Esta página fue traducida del inglés por la comunidad, pero no se mantiene activamente, por lo que puede estar desactualizada. Si desea ayudar a mantenerlo, descubra cómo activar las configuraciones regionales inactivas.

Introducción a Express/Node

En este primer articulo de Express resolveremos las preguntas "¿Qué es Node?" y "¿Qué es Express?", y te daremos una visión general de que hace especial al framework web "Express". Delinearemos las características principales, y te mostraremos algunos de los principales bloques de construcción de una aplicación en Express (aunque en este punto no tendrás todavía un entorno de desarrollo en que probarlo).

Pre-requisitos:	Conocimientos básicos de informática. Noción general sobre programación lado servidor de sitios web, y en particular los mecanismos de las interacciones cliente-servidor en sitios web.
Objetivo:	Ganar familiaridad con lo que es Express y cómo encaja con Node, qué funcionalidad proporciona y los pilares de construcción de una aplicación Express.

¿Qué son Express y Node?

Node (o más correctamente: *Node.js*) es un entorno que trabaja en tiempo de ejecución, de código abierto, multi-plataforma, que permite a los desarrolladores crear toda clase de herramientas de lado servidor y aplicaciones en JavaScript. La ejecución en tiempo real está pensada para usarse fuera del contexto de un explorador web (es decir, ejecutarse directamente en una computadora o sistema operativo de servidor). Como tal, el entorno omite las APIs de JavaScript específicas del explorador web y añade soporte para APIs de sistema operativo más tradicionales que incluyen HTTP y bibliotecas de sistemas de ficheros.

Desde una perspectiva de desarrollo de servidor web, Node tiene un gran número de ventajas:

- ¡Gran rendimiento! *Node* ha sido diseñado para optimizar el rendimiento y la escalabilidad en aplicaciones web y es un muy buen complemento para muchos problemas comunes de desarrollo web (ej, aplicaciones web en tiempo real).
- El código está escrito en "simple JavaScript", lo que significa que se pierde menos tiempo ocupándose de las "conmutaciones de contexto" entre lenguajes cuando estás escribiendo tanto el código del explorador web como del servidor.

- JavaScript es un lenguaje de programación relativamente nuevo y se beneficia de los avances en diseño de lenguajes cuando se compara con otros lenguajes de servidor web tradicionales (ej, Python, PHP, etc.) Muchos otros lenguajes nuevos y populares se compilan/convierten a JavaScript de manera que puedes también usar CoffeeScript, ClosureScript, Scala, LiveScript, etc.
- El gestor de paquetes de *Node* (NPM del inglés: Node Packet Manager) proporciona acceso a cientos o miles de paquetes reutilizables. Tiene además la mejor en su clase resolución de dependencias y puede usarse para automatizar la mayor parte de la cadena de herramientas de compilación.
- Es portable, con versiones que funcionan en Microsoft Windows, OS X, Linux, Solaris, FreeBSD, OpenBSD, WebOS, y NonStop OS. Además, está bien soportado por muchos de los proveedores de hospedaje web, que proporcionan infraestructura específica y documentación para hospedaje de sitios *Node*.
- Tiene un ecosistema y comunidad de desarrolladores de terceros muy activa, con cantidad de gente deseosa de ayudar.

Puedes crear de forma sencilla un servidor web básico para responder cualquier petición simplemente usando el paquete HTTP de *Node*, como se muestra abajo. Este, creará un servidor y escuchará cualquier clase de peticiones en la URL `http://127.0.0.1:8000/` ; cuando se reciba una petición, se responderá enviando en texto la respuesta: "Hola Mundo!".

```
// Se carga el módulo de HTTP
var http = require("http");

// Creación del servidor HTTP, y se define la escucha
// de peticiones en el puerto 8000
http.createServer(function(request, response) {

   // Se define la cabecera HTTP, con el estado HTTP (OK: 200) y el tipo de
   response.writeHead(200, {'Content-Type': 'text/plain'});
```

```
    // Se responde, en el cuerpo de la respuesta con el mensaje "Hello World"
    response.end('Hola Mundo!\n');
}).listen(8000);

// Se escribe la URL para el acceso al servidor
console.log('Servidor en la url http://127.0.0.1:8000/');
```

Otras tareas comunes de desarrollo web no están directamente soportadas por el mismo *Node*. Si quieres añadir el manejo específico de diferentes verbos HTTP (ej, `GET`, `POST`, `DELETE`, etc.), gestionar de forma separada las peticiones por medio de diferentes direcciones URL ("rutas"), servir ficheros estáticos o usar plantillas para crear la respuesta de forma dinámica, necesitarás escribir el código por tí mismo, o ¡puedes evitar reinventar la rueda usando un framework web!

Express es el framework web más popular de *Node*, y es la librería subyacente para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

- Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).
- Integración con motores de renderización de "vistas" para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
- Añadir procesamiento de peticiones "middleware" adicional en cualquier punto dentro de la tubería de manejo de la petición.

A pesar de que *Express* es en sí mismo bastante minimalista, los desarrolladores han creado paquetes de middleware compatibles para abordar casi cualquier problema de desarrollo web. Hay librerías para trabajar con cookies, sesiones, inicios de sesión de usuario, parámetros URL, datos `POST`, cabeceras de seguridad y *muchos* más. Puedes encontrar una lista de paquetes middleware mantenida por el equipo de Express en Express Middleware (junto con una lista de algunos de los paquetes más populares de terceros).

> **Nota:** esta flexibilidad es una espada de doble filo. Hay paquetes de middleware para abordar casi cualquier problema o requerimiento, pero deducir cuáles son los paquetes

> adecuados a usar algunas veces puede ser un auténtico reto. Tampoco hay una "forma correcta" de estructurar una aplicación, y muchos ejemplos que puedes encontrar en la Internet no son óptimos, o sólo muestran una pequeña parte de lo que necesitas hacer para desarrollar una aplicación web.

¿Dónde comenzó?

Node fué lanzado inicialmente, sólo para Linux, en 2009. El gestor de paquetes NPM fué lanzado en 2010, y el soporte nativo para Windows fue añadido en 2012. La versión actual LTS (Long Term Suppport) es Node v12.18.0 mientras que la última versión es Node 14.4.0. Ésto es sólo una pequeña foto de una historia muy rica; profundiza en Wikipedia si quieres saber más).

Express fue lanzado inicialmente en Noviembre de 2010 y está ahora en la versión 4.17.1 de la API. Puedes comprobar en el changelog la información sobre cambios en la versión actual, y en GitHub notas de lanzamiento históricas más detalladas.

¿Qué popularidad tiene Node/Express?

La popularidad de un framework web es importante porque es un indicador de se continuará manteniendo y qué recursos tienen más probabilidad de estar disponibles en términos de documentación, librerías de extensiones y soporte técnico.

No existe una medida disponible de inmediato y definitiva de la popularidad de los frameworks de lado servidor (aunque sitios como Hot Frameworks intentan asesorar sobre popularidad usando mecanismos como contar para cada plataforma el número de preguntas sobre proyectos en GitHub y StackOverflow). Una pregunta mejor es si Node y Express son lo "suficientemente populares" para evitar los problemas de las plataformas menos populares. ¿Continúan evolucionando? ¿Puedes conseguir la ayuda que necesitas? ¿Hay alguna posibilidad de que consigas un trabajo remunerado si aprendes Express?

De acuerdo con el número de compañías de perfil alto que usan Express, el número de gente que contribuye al código base, y el número de gente que proporciona soporte tanto libre como pagado, podemos entonces decir que sí, !*Express* es un framework popular!

¿Es Express dogmático?

Los frameworks web frecuentemente se refieren a sí mismos como "dogmáticos" ("*opinionated*") o "no dogmáticos" ("*unopinionated*")

("*opinionated*") o "no dogmáticos" ("*unopinionated*").

Los frameworks dogmáticos son aquellos que opinan acerca de la "manera correcta" de gestionar cualquier tarea en particular. Ofrecen soporte para el desarrollo rápido en un *dominio en particular* (resolver problemas de un tipo en particular) porque la manera correcta de hacer cualquier cosa está generalmente bien comprendida y bien documentada. Sin embargo pueden ser menos flexibles para resolver problemas fuera de su dominio principal, y tienden a ofrecer menos opciones para elegir qué componentes y enfoques pueden usarse.

Los framewoks no dogmáticos, en contraposición, tienen muchas menos restricciones sobre el modo mejor de unir componentes para alcanzar un objetivo, o incluso qué componentes deberían usarse. Hacen más fácil para los desarrolladores usar las herramientas más adecuadas para completar una tarea en particular, si bien al coste de que necesitas encontrar esos componentes por tí mismo.

Express es no dogmático, transigente. Puedes insertar casi cualquier middleware compatible que te guste dentro de la cadena de manejo de la petición, en casi cualquier orden que te apetezca. Puedes estructurar la app en un fichero o múltiples ficheros y usar cualquier estructura de directorios. ¡Algunas veces puedes sentir que tienes demasiadas opciones!

¿Cómo es el código para Express?

En sitios web o aplicaciones web dinámicas, que accedan a bases de datos, el servidor espera a recibir peticiones HTTP del navegador (o cliente). Cuando se recibe una petición, la aplicación determina cuál es la acción adecuada correspondiente, de acuerdo a la estructura de la URL y a la información (opcional) indicada en la petición con los métodos `POST` o `GET`. Dependiendo de la acción a realizar, puede que se necesite leer o escribir en la base de datos, o realizar otras acciones necesarias para atender la petición correctamente. La aplicación ha de responder al navegador, normalmente, creando una página HTML dinámicamente para él, en la que se muestre la información pedida, usualmente dentro de un elemento especifico para este fin, en una plantilla HTML.

Express posee métodos para especificar que función ha de ser llamada dependiendo del verbo HTTP usado en la petición (`GET`, `POST`, `SET`, etc.) y la estructura de la URL ("ruta"). También tiene los métodos para especificar que plantilla ("view") o gestor de visualización utilizar, donde están guardadas las plantillas de HTML que han de usarse y como generar la visualización adecuada para cada caso. El middleware de *Express*, puede usarse también para añadir funcionalidades para la gestión de cookies, sesiones y usuarios, mediante el uso

de parámetros, en los métodos `POST` / `GET`. Puede utilizarse además cualquier sistema de trabajo con bases de datos, que sea soportado por *Node* (*Express* no especifica ningún método preferido para trabajar con bases de datos).

En las siguientes secciones, se explican algunos puntos comunes que se pueden encontrar cuando se trabaja con código de *Node* y *Express*.

Hola Mundo! - en Express

Primero consideremos el tradicional ejemplo de Hola Mundo! (se comentará cada parte a continuación).

> **Consejo:** Si tiene *Node* y *Express* instalado (o piensa instalarlos posteriormente) puede guardar este código en un archivo llamado **app.js** y ejecutarlo posteriormente en la linea de comandos invocándolo mediante: `node app.js`.

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

app.get('/', function(req, res) {
  res.send('Hola Mundo!');
});

app.listen(3000, function() {
  console.log('Aplicación ejemplo, escuchando el puerto 3000!');
});
```

Las primeras dos líneas incluyen (mediante la orden `require()`) el módulo de Express y crean una aplicación de Express. Este elemento se denomina comúnmente `app`, y posee métodos para el enrutamiento de las peticiones HTTP, configuración del 'middleware', y visualización de las vistas de HTML, uso del motores de 'templates', y gestión de las configuraciones de las aplicaciones que controlan la aplicación (por ejemplo el entorno, las definiciones para enrutado ... etcetera.)

Las líneas que siguen en el código (las tres líneas que comienzan con `app.get`) muestran una definición de ruta que se llamará cuando se reciba una petición HTTP `GET` con una dirección (`'/'`) relativa al directorio raíz. La función 'callback' coge una petición y una respuesta como argumentos, y ejecuta un `send()` en la respuesta, para enviar la cadena de caracteres: "Hola Mundo!".

El bloque final de código, define y crea el servidor, escuchando el puerto 3000 e imprime un comentario en la consola. Cuando se está ejecutando el servidor, es posible ir hasta la dirección `localhost:3000` en un navegador, y ver como el servidor de este ejemplo devuelve el mensaje de respuesta.

Importando y creando módulos

Un modulo es una librería o archivo JavaScript que puede ser importado dentro de otro código utilizando la función `require()` de Node. Por sí mismo, *Express* es un modulo, como lo son el middleware y las librerías de bases de datos que se utilizan en las aplicaciones *Express.*

El código mostrado abajo, muestra como puede importarse un modulo con base a su nombre, como ejemplo se utiliza el framework *Express* . Primero se invoca la función `require()` , indicando como parámetro el nombre del módulo o librería como una cadena (`'express'`), posteriormente se invoca el objeto obtenido para crear una aplicación Express .

Posteriormente, se puede acceder a las propiedades y funciones del objeto Aplicación.

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

También podemos crear nuestros propios módulos que puedan posteriormente ser importados de la misma manera.

> **Consejo:** Usted puede desear crear sus propios módulos, esto le permitirá organizar su código en partes más administrables; una aplicación que reside en un solo archivo es difícil de entender y manejar.
>
> El utilizar módulos independientes también le permite administrar el espacio de nombres, de esta manera unicamente las variables que exporte explícitamente son importadas cuando utilice un módulo.

Para hacer que los objetos esten disponibles fuera de un modulo, solamente es necesario asignarlos al objeto `exports` . Por ejemplo, el modulo mostrado a continuación **square.js** es un archivo que exporta los métodos `area()` y `perimeter()` :

```
exports.area = function(width) { return width * width; };
```

```
exports.area = function(width) { return width * width; };
exports.perimeter = function(width) { return 4 * width; };
```

Nosotros podemos importar este módulo utilizando la función `require()`, y entonces podremos invocar los métodos exportados de la siguiente manera:

```
// Utilizamos la función require() El nombre del archivo se ingresa sin e
var square = require('./square');
// invocamos el metodo area()
console.log('El área de un cuadrado con lado de 4 es ' + square.area(4));
```

> **Nota:** Usted también puede especificar una ruta absoluta a la ubicación del módulo (o un nombre como se realizó inicialmente).

Si usted desea exportar completamente un objeto en una asignación en lugar de construir cada propiedad por separado, debe asignarlo al módulo `module.exports` como se muestra a continuación (también puede hacer esto al inicio de un constructor o de otra función.)

```
module.exports = {
  area: function(width) {
    return width * width;
  },

  perimeter: function(width) {
    return 4 * width;
  }
};
```

Para más información acerca de módulos vea Modulos (*Node* API docs).

Usando APIs asíncronas

El código JavaScript usa frecuentemente APIs asíncronas antes que sincrónicas para operaciones que tomen algún tiempo en completarse. En una API sincrónica cada operación debe completarse antes de que la siguiente pueda comenzar. Por ejemplo, la siguiente función de registro es síncrona, y escribirá en orden el texto en la consola (Primero, Segundo).

```
console.log('Primero');
console.log('Segundo');
```

En contraste, en una API asincrónica, la API comenzará una operación e inmediatamente retornará (antes de que la operación se complete). Una vez que la operación

finalice, la API usará algún mecanismo para realizar operaciones adicionales. Por ejemplo, el código de abajo imprimirá "Segundo, Primero" porque aunque el método `setTimeout()` es llamado primero y retorna inmediatamente, la operación no se completa por varios segundos.

```
setTimeout(function() {
   console.log('Primero');
   }, 3000);
console.log('Segundo');
```

Usar APIs asíncronas sin bloques es aun mas importante en *Node* que en el navegador, porque *Node* es un entorno de ejecución controlado por eventos de un solo hilo. "Un solo hilo" quiere decir que todas las peticiones al servidor son ejecutadas en el mismo hilo (en vez de dividirse en procesos separados). Este modelo es extremadamente eficiente en términos de velocidad y recursos del servidor, pero eso significa que si alguna de sus funciones llama a métodos sincrónicos que tomen demasiado tiempo en completarse, bloquearan no solo la solicitud actual, sino también cualquier otra petición que este siendo manejada por tu aplicación web.

Hay muchas maneras para una API asincrónica de notificar a su aplicación que se ha completado. La manera mas común es registrar una función callback cuando usted invoca a una API asincrónica, la misma será llamada de vuelta cuando la operación se complete. Éste es el enfoque utilizado anteriormente.

> **Tip:** Usar "callbacks" puede ser un poco enmarañado si usted tiene una secuencia de operaciones asíncronas dependientes que deben ser llevadas a cabo en orden, porque esto resulta en múltiples niveles de "callbacks" anidadas. Este problema es comúnmente conocido como "callback hell" (callback del infierno). Este problema puede ser reducido con buenas practicas de código (vea http://callbackhell.com/), usando un modulo como async , o incluso avanzando a características de ES6 como las promesas.

> **Nota:** Una convención común para *Node* y *Express* es usar callbacks de error primero. En esta convención el primer valor en su función callback es un error, mientras que los argumentos subsecuentes contienen datos correctos. Hay una buena explicación de porque este enfoque es útil en este blog:
> The Node.js Way - Understanding Error-First Callbacks (fredkschott.com).

Creando manejadores de rutas

En nuestro ejemplo anterior de "Hola Mundo!" en *Express* (véase mas arriba), definimos una función (callback) manejadora de ruta para peticiones HTTP `GET` a la raíz del sitio (`'/'`).

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

La función callback toma una petición y una respuesta como argumentos. En este caso el método simplemente llama a `send()` en la respuesta para retornar la cadena "Hello World!". Hay un número de otros métodos de respuesta para finalizar el ciclo de solicitud/respuesta, por ejemplo podrá llamar a `res.json()` para enviar una respuesta JSON o `res.sendFile()` para enviar un archivo.

> **JavaScript tip:** Usted puede utilizar cualquier nombre que quiera para los argumentos en las funciones callback; cuando la callback es invocada el primer argumento siempre sera la petición y el segundo siempre sera la respuesta. Tiene sentido nombrarlos de manera que pueda identificar el objeto con el que esta trabajando en el cuerpo de la callback.

El objeto que representa una aplicación de *Express*, también posee métodos para definir los manejadores de rutas para el resto de los verbos HTTP: `post()`, `put()`, `delete()`, `options()`, `trace()`, `copy()`, `lock()`, `mkcol()`, `move()`, `purge()`, `propfind()`, `proppatch()`, `unlock()`, `report()`, `mkactivity()`, `checkout()`, `merge()`, `m- search ()`, `notify()`, `subscribe()`, `unsubscribe()`, `patch()`, `search()`, y `connect()`.

Hay un método general para definir las rutas: `app.all()`, el cual será llamado en respuesta a cualquier método HTTP. Se usa para cargar funciones del middleware en una dirección particular para todos los métodos de peticiones. El siguiente ejemplo (de la documentación de *Express*) muestra el uso de los manejadores a `/secret` sin tener en cuenta el verbo HTTP utilizado (siempre que esté definido por el módulo http).

```
app.all('/secret', function(req, res, next) {
  console.log('Accediendo a la seccion secreta ...');
  next(); // pasa el control al siguiente manejador
});
```

Las rutas le permiten igualar patrones particulares de caracteres en la URL, y extraer algunos valores de ella y pasarlos como parámetros al manejador de rutas (como atributo del objeto petición pasado como parámetro).

Usualmente es útil agrupar manejadores de rutas para una parte del sitio juntos y accederlos usando un prefijo de ruta en común. (Ej: un sitio con una Wiki podría tener todas las rutas relacionadas a dicha sección en un archivo y siendo accedidas con el prefijo de ruta /wiki/. En *Express* esto se logra usando el objeto `express.Router` . Ej: podemos crear nuestra ruta

wiki en un módulo llamado wiki.js, y entonces exportar el objeto `Router` , como se muestra debajo:

```
// wiki.js - Modulo de rutas Wiki

var express = require('express');
var router = express.Router();

// Home page route
router.get('/', function(req, res) {
  res.send('Página de inicio Wiki');
});

// About page route
router.get('/about', function(req, res) {
  res.send('Acerca de esta wiki');
});

module.exports = router;
```

> **Nota:** Agregar rutas al objeto `Router` es como agregar rutas al objeto `app` (como se vio anteriormente).

Para usar el router en nuestro archivo app principal, necesitamos `require()` el módulo de rutas (**wiki.js**), entonces llame `use()` en la aplicación *Express* para agregar el Router al software intermediario que maneja las rutas. Las dos rutas serán accesibles entonces desde `/wiki/` y `/wiki/about/` .

```
var wiki = require('./wiki.js');
// ...
app.use('/wiki', wiki);
```

Le mostraremos mucho más sobre como trabajar con rutas, y en particular, acerca de como usar el `Router`, más adelante en la sección Rutas y controladores .

Usando middleware

El "middleware" es ampliamente utilizado en las aplicaciones de *Express:* desde tareas para servir archivos estáticos, a la gestión de errores o la compresión de las respuestas HTTP. Mientras las funciones de enrutamiento, con el objeto express.Router , se encargan del ciclo petición-respuesta, al gestionar la respuesta adecuada al cliente, las funciones de

middleware normalmente realizan alguna operación al gestionar una petición o respuesta y a continuación llaman a la siguiente función en la "pila", que puede ser otra función de middleware u otra función de enrutamiento. El orden en el que las funciones de middleware son llamadas depende del desarrollador de la aplicación.

> **Nota:** El middleware puede realizar cualquier operación: hacer cambios a una petición, ejecutar código, realizar cambios a la petición o al objeto pedido, puede también finalizar el ciclo de petición-respuesta. Si no finaliza el ciclo debe llamar a la función `next()` para pasar el control de la ejecución a la siguiente función del middleware (o a la petición quedaría esperando una respuesta ...).

La mayoría de las aplicaciones usan middleware desarrollado por terceras partes, para simplificar funciones habituales en el desarrollo web, como puede ser: gestión de cookies, sesiones, autentificado de usuarios, peticiones `POST` y datos en JSON, registros de eventos, etc. Puede encontrar en el siguiente enlace una lista de middleware mantenido por el equipo de *Express* (que también incluye otros paquetes populares de terceras partes). Las librerías de *Express* están disponibles con la aplicación NPM (Node Package Manager).

Para usar estas colecciones, primero ha de instalar la aplicación usando NPM. Por ejemplo para instalar el registro de peticiones HTTP morgan , se haría con el comando Bash:

```
$ npm install morgan
```

Entonces podría llamar a la función `use()` en un objeto de aplicación *Express* para utilizar este middleware a su aplicación.

```
var express = require('express');
var logger = require('morgan');
```

```
var logger = require('morgan');
var app = express();
app.use(logger('dev'));
...
```

> **Note:** Las funciones Middleware y routing son llamadas en el orden que son declaradas. Para algunos middleware el orden es importante (por ejemplo si el middleware de sesion depende del middleware de cookie, entonces el manejador de cookie tiene que ser llamado antes). Casi siempre es el caso que el middleware es llamado antes de configurar
>
> las rutas, o tu manejador de rutas no tendra acceso a la funcionalidad agregada por tu middleware.

Tu puedes escribir tu propia funcion middleware, y si quieres hacerlo así (solo para crear código de manejo de error). La única diferencia entre una función middleware y un callback manejador de rutas es que las funciones middleware tienen un tercer argumento `next`, cuyas funciones middleware son esperadas para llamarlas si ellas no completan el ciclo request (cuando la función midleware es llamada, esta contiene la próxima función que debe ser llamada).

Puede agregar una función middleware a la cadenan de procesamiento con cualquier `app.use()` o `app.add()`, dependiendo de si quiere aplicar el middleware a todas las respuestas o a respuestas con un verbo particular HTTP (`GET`, `POST`, etc). Usted especifica rutas, lo mismo en ambos casos, aunque la ruta es opcional cuando llama **app.use()**.

El ejemplo de abajo muestra como puede agregar la función middleware usando ambos métodos, y con/sin una ruta.

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

// An example middleware function
var a_middleware_function = function(req, res, next) {
  // ... perform some operations
  next(); // Call next() so Express will call the next middleware function i
}

// Function added with use() for all routes and verbs
app.use(a_middleware_function);
```

```
// Function added with use() for a specific route
app.use('/someroute', a_middleware_function);

// A middleware function added for a specific HTTP verb and route
app.get('/', a_middleware_function);

app.listen(3000);
```

> **JavaScript Tip:** Arriba declaramos la función middleware separadamente y la configuramos como el callback. En nuestra función previous manejadora de ruta declaramos la función callback cuando esta fué usada. En JavaScript, cuealquer aproximación es valida.

La documentación Express tiene mucha mas documentación excelente acerca del uso y escritura de middleware Express.

Sirviendo archivos estáticos

Puede utilizar el middleware express.static para servir archivos estáticos, incluyendo sus imagenes, CSS y JavaScript (`static()` es la única función middleware que es actualmente **parte** de *Express*). Por ejemplo, podria utilizar la linea de abajo para servir imágenes, archivos CSS, y archivos JavaScript desde un directorio nombrado **'public'** al mismo nivel desde donde llama a node:

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

Cualesquiere archivos en el directorio público son servidos al agregar su nombre de archivo (*relativo* a la ubicación del directorio "público") de la ubicación URL. Por ejemplo:

```
http://localhost:3000/images/dog.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/about.html
```

Puede llamar `static()` multiples ocasiones a servir multiples directorios. Si un archivo no puede ser encontrado por una función middleware entonces este simplemente será pasado en la subsequente middleware (el orden en que el middleware está basado en su orden de declaración)

declaración).

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

Tambien puede crear un prefijo virtual para sus URLs estáticas, aun más teniendo los archivos agregados en la ubicación URL. Por ejemplo, aqui especificamos a mount path tal que los archivos son bajados con el prefijo "/media":

```
app.use('/media', express.static('public'));
```

Ahora, puede bajar los archivos que estan en el directorio `publico` del path con prefijo `/media`.

```
http://localhost:3000/media/images/dog.jpg
http://localhost:3000/media/video/cat.mp4
http://localhost:3000/media/cry.mp3
```

Para más información, ver Sirviendo archivos estáticos en Express.

Manejando errores

Los errores majejados por una o más funciones especiales middleware que tienen cuatro argumentos, en lugar de las usuales tres: `(err, req, res, next)`. For example:

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Estas pueden devolver cualquier contenido, pero deben ser llamadas despues de todas las otras `app.use()` llamadas de rutas tal que ellas son las últimas middleware en el proceso de manejo de request!

Express viene con un manejador de error integrado, el que se ocupa de error remanente que pudiera ser encontrado en la app. Esta función middleware manejador de error esta agregada al final del stack de funciones middleware. Si pasa un error a `next()` y no lo maneja en un manejador de error, este sera manejado por el manejador de error integrado; el error sera escrito en el cliente con el rastreo de pila.

> **Note:** El rastreo de pila no esta incluido en el ambiente de producción. Para ejecutarlo en modo de producción necesita configurar la variable de ambiente NODE_ENV to

modo de producción necesita configurar la variable de ambiente `NODE_ENV` to `'production'`.

Note: HTTP404 y otros codigos de estatus de "error" no son tratados como errores. Si quiere manejar estos, puede agregar una función middleware para hacerlo. Para mas información vea las FAQ .

Para mayor información vea Manejo de error (Docs. Express).

Usando Bases de datos

Las apps de *Express* pueden usar cualquier mecanismo de bases de datos suportadas por *Node* (*Express* en sí mismo no define ningúna conducta/requerimiento specifico adicional para administración de bases de datos). Hay muchas opciones, incluyendo PostgreSQL, MySQL, Redis, SQLite, MongoDB, etc.

Con el propósito de usar éste, debe primero instalar el manejador de bases de datos utilizando NPM. Por ejemplo, para instalar el manejador para el popular NoSQL MongoDB querría utilizar el comando:

```
$ npm install mongodb
```

La base de datos por si misma puede ser instalada localmente o en un servidor de la nube. En su codigo Express requiere el manejador, conectarse a la base de datos, y entonces ejecutar operaciones crear, leer, actualizar, y borrar (CLAB). }El ejemplo de abajo (de la documentación Express documentation) muestra como puede encontrar registros en la colección "mamiferos" usando MongoDB.

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

MongoClient.connect('mongodb://localhost:27017/animals', function(err, db) {
  if (err) throw err;

  db.collection('mammals').find().toArray(function (err, result) {
    if (err) throw err;

    console.log(result);
  });
});
```

Otra aproximación popular es acceder a su base de datos indirectamente, via an Mapeo Objeto Relacional ("MOR"). En esta aproximación usted define sus datos como "objetos" o "modelos" y el MOR mapea estos a través del deliniamiento basico de la base de datos. Esta aproximación tiene el beneficio de que como un desrrollador puede continuar pensando en términos de objetos de JavaScript mas que en semántica de bases de datos, y en esto hay un lugar obvio para ejecutar la validación y chequeo de entrada de datos. Hablaremos más de bases de datos en un artículo posterior.

Para más información ver Integracion de Bases de Datos (docs Express).

Renderización de data (vistas)

El Motor de plantilla (referido como "motor de vistas" por *Express*) le permite definir la estructura de documento de salida en una plantilla, usando marcadores de posición para datos que seran llenados cuando una pagina es generada. Las plantillas son utilizadas generalmete para crear HTML, pero tambien pueden crear otros tipos de documentos. Express tiene soporte para numerosos motores de plantillas , y hay una util comparación de los motores más populares aquí: Comparando Motores de Plantillas de JavaScript: Jade, Mustache, Dust and More .

En su código de configuración de su aplicación usted configura el motor de plantillas para usar y su localización Express podiría buscar plantillas usando las configuraciones de 'vistas' y 'motores de vistas', mostrado abajo (tendría también que instalar el paquete conteniendo su librería de plantillas!)

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

// Set directory to contain the templates ('views')
app.set('views', path.join(__dirname, 'views'));

// Set view engine to use, in this case 'some_template_engine_name'
app.set('view engine', 'some_template_engine_name');
```

La apariencia de la plantilla dependera de qué motor use. Asumiendo que tiene un archivo de plantillas nombrado "index.<template_extension>" este contiene placeholders para variables de datos nombradas 'title' y "message", podría llamar `Response.render()` en una función manejadora de rutas para crear y enviar la HTML response:

```
app.get('/', function(req, res) {
  res.render('index', { title: 'About dogs', message: 'Dogs rock!' });
```

```
});
```

Para más información vea Usando motores de plantillas con Express (docs Express).

Estructura de Archivos

Express no hace asunciones en términos de estructura o que componentes usted usa. Rutas, vistas, archivos estáticos, y otras lógicas de aplicación específica puede vivir en cualquier número de archivos con cualquier estructura de directorio. Mientras que esto es perfectamente posible, se puede tener toda la aplicación en un solo archivo, en *Express*, tipicamente esto tiene sentido al desplegar su aplicacion dentro de archivos basados en función (e.g. administracion de cuentas, blogs, tableros de discusion) y dominio de problema arquitectonico (e.g. modelo, vista or controlador si tu pasas a estar usando una arquitectura MVC).

En un tópico posterior usaremos el Generador de Aplicaciones *Express Application Generator*, el que crea un esquelo de una app modular que podemos facilmente extender para crear aplicaciones web.

Resumen

¡Felicitaciones, ha completado el primer paso en su viaje Express/Node! Ahora debes comprender los principales beneficios de Express y Node, y más o menos cómo se verían las partes principales de una aplicación Express (rutas, middleware, manejo de errores y plantillas). ¡También debe comprender que con Express como un framework unopinionated, la forma en que une estas partes y las bibliotecas que usa dependen en gran medida de usted!

Por supuesto, Express es deliberadamente un un framework de aplicaciones web muy ligero, por lo que gran parte de sus beneficios y potencial proviene de bibliotecas y características de terceros. Lo veremos con más detalle en los siguientes artículos. En nuestro próximo artículo, veremos cómo configurar un entorno de desarrollo de Node, para que pueda comenzar a ver código de Express en acción.

Ver también

- Modules (Node API docs)
- Express (home page)
- Basic routing (Express docs)
- Routing guide (Express docs)

- Using template engines with Express (Express docs)
- Using middleware (Express docs)
- Writing middleware for use in Express apps (Express docs)
- Database integration (Express docs)
- Serving static files in Express (Express docs)
- Error handling (Express docs)

En este modulo

- Express/Node introduction
- Setting up a Node (Express) development environment
- Express Tutorial: The Local Library website
- Express Tutorial Part 2: Creating a skeleton website
- Express Tutorial Part 3: Using a Database (with Mongoose)
- Express Tutorial Part 4: Routes and controllers
- Express Tutorial Part 5: Displaying library data
- Express Tutorial Part 6: Working with forms
- Express Tutorial Part 7: Deploying to production

Last modified: 23 ago 2021, by MDN contributors