



Clase 41. Programación Backend

Desarrollo de un servidor web basado en capas completo



OBJETIVOS DE LA CLASE

- Identificar los marcos de MERN stack.
- Configurar CORS.
- Crear una aplicación completa con API RESTful y un front-end simple.

CRONOGRAMA DEL CURSO

Clase 40



**Arquitectura del servidor:
Persistencia**

Clase 41



**Desarrollo de un
servidor web basado en
capas**

Clase 42



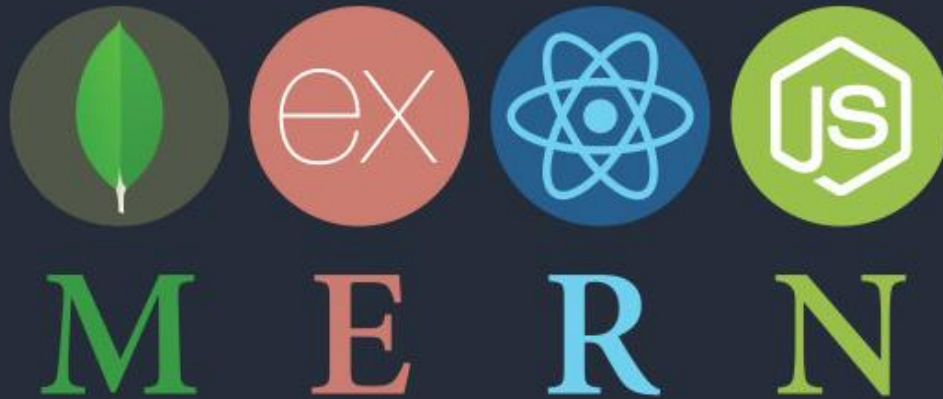
**Testeo de
funcionalidades**

STACK MERN

¿De qué se trata?



- Como vimos al principio del curso, el stack MERN es un conjunto de marcos/tecnologías utilizados para el desarrollo web de aplicaciones que consta de MongoDB, React JS, Express JS y Node JS como sus componentes.
- La combinación de estas cuatro tecnologías nos permite como desarrolladores crear sitios web (y aplicaciones) completos usando React (con JavaScript o TypeScript) del lado del cliente (front-end) y Node JS del lado del servidor (back-end). Así podremos dominar tanto la parte visual (la experiencia del usuario) como la parte lógica del servidor.
- Entonces con este stack, usamos Javascript tanto del lado del cliente como del lado del servidor.



CORS

¿De qué se trata?



- El Intercambio de Recursos de Origen Cruzado, CORS, es un mecanismo para permitir o restringir los recursos solicitados en un servidor web dependiendo de dónde se inició la solicitud HTTP.
- Esto se utiliza para proteger un determinado servidor web del acceso de otro sitio web o dominio. Por ejemplo, solo los dominios permitidos podrán acceder a los archivos alojados en un servidor, como una hoja de estilo, una imagen o un script.
- Por razones de seguridad, los navegadores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.

¿Cómo se utiliza?



- Por ejemplo, si nos encontramos en **http://example.com/page1** y estamos haciendo referencia a una imagen de **http://image.com/myimage.jpg**, no podremos recuperar esa imagen a menos que **http://image.com** permita compartir orígenes cruzados con **http://example.com**.
- Hay un encabezado HTTP llamado *origin* en cada solicitud HTTP el cual define desde dónde se originó la solicitud de dominio. Podemos usar la información del encabezado para restringir o permitir que los recursos de nuestro servidor web los protejan.

Configurando CORS



- Npm tiene un módulo llamado CORS, para poder configurar fácilmente las cabeceras, y decidir si permitimos o no el acceso a ciertas solicitudes de dominio cruzado.
- En primer lugar, instalamos el módulo con el comando:
- Luego, lo requerimos en el archivo *server.js*.

```
$ npm install cors
```

```
var express = require('express')  
var cors = require('cors')  
var app = express()
```

Configurando CORS

Ejemplo
en vivo



- Si deseamos habilitar CORS para todas las solicitudes, simplemente podemos usar el middleware cors antes de configurar las rutas, configurándolo a nivel global:

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

const app = express();

app.use(cors())

.....
```

- ★ Esto nos permitirá acceder a todas las rutas desde cualquier lugar de la web si eso es lo que necesitamos. Entonces, las rutas que configuremos serán accesibles para todos los dominios.

Configurando CORS

Ejemplo
en vivo



- Si necesitamos que una determinada ruta sea accesible y no otras rutas, podemos configurar cors en una determinada ruta como middleware en lugar de configurarlo para toda la aplicación:

```
app.get('/', cors(), (req, res) => {  
  res.json({  
    message: 'Hello World'  
  });  
});
```

- Esto permitirá que una determinada ruta sea accesible por cualquier dominio. Entonces, en este caso, solo la ruta “/” será accesible para cada dominio. Las demás rutas solo serán accesibles para las solicitudes que se iniciaron en el mismo dominio que la API en la que estén definidas.

Configurando CORS con Options

Ejemplo
en vivo



- Podemos usar las opciones de configuración con CORS para personalizar ésto aún más.
- Podemos usar la configuración para permitir el acceso de un solo dominio o subdominios, configurar métodos HTTP que estén permitidos, como GET y POST, según nuestros requisitos.
- Así es como podemos permitir el acceso de un solo dominio usando las opciones de CORS:

```
var corsOptions = {  
  origin: 'http://localhost:8080',  
  optionsSuccessStatus: 200 // For Legacy browser support  
}  
  
app.use(cors(corsOptions));
```

Configurando CORS con Options

Ejemplo
en vivo



- También podemos configurar los métodos HTTP que estén permitidos:

```
var corsOptions = {  
  origin: 'http://localhost:8080',  
  optionsSuccessStatus: 200 // For legacy browser support  
  methods: "GET, PUT"  
}  
  
app.use(cors(corsOptions));
```

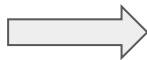
- Si enviamos una solicitud POST desde http://localhost: 8080, el navegador la bloqueará, ya que solo se admiten GET y PUT según los métodos especificados en esta configuración.

Configurando CORS dinámico con Function

Ejemplo
en vivo



- Si las configuraciones no satisfacen nuestros requisitos, podemos crear una función personalizada para CORS.
- Por ejemplo, supongamos que deseamos permitir el uso compartido de CORS para archivos .jpg <http://something.com> y <http://example.com>.



