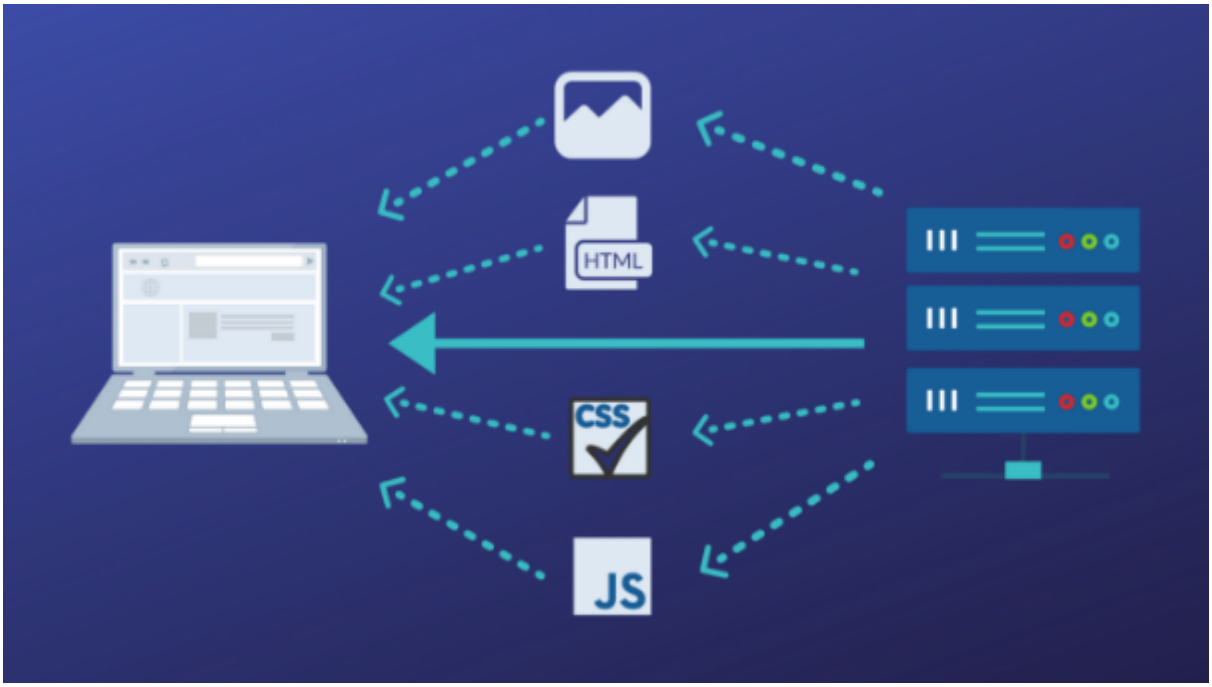
UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Sirviendo archivos estáticos (*)



Express proporciona un middleware incorporado: **"express.static"** para servir archivos estáticos, como html, imágenes, CSS, JavaScript, etc.

Simplemente necesita pasar el nombre del directorio donde guarda sus activos estáticos al middleware **express.static** para comenzar a servir los archivos directamente. Por ejemplo, si guarda sus archivos de imágenes, CSS y JavaScript en un directorio llamado *public*, puede hacerlo codificando lo siguiente:

```
app.use(express.static('public'));
```

Podríamos tener algunas imágenes en el subdirectorio public/images de la siguiente manera:

```
node_modules
index.js
public/
public/images
public/images/logo.png
```

Ejecute la aplicación y luego abra el explorador en la URL: http://127.0.0.1:3000/images/logo.png y allí podrá ver la imagen correspondiente a su archivo *logo.png*.

- Ver módulo *"path"* de nodejs que proporciona funcionalidad para trabajar con archivos y carpetas, necesarias para referenciar los archivos que queremos exponer mediante Express.
- y propiedad *"_dirname"* de NodeJs necesarias para ubicar los archivos estáticos que queremos exponer mediante Express.

Middlewares (*)

El concepto de middlewares es popular en muchos frameworks de Desarrollo Web. En el caso particular de Express, este depende fuertemente de esta construcción.

Un Middleware tiene como propósito tomar dos piezas de la aplicación y conectarlas, como un puente en el que fluye la información. Normalmente decimos que una rutina de código tiene como propósito recibir información y retornarla transformada, la única característica especial de un Middleware es que la información la obtiene de otra función de código para luego enviársela a una función distinta más.

Los middlewares en Express se montan por múltiples razones, una de ellas por ejemplo es validar la información antes de que llegue a la rutina que enviará respuesta hacia el cliente, también pueden usarse para hacer una consulta y guardar información antes de que pase a las funciones que responderán.

Un middleware en Express es una función que recibe 3 argumentos:

```
function(req,res,next){ … }
```

Los primeros dos argumentos, como cualquier función que responde peticiones del cliente contiene la información de la solicitud en el primer argumento Request (req), y el objeto Response (res) como segundo argumento nos sirve para modificar la respuesta que se enviará al cliente.

El tercer argumento (next) es muy utilizado en un middleware de una función de respuesta. Este tercer argumento es una función que contiene el próximo middleware o función que va a ejecutar luego de que la actual termine su ejecución.

Esta función *next* termina la ejecución de nuestro middleware y puede hacerlo de dos formas:

- Con éxito, la función next en este caso no recibe argumentos al ejecutarse, indicándole al siguiente punto de la ejecución que todo salió bien.

```
function miMiddleware(req,res,next){
  next();
}
```

- Con un error, el mismo se envía como argumento de la función, indicando al siguiente punto de la ejecución que algo salió mal y no puede continuar con la respuesta de la petición del cliente.

```
function miMiddleware(req,res,next){
  if(user.permisos != "admin"){
    next(new Error('No tienes permisos para estar aquí'));
```

```
}
}
```

Estas funciones se montan:

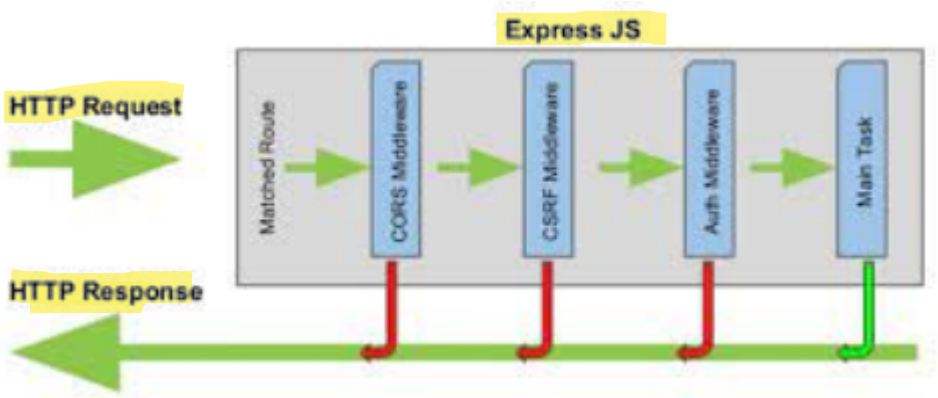
- en el proceso de respuesta a una petición usando el método use del objeto app. El middleware se ejecuta en el orden en que se define en el código, y se aplica a todas las solicitudes que llegan a la aplicación.

```
const express = require('express');
const app = express();
function miMiddleware(req,res,next){
  console.log('Este es un middleware');
  next();
}
app.use(miMiddleware); //Esto indica que antes de cualquier función de respuesta se debe ejecutar este middleware
```

- O bien como parte de la respuesta de una ruta:

```
const express = require('express');
const app = express();
function miMiddleware(req,res,next){
  next();
}
app.get('/',miMiddleware,function(req,res){
//Se ejecutará esta función luego del middleware
});
```

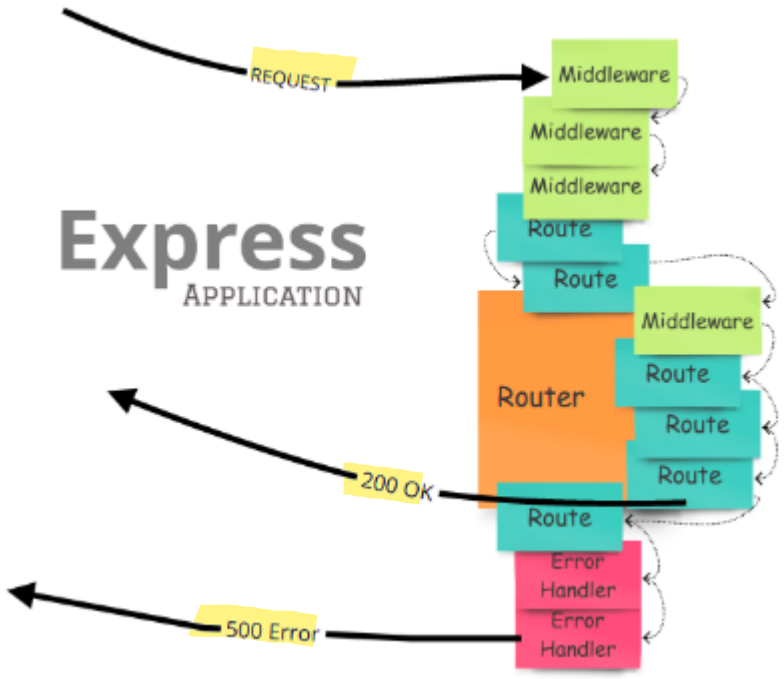
En ambos casos, es posible que podamos colocar cuantos middlewares necesitemos definir, lo importante es que cada uno llame la función **next();**, sin argumentos, para que el siguiente middleware se ejecute hasta llegar a la función de respuesta.



A continuación vemos un ejemplo de middleware que servirá para generar un registro (log) de las peticiones del cliente

```
app.use((req, res, next) => {
  let current_datetime = new Date();
  let formatted_date =
  current_datetime.getFullYear() +
  "-" +
  (current_datetime.getMonth() + 1) +
  "-" +
  current_datetime.getDate() +
  " " +
  current_datetime.getHours() +
  ":" +
  current_datetime.getMinutes() +
  ":" +
  current_datetime.getSeconds();
  let method = req.method;
  let URL = req.URL;
  let status = res.statusCode;
  let log = `[${formatted\_date}] ${method}:${URL} ${status}`;
  console.log(log);
  next();
})
```

MANEJO DE ERRORES:



El manejo de errores es uno de los temas más importantes en la programación. Ayuda a los usuarios finales a comprender lo que está mal y hace que el código sea más fácil de corregir y mantener. Como premisa general necesitamos que todos los posibles errores de nuestra aplicación no hagan caer nuestro servidor (daemon, o servicio) y queden registrados para poder analizarlos y tomar las medidas necesarias para corregirlos o mitigarlos.

El manejo de errores en express se hace con un middleware, que debe definirse al final de otras llamadas a rutas.

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Actualmente tenemos inconvenientes con procesar su
solicitud, intente nuevamente mas tarde!');
});
```

Los errores que ocurren en el código síncrono dentro de los controladores de ruta y el middleware no requieren trabajo adicional. Si el código síncrono arroja un error, Express lo detectará y lo procesará.

```
// función con error sincrono
app.get('/testerror', (req, res) => {
  throw new Error('probando desencadenar un error');
});
```

- Ver implementación en https://stackblitz.com/edit/dds-backend?file=index.js

Pero para el caso de los errores devueltos por funciones asincrónicas invocadas por controladores de ruta y middleware, debemos pasarlos a la función **next()**, para que Express pueda detectarlos y procesarlos, caso contrario harán caer nuestro servidor.

```
// función con error asincrono, con uso de la función next()
router.get("/api/articulosfamilias-testerror", async function (req, res, next) {
  try {
    let items = await db.articulosfamilias.findAll({
        attributes: ["CampoInexistenteParaGenerarUnError"],
        });
    res.json(items);
  } catch (error) {
    next(error);
  }
});
```

Para evitar tener que capturar todos los errores asíncronos en forma manual, usaremos un middleware específico para tal tarea, seguiremos los siguientes pasos.

1. Instalar paquete: ***npm install express-async-errors***
2. Modificar todos los métodos asíncronos para que usen async/await en lugar de promesas, para que funcione la librería y evitemos que se caiga el servidor web.

- Ejercicio: modificar el controlador de errores para que registre los errores en un archivo, que luego solo como desarrolladores tendremos acceso.

- Ejercicio: Modificar las validaciones que hacemos con Sequelize, tanto en el alta como en la modificaciones, encapsulándola en una única función común a ambas (alta y modificación).

Excepciones no controladas

¿Que sucede cuando ocurre un error regular y no es detectado por try..catch?: El script finaliza con un mensaje en la consola. El mismo puede ser capturado por el evento de Node ***"uncaughtException"*** que se emite precisamente cuando una excepción JavaScript no capturada vuelve al ciclo de eventos de node.js. Mediante este evento no podremos evitar que la aplicación se caiga pero nos permitirá registrar la causa de la misma, para analizarla posteriormente.

```
import process from 'node:process';
process.on('uncaughtExceptionMonitor', (err, origin) => {
  console.log('Olvidamos manejar este error en el código:  ',  error);
  process.exit(1); // salir de la aplicación
});
```

Análogamente existe el evento "***unhandledRejection***", que se emite cuando una Promesa es rechazada y no hay ningún catch de errores adjuntado a la misma dentro del bucle de eventos de node.js. El cual también deberíamos usar.

```
process.on('unhandledRejection', (error, promise) => {
  console.log(' Olvidamos de manejar este error en la promesa', promise);
  console.log(' El error fue: ', error );
});
```

Como vemos siempre existe que la posibilidad que nuestra aplicación se interrumpa por algún tipo de error o malfuncionamiento, para salvar está situación cuando publiquemos en un entorno de producción nos valdremos de algun gestor de procesos que nos dara otra capa más de proteccion para asegurar el funcionamiento de nuestro servidor.

Gestores de procesos para las aplicaciones Express en producción.

Cuando ejecuta aplicaciones Express en producción, es muy útil utilizar un ***gestor de procesos*** para realizar las siguientes tareas:

- Reiniciar la aplicación automáticamente si se bloquea o se interrumpe.
- Obtener información útil sobre el rendimiento en tiempo de ejecución y el consumo de recursos.
- Modificar dinámicamente los valores para mejorar el rendimiento.
- Controlar la agrupación en clúster.

Un gestor de procesos es una especie de servidor de aplicaciones: un "contenedor" de aplicaciones que facilita el despliegue, proporciona una alta disponibilidad y permite gestionar la aplicación en el tiempo de ejecución.

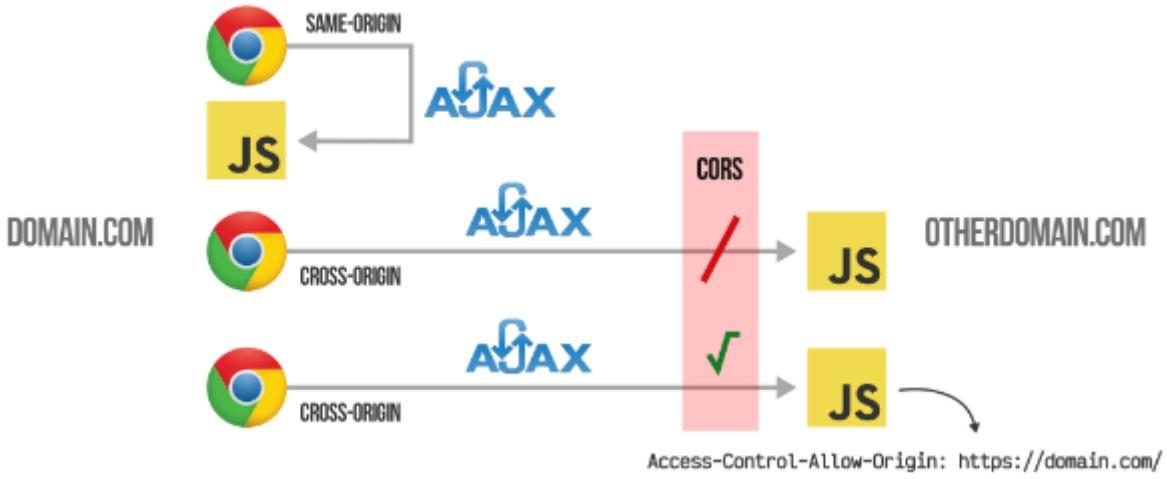
Los gestores de procesos más conocidos para Express y otras aplicaciones Node.js son los siguientes:

- StrongLoop Process Manager
- PM2
- Forever

```
Nota: En nuestros ejemplos cuando publiquemos nuestro paso a paso en Microsoft
Azure
 mediante la funcionalidad “App Service Linux, stack Node”, esta nos proveerá
 automáticamente del gestor de proyectos PM2.  Azure App Service le permite crear
 y hospedar aplicaciones web, back-ends móviles y API RESTful en varios lenguaje
 disponibles de programación sin tener que administrar la infraestructura.
```

CORS (*)

CORS significa **Cross-Origin Resource Sharing**, y es una política a nivel de navegador web que se aplica para prevenir que el dominio determinado evite acceder a recursos de otro dominio usando solicitudes del tipo AJAX como cuando usamos **fetch()** o **XMLHttpRequest** o cualquier librería como por ej Axios



Referencia: https://lenguajejs.com/javascript/peticiones-http/cors/

Lo primero que necesitamos saber es que si tenemos dos dominios, por ejemplo dds-frontend.com y dds-backend.com en principio no pueden comunicarse. Si nosotros queremos que por ejemplo dds-backend.com pueda permitir a otros dominios acceder a sus recursos, podemos hacerlo a través del módulo de cors.

Entonces tenemos que instalar y configurar dicha librería

- Comando para instalar:

```
    npm i cors
```

- configurar en el sistema

```
app.use(cors());
var express = require('express');
```

```
var cors = require('cors');
var app = express();
app.use(cors());
app.get('/products/:id', function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for all origins!'})
});

app.listen(80, function () {
  console.log('CORS-enabled web server listening on port 80')
});
```

Con esto ya estamos permitiendo a nuestro dominio recibir solicitudes de otros dominios. Pero si queremos limitar solo a ciertos dominios acceder a nuestros recursos podemos igual hacerlo a través de una lista blanca, en donde definimos los dominios y validamos que cada que haya una solicitud a una ruta específica se ejecute ese procedimiento de confirmación para aprobar o descargar el dominio.

En nuestro código paso a paso podemos ver su configuracion en el archivo **index.js**

Código Online: https://stackblitz.com/edit/dds-backend?file=index.js