title

- Añadimos el método GET para **recuperar una película**:

Define los parámetros a extraer de la URL: lo que va tras “/movies/” se guarda en el parámetro “title”

```
app.get('/movies/:title', function (req, res) {  
  let title = req.params.title;  
  res.status(200).send(movies.filter(m => m.title === title));  
});
```

Saca el título de los parámetros

- Accedemos a <http://localhost:8080/movies/The Lion King> para verificar que nos devuelve la información de la película.

- Añadimos el método PUT para **actualizar una película**:

findIndex encuentra la película dentro del array, movies, y *splice* elimina 1 espacio sin dejar huecos

```
app.put('/movies/:title', function (req, res) {  
  let title = req.params.title;  
  movies.splice(movies.findIndex(m => m.title === title), 1);  
  movies.push(req.body);  
  res.status(200).send("Film updated!");  
});
```

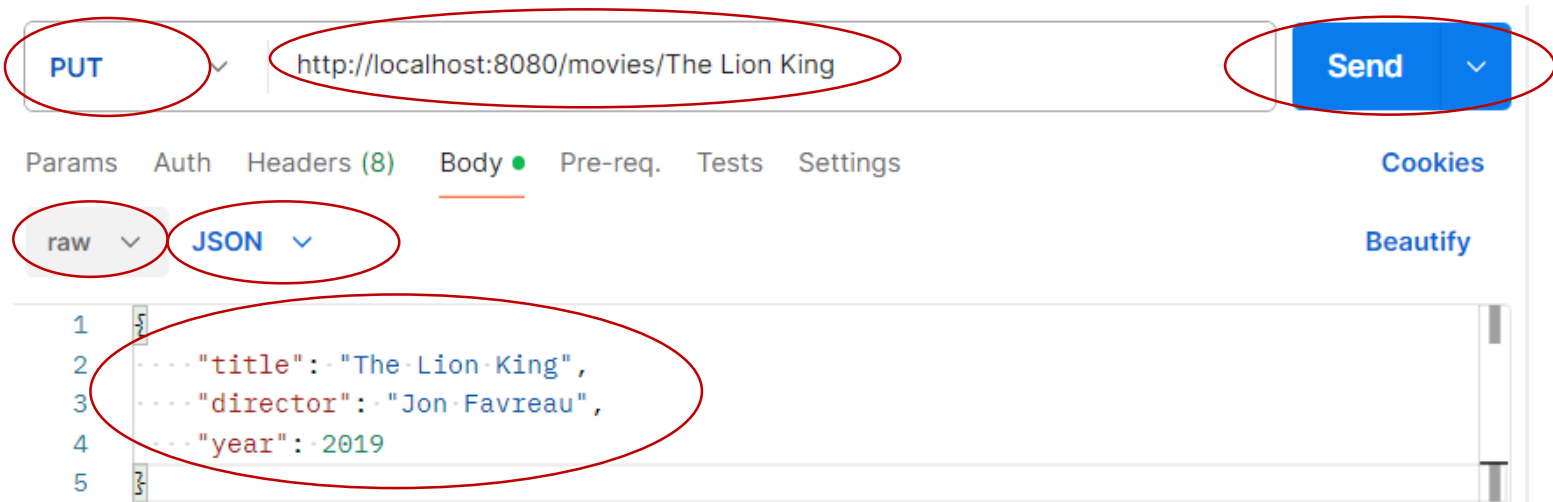
Paso 4 – Probar el método PUT /movies/title

- Abrimos **Postman** y configuramos un envío de petición PUT:

- Método: PUT --- URL: <http://localhost:8080/movies/The Lion King>
- Body (marcar 'raw') y seleccionar 'JSON (application/json)':

```
{  
  "title": "The Lion King",  
  "director": "Jon Favreau",  
  "year": 2019  
}
```

- Reiniciar servidor Node / Pulsar botón azul “Send” en Postman.
- Acceder de nuevo a la URL y verificar que se modificó la película.

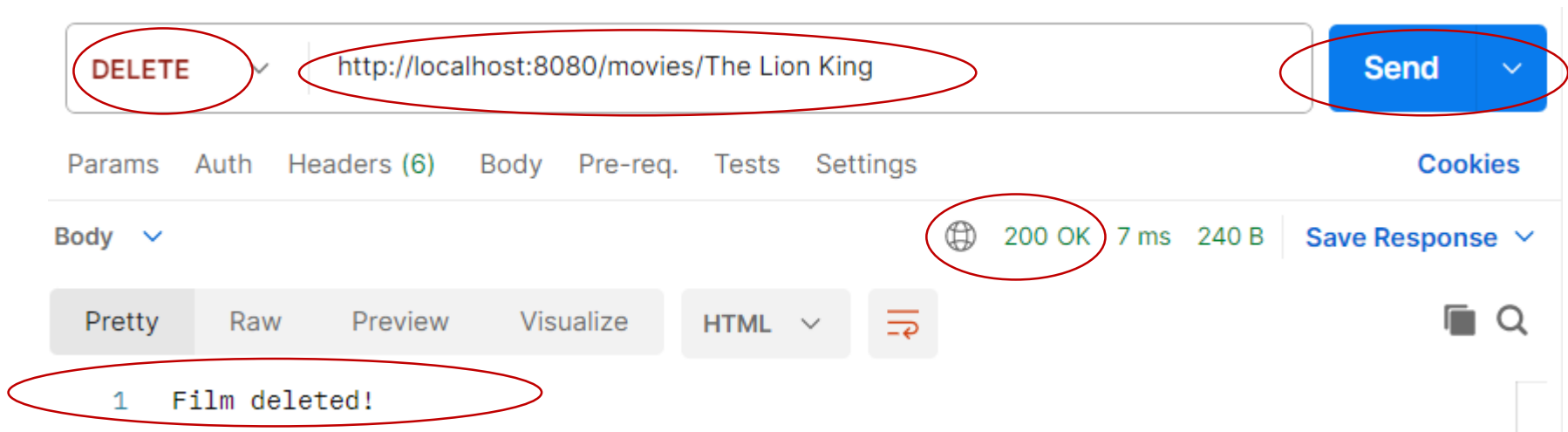


Paso 5 – Método DELETE /movies/title

- Añadimos el método DELETE para eliminar una película:

```
app.delete('/movies/:title', function (req, res) {  
  let title = req.params.title;  
  movies = movies.filter(m => m.title !== title);  
  res.status(200).send("Film deleted!");  
});
```

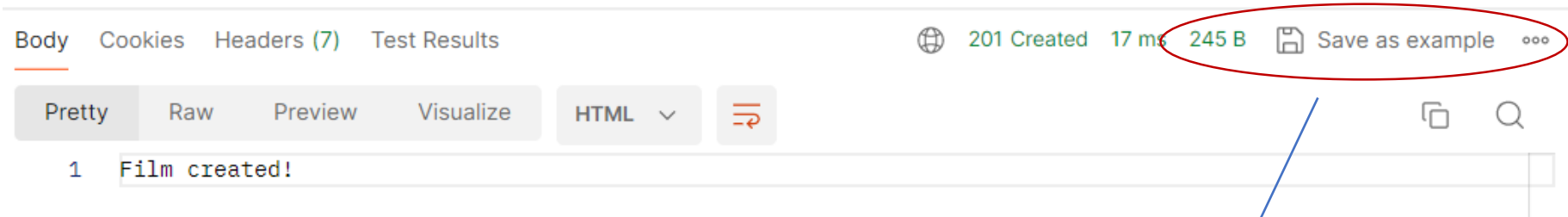
- Abrimos **Postman** y configuramos un envío de petición DELETE:
 - Método: DELETE
 - URL: <http://localhost:8080/movies/The Lion King>



Último paso – Guardar resultados de POSTMAN

Es posible guardar la colección de nuestras peticiones HTTP, así como **los resultados obtenidos tras su ejecución**:

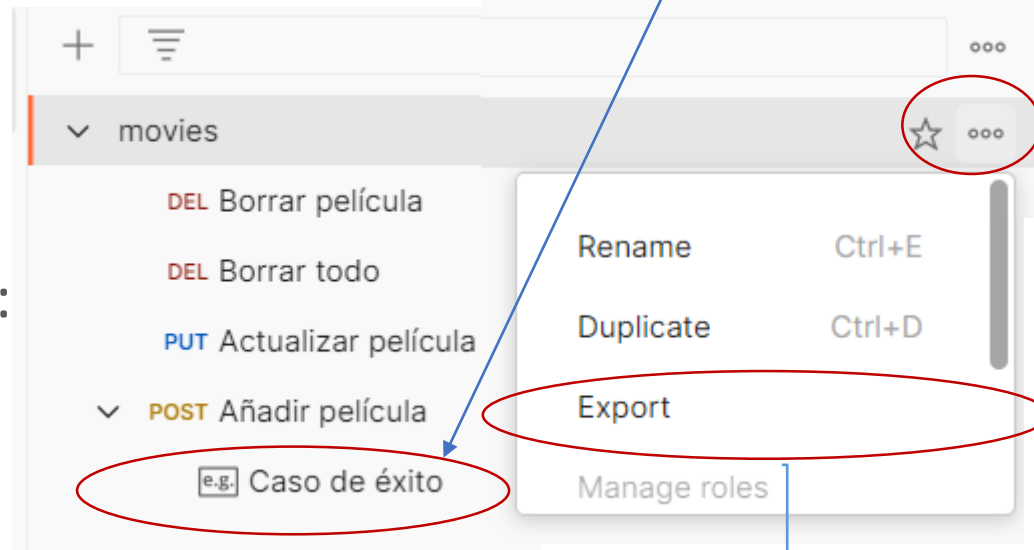
- **Guardar resultado ejecución**: “Save as example”:



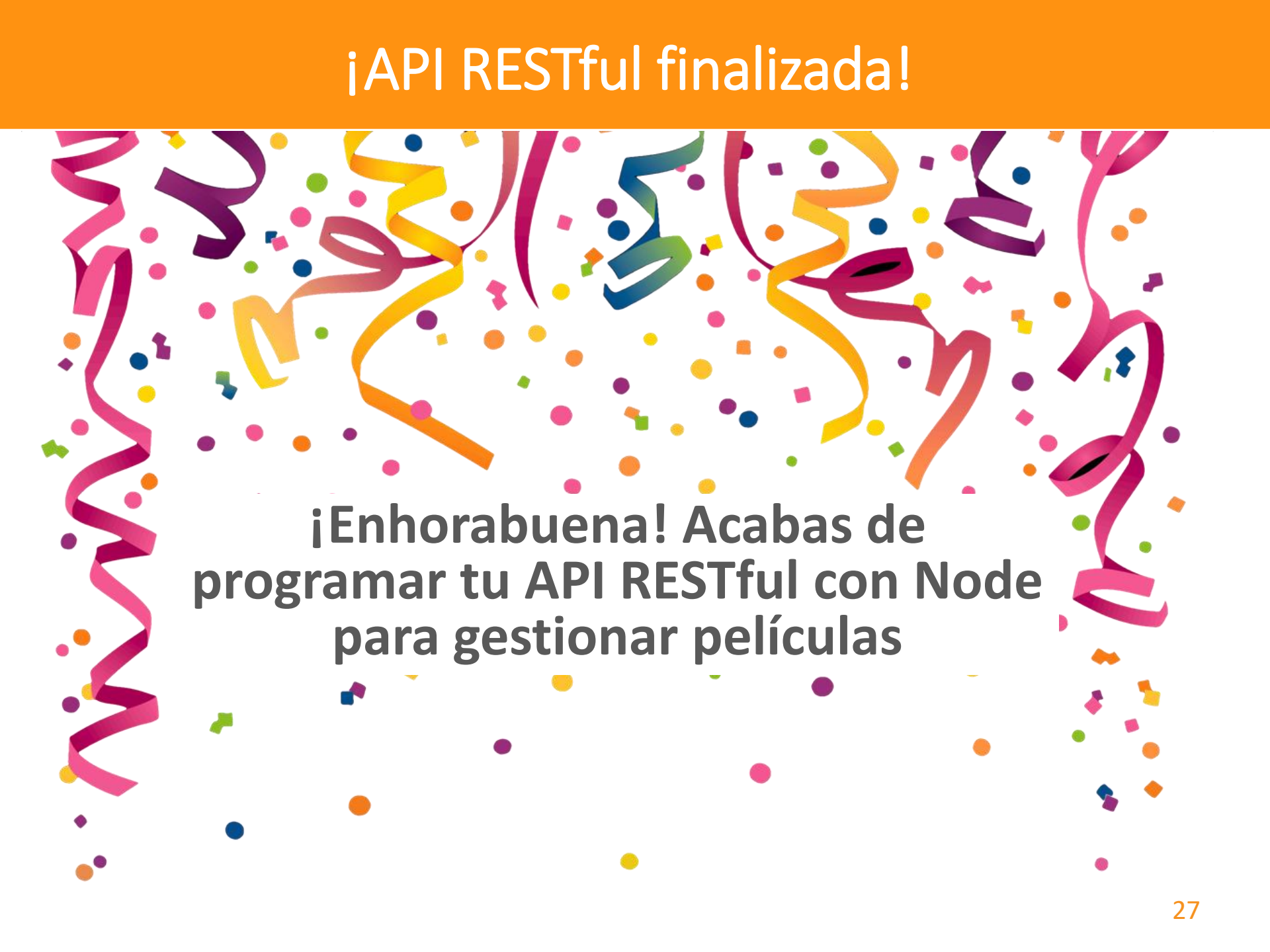
- **Exportar colección**: Sobre la colección creada, desplegar el menú y opción “Export”.

Se generará un fichero JSON:

```
"response": [
  {
    "name": "Caso de éxito",
    "originalRequest": {
      "method": "POST",
      "header": [],
      "body": {
        "mode": "raw",
```



¡API RESTful finalizada!

The background of the slide is decorated with a festive pattern of colorful streamers and confetti. The streamers are in shades of pink, purple, yellow, and blue, and the confetti consists of small circles and squares in various colors like orange, green, and blue.

**¡Enhorabuena! Acabas de
programar tu API RESTful con Node
para gestionar películas**

ÍNDICE

- Introducción
- Conceptos básicos de REST
- RESTful Hello World
- Creando API de películas
- **Mejoras: *routes*, CORS y persistencia**
- Conclusiones

AGRUPAR POR ROUTES – Creación de *routes*

- Cuando tenemos más de un recurso en nuestra API (*movies*, *people*, *books*, ...) el *index.js* empieza a sobrecargarse.
- Para solucionar eso, Express utiliza los **routes**, que es una forma de organizar y empaquetar recursos dentro de una API.

PASO 1: (Partimos de nuestra API de películas)

- Creamos la carpeta *routes* en la raíz del proyecto.
- Creamos el fichero *movies.js* dentro de *routes*.

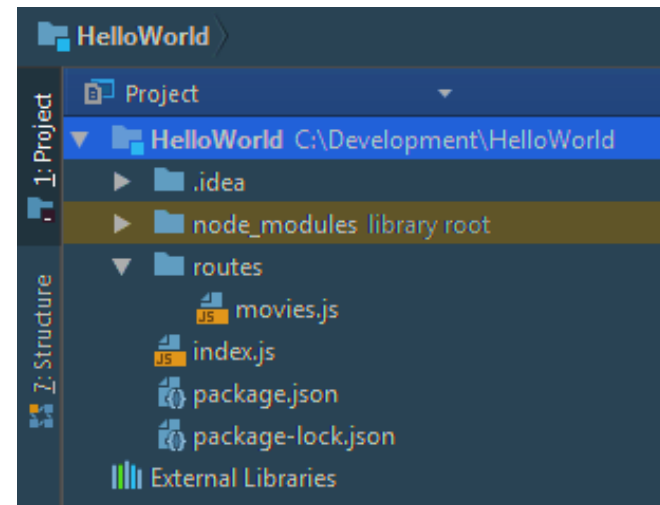
PASO 2:

- Abrimos el fichero *movies.js* y añadimos:

```
'use strict';
```

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

```
const router = express.Router();
```



Programar el *route* de movies

- A continuación **cortamos del *index.js*** todas las operaciones (GET, POST, PUT, DELETE) definidas para “*movies*” en este fichero:
 - Sustituyendo la palabra ‘app’ por ‘router’; y
 - eliminando ‘movies’ de la ruta del método, ejemplo:

ANTES

```
app.get('/movies', function (req, res) {  
  res.send(movies);  
});
```

AHORA

```
router.get('/', function (req, res) {  
  res.send(movies);  
});
```

- Cortamos del *index.js* nuestro array de películas y lo pegamos en *movies.js*, antes de las operaciones.
- Añadimos la siguiente línea al final del fichero *movies.js* para **exportar el enrutador** que hemos creado:

```
module.exports = router;
```

Preparar el *index.js* y probar

- Eliminadas las operaciones de API del *index.js*, antes de la inicialización del servidor (`const server = ...`) añadimos lo siguiente para importar el enrutador y utilizarlo en nuestra API:

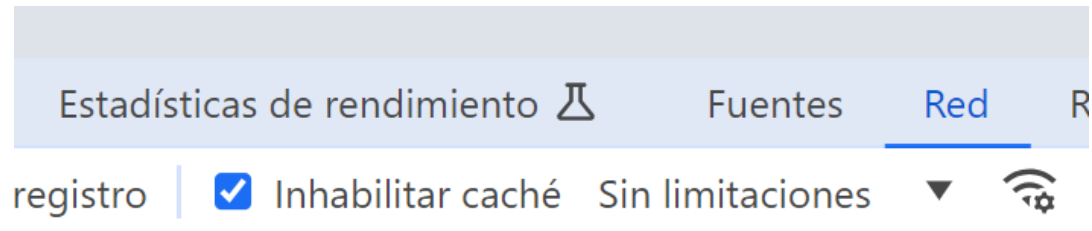
```
const movies = require('./routes/movies');  
app.use('/movies', movies);
```

- Reiniciamos el servidor y accedemos a la siguiente URL para verificar que la API funciona: <http://localhost:8080/movies>

```
pedro@pedro-espada:~/Documentos/PNET/PeliculasRoutes$ node index.js  
Server up and running on localhost:8080  
GET /movies 200 11.827 ms - 149
```

- Si ves un estado 304, no te preocupes, se debe a la caché:

Si quieres, puedes inhabilitarla en el navegador



HABILITANDO CORS

CORS (Cross-Origin Resource Sharing)

- Es un mecanismo de seguridad para las transacciones HTTP con AJAX.
- Está basado en la política *Same origin*, pero es algo más flexible (comunicación entre servidor y navegador).
- Cuando hacemos una petición *HTTP CORS* de un servidor a otro diferente, si el destinatario no tiene configuradas las CORS, denegará la solicitud.

Más info sobre CORS:

https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS/

Más info sobre diferentes configuraciones:

<https://github.com/expressjs/cors>

Descargar, incluir y habilitar CORS

- Vamos a instalar las dependencias de CORS para nuestro proyecto, por lo tanto ejecutamos el siguiente comando:

```
npm install cors --save
```

```
pedro@pedro-espada:~/Documentos/PNET/PeliculasRoutes$ npm install cors --save
added 2 packages, and audited 67 packages in 1s
```

- En nuestro *index.js* agregamos la dependencia de CORS justo después de las demás dependencias:

```
const cors = require('cors');
```

- En el uso más simple, añadimos la siguiente línea de código para habilitar todos los orígenes:

```
app.use(cors());
```

AÑADIR PERSISTENCIA

- Hasta el momento, como habrás podido comprobar, los datos se están almacenando en un array que, cuando reinicias el servidor, todo vuelve a como estaba por defecto. En cualquier entorno de producción eso es inviable.
- **Solución:** dotar de persistencia a nuestra API para que los datos se mantengan.
- Podemos utilizar bases de datos SQL y NoSQL. En este caso usaremos **MongoDB Atlas**.



- El primer paso es registrarse en la plataforma (*ver manual de instalación del Tema 3*).

Paso 1.1 – Configurar seguridad

Primer paso:

- Acceder a “Database Access”. Indicamos un usuario y una contraseña.
- **No uses contraseñas personales** (puedes necesitar compartirlas).
- **Pero no olvides la contraseña** ya que tendrás que usarla más tarde.
- Una buena opción para que la contraseña sea robusta es dar al botón de autogeneración.
- Selecciona “Atlas admin” como “Built-in Role”
- Click en “Add User”.

Built-in Role

Select one [built-in role](#) for this user.

Atlas admin ▼

1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password

Certificate

i We autogenerated a username and password for your first database user in this project. See MongoDB Cloud registration information.


Create a database user using a username and password. Users will be given the *read and write* privilege by default. You can update these permissions and/or create additional users later. Your autogenerated credentials are different to your MongoDB Cloud username and password.

Username

testPNET

Password 

.....

 Autogenerate Secure Password

Create User

Paso 1.2 – Configurar seguridad

Segundo paso:

- Acceder a “Network Access”. Hacer click en “Add IP Address” y seleccionar “Add Your Current IP Address”. *Se puede añadir una descripción.*
- Esto bloquearía peticiones a nuestra BBDD que no sean desde esta IP.

2 Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.



My Local Environment

Use this to add network IP addresses to the IP Access List. This can be modified at any time.



Cloud Environment

Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

ADVANCED

Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters.

IP Address

Description

Enter IP Address

Enter description

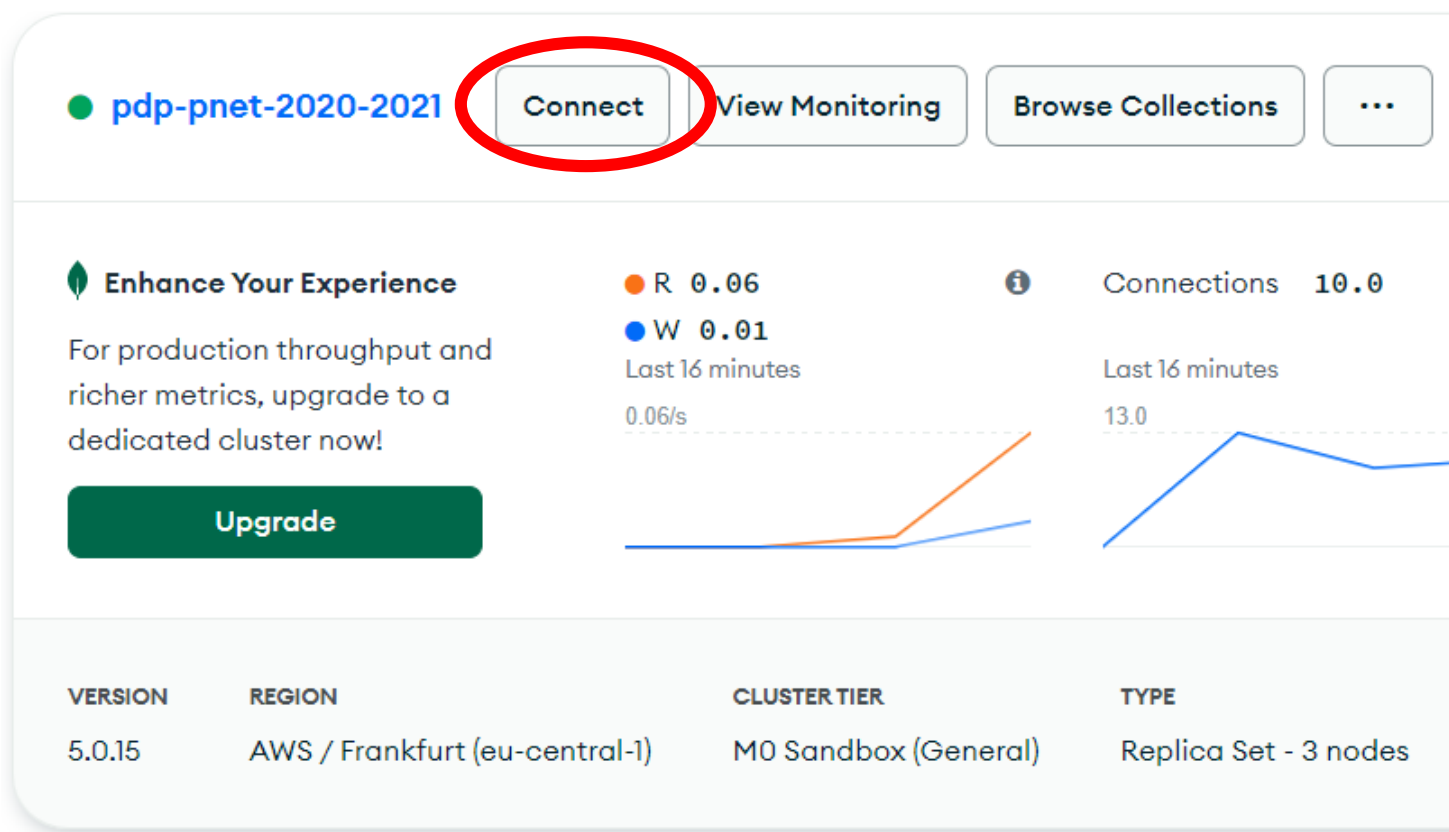
Add My Current IP Address

Add Entry

Paso 1.3 – Configurar seguridad

Tercer paso:

- Una vez finalizado el proceso de registro y de creación del clúster pasamos a configurar la seguridad.
- Entramos en Database, y hacemos click en “Connect”.



The screenshot shows the MongoDB Atlas interface for a cluster named **pdp-pnet-2020-2021**. The **Connect** button is highlighted with a red circle. Below the cluster name, there are three buttons: **Connect**, **View Monitoring**, and **Browse Collections**, followed by a menu icon (three dots).

Below the buttons, there is a section titled **Enhance Your Experience** with the text: "For production throughput and richer metrics, upgrade to a dedicated cluster now!". There is a green **Upgrade** button.

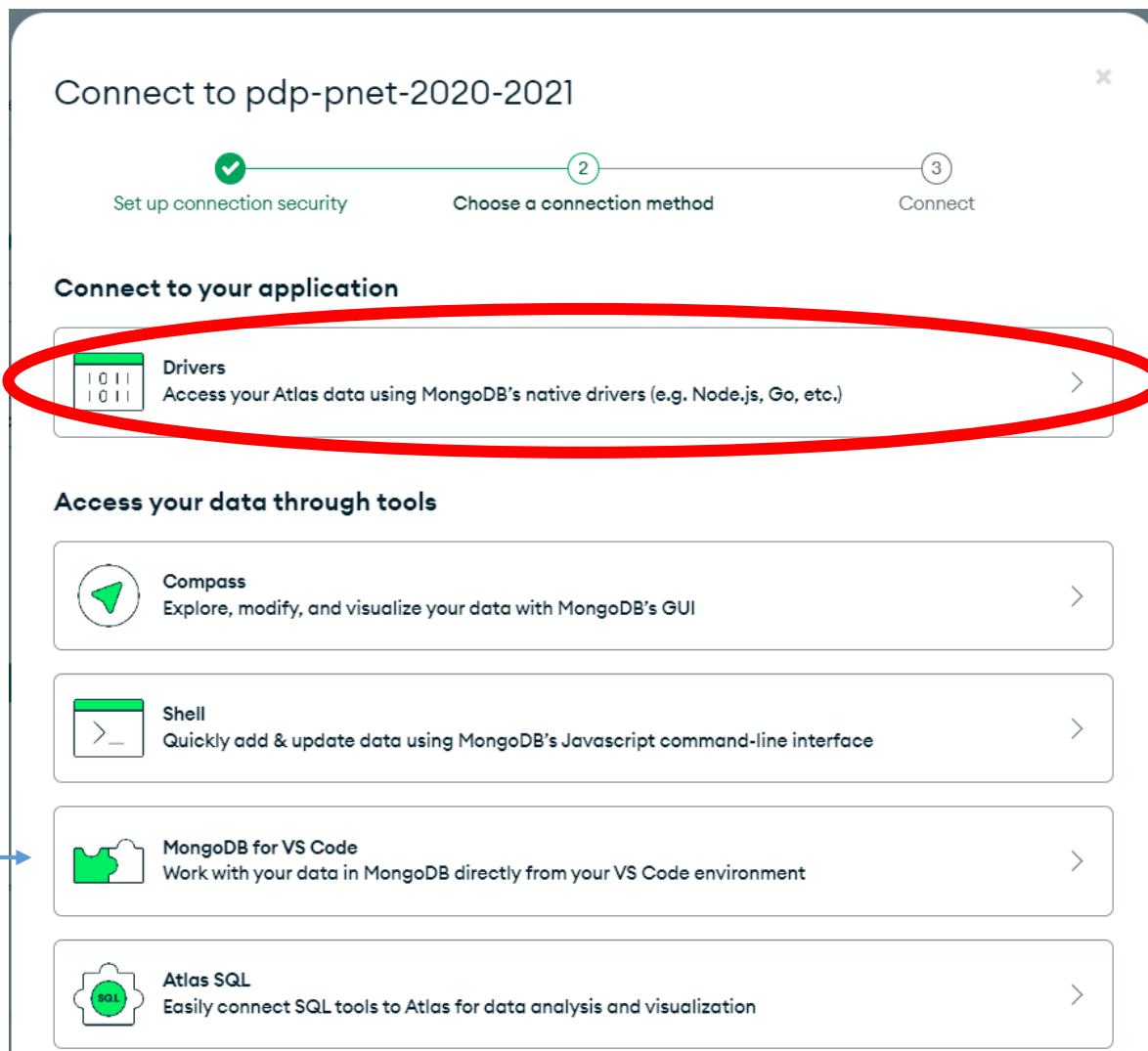
To the right of the upgrade section, there are two line graphs. The first graph shows **R 0.06** (orange line) and **W 0.01** (blue line) for the **Last 16 minutes**. The second graph shows **Connections 10.0** (blue line) for the **Last 16 minutes**.

At the bottom, there is a table with the following data:

VERSION	REGION	CLUSTER TIER	TYPE
5.0.15	AWS / Frankfurt (eu-central-1)	M0 Sandbox (General)	Replica Set - 3 nodes

Paso 1.4 – Configurar seguridad

Cuarto paso: Elije la opción “Drivers”



*Notar también
la existencia
de esta opción*

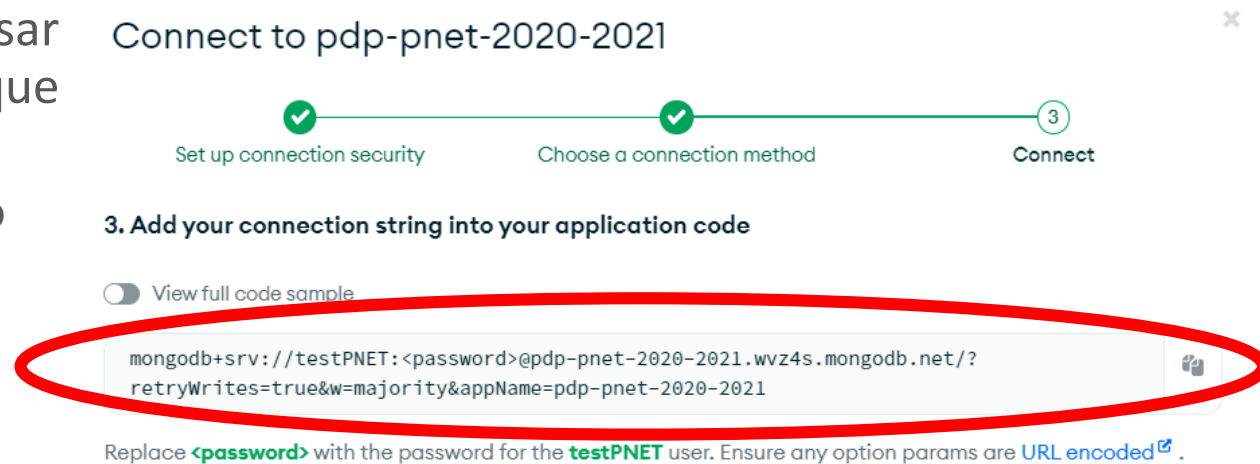
Paso 1.5 – Configurar seguridad

- **Quinto paso:** Dejamos todo por defecto. En esta ocasión lo importante es el “string” que aparece en la cajita del punto 2 (*copiar con el botón a la derecha*):

```
mongodb+srv://<user>:<password>@pdp-pnet-2020-2021.wvz4s.mongodb.net/  
myFirstDatabase?retryWrites=true&w=majority
```

- Cuando vayamos a usar dicha cadena, hay que sustituir:

- **<user>** por el usuario que pusimos en el paso 1.1.
- **<password>** por la contraseña que pusimos en el paso 1.1.

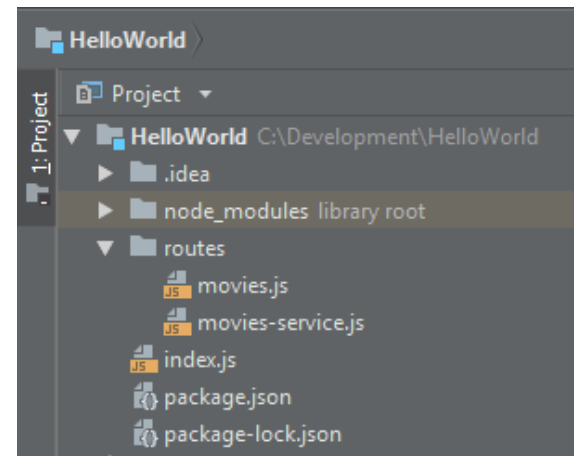


- Por lo tanto, siguiendo el ejemplo, mi cadena de conexión será:

```
mongodb+srv://testPNET:MICONTRASEÑA@pdp-pnet-2020-2021.wvz4s.mongodb.net/  
myFirstDatabase?retryWrites=true&w=majority&appName=pdp-pnet-2020-2021
```

Paso 2.1 – Añadir en movies-service.js

- Una vez con nuestra BBDD creada, es el momento de configurar nuestra API para que la emplee.
- Creamos el fichero “*movies-service.js*” dentro de la carpeta *routes*, quedando nuestra estructura de ficheros similar a:



- Añadimos las siguientes **constantes**:

```
'use strict';
```

```
const MongoClient = require('mongodb').MongoClient;
```

```
let db;
```

```
let ObjectId = require('mongodb').ObjectId;
```

```
const Movies = function () {  
};
```

Paquete [mongodb](#)

Paso 2.2 – Añadir en movies-service.js

- Añadimos lo siguiente. En rojo lo que varía para cada proyecto:
(Mongo URI que configuramos antes y el nombre de la BBDD a conectar).

```
Movies.prototype.connectDb = function (callback) {  
    MongoClient.connect("mongodb+srv://testPNET:MICONTRASEÑA@pdp-pnet-  
2020-2021.wvz4s.mongodb.net/myFirstDatabase?retryWrites=true&w=majority",  
        {useNewUrlParser: true, useUnifiedTopology: true},  
        function (err, database) {  
            if (err) {  
                console.log(err);  
                callback(err);  
            }  
  
            db=database.db('pdp-pnet-2020-2021').collection('movies');  
            console.log("Conexión correcta");  
            callback(err, database);  
        });  
};
```

- Cambios forzosos
- Opcional

Paso 2.3 – Añadir en movies-service.js

- Añadimos los métodos para trabajar con la base de datos:

Añadir a la BBDD:

```
Movies.prototype.add = function (movie, callback) {  
    return db.insertOne(movie, callback);  
};
```

¿Qué es una *función callback*?

https://www.w3schools.com/js/js_callback.asp

Recuperar de la BBDD:

```
Movies.prototype.get = function (_id, callback) {  
    return db.find({_id: ObjectId(_id)}).toArray(callback);  
};
```

En este caso, buscamos el identificador de MongoDB
Pero podríamos buscar por otros campos también

```
Movies.prototype.getAll = function (callback) {  
    return db.find({}).toArray(callback);  
};
```

Nos devuelve
un array con
los resultados

Paso 2.4 – Añadir en movies-service.js

- Añadimos los métodos para trabajar con la base de datos.

Actualizar la BBDD:

```
Movies.prototype.update = function (_id, updatedMovie, callback) {  
  delete updatedMovie._id;  
  return db.updateOne(  
    {_id: ObjectId(_id)}, {$set: updatedMovie}, callback);  
};
```

Actualiza el registro con los valores recibidos

Eliminar de la BBDD:

```
Movies.prototype.remove = function (_id, callback) {  
  return db.deleteOne({_id: ObjectId(_id)}, callback);  
};
```

```
Movies.prototype.removeAll = function (callback) {  
  return db.deleteMany({}, callback);  
};
```

```
module.exports = new Movies();
```

Finalmente, se realiza la exportación de las operaciones

Paso 3.1 – Editar movies.js

- Vamos a reconfigurar nuestro *movies.js* para que realice las operaciones sobre la base de datos y no sobre el array.

- En primer lugar eliminamos el array que teníamos:

```
let movies = [...];
```

- Importamos el fichero que habíamos creado previamente (*movies-service.js*), añadiendo la siguiente línea (en negrita) justo después de la inicialización del router:

```
const express = require('express');  
const router = express.Router();  
const moviesService = require('./movies-service');
```

Paso 3.2 – Editar movies.js

- Actualizamos el código de nuestro método GET /movies

Antes:

```
router.get('/', function (req, res) {  
  res.status(200).send(movies);  
});
```

Cuando se produzca algún error

Cuando no haya películas a devolver

Caso de éxito

Después:

```
router.get('/', function (req, res) {  
  moviesService.getAll((err, movies) => {  
    if (err) {  
      res.status(500).send({  
        msg: err  
      });  
    } else if (movies.length == 0) {  
      res.status(500).send({  
        msg: "movies null"  
      });  
    } else {  
      res.status(200).send(movies);  
    }  
  })  
});
```

Paso 3.3 – Editar movies.js

- Actualizamos el código de nuestro método POST /movies

Antes:

```
router.post('/', function (req, res) {  
  movies.push(req.body);  
  res.status(201).send("Film created!");  
});
```

Opcional: En primer lugar, se puede comprobar que no se introduce una película vacía.

Devolvemos un error 400 para indicar que el error proviene del cliente.

Después:

```
router.post('/', function (req, res) {  
  let movie = req.body;  
  if(Object.entries(movie).length === 0){  
    res.status(400).send({msg: 'Empty movie'});  
  }else{  
    moviesService.add(movie, (err, movie) => {  
      if (err) {  
        res.status(500).send({ msg: err });  
      } else {  
        res.status(201).send({  
          msg: 'Film created!'  
        });  
      }  
    });  
  }  
});
```

Paso 3.4 – Editar movies.js

- Actualizamos el código de nuestro método DELETE /movies

Antes:

```
router.delete('/', function (req, res) {  
  movies = [];  
  res.status(200).send("Films deleted!");  
});
```

Después:

```
router.delete('/', function (req, res) {  
  moviesService.removeAll((err) => {  
    if (err) {  
      res.status(500).send({  
        msg: err  
      });  
    } else {  
      res.status(200).send({  
        msg: 'Films deleted!'  
      });  
    }  
  });  
});
```

Paso 3.5 – Editar movies.js

- Actualizamos el código de nuestro método GET /movie/{id}

Antes:

```
router.get('/:title', function (req, res) {
  let title = req.params.title;
  res.status(200).send(
    movies.filter(
      m => m.title === title
    )
  );
});
```

Ahora nuestra 'clave primaria' no será el atributo 'title', sino el '_id' único que genera MongoDB para cada documento.

Sin embargo, esto no significa que no podamos usar una clave propia, es decir, un identificador único que sea parte de los registros almacenados en la BBDD.

Después:

```
router.get('/:_id', function (req, res) {
  let _id = req.params._id;
  moviesService.get(_id, (err, movie) => {
    if (err) {
      res.status(500).send({
        msg: err
      });
    } else if (movie.length == 0) {
      res.status(500).send({
        msg: "movies null"
      });
    } else {
      res.status(200).send(movie);
    }
  }
  );
});
```


Paso 3.6 – Editar movies.js

- Actualizamos el código de nuestro método PUT /movie/{id}

Antes:

```
router.put('/:title', function (req, res) {  
  let title = req.params.title;  
  movies.splice(movies.findIndex(m =>  
    m.title === title), 1);  
  movies.push(req.body);  
  res.status(200).send("Film  
    updated!");  
});
```

Un error posible es que el identificador de MongoDB proporcionado no esté bien formado.

Tenemos en cuenta el caso en que no se haya actualizado ningún parámetro de la reserva.

Después:

```
router.put('/:id', function (req, res) {  
  const _id = req.params._id;  
  const updatedMovie = req.body;  
  moviesService.update(_id, updatedMovie, (err, numUpdates) => {  
    if (err) {  
      res.status(500).send({msg: err});  
    } else if (numUpdates.modifiedCount === 0) {  
      res.status(500).send({  
        msg: "not updated"  
      });  
    } else {  
      res.status(200).send({  
        msg: 'Film updated!'});  
    }  
  });  
});
```

Paso 3.7 – Editar movies.js

- Actualizamos el código de nuestro método DELETE /movie/{id}

Antes:

```
router.delete('/:title', function (req, res) {
  let title = req.params.title;
  movies = movies.filter(
    m => m.title !== title);
  res.status(200).send("Film deleted!");
});
```

Después:

```
router.delete('COMPLETAR', function (req, res) {
  let _id = COMPLETAR;
  moviesService.COMPLETAR(_id, (err) => {
    COMPLETAR
  });
});
```

Ejercicio: Completar las partes que faltan, indicadas por los carteles “**COMPLETAR**”:

- Ruta
- Recuperación del `_id`.
- Método del *movies-service.js*.
- Cuerpo del manejador (contemplando el caso de error y éxito).

Paso 4 – Editar index.js

- Importamos el fichero *movies-service.js* (negrita) en nuestro *index.js* justo antes de la importación del fichero *movies.js*:

```
const moviesService = require('./routes/movies-service');  
const movies = require('./routes/movies');
```

- Actualizamos el código de la inicialización del servidor. Ahora, antes de arrancar el servidor, tenemos que conectar con la BBDD.

Antes:

```
server.listen(PORT, function () {  
  console.log('Server up and running on localhost:' + PORT);  
});
```

Después:

```
moviesService.connectDb(function (err) {  
  if (err) {  
    console.log('Could not connect with MongoDB - moviesService');  
    process.exit(1);  
  }  
  server.listen(PORT, function () {  
    console.log('Server up and running on localhost:' + PORT);  
  });  
});
```

Paso 5 – Instalar dependencia e iniciar API

- Ejecutamos lo siguiente en la terminal para instalar el paquete de MongoDB *(recuerda instalar los paquetes de express, morgan y cors)*

```
npm install mongodb@4.14.0 --save
```

```
pedro@pedro-espada:~/Documentos/PNET/Peliculas-BBDD$ npm install mongodb@4.14.0 --save
```

- Es el momento de iniciar nuestra API, para ello ejecutamos en la raíz de nuestro proyecto:

```
node index.js
```

- Si todo ha ido bien, en consola se mostrará el mensaje:

‘Conexión correcta’ (opcional)

‘Server up and running on localhost:8080’

- Podemos acceder a <http://localhost:8080/movies> y verificar que devuelve un array vacío *(no hay nada en la BBDD de momento)*.

Paso 5.1 – Si tienes problemas de conexión

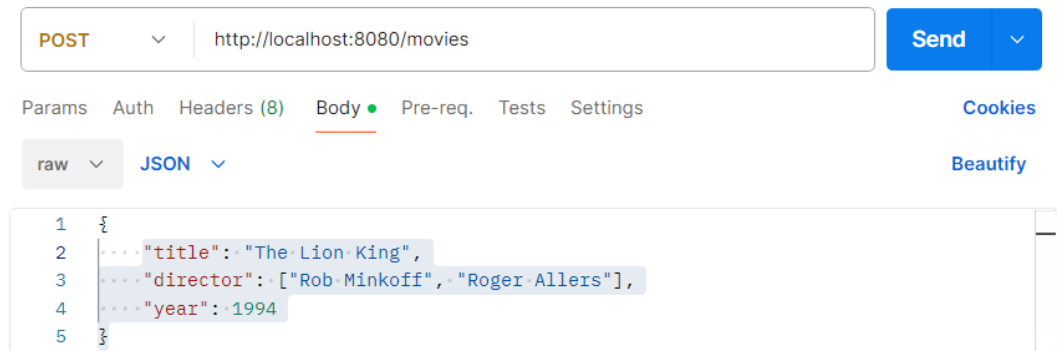
- Si no se conecta **y la URI de conexión está bien formada (revísalo)**, comprueba lo siguiente:
 - No estás usando un VPN.
 - Antivirus o firewall desactivado (raro pero posible).
- Si aún así nada, prueba a dar acceso a **TODAS LAS IPS**. Para ello accede a:
 - “Network Access” en el menú lateral izquierdo.
 - Click en el botón “Add IP address”.
 - Se abrirá una ventana emergente.
 - Escribir 0.0.0.0/0 en “Access List Entry”
 - Confirmamos y esperamos que se guarden los cambios.

Documentación posibles problemas de conexión:

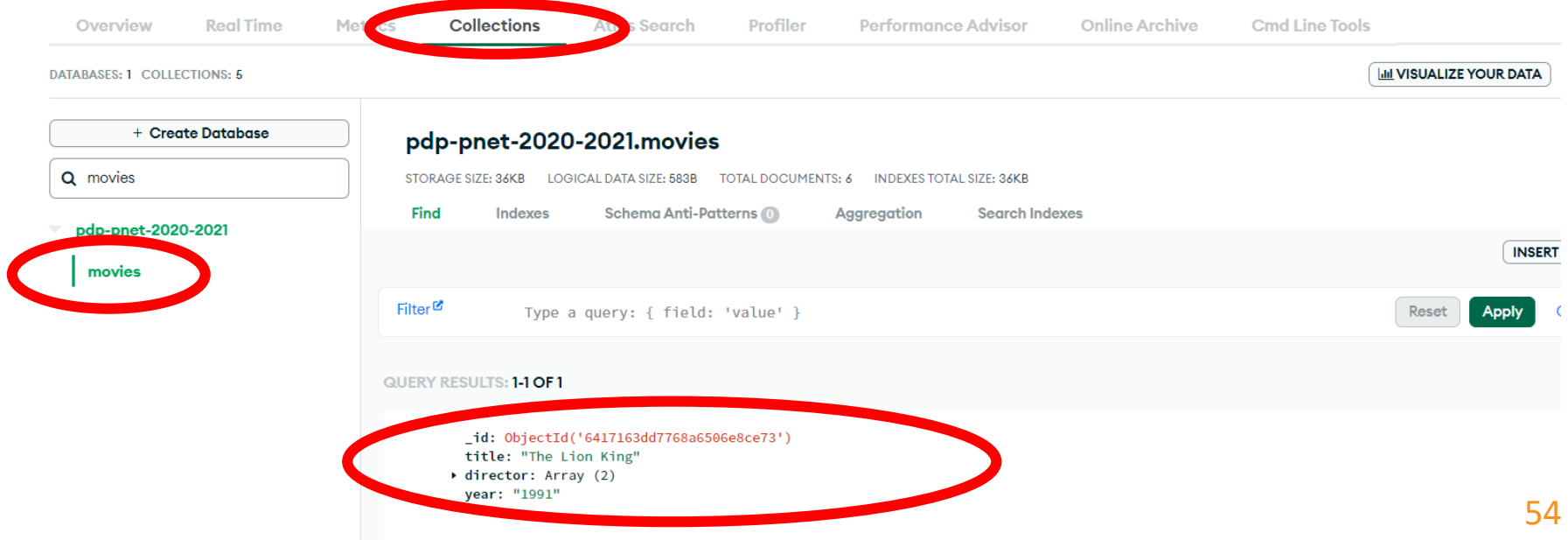
<https://docs.atlas.mongodb.com/troubleshoot-connection/>

Paso 7 – Probar API

- Para comenzar a introducir datos en la BBDD, podemos replicar las pruebas de POST que hicimos anteriormente con **Postman**.



- Podemos observar que se crea el documento en **MongoDB Atlas**:



Paso 7.2 – Probar API

- Realizaremos ahora una operación de PUT con **Postman**.
- En este caso tendremos que sustituir el 'title' en la URL por el 'id' único que genera MongoDB.

Antes:

<http://localhost:8080/movies/The Lion King>

Ahora (id de ejemplo - CAMBIAR):

<http://localhost:8080/movies/6417163dd7768a6506e8ce73>

The screenshot shows the Postman interface for a PUT request. The URL is `http://localhost:8080/movies/6417163dd7768a6506e8ce73`. The request body is a JSON object: `{ "title": "The Lion King", "director": "Jon Favreau", "year": 2019 }`. The response status is `200 OK` with a response time of `82 ms` and a body size of `290 B`. The response body is a JSON object: `{ "msg": "Film updated!" }`. The right sidebar shows the query results: `_id: ObjectId('6417163dd7768a6506e8ce73')`, `title: "The Lion King"`, `director: "Jon Favreau"`, and `year: 2019`.

```
PUT http://localhost:8080/movies/6417163dd7768a6506e8ce73
```

Params Auth Headers (8) **Body** Pre-req. Tests Settings Cookies

raw JSON Beautify

```
1 {
2   "title": "The Lion King",
3   "director": "Jon Favreau",
4   "year": 2019
5 }
```

Body 200 OK 82 ms 290 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "msg": "Film updated!"
3 }
```

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('6417163dd7768a6506e8ce73')
title: "The Lion King"
director: "Jon Favreau"
year: 2019
```

ÍNDICE

- Introducción
- Conceptos básicos de REST
- RESTful Hello World
- Creando API de películas
- Mejoras: *routes*, CORS y persistencia
- **Conclusiones**

¿QUÉ HEMOS LOGRADO?

1. Hemos creado una **API RESTful** completamente funcional:
 - Versión con *arrays*.
 - Versión con **persistencia** usando MongoDB.
2. Todas las peticiones y respuestas trabajan con **JSON**.
3. Está basada en **NodeJs** y utiliza **Express**.
4. Podemos tener tantos recursos como queramos dentro de nuestra API, agrupándolos en *routes*. En la API podríamos mezclar *routes* que usaran recursos con y sin persistencia.
5. Podemos probar nuestras APIs mediante programas de envío de peticiones REST, como **Postman** (existen otras alternativas, como [CURL](#) o [Advanced Rest Client](#)).

¿Qué mas puedo hacer en mi proyecto?

- **Control de errores** que puedan suceder (si el *id* es inválido, si un registro puede modificarse, si tiene estructura adecuada, etc.).
- **Nuevas operaciones:** explorar las funciones de *MongoDB*: <https://www.mongodb.com/docs/manual/crud/#std-label-crud>
- **Búsquedas por campo** en la BD: Ver [Query documents](#) y [find\(\)](#).

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

- **Mejorar los *middlewares*:** encadenamiento, manejo de errores... <https://expressjs.com/es/guide/using-middleware.html>

```
app.use('/user/:id', function (req, res, next) {  
  console.log('Request Type:', req.method);  
  next();  
});
```

```
app.get('/user/:id', function (req, res, next) {  
  res.send('USER');  
});
```

- Conexión de *frontend* y *backend* → **Próxima sesión**

Bibliografía

Bibliografía:

- 📖 **Web Services & SOA: Principles and Technology.** M.P. Papazoglou. Pearson – Prentice Hall, 2012.
- 📖 **RESTful web API design with Node.js 10 : learn to create robust RESTful web services with Node.js, MongoDB, and Express.js,** Valentin Bojinov, Packt Publishing, 3ª Ed, 2018.
- 📖 **RESTful Web APIs.** Leonard Richardson; Mike Amudsen; Sam Ruby, O'Reilly Media, 1ª Ed., 2013.

Recursos web:

- Listado middlewares de *express*:
<https://expressjs.com/en/resources/middleware.html>