```
const allowlist = ['http://something.com', 'http://example.com'];

const corsOptionsDelegate = (req, callback) => {
  let corsOptions;

  let isDomainAllowed = whitelist.indexOf(req.header('Origin')) !== -1;
  let isExtensionAllowed = req.path.endsWith('.jpg');

  if (isDomainAllowed && isExtensionAllowed) {
    // Enable CORS for this request
    corsOptions = { origin: true }
  } else {
    // Disable CORS for this request
    corsOptions = { origin: false }
  }
  callback(null, corsOptions)
}

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

Configurando CORS dinámico con Function

Ejemplo
en vivo



- El *callback* acepta dos parámetros:
El primero es un error donde pasamos **null** y el segundo son opciones donde pasamos **{origin: false}**.
- Por lo tanto, una aplicación web alojada en `http://something.com` o `http://example.com` podría hacer referencia a una imagen con la extensión `.jpg` desde el servidor, como hemos configurado en nuestra función personalizada.

```
const allowlist = ['http://something.com', 'http://example.com'];

const corsOptionsDelegate = (req, callback) => {
  let corsOptions;

  let isDomainAllowed = whitelist.indexOf(req.header('Origin')) !== -1;
  let isExtensionAllowed = req.path.endsWith('.jpg');

  if (isDomainAllowed && isExtensionAllowed) {
    // Enable CORS for this request
    corsOptions = { origin: true }
  } else {
    // Disable CORS for this request
    corsOptions = { origin: false }
  }
  callback(null, corsOptions)
}

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


APLICACIÓN COMPLETA

LADO SERVIDOR: API RESTful



BREAK

¡10 MINUTOS Y VOLVEMOS!



SERVIDOR MVC COMPLETO

Tiempo: 15 a 20 minutos

SERVIDOR MVC COMPLETO

Desafío
generico



Tiempo: 15 a 20 minutos

Realizar un esqueleto de servidor MVC basado en Node.js y express.

Este debe tener separado en capas, donde se encuentren carpetas para resolver:

- La capa de ruteo
- El controlador
- La lógica de negocio
- Las validaciones de nuestros datos
- La capa de persistencia (DAO, DTO)

Realizar una simple ruta get y una post para pedir e incorporar palabras a un array de strings persistidos en memoria, siguiendo la lógica de la separación del proceso en capas.

SERVIDOR MVC COMPLETO

Desafío
generico



Tiempo: 15 a 20 minutos

Cada palabra que ingrese por post se debe almacenar en el array dentro de un objeto que contenga un timestamp. Ej.

```
[  
  { id: 1, palabra: "Hola", timestamp: 1624450180112 },  
  { id: 2, palabra: "que", timestamp: 1624450189685 },  
  { id: 3, palabra: "tal", timestamp: 1624450195068 }  
  ...  
]
```

Con el get se traerá la frase completa en formato string.

Probar la operación con postman.

LADO CLIENTE: PROYECTO EN REACT

Configuración del front-end



- Creamos un proyecto en React para nuestro front-end de la aplicación.
- En éste, vamos a consumir la API que creamos, y crear los componentes que necesitemos para poder mostrar el listado de noticias, crear nuevas, actualizarlas y borrarlas.
- Vamos a usar *Axios* para hacer los llamados a la API RESTful.
- Vamos a usar el módulo *faker* para crear noticias de forma aleatoria.



CONSUMIR NUESTRA API REST

Tiempo: 5 minutos

CONSUMIR NUESTRA API REST

Desafío
generico



Tiempo: 5 minutos

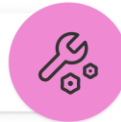
- Realizar una sencilla página web front en HTML/JS (send.html) que al ejecutarse dentro del navegador, en un proceso independiente al servidor del desafío anterior (puede estar servida por el live server de visual studio code), le envíe a este por post una palabra al azar.

No hace falta realizar la vista, el HTML estará para contener el script de ejecución.

- Utilizar axios en el front para emitir dicho request.

CONSUMIR NUESTRA API REST

Desafío
generico



Tiempo: 5 minutos

- Así mismo, realizaremos otra página web (receive.html) similar a la anterior, que al ejecutar su script interno, genere un request al mismo servidor en su ruta get para obtener la frase completa almacenada, representando por consola o en la vista del documento dicha frase.

Considerar el uso de CORS en el servidor para permitir los request de dominios cruzados.

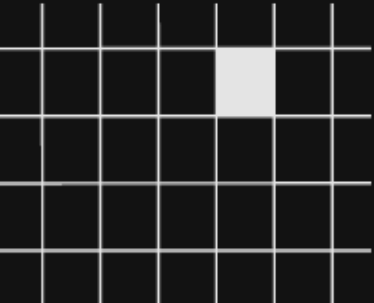
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- MERN stack
 - CORS
 - Aplicación con API RESTful en el lado servidor y un front-end simple en lado cliente.
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN