

# Diplomatura en programación web full stack con React JS

Módulo 6:

# Administración, API Rest y despliegue

Unidad 3:

## CRUD (parte 3)



## Presentación

En esta unidad implementaremos la funcionalidad de subir archivos a nuestro servidor. Junto a esto utilizaremos el servicio de Cloudinary para almacenar y transformar las imágenes subidas. Crearemos además una API Rest para que nuestro frontend consuma los datos cargados en el administrador.



## Objetivos

### Que los participantes logren...

- Implementar las funcionalidades necesarias para realizar la subida de archivos al servidor.
- Conectar el sitio con el servicio de Cloudinary para almacenar las imágenes subidas y obtener versiones recortadas de las mismas.
- Crear una API Rest en Express y consumirla desde React.



## Bloques temáticos

1. Subida de imágenes.
2. Creación de API Rest.
3. Consumo de API con React.

# 1. Subida de archivos

Nuestro sitio web permitirá al administrador subir imágenes a las novedades que ya tenemos implementadas. Para esto será necesario hacer algunas modificaciones en el código y estructura de base de datos que ya tenemos.

Necesitaremos primero instalar algunas dependencias nuevas:

## express-fileupload

La dependencia `express-fileupload` nos va a permitir, mediante un middleware, capturar y manipular archivos subidos mediante formularios a nuestro sitio web.



```
npm i express-fileupload
```

## cloudinary

Utilizaremos el servicio gratuito de Cloudinary (<https://cloudinary.com/>) para alojar y manipular las imágenes que se vayan subiendo a las novedades. La dependencia la instalamos de la siguiente manera:

```
npm i cloudinary
```

Para poder utilizar el servicios necesitaremos registrarnos en el sitio y, una vez terminado este proceso, copiar el valor de API Environment variable a nuestro archivos .env con el nombre **CLOUDINARY\_URL**

## Dashboard

### Account Details

Cloud name:	dnrbsy3q7	
API Key:	342699	<a href="#">Copy to clipboard</a>
API Secret:	*****	<a href="#">Copy to clipboard</a> <a href="#">Reveal</a>
API Environment variable:	CLOUDINARY_URL=cloudinary://*****:*****@	<a href="#">Copy to clipboard</a>

⚙ .env

```
1 MYSQL_HOST=localhost
2 MYSQL_DB_NAME=cerveceria
3 MYSQL_USER=root
4 MYSQL_PASSWORD=
5
6 CLOUDINARY_URL=cloudinary://342699796164784:006-Eck-Cr*****
7
```

## Agregar imágenes a la novedades

### Paso 1

Incluimos la dependencia de **express-fileupload** y los configuramos como middleware de nuestra aplicación de express. Es importante configurar el uso de

archivos temporales a fin de que podamos, una vez subidos, enviarlos al servidor de Cloudinary donde quedarán almacenados.

```
var fileUpload = require('express-fileupload');

app.use(fileUpload({
  useTempFiles: true,
  tempFileDir: '/tmp/'
}));
```

## Paso 2

Agregamos en la tabla de novedades, la columna **img\_id**, con el tipo de datos varchar, cantidad de caracteres 250 y tildamos la opción NULL (para que permitir la carga de novedades sin imágenes)

## Paso 3

En el archivo **view/admin/agregar.hbs**, agregamos en la etiqueta **<form>**: el atributo **enctype** con el valor **multipart/form-data**. Esto hará que el formulario envíe no solo los campos que veníamos usando hasta ahora, sino que también enviará los archivos que incluyamos.

```
<form action="/admin/novedades/agregar" method="post"
  enctype="multipart/form-data" />
```



Debajo de la etiqueta `input` cuerpo agregamos el **input** del tipo **file** para poder adjuntar la imagen y le ponemos de nombre imagen.

```
<div class="form-group">
  <label>Imagen: <input type="file" name="imagen"
                      id="imagen" /></label>
</div>
```

## Paso 4

Modificamos nuestro archivo **routes/admin/novedades.js** e incluimos tanto la dependencia de **cloudinary** como la de **util** que usaremos para convertir el método **upload** en una función asincrónica.

```
var util = require('util');
var cloudinary = require('cloudinary').v2;

const uploader = util.promisify(cloudinary.uploader.upload);
```

Dentro del controlador `router.post (/agregar)` vamos a verificar, antes de enviar la imagen a nuestro modelo, inicializamos una variable `img_id` como vacía además chequeamos si existe el objeto `files` dentro de la petición y si incluye uno o más objetos dentro suyo.

Esta es la función que realiza express-fileupload; agregar en la petición el objeto files, y dentro de él facilitarnos todos los inputs del tipo file que se hayan enviado con su nombre.

```
router.post('/agregar', async (req, res, next) => {
  try {
    var img_id = '';
    if (req.files && Object.keys(req.files).length > 0) {
      imagen = req.files.imagen;
      img_id = (await uploader(imagen.tempFilePath)).public_id;
    }

    if (req.body.titulo != "" && req.body.subtitulo != ""
      && req.body.cuerpo != "") {

      await novedadesModel.insertNovedad({
        ...req.body,
        img_id
      });

      res.redirect('/admin/novedades')
    }
  }
});
```

En caso de que se hayan enviado archivos, buscaremos uno llamado imagen (recordar el atributo name del input que agregamos) y utilizaremos la ruta temporal donde fue almacenado en nuestra computadora como archivo de origen para enviar al uploader de Cloudinary. Por último, al completarse la subida a cloudinary guardamos el valor de la propiedad public\_id en nuestra variable img\_id.

Este proceso hace que la variable img\_id, que enviamos junto con el resto de los campos a la base de datos, solo contenga un valor (el id publico que cloudinary asignó a nuestra imagen y que necesitaremos más adelante para obtenerla y manipularla) en caso de que se haya subido una imagen. En caso contrario, enviaremos un valor vacío y la novedad quedará almacenada sin imagen.

## Mostrar imágenes en el listado del administrador

### Paso 1

En el archivo **routes/admin/novedades.js** agregamos la lógica necesaria para enviar junto con cada novedad la imagen que se haya subido en caso de haber una o nada en caso de no haberse subido una imagen a esa novedad.

Para esto utilizaremos el método `image` de `cloudinary` que nos devuelve una etiqueta `img` html completa.



```
router.get('/', async function (req, res, next) {
  var novedades = await novedadesModel.getNovedades();

  novedades = novedades.map(novedad => {
    if (novedad.img_id) {
      const imagen = cloudinary.image(novedad.img_id, {
        width: 100,
        height: 100,
        crop: 'fill'
      });
      return {
        ...novedad,
        imagen
      }
    } else {
      return {
        ...novedad,
        imagen: ''
      }
    }
  });

  res.render('admin/novedades', {
    layout: 'admin/layout',
    usuario: req.session.nombre,
    novedades
  });
});
```

## Paso 2

Finalmente en el archivo **view/admin/novedades.hbs** agregaremos en el bucle `each` la variable de la imagen con triple llave debido a que puede contener HTML

(la etiqueta img).

```
<tr>
  <th scope="col">#</th>
  <th scope="col">Título</th>
  <th scope="col">Imagen</th>
  <th class="text-center" scope="col">Acciones</th>
</tr>

<tr>
  <th scope="row">{{id}}</th>
  <td>{{ titulo }}</td>
  <td>{{ imagen }}</td>
</tr>
```

## Modificar una novedad con imagen

### Paso 1

Modificamos **views/admin/modificar.hbs** e incluimos el input para poder enviar la imagen. A diferencia del formulario de agregar, ahora incluiremos también un input del tipo checkbox para permitir a los usuarios eliminar la imagen asociada a la novedad.

Es necesario también que incluyamos un input del tipo hidden para enviar junto con el formulario, el id de la imagen actual de la novedad en caso de haber una.

```
<div class="form-group">
  <p>
    <label>Imagen: <input type="file" name="imagen"
      id="imagen" /></label>
  </p>
  <p>
    <label><input type="checkbox" name="img_delete"
      id="img_delete" value="1">
      Eliminar imagen actual?</label>
    <input type="hidden" name="img_original"
      value="{{novedad.img_id}}">
  </p>
</div>
```

## Paso 2

En el archivo **routes/admin/novedades.js** agregamos el método **destroy** de cloudinary que nos permite eliminar imágenes del servicio. Al igual que hicimos con el uploader, utilizaremos la librería util para convertirlo en un método asíncrono.

El código para manejar la modificación de la imagen en la novedad puede parecer complejo a primera vista pero no lo es. Primero inicializamos la variable **img\_id** con el valor que haya venido del input hidden y la variable **borrar\_img\_vieja** que usaremos para saber si tenemos que llamar al método **destroy**. De esta forma, pase lo que pase, al enviar el valor a la base de datos, preservaremos el valor original.

A continuación revisamos si el usuario decidió eliminar la imagen novedad, en cuyo caso asignamos **null** a **img\_id** para limpiar el valor de la base de datos.

Si el usuario no tildó la casilla para eliminar la imagen actual repetimos el proceso que hicimos en agregar para ver si se envió una imagen junto con el formulario. En

caso afirmativo la subimos a cloudinary, almacenamos el id publico en nuestra variable `img_id` y seteamos `borrar_img_vieja` a verdadero.

En el último paso revisamos el valor de `borrar_img_vieja` y en caso de ser verdadero, ya sea porque el usuario tilo la casilla o subió una imagen nueva, llamamos al método `destroy` para eliminar la imagen antigua de cloudinary.

Al finalizar todo este proceso, el valor de `img_id` contendrá una de tres opciones según el camino que haya seguido el código, `null`, el valor del `input hidden` o el id publico de una nueva imagen.

```
const destroy = util.promisify(cloudinary.uploader.destroy);

router.post('/modificar', async (req, res, next) => {
  try{
    let img_id = req.body.img_original;
    let borrar_img_vieja = false;
    if (req.body.img_delete === "1") {
      img_id = null;
      borrar_img_vieja = true;
    } else {
      if (req.files && Object.keys(req.files).length > 0) {
        imagen = req.files.imagen;
        img_id = (await
          uploader(imagen.tempFilePath)).public_id;
        borrar_img_vieja = true;
      }
    }
    if (borrar_img_vieja && req.body.img_original) {
      await (destroy(req.body.img_original));
    }
  }
}
```

## Paso 3

En la variable obj agregamos img\_id para enviarlo a la base de datos junto con el resto de los campos.

```
var obj = {  
  titulo:req.body.titulo,  
  subtitulo:req.body.subtitulo,  
  cuerpo: req.body.cuerpo,  
  img_id  
}
```

## Para eliminar una imagen

### Paso 1

En el archivo **routes/admin/novedades.js** dentro del controlador de la ruta que elimina las novedades vamos a verificar primero si la novedad tiene una imagen, en cuyo caso, como vamos a eliminar la novedad de nuestra base de datos, la eliminamos también de cloudinary con el método destroy.





```
router.get('/eliminar/:id', async (req, res, next) => {  
  var id = req.params.id;  
  
  let novedad = await novedadesModel.getNovedadById(id);  
  if (novedad.img_id) {  
    await (destroy(novedad.img_id));  
  }  
  
  await novedadesModel.deleteNovedadesById(id);  
  res.redirect('/admin/novedades')  
});
```

## 2. Creación de API Rest

Las API Rest son interfaces que las aplicaciones exponen para permitir la comunicación con otras y/o el consumo de datos controlado por parte de terceros. Muchas aplicaciones exponen sus servicios mediante la implementación de APIs Rest, permitiéndonos utilizar funcionalidades o acceder a datos a los cuales de otra forma sería imposible acceder.

En el caso de nuestra aplicación crearemos un endpoint (así se denomina cada URL que compone una API Rest) para exponer las novedades que se hayan cargado en la base de datos mediante el administrador. Esta sencilla API permitirá que nuestro sitio o cualquier otra aplicación (podría ser otro sitio o una aplicación de escritorio o para celulares) consuma nuestros datos

### CORS


CORS (del inglés Cross-Origin Resource Sharing) es un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un sitio obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio) al que pertenece. Es una práctica de seguridad común para prevenir el uso no autorizado de recursos.

Instalamos la dependencia necesaria con el siguiente comando.




```
npm i cors
```

Luego importamos la nueva dependencia en nuestro archivo **app.js**.



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

Para una mejor organización, vamos a separar todos los controladores para las rutas de nuestra API en un nuevo ruteador de express. Para eso creamos un archivo nuevo **routes/api.js** y lo importamos también en nuestro archivo **app.js**.



```
var apiRouter = require('./routes/api');  
  
app.use('/api', cors(), apiRouter);
```

Como se puede ver nuestro ruteador se encargará de todas las rutas que empiecen con **/api** e implementará a modo de middleware la librería de CORS que instalamos. Esto permitirá que cualquier ruta de nuestra API pueda ser consumida desde dominios diferentes al que aloje a nuestra aplicación.

El contenido de **routes/api.js** es el siguiente.

```
var express = require('express');
var router = express.Router();
var novedadesModel = require('../models/novedadesModel');

router.get('/novedades', async function (req, res, next) {
  let novedades = await novedadesModel.getNovedades();

  res.json(novedades);
});

module.exports = router;
```

Devolver los datos de nuestra base de datos mediante la API, como se puede apreciar, es muy sencillo. Sólo debemos devolver como respuesta JSON el listado de novedades que nuestro modelo ha generado.

Para poder probar nuestra API podemos acceder a la ruta <http://localhost:3000/api/novedades> utilizando Chrome u otro navegador. Dependiendo de la configuración o extensiones que tengamos instaladas podemos ver el contenido con un formato legible o el texto todo junto.

Una alternativa muy utilizada cuando desarrollamos APIs con muchos endpoints y necesitamos testearlas, más allá de que tengamos una aplicación que las consuma, es una aplicación llamada Postman. Ésta se puede descargar de <https://www.postman.com/downloads/> para Windows, macOS o Linux. Allí también se encuentra la documentación con explicaciones y ejemplos de su uso.

## Manipulación de imágenes.

Si prestamos atención hasta ahora solo estamos devolviendo en nuestra API el ID de la imagen que nos devolvió Cloudinary. Si queremos devolver la url de dicha imagen solamente debemos incorporar la librería de Cloudinary a nuestro código y consultar las urls de las imágenes de las cuales tenemos el ID.

Cabe destacar que Cloudinary además soporta varios métodos de manipulación de imágenes como redimensión, recorte o efectos que se pueden aplicar de forma muy sencilla en nuestro código.



```
var express = require('express');
var router = express.Router();
var novedadesModel = require('../models/novedadesModel');
var cloudinary = require('cloudinary').v2;

router.get('/novedades', async function (req, res, next) {
  let novedades = await novedadesModel.getNovedades();

  novedades = novedades.map(novedades => {
    if (novedades.img_id) {
      const imagen = cloudinary.url(novedades.img_id, {
        width: 960,
        height: 200,
        crop: 'fill'
      });
      return {
        ...novedades,
        imagen
      }
    } else {
      return {
        ...novedades,
        imagen: ''
      }
    }
  });

  res.json(novedades);
});

module.exports = router;
```

En nuestro ejemplo estamos aplicando un recorte (crop) del tipo fill que redimensiona la imagen y elimina los excedentes utilizando los tamaños proporcionados.

Los nombres y los parámetros de las otras manipulaciones o efectos que podemos aplicar sobre nuestras imágenes se pueden consultar en la documentación del sitio de Cloudinary.

## 3. Consumo de API con React

Ahora vamos a consumir la API que acabamos de crear en express utilizando el sitio que estamos desarrollando en React. Para esto utilizaremos una librería llamada axios, la cual nos facilita la realización de peticiones HTTP. La instalamos dentro de nuestra carpeta **frontend** con el siguiente comando.



```
npm i axios
```

Antes de listar las novedades que vengan desde la API vamos a crear el componente de React que representará a cada una de ellas. Para eso creamos un nuevo archivo en **components/novedades/NovedadItem.js** con el siguiente contenido.



```
import React from 'react';

const NovedadItem = (props) => {
  const { title, subtitle, imagen, body } = props;

  return (
    <div className="novedades">
      <h1>{title}</h1>
      <h2>{subtitle}</h2>
      <img src={imagen} />
      <div dangerouslySetInnerHTML={{ __html: body }} />
      <hr />
    </div>
  );
}

export default NovedadItem;
```

Podemos apreciar que es un componente bastante sencillo, que hace uso de destructuring para acceder de forma más cómoda a los valores pasados en las props.

Algo que puede llamarnos la atención a primera vista es el uso de la propiedad **dangerouslySetInnerHTML** en uno de los elementos div. Esta propiedad, con este nombre tan largo y explícito cumple la función de mostrar el código HTML tal cual venga (lo cual en ciertos casos puede ser inseguro). De esta forma React se asegura de que somos conscientes del riesgo que estamos tomando. Esta propiedad además recibe como valor un objeto que debe contener una propiedad llamada **\_\_html** con el contenido a mostrar.



A continuación, en el archivo **pages/NovedadesPage.js** vamos a importar axios y el componente que recién creamos. Vamos a hacer uso de otras funcionalidades de React como useEffect y useState para controlar la carga de las noticias y la información que vea el usuario de nuestro sitio.

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import NovedadItem from '../components/novedades/NovedadItem';

const NoveadesPage = (props) => {

  const [loading, setLoading] = useState(false);
  const [novedades, setNovedades] = useState([]);

  useEffect(() => {
    const cargarNovedades = async () => {
      setLoading(true);
      const response = await axios.get('http://localhost:3000/api/novedades');
      setNovedades(response.data);
      setLoading(false);
    };

    cargarNovedades();
  }, []);

  return (
    <section className="holder">
      <h2>Novedades</h2>
      {loading ? (
        <p>Cargando...</p>
      ) : (
        novedades.map(item => <NovedadItem key={item.id}
          title={item.titulo} subtitle={item.subtitulo}
          imagen={item.imagen} body={item.cuerpo} />)
      )}
    </section>
  );
}

export default NoveadesPage;
```

En este caso en particular hacemos uso de useEffect para ejecutar código al momento de que nuestro componente NovedadesPage sea incorporado al DOM,

pero antes de que el usuario lo vea. Le pasamos como parámetros una función anónima con el código que queremos ejecutar y un array de dependencias, que al estar vacío hace que el código solo corra cuando necesitamos.

Dentro de la función definimos una segunda función para poder saltar la limitación de no poder usar código asíncronico dentro de un hook de React e inmediatamente la llamamos. Esta función es la encargada de hacer la llamada a la API utilizando axios y manipular el estado de la variable loading y novedades (variables de estado).

useState es un hook de React que permite manipular de forma sencilla el estado dentro de componentes funcionales que previamente no podían contar con esta funcionalidad. En nuestro caso utilizamos 2 variables y sus correspondientes setters: loading y setLoading, y novedades y setNovedades. En ambos casos definimos un valor inicial para estas variables: false para loading y un array vacío para las novedades.

Al iniciar la carga dentro de useEffect seteamos el valor de la variable loading a true, lo que hará que el usuario vea el texto "cargando...". Al llegar la respuesta de nuestra consulta a la API y guardar su resultado utilizando setNovedades, volvemos a poner el valor de loading en false para mostrar al usuario las novedades que hayamos recibido.

Toda la lógica que acabamos de definir esta implementada mediante el uso del renderizado condicional utilizando un operador ternario que nos permite cambiar el contenido mostrado de acuerdo al valor de la variable loading y por último utilizamos un map sobre el array de novedades para duplicar nuestro componente NovedadItem tantas veces como noticias hayamos recibido.

Es muy importante recordar al duplicar elementos en React utilizar la propiedad key, la cual permite a React identificar al elemento de forma rápida.



## Bibliografía utilizada y sugerida

### Artículos de revista en formato electrónico:

**Cloudinary.** Disponible desde la URL: <https://cloudinary.com/documentation/>

**Handlebars.** Disponible desde la URL: <https://handlebarsjs.com/>

**MDN Web Docs.** Disponible desde la URL: <https://developer.mozilla.org/>

**npmjs.** Disponible desde la URL: <https://www.npmjs.com/>

**React.** Disponible desde la URL: <https://es.reactjs.org/>