

THE EXPRESS HANDBOOK

FLAVIO COPES

Tabla de contenido

[El manual expreso](#)

[Visión general exprés](#)

[Solicitar parámetros](#)

[Enviando una respuesta](#)

[Envío de una respuesta JSON](#)

[Administrar cookies](#)

[Trabajar con encabezados HTTP](#)

[Redirecciones](#)

[Enrutamiento](#)

[CORS](#)

[Plantillas](#)

[La guía del pug](#)

[software intermedio](#)

[Sirviendo archivos estáticos](#)

[Enviar archivos](#)

[Sesiones](#)

[Validando entrada](#)

[Insumo desinfectante](#)

[Manejo de formularios](#)

[Subida de archivos en formularios](#)

[Un servidor Express HTTPS con un certificado autofirmado](#)

[Configurar Let's Encrypt para Express](#)

El manual expreso

El Manual Express sigue la regla del 80/20: aprende en el 20% del tiempo el 80% de un tema.

Creo que este enfoque ofrece una visión completa. Este libro no trata de cubrir todo lo relacionado con Express. Si crees que debería incluirse algún tema en concreto, dímelo.

Puedes contactarme en Twitter [@flaviocopes](#).

Espero que el contenido de este libro lo ayude a lograr lo que desea: **aprender lo básico Express**.

Este libro está escrito por Flavio. Publico **tutoriales de desarrollo web** todos los días en mi sitio web [flaviocopes.com](#).

¡Disfrutar!

Visión general exprés

Express es un marco web Node.js. Node.js es una herramienta increíble para crear aplicaciones y servicios de red. Express se basa en sus características para proporcionar una funcionalidad fácil de usar que satisfaga las necesidades del uso del caso.



Express es un [Node.js](#) Marco web.

Node.js es una herramienta increíble para crear aplicaciones y servicios de red.

Express se basa en sus características para proporcionar una funcionalidad fácil de usar que satisfaga las necesidades del caso de uso del Servidor Web.

Es de código abierto, gratuito, fácil de ampliar, muy eficaz y tiene montones y montones de paquetes que puede colocar y usar, para realizar todo tipo de cosas.

Instalación

Puede instalar Express en cualquier proyecto con [npm](#):

```
npm instalar express --save
```

o [hilo](#):

```
añadir hilo express
```

Ambos comandos también funcionarán en un directorio vacío, para iniciar su proyecto desde cero, aunque npm no crea ningún archivo `package.json` y Yarn crea uno básico.

Simplemente ejecute `npm init` o `yarn init` si está comenzando un nuevo proyecto desde cero.

Hola Mundo

Estamos listos para crear nuestro primer servidor Web Express.

Aquí hay algo de código:

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

app.get('/', (req, res) => res.send('¡Hola mundo!')) app.listen(3000, ()
=> console.log('Servidor listo'))
```

Guarde esto en un archivo `index.js` en la carpeta raíz de su proyecto e inicie el servidor usando

```
nodo index.js
```

Puede abrir el navegador en el puerto 3000 en localhost y debería ver Hello World!
mensaje.

Aprenda los conceptos básicos de Express al comprender el código Hello World

Esas 4 líneas de código hacen mucho detrás de escena.

Primero, importamos el paquete `expres` al valor `expres` .

Instanciamos una aplicación llamando a su método `app()` .

Una vez que tenemos el objeto de la aplicación, le decimos que escuche las solicitudes GET en la ruta `/` , usando el método `get()` .

Hay un método para cada **verbo HTTP**: `get()` , `post()` , `put()` , `delete()` , `patch()` :

```
app.get('/', (requerido, res) => { /* */ }) app.post('/',  
(requerido, res) => { /* */ }) app.put('/', (requerido,  
res) => { /* */ }) app.delete('/', (requerido, res) => { /*  
*/ }) app.patch('/', (requerido, res ) => { /* */ })
```

Esos métodos aceptan una función de devolución de llamada, que se llama cuando se inicia una solicitud, y nosotros necesita manejarlo.

Pasamos en una función de flecha:

```
(req, res) => res.send('¡Hola Mundo!')
```

Express nos envía dos objetos en esta devolución de llamada, a los que llamamos `req` y `res` , los `Request` y `Response` , que representan objetos `Request` y `Response`.

`Request` es la solicitud HTTP. Puede brindarnos toda la información al respecto, incluidos los parámetros de la solicitud, los encabezados, el cuerpo de la solicitud y más.

`Response` es el objeto de respuesta HTTP que enviaremos al cliente.

Lo que hacemos en esta devolución de llamada es enviar el mensaje '¡Hola mundo!' cadena al cliente, usando el `Response.send()` .

Este método establece esa cadena como el cuerpo y cierra la conexión.

La última línea del ejemplo en realidad inicia el servidor y le dice que escuche en el puerto 3000 . Pasamos una devolución de llamada que se llama cuando el servidor está listo para aceptar nuevas solicitudes.

Solicitar parámetros

Una referencia práctica a todas las propiedades del objeto de solicitud y cómo usarlas

Solicitar parámetros

Mencioné cómo el objeto Solicitud contiene toda la información de la solicitud HTTP.

Estas son las principales propiedades que probablemente usará:

Propiedad	Descripción
.aplicación	contiene una referencia al objeto de la aplicación Express
.baseUrl	la ruta base en la que responde la aplicación
.cuerpo	contiene los datos enviados en el cuerpo de la solicitud (deben ser analizados y se rellena manualmente antes de poder acceder a él)
.galletas	contiene las cookies enviadas por la solicitud (necesita el analizador de cookies software intermedio)
.nombre de host	el nombre de host del servidor
.ip	la ip del servidor
.método	el método HTTP utilizado
.parámetros	la ruta nombró parámetros
.sendero	la ruta URL
.protocolo	el protocolo de solicitud
.consulta	un objeto que contiene todas las cadenas de consulta utilizadas en la solicitud
.seguro	verdadero si la solicitud es segura (usa HTTPS)
.signedCookies	contiene las cookies firmadas enviadas por la solicitud (necesita el middleware del analizador de cookies)
.xhr	verdadero si la solicitud es XMLHttpRequest

Cómo recuperar los parámetros de cadena de consulta GET usando expreso

La cadena de consulta es la parte que viene después de la ruta de la URL y comienza con un signo de interrogación ? .

Ejemplo:

```
?nombre=flavio
```

Se pueden agregar múltiples parámetros de consulta usando & :

```
?nombre=flavio&edad=35
```

¿Cómo obtienes esos valores de cadena de consulta en Express?

Express lo hace muy fácil al completar el objeto `Request.query` para nosotros:

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

app.get('/', (requerido, res) =>
  { console.log(requerido.consulta) })

aplicación.escucha(8080)
```

Este objeto se rellena con una propiedad para cada parámetro de consulta.

Si no hay parámetros de consulta, es un objeto vacío.

Esto hace que sea fácil iterarlo usando el bucle `for...in`:

```
for ( clave const en req.query) {
  console.log(clave, req.consulta[clave])
}
```

Esto imprimirá la clave de propiedad de consulta y el valor.

También puede acceder a propiedades individuales:

```
req.query.name //flavio
req.query.age //35
```

Cómo recuperar los parámetros de cadena de consulta POST usando Express

Los parámetros de consulta POST son enviados por clientes HTTP, por ejemplo, mediante formularios, o al realizar una solicitud POST enviando datos.

¿Cómo se puede acceder a estos datos?

Si los datos se enviaron como JSON, utilizando Content-Type: application/json , utilizará el Express.json() middleware:

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

aplicación.use(express.json())
```

Si los datos se enviaron como JSON, utilizando Content-Type: application/x-www-form-urlencoded , utilizará el middleware express.urlencoded() :

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

app.use(express.urlencoded())
```

En ambos casos, puede acceder a los datos haciendo referencia a ellos desde Request.body :

```
app.post('/formulario', (req, res) => { const
  nombre = req.body.name })
```

Nota: las versiones anteriores de Express requerían el uso del **módulo** analizador de cuerpo para procesar los datos POST. Este ya no es el caso a partir de Express 4.16 (lanzado en septiembre de 2017) y versiones posteriores.

Enviando una respuesta

Cómo enviar una respuesta al cliente usando Express

En el ejemplo de Hello World, usamos el método `Response.send()` para enviar una cadena simple como respuesta y cerrar la conexión:

```
(req, res) => res.send('¡Hola mundo!')
```

Si pasa una cadena, establece el encabezado `Content-Type` en `text/html`.

si pasa un objeto o una matriz, establece el encabezado de tipo de contenido `application/json` y analiza ese parámetro en JSON.

`send()` establece automáticamente el encabezado de respuesta HTTP `Content-Length`.

`send()` también cierra automáticamente la conexión.

Usa `end()` para enviar una respuesta vacía

Una forma alternativa de enviar la respuesta, sin ningún cuerpo, es mediante el uso de `Response.end()` método:

```
res.end()
```

Establecer el estado de respuesta HTTP

Usa `Response.status()` :

```
res.status(404).end()
```

o

```
res.status(404).send('Archivo no encontrado')
```

`sendStatus ()` es un atajo:

```
res.sendStatus (200) //  
=== res.status (200) .send ('OK')  
  
res.sendStatus (403) //  
=== res.status (403) .send ('Prohibido')
```

```
res.sendStatus(404) //  
=== res.status(404).send('No encontrado')  
  
res.sendStatus(500) //  
=== res.status(500).send('Error interno del servidor')
```

Envío de una respuesta JSON

Cómo servir datos JSON usando la biblioteca Node.js Express

Cuando escucha conexiones en una ruta en Express, la función de devolución de llamada se invocará en cada llamada de red con una instancia de objeto de Solicitud y una instancia de objeto de Respuesta.

Ejemplo:

```
app.get('/', (req, res) => res.send('¡Hola mundo!'))
```

Aquí usamos el método `Response.send()` , que acepta cualquier cadena.

Puede enviar JSON al cliente mediante `Response.json()` , un método útil.

Acepta un objeto o matriz y lo convierte a JSON antes de enviarlo:

```
res.json({nombre de usuario: 'Flavio' })
```

Administrar cookies

Cómo usar el método `Response.cookie()` para manipular tus cookies

Utilice el método `Response.cookie()` para manipular sus cookies.

Ejemplos:

```
res.cookie('nombre de usuario', 'Flavio')
```

Este método acepta un tercer parámetro que contiene varias opciones:

```
res.cookie('nombre de usuario', 'Flavio', { dominio: '.flaviocopes.com', ruta: '/administrador', sec  
seguro: cierto })  
  
res.cookie('nombre de usuario', 'Flavio', { expira: nueva Fecha(Fecha.ahora()) + 900000), httpOnly: verdadero  
}))
```

Los parámetros más útiles que puede configurar son:

Valor	Descripción
dominio	el nombre de dominio de la cookie
caduca	establecer la fecha de caducidad de la cookie . Si falta, o 0, la cookie es una sesión Galleta
solo http	configure la cookie para que solo sea accesible por el servidor web. Ver HttpOnly
MaxAge	establecer el tiempo de caducidad relativo al tiempo actual, expresado en milisegundos
sendero	la ruta de las cookies . Predeterminado a /
seguro	Marca la cookie solo HTTPS
firmado	configurar la cookie para que se firme
mismoSitio	Valor de SameSite

Una cookie se puede borrar con

```
res.clearCookie('nombre de usuario')
```

Trabajar con encabezados HTTP

Aprenda a acceder y cambiar los encabezados HTTP usando Express

Acceder a valores de encabezados HTTP desde una solicitud

Puede acceder a todos los encabezados HTTP utilizando la propiedad `Request.headers` :

```
app.get('/', (req, res) =>
  { console.log(req.headers) })
```

Utilice el método `Request.header()` para acceder a un valor de encabezado de solicitud individual:

```
app.get('/', (req, res) =>
  { req.header('User-Agent')
  })
```

Cambiar cualquier valor de encabezado HTTP de una respuesta

Puede cambiar cualquier valor de encabezado HTTP usando `Response.set()` :

```
res.set('Tipo de contenido', 'texto/html')
```

Sin embargo, hay un atajo para el encabezado de tipo de contenido:

```
res.type('.html')
// => 'texto/html'

res.type('html')
// => 'texto/html'

res.type('json') // =>
'aplicación/json'

res.type('aplicación/json') // =>
'aplicación/json'

res.type('png') // =>
imagen/png:
```


Redirecciones

Cómo redirigir a otras páginas del lado del servidor

Los redireccionamientos son comunes en el desarrollo web. Puede crear una redirección utilizando el

Método `Response.redirect()` :

```
res.redirect('/ir-allí')
```

Esto crea una redirección 302.

Una redirección 301 se realiza de esta manera:

```
res.redirect(301, '/ir-allí')
```

Puede especificar una ruta absoluta (`/go-there`), una URL absoluta (`https://anothersite.com`), una ruta relativa (`go-there`) o usar `..` para retroceder un nivel:

```
res.redirect('../ir-allí') res.redirect('..')
```

También puede redirigir de nuevo al valor del encabezado HTTP Referer (predeterminado en `/` si no está configurado) usando

```
res.redirect('atrás')
```


Enrutamiento

El enrutamiento es el proceso de determinar qué debe suceder cuando se llama a una URL, o también qué partes de la aplicación deben manejar una solicitud entrante específica.

El enrutamiento es el proceso de determinar qué debe suceder cuando se llama a una URL, o también qué partes de la aplicación deben manejar una solicitud entrante específica.

En el ejemplo de Hello World usamos este código

```
app.get('/', (requerido, res) => { /* */ })
```

Esto crea una ruta que asigna el acceso a la URL del dominio raíz / usando el método HTTP GET a la respuesta que queremos proporcionar.

Parámetros con nombre

¿Qué pasa si queremos escuchar solicitudes personalizadas, tal vez queremos crear un servicio que acepte una cadena y la devuelva en mayúsculas, y no queremos que el parámetro se envíe como una cadena de consulta, sino como parte de la URL? Usamos parámetros con nombre:

```
app.get('/ mayúsculas /: theValue', (req, res) => res.send (req.params.theValue.toUpperCase ()))
```

Si enviamos una solicitud a /mayúsculas/prueba , obtendremos PRUEBA en el cuerpo de la respuesta.

Puede usar varios parámetros con nombre en la misma URL y todos se almacenarán en

```
req.params .
```

Use una expresión regular para hacer coincidir una ruta

Puedes usar [expresiones regulares](#) para hacer coincidir varias rutas con una declaración:

```
app.get(/post/, (req, res) => { /* */ })
```

coincidirá con /post , /post/first , /thepost , /posting/something , etc.

CORS

Cómo permitir solicitudes entre sitios configurando CORS

Una aplicación de JavaScript que se ejecuta en el navegador generalmente solo puede acceder a los recursos HTTP en el mismo dominio (origen) que lo atiende.

La carga de imágenes o scripts/estilos siempre funciona, pero las llamadas XHR y Fetch a otro servidor fallarán, a menos que ese servidor implemente una forma de permitir esa conexión.

Esta forma se llama CORS, **Cross-Origin Resource Sharing**.

También cargar Web Fonts usando @font-face tiene una política del mismo origen por defecto y otras cosas menos populares (como texturas WebGL y recursos drawImage cargados en la API de Canvas).

Una cosa muy importante que necesita CORS son los **Módulos ES**, introducidos recientemente en los modernos navegadores

Si no configura una política de CORS **en el servidor** que permita servir orígenes de terceros, la solicitud fallará.

Obtener ejemplo:

```
> fetch('https://google.com')
< ▶ Promise {<pending>}
```

```
✖ Failed to load https://google.com/: Redirect from https://google.com/ to 'https://www.google.it/?gfe_rd=cr&dcr=0&ei=TiDHWtehBcPCXprvpIgF' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'https://flaviocopes.com' is therefore not allowed access. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
✖ Uncaught (in promise) TypeError: Failed to fetch
> |
```

Ejemplo XHR:

```
> const xhr = new XMLHttpRequest()
  xhr.onreadystatechange = () => {
    if (xhr.readyState === 4) {
      xhr.status === 200 ? console.log(xhr.responseText) : console.error('error')
    }
  }
  xhr.open('GET', 'https://google.com')
  xhr.send()
< undefined
```

```
✖ Failed to load https://google.com/: Redirect from 'https://google.com/' to 'http://localhost:1313' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:1313' is therefore not allowed access.
✖ ▶ error
  ▶ XHR finished loading: GET "https://google.com/".
> |
```

Un recurso de origen cruzado falla si es:

- a un **dominio** diferente
- a un **subdominio** diferente
- a un **puerto** diferente
- a un **protocolo** diferente

y está ahí para su seguridad, para evitar que usuarios maliciosos exploten la Plataforma Web.

Pero si controla tanto el servidor como el cliente, tiene todas las buenas razones para permitirles hablar entre sí.

¿Cómo?

Depende de su pila del lado del servidor.

Compatibilidad con navegador

Bastante bien (básicamente todos excepto IE <10):

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android *	Blackberry	Opera Mobile *	Chrome Android
6		54	60	³ 8	45	³ 8.4		¹ 4			
7	12	55	61	³ 9	46	³ 9.2		¹ 4.1			
² 8	13	56	62	³ 9.1	47	³ 9.3		¹ 4.3			
² 9	14	57	63	³ 10	48	³ 10.2		4.4		12	
¹ 10	15	58	64	³ 10.1	49	³ 10.3		4.4.4	¹ 7	12.1	
11	16	59	65	³ 11	50	³ 11.2	all	62	10	37	64
	17	60	66	³ 11.1	51	³ 11.3					
		61	67	³ TP	52						
			68								

Ejemplo con Express

Si usa Node.js y Express como marco, use el [paquete de middleware CORS](#).

Aquí hay una implementación simple de un servidor Express Node.js:

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

```
app.get('/sin-cors', (req, res, next) => { res.json({ msg: 'ÿ no
  CORS, no party!' })
})

const servidor = app.listen(3000, () => {
  console.log('Escuchando en el puerto %s', server.address().port) })
```

Si presiona /sin-cors con una solicitud de recuperación de un origen diferente, aumentará el Problema CORS.

Todo lo que necesita hacer para que las cosas funcionen es solicitar el paquete cors vinculado anteriormente y pasarlo como una función de middleware a un controlador de solicitudes de punto final:

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




app.get('/with-cors', cors(), (req, res, next) => {
  res.json({ msg: '¡WHOA con CORS funciona! ÿ ÿ ' }) })

/* el resto de la aplicación */
```

Hice un ejemplo simple de Glitch. [Aquí está el cliente](https://glitch.com/edit/#!/flavio-cors-client) funcionando, y aquí está su código: <https://glitch.com/edit/#!/flavio-cors-client>.

Este es el servidor Node.js Express: <https://glitch.com/edit/#!/flaviocopes-cors-example-express>




Observe cómo la solicitud que falla porque no maneja correctamente los encabezados CORS aún se recibe, como puede ver en el panel Red, donde se encuentra el mensaje que envió el servidor:

Name	× Headers	Preview	Response	Timing
 flavio-cors-client api.glitch.com/proj...		▼ {msg: "😞 no CORS, no party!"}		
 without-cors flaviocopes-cors-e...		msg: "😞 no CORS, no party!"		
 with-cors flaviocopes-cors-e...				
3 / 11 requests 1.2 KB / ...				

Permitir solo orígenes específicos

Sin embargo, este ejemplo tiene un problema: el servidor aceptará CUALQUIER solicitud como origen cruzado.

Como puede ver en el panel Red, la solicitud que pasó tiene un acceso de encabezado de respuesta controlar-permitir-origen: *

Name	×	Headers	Preview	Response	Timing
 flavio-cors-client api.glitch.com/proj...		Referrer Policy: no-referrer-when-downgrade			
 without-cors flaviocopes-cors-e...		▼ Response Headers			
 with-cors flaviocopes-cors-e...		access-control-allow-origin: *			
		content-length: 45			
		content-type: application/json; charset=utf-8			
		date: Fri, 06 Apr 2018 16:10:18 GMT			
		etag: W/"2d-0UnEpHCoz1V3cgJ99D+m3n/b0p4"			
		status: 200			
		x-powered-by: Express			
3 / 11 requests 1.2 KB / ...		▼ Request Headers			

Debe configurar el servidor para que solo permita el servicio de un origen y bloquee todos los demás.

Usando la misma biblioteca de cors Node, así es como lo haría:

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

const corsOptions = {
  origen: 'https://tudominio.com'
}

app.get('/productos/:id', cors(corsOptions), (req, res, next) => {
  //...
})
```

También puedes servir más:

```
const whitelist = ['http://example1.com', 'http://example2.com'] const corsOptions =
{ origen: función (origen, devolución de llamada) {

  if (lista blanca.indexOf(origen) !== -1) {
    devolución de llamada (nulo,
verdadero) } más { devolución de
llamada (nuevo error ('No permitido por CORS'))
  }
}
}
```

Verificación previa

Hay algunas solicitudes que se manejan de manera "simple". Todas las solicitudes GET pertenecen a este grupo.

También lo hacen *algunas* solicitudes POST y HEAD .

Las solicitudes POST también están en este grupo, si cumplen el requisito de usar un tipo de contenido de

- application/x-www-form-urlencoded
- multipart/datos de formulario
- Texto sin formato

Todas las demás solicitudes deben pasar por una fase de aprobación previa, llamada verificación previa. El navegador hace esto para determinar si tiene permiso para realizar una acción, emitiendo una solicitud de OPCIONES .

Una solicitud de verificación previa contiene algunos encabezados que el servidor usará para verificar los permisos (se omiten los campos irrelevantes):

```
OPCIONES /the/resource/you/request Access-Control-Request-Method: POST Access-Control-Request-Headers: origin, x-requested-with, accept Origin: https://your-origin.com
```

El servidor responderá con algo como esto (se omiten los campos irrelevantes):

```
HTTP/1.1 200 Aceptar
Access-Control-Allow-Origin: https://your-origin.com Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE
```

Verificamos POST, pero el servidor nos dice que también podemos emitir otros tipos de solicitudes HTTP para ese recurso en particular.

Siguiendo el ejemplo anterior de Node.js Express, el servidor también debe manejar la solicitud de OPCIONES:

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

//permitir OPCIONES en un solo recurso app.options('/the/resource/you/request', cors())

//permitir OPCIONES en todos los recursos
app.options('*', cors())
```


Plantillas

Express es capaz de manejar motores de plantillas del lado del servidor. Los motores de plantillas nos permiten agregar datos a una vista y generar HTML dinámicamente.

Express es capaz de manejar motores de plantillas del lado del servidor.

Los motores de plantillas nos permiten agregar datos a una vista y generar HTML dinámicamente.

Express usa Jade como predeterminado. Jade es la versión antigua de Pug, concretamente Pug 1.0.

El nombre se cambió de Jade a Pug debido a un problema de marca registrada en 2016, cuando el proyecto lanzó la versión 2. Todavía puede usar Jade, también conocido como Pug 1.0, pero en el futuro, es mejor usar Pug 2.0

Aunque la última versión de Jade tiene 3 años (en el momento de escribir este artículo, verano de 2018), sigue siendo la predeterminada en Express por motivos de compatibilidad con versiones anteriores.

En cualquier proyecto nuevo, debe usar Pug u otro motor de su elección. El sitio oficial de Pug es <https://pugjs.org/>.

Puede usar muchos motores de plantillas diferentes, incluidos Pug, Handlebars, Moustache, EJS y más.

Usando pug

Para usar Pug primero debemos instalarlo:

```
npm instalar pug
```

y al inicializar la aplicación Express, debemos configurarla:

```
const express = require('express') const app
= express() app.set('view engine', 'pug')
```

Ahora podemos comenzar a escribir nuestras plantillas en archivos .pug .

Crear una vista acerca de:

```
app.get('/acerca de', (req, res) =>
  { res.render('acerca de')
  })
```


y la plantilla en `views/about.pug` :

```
p Hola de Flavio
```

Esta plantilla creará una etiqueta `p` con el contenido `Hola de Flavio` .

Puede interpolar una variable usando

```
app.get('/sobre', (req, res) => {  
  res.render('acerca de', { nombre: 'Flavio' }) })
```

```
p Hola de #{nombre}
```

Esta es una breve introducción a Pug, en el contexto de su uso con Express. Consulte la [guía Pug](#) para obtener más información sobre cómo usar Pug.

Si está acostumbrado a motores de plantillas que usan HTML e interpolan variables, como los manillares (que se describen a continuación), es posible que tenga problemas, especialmente cuando necesite convertir HTML existente a Pug. Este convertidor en línea de HTML a Jade (que es muy similar, pero un poco diferente a Pug) será de gran ayuda: <https://jsonformatter.org/html-to-jade>

Vea también las [diferencias entre Jade y Pug](#) .

Uso de manubrios

Intentemos usar manubrios en lugar de Pug.

Puede instalarlo usando `npm install hbs` .

Coloque un archivo de plantilla `about.hbs` en la carpeta `views/` :

```
Hola de {{nombre}}
```

y luego use esta configuración Express para servirla en `/about` :

```
const express = require('express') const app  
= express() const hbs = require('hbs')  
  
app.set(' motor de vista', 'hbs')  
app.set('vistas', ruta.join(__dirname, 'vistas'))  
  
app.get('/sobre', (req, res) => {
```

```
    res.render('acerca de', { nombre: 'Flavio' })
  })

  app.listen(3000, () => console.log('Servidor listo'))
```

También puede **renderizar una aplicación React del lado del servidor**, utilizando el paquete [express-react-views](#) .

Comience con `npm install express-react-views react react-dom` .

Ahora, en lugar de requerir `hbs` , requerimos `express-react-views` y lo usamos como motor, usando archivos `jsx` :

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

app.set('view engine', 'jsx') app.engine('jsx',
require('express-react-views').createEngine())

app.get('/sobre', (req, res) => {
  res.render('acerca de', { nombre: 'Flavio' })
})

app.listen(3000, () => console.log('Servidor listo'))
```

Simplemente coloque un archivo `about.jsx` en `views` , y llamando `/acerca` de debería presentar un "Hola desde vistas/ cadena Flavio":

```
const Reaccionar = require('reaccionar')

class HelloMessage extends React.Component {
  render()
    { devuelve <div>Hola desde {this.props.name}</div>
    }
}

module.exports = HolaMensaje
```

La guía del pug

Cómo usar el motor de plantillas Pug

- [Introducción al barro amasado](#)
- [¿Cómo se ve Pug?](#)
- [Instalar barro amasado](#)
- [Configurar Pug para que sea el motor de plantillas en Express](#)
- [Tu primera plantilla Pug](#)
- [Interpolando variables en Pug](#)
- [Interpolar un valor de retorno de función](#)
- [Agregar atributos de identificación y clase a los elementos](#)
- [Establecer el tipo de documento](#)
- [Etiquetas meta](#)
- [Adición de guiones y estilos](#)
- [Guiones en línea](#)
- [Bucles](#)
- [Condicionales](#)
- [Establecer variables](#)
- [Variables incrementales](#)
- [Asignación de variables a valores de elementos](#)
- [Iterando sobre variables](#)
- [Incluyendo otros archivos Pug](#)
- [Definición de bloques](#)
- [Ampliación de una plantilla base](#)
- [Comentarios](#)
 - [Visible](#)
 - [Invisible](#)

Introducción al barro amasado

¿Qué es pug? Es un motor de plantillas para aplicaciones Node.js del lado del servidor.

Express es capaz de manejar motores de plantillas del lado del servidor. Los motores de plantillas nos permiten agregar datos a una vista y generar HTML dinámicamente.

Pug es un nombre nuevo para algo viejo. Es *Jade 2.0*.

El nombre se cambió de Jade a Pug debido a un problema de marca registrada en 2016, cuando el proyecto lanzó la versión 2. Todavía puede usar Jade, también conocido como Pug 1.0, pero en el futuro, es mejor usar Pug. 2.0

Vea también las [diferencias entre Jade y Pug](#) .

Express usa Jade como predeterminado. Jade es la versión antigua de Pug, concretamente Pug 1.0.

Aunque la última versión de Jade tiene 3 años (en el momento de escribir este artículo, verano de 2018), sigue siendo la predeterminada en Express por motivos de compatibilidad con versiones anteriores.

En cualquier proyecto nuevo, debe usar Pug u otro motor de su elección. El sitio oficial de Pug es <https://pugjs.org/>.

¿Cómo se ve Pug?

```
p Hola de Flavio
```

Esta plantilla creará una etiqueta `p` con el contenido Hola de Flavio .

Como puedes ver, Pug es bastante especial. Toma el nombre de la etiqueta como lo primero en una línea, y el resto es el contenido que va dentro de ella.

Si está acostumbrado a motores de plantillas que usan HTML e interpolan variables, como los manillares (que se describen a continuación), es posible que tenga problemas, especialmente cuando necesite convertir HTML existente a Pug. Este convertidor en línea de HTML a Jade (que es muy similar, pero un poco diferente a Pug) será de gran ayuda: <https://jsonformatter.org/html-to-jade>

Instalar barro amasado

Instalar Pug es tan simple como ejecutar `npm install` :

```
npm instalar pug
```

Configurar Pug para que sea el motor de plantillas en Express

y al inicializar la aplicación Express, debemos configurarla:

```
const ruta = require('ruta') const
express = require('express') const app =
express()
```

```
app.set('motor de vista', 'pug')
app.set('vistas', ruta.join(__dirname, 'vistas'))
```

Tu primera plantilla Pug

Crear una vista acerca de:

```
app.get('/acerca de', (req, res) =>
  { res.render('acerca de')
  })
```

y la plantilla en views/about.pug :

```
p Hola de Flavio
```

Esta plantilla creará una etiqueta p con el contenido Hola de Flavio .

Interpolando variables en Pug

Puede interpolar una variable usando

```
app.get('/sobre', (req, res) => {
  res.render('acerca de', { nombre: 'Flavio' })
})
```

```
p Hola de #{nombre}
```

Interpolar un valor de retorno de función

Puede interpolar el valor de retorno de una función usando

```
app.get('/sobre', (req, res) => {
  res.render('acerca de', { getNombre: () => 'Flavio' })
})
```

```
p Hola de #{getName()}
```

Agregar atributos de identificación y clase a los elementos

p#título
p.título

Establecer el tipo de documento

tipo de documento html

Etiquetas meta

```
html
  cabeza
    meta(charset='utf-8') meta(http-
    equiv='X-UA-Compatible', content='IE=edge') meta(name='description',
    content='Alguna descripción') meta(name ='viewport', content='width=device-
    width, initial-scale=1')
```

Adición de guiones y estilos

```
html
  cabeza
    secuencia de
    comandos(src="secuencia de comandos.js") secuencia de comandos(src='//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js')

    enlace (rel='hoja de estilo', href='css/main.css')
```

Guiones en línea

```
alerta de secuencia de comandos ('prueba')

guion
  (función(b,o,i,l,e,r){b.GoogleAnalyticsObject=b[l]||(b[l]= función(){(b[l].q=b[l] .q||
  []).push(argumentos)}};b[l].l=+nueva fecha;e=o.createElement(i);r=o.getElementsByTagName(i)
  [0];e.src ='//www.google-analytics.com/analytics.js';r.parentNode.insertBefore(e,r)}
  (window,document,'script','ga')); ga('crear','UA-XXXXX-X');ga('enviar','vista de página');
```

Bucles

```
ul
  cada color en ['Rojo', 'Amarillo', 'Azul']
    li = color

ul
  cada color, índice en ['Rojo', 'Amarillo', 'Azul']
    li= 'Número de color' + índice + ': ' + color
```

Condicionales

```
si nombre
  h2 Hola de #{nombre}
más
  h2 hola
```

else-if también funciona:

```
si nombre
  h2 Hola de #{nombre}
más si otro nombre
  h2 Hola de #{otroNombre} más

  h2 hola
```

Establecer variables

Puede establecer variables en las plantillas de Pug:

```
- var nombre = 'Flavio'
- var edad = 35 -
var roger = {nombre: 'Roger'} - var perros
= ['Roger', 'Syd']
```

Variables incrementales

Puede incrementar una variable numérica usando ++ :

```
edad++
```

Asignación de variables a valores de elementos

```
p= nombre
```

```
span.edad= edad
```

Iterando sobre variables

Puede usar `para` o `cada uno` . No hay diferencia.

```
para perro en perros  
  li = perro
```

```
ul  
  cada perro en perros  
    li = perro
```

Puede usar `.length` para obtener la cantidad de elementos:

```
p Hay #{valores.longitud}
```

`while` es otro tipo de bucle:

```
- var n = 0;  
  
ul  
  mientras que n <= 5  
    li= n++
```

Incluyendo otros archivos Pug

En un archivo Pug puede incluir otros archivos Pug:

```
incluir otroarchivo.pug
```

Definición de bloques

Un sistema de plantillas bien organizado definirá una plantilla base y luego todas las demás plantillas.
extenderse desde ella.

La forma en que se puede ampliar una parte de una plantilla es mediante el uso de bloques:

```
html
  cabeza
    secuencia de
      comandos(src="secuencia de comandos.js") secuencia de comandos(src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js")

    enlace (rel='hoja de estilo', href='css/main.css')
    cabeza de bloque

  cuerpo
    bloque cuerpo
      h1 Página de inicio
      p bienvenido
```

En este caso, un bloque, el cuerpo , tiene algo de contenido, mientras que la cabeza no. cabeza está destinada a puede usarse para agregar contenido adicional al encabezado, mientras que el contenido del cuerpo está hecho para ser anulado por otras páginas.

Ampliación de una plantilla base

Una plantilla puede extender una plantilla base usando la palabra clave extends :

```
se extiende a casa.pug
```

Una vez hecho esto, debe redefinir los bloques. Todo el contenido de la plantilla debe ir en bloques, de lo contrario el motor no sabe dónde ponerlos.

Ejemplo:

```
se extiende a casa.pug

bloque cuerpo
  h1 Otra página
  p hola!
  ul
    li Algo li Algo más
```

Puede redefinir uno o más bloques. Los que no se redefinieron se mantendrán con el contenido de la plantilla original.

Comentarios

Los comentarios en Pug pueden ser de dos tipos: visibles o no visibles en el HTML resultante.

Visible

En línea:

```
// algun comentario
```

Bloquear:

```
//  
alguno  
comentario
```

Invisible

En línea:

```
//- algún comentario
```

Bloquear:

```
//-  
alguno  
comentario
```

software intermedio

Un middleware es una función que se conecta al proceso de enrutamiento y realiza alguna operación en algún punto, dependiendo de lo que quiera hacer.

Un middleware es una función que se conecta al proceso de enrutamiento y realiza alguna operación en algún punto, dependiendo de lo que quiera hacer.

Se usa comúnmente para editar los objetos de solicitud o respuesta, o para finalizar la solicitud antes de que finalice. llega al código del controlador de ruta.

Se agrega a la pila de ejecución de esta manera:

```
app.use((requerido, res, siguiente) => { /* */ })
```

Esto es similar a definir una ruta, pero además de las instancias de los objetos Solicitud y Respuesta, también tenemos una referencia a la *siguiente* función de middleware, que asignamos al siguiente variable .

Siempre llamamos a `next()` al final de nuestra función de middleware, para pasar la ejecución al siguiente controlador, a menos que queramos finalizar prematuramente la respuesta y enviarla de vuelta al cliente.

Por lo general, utiliza middleware prefabricado, en forma de paquetes npm . Una gran lista de los disponibles están [aquí](#).

Un ejemplo es `cookie-parser` , que se usa para analizar las cookies en el objeto `req.cookies` . Lo instala usando `npm install cookie-parser` y puede usarlo así:

```
const express = require('express') const app
= express() const cookieParser = require('cookie-
parser')

app.get('/', (req, res) => res.send('¡Hola mundo!'))

app.use(cookieParser())
app.listen(3000, () => console.log('Servidor listo'))
```

También puede configurar una función de middleware para que se ejecute solo para rutas específicas, no para todas, usándola como el segundo parámetro de la definición de ruta:

```
const myMiddleware = (req, res, next) => {
  /* ... */
  siguiente()
}

app.get('/', myMiddleware, (req, res) => res.send('¡Hola mundo!'))
```

Si necesita almacenar datos que se generan en un middleware para pasarlos a funciones de middleware posteriores o al controlador de solicitudes, puede usar el objeto `Request.locals` . Adjuntará esos datos a la solicitud actual:

```
req.locals.name = 'Flavio'
```

Sirviendo archivos estáticos

Cómo servir activos estáticos directamente desde una carpeta en Express

Es común tener imágenes, CSS y más en una subcarpeta pública y exponerlos al nivel raíz:

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

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

/* ... */

app.listen(3000, () => console.log('Servidor listo'))
```

Si tiene un archivo `index.html` en `public/`, se servirá si ahora accede a la URL del dominio raíz (`http://localhost:3000`)

Enviar archivos

Express proporciona un método práctico para transferir un archivo como adjunto: `Response.download()`

Express proporciona un método útil para transferir un archivo como archivo adjunto: `Response.download()`.

Una vez que un usuario llega a una ruta que envía un archivo usando este método, los navegadores le pedirán al usuario que descargar.

El método `Response.download()` le permite enviar un archivo adjunto a la solicitud y el navegador, en lugar de mostrarlo en la página, lo guardará en el disco.

```
app.get('/', (req, res) => res.download('./file.pdf'))
```

En el contexto de una aplicación:

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

app.get('/', (req, res) => res.download('./file.pdf')) app.listen(3000, () =>
console.log('Servidor listo'))
```

Puede configurar el archivo para que se envíe con un nombre de archivo personalizado:

```
res.download('./file.pdf', 'user-facing-filename.pdf')
```

Este método proporciona una función de devolución de llamada que puede usar para ejecutar código una vez que el archivo enviado:

```
res.download('./file.pdf', 'user-facing-filename.pdf', (err) => {
  si (err) {
    // manejar el error
    devolver
  } else { //
    hacer algo
  } })
```

Sesiones

Cómo usar sesiones para identificar usuarios en todas las solicitudes

Por defecto, las solicitudes Express son secuenciales y ninguna solicitud puede vincularse entre sí. No hay forma de saber si esta solicitud proviene de un cliente que ya realizó una solicitud anteriormente.

Los usuarios no pueden ser identificados a menos que se utilice algún tipo de mecanismo que lo haga posible.

Eso es lo que son las sesiones.

Cuando se implemente, a cada usuario de su API o sitio web se le asignará una sesión única, y esto le permitirá almacenar el estado del usuario.

Usaremos el módulo de sesión express , que es mantenido por el equipo de Express.

Puedes instalarlo usando

```
npm instalar sesión expresa
```

y una vez que haya terminado, puede crear una instancia en su aplicación con

```
const session = require('express-session')
```

Este es un middleware, por lo que lo *instala* en Express usando

```
const express = require('express') const
session = require('express-session')

const app = express()
app.use(sesión(
  'secreto': '343ji43j4n3jn4jk3n' ))
```

Una vez hecho esto, todas las solicitudes a las rutas de la aplicación ahora usan sesiones.

secret es el único parámetro requerido, pero hay muchos más que puede usar. debería ser un cadena aleatoriamente única para su aplicación.

La sesión se adjunta a la solicitud, por lo que puede acceder a ella utilizando req.session aquí:

```
app.get('/', (req, res, next) => { // req.sesión

}
```

Este objeto se puede usar para obtener datos de la sesión y también para establecer datos:

```
req.sesión.nombre = 'Flavio'  
console.log(req.sesión.nombre) // 'Flavio'
```

Estos datos se serializan como JSON cuando se almacenan, por lo que es seguro usar objetos anidados.

Puede usar sesiones para comunicar datos al middleware que se ejecuta más adelante, o para recuperarlos más tarde en solicitudes posteriores.

¿Dónde se almacenan los datos de la sesión? depende de cómo configure el módulo de sesión rápida.

Puede almacenar datos de sesión en

- **memoria**, no pensada para producción
- una base de **datos** como MySQL o Mongo un
- **caché de memoria** como Redis o Memcached

Hay una gran lista de terceros paquetes que implementan una amplia variedad de diferentes tiendas de almacenamiento en caché compatibles en <https://github.com/expressjs/session>

Todas las soluciones almacenan la identificación de la sesión en una cookie y mantienen los datos del lado del servidor. El cliente recibirá la identificación de la sesión en una cookie y la enviará junto con cada solicitud HTTP.

Haremos referencia a ese lado del servidor para asociar la identificación de la sesión con los datos almacenados localmente.

La memoria es la predeterminada, no requiere una configuración especial de su parte, es lo más simple, pero está destinada solo para fines de desarrollo.

La mejor opción es un caché de memoria como Redis, para el cual debe configurar su propio infraestructura.

Otro paquete popular para administrar sesiones en Express es la diferencia de sesión de cookie : almacena , que tiene un gran datos del lado del cliente en la cookie. No recomiendo hacerlo porque almacenar datos en cookies significa que se almacenan en el lado del cliente y se envían de un lado a otro en cada solicitud realizada por el usuario. También tiene un tamaño limitado, ya que solo puede almacenar 4 kilobytes de datos. Las cookies también deben estar protegidas, pero de forma predeterminada no lo están, ya que las cookies seguras son posibles en los sitios HTTPS y debe configurarlas si tiene servidores proxy.

Validando entrada

Aprenda a validar cualquier dato que ingrese como entrada en sus terminales Express

Supongamos que tiene un punto final POST que acepta los parámetros de nombre, correo electrónico y edad:

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

aplicación.use(express.json())

app.post('/formulario', (req, res) => { const
  nombre = req.body.name const email =
  req.body.email const edad = req.body.age })
```

¿Cómo valida esos resultados del lado del servidor para asegurarse de que

- nombre es una cadena de al menos 3 caracteres?
- ¿El correo electrónico es un correo electrónico real?
- ¿La edad es un número, entre 0 y 110?

La mejor manera de manejar la validación de cualquier tipo de entrada proveniente del exterior en Express es usando el [paquete express-validator](#) :

```
npm instalar validador expreso
```

Necesita el objeto de verificación del paquete:

```
const { comprobar } = require('express-validator/check')
```

Pasamos una serie de llamadas check() como segundo argumento de la llamada post() . Cada

La llamada check() acepta el nombre del parámetro como argumento:

```
app.post('/formulario', [
  check('nombre').isLength({ min: 3 }),
  check('correo electrónico').isEmail(),
  check('edad').isNumeric() ], (req, res) => { const
  nombre = req.body.name const email =
  req.body.email const edad = req.body.age })
```

Aviso que usé

- `esLongitud()`
- `esCorreo electrónico()`
- `esnumérico()`

Hay muchos más de estos métodos, todos provenientes de [validator.js](#), incluido:

- `contains()` , comprobar si el valor contiene el valor especificado
- `equals()` , comprobar si el valor es igual al valor especificado
- `esAlfa()`
- `esAlfanumérico()`
- `esAscii ()`
- `esBase64 ()`
- `es booleano ()`
- `esMoneda()`
- `esDecimal()`
- `esta vacio()`
- `es FQDN ()` , Qué es un nombre de dominio completo?
- `esFlotante()`
- `esHash()`
- `esHexColor()`
- `es IP ()`
- `isIn()` , compruebe si el valor está en una matriz de valores permitidos
- `esInt ()`
- `esJSON()`
- `esLatLong ()`
- `esLongitud()`
- `es minúsculas ()`
- `esTeléfonoMóvil()`
- `esnumérico()`
- `esCódigoPostal()`
- `es URL ()`
- `esMayúsculas()`
- `isWhitelisted()` , compara la entrada con una lista blanca de caracteres permitidos

Puede validar la entrada con una expresión regular usando `matches()` .

Las fechas se pueden verificar usando

- verifique si la fecha ingresada es posterior a la que pasa `isAfter()` ,
- verifique si la fecha ingresada es anterior a la que pasa `isBefore()` ,
- `esISO8601()`

- `esRFC3339()`

Para obtener detalles exactos sobre cómo usar esos validadores, consulte <https://github.com/chriso/validator.js#validators>.

Todos esos controles se pueden combinar canalizándolos:

```
check('nombre').isAlpha().isLength({ min: 10 })
```

Si hay algún error, el servidor envía automáticamente una respuesta para comunicar el error. Por ejemplo, si el correo electrónico no es válido, esto es lo que se devolverá:

```
{
  "errores":
    [{ "ubicación": "cuerpo",
      "mensaje": "Valor no válido",
      "parámetro": "correo electrónico" }]
}
```

Este error predeterminado se puede anular para cada verificación que realice, usando `withMessage()` :

```
check('nombre').isAlpha().withMessage('Debe tener solo caracteres alfabéticos').isLength({ min: 10 }).withMessage('Debe tener al menos 10 caracteres')
```

¿Qué sucede si desea escribir su propio validador especial y personalizado? Puedes usar la costumbre `validador`

En la función de devolución de llamada, puede rechazar la validación lanzando una excepción o devolviendo una promesa rechazada:

```
app.post('/formulario', [
  check('nombre').isLength({ min: 3 }), check('correo electrónico').personalizado(correo electrónico => { if (ya tiene correo electrónico(correo electrónico)) {
    lanzar un nuevo error ('Correo electrónico ya registrado')
  } } ),
  check('edad').isNumeric() ], (req, res) => { const nombre = req.body.name const email = req.body.email const edad = req.body.age })
```

El validador personalizado:

```
check('email').custom(email => { if
  (yaHaveEmail(email)) {
    lanzar un nuevo error ('Correo electrónico ya registrado')

  } })
```

se puede reescribir como

```
check('email').custom(email => { if
  (yaHaveEmail(email)) {
    return Promise.reject('Email ya registrado')

  } })
```

Insumo desinfectante

Ha visto cómo validar la entrada que proviene del mundo exterior a su aplicación Express.

Hay una cosa que aprende rápidamente cuando ejecuta un servidor público: nunca confíe en la entrada.

Incluso si desinfecta y se asegura de que las personas no puedan ingresar cosas raras usando el código del lado del cliente, aún estará sujeto a que las personas usen herramientas (incluso solo las herramientas de desarrollo del navegador) para enviar POST directamente a sus puntos finales.

O bots que intentan todas las combinaciones posibles de exploits conocidas por los humanos.

Lo que debe hacer es desinfectar su entrada.

El [paquete express-validator](#) que ya usa para validar la entrada también puede usarse convenientemente para realizar la desinfección.

Supongamos que tiene un punto final POST que acepta los parámetros de nombre, correo electrónico y edad:

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

aplicación.use(express.json())

app.post('/formulario', (req, res) => { const
  nombre = req.body.name const email =
  req.body.email const edad = req.body.age })
```

Puede validarlo usando:

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

aplicación.use(express.json())

app.post('/formulario', [
  check('nombre').isLength({ min: 3 }),
  check('correo electrónico').isEmail(),
  check('edad').isNumeric() ], (req, res) => { const
  nombre = req.body.name const email =
  req.body.email const edad = req.body.age })
```

Puede agregar desinfección canalizando los métodos de desinfección después de los de validación:

```
app.post('/formulario', [
  check('nombre').isLength({ min: 3 }).trim().escape(),
  cheque('correo').isEmail().normalizeEmail(),
  check('edad').isNumeric().trim().escape()
], (requerido, res) => {
  //...
})
```

Aquí usé los métodos:

- `trim()` recorta caracteres (espacios en blanco por defecto) al principio y al final de un cuerda
- `escape()` reemplaza a `<`, `>`, `&`, `'`, `"` y `/` con sus correspondientes entidades HTML
- `normalizeEmail()` canonicaliza una dirección de correo electrónico. Acepta varias opciones a minúsculas direcciones de correo electrónico o subdirecciones (por ejemplo `flavio+newsletters@gmail.com`)

Otros métodos de desinfección:

- `blacklist()` elimina los caracteres que aparecen en la lista negra
- `whitelist()` elimina los caracteres que no aparecen en la lista blanca
- `unescape()` reemplaza las entidades codificadas en HTML con `< ltrim()`, `>`, `&`, `'`, `"` y `/`
- como `trim()`, pero solo recorta los caracteres al comienzo de la cadena
- `rtrim()` como `trim()`, pero solo recorta los caracteres al final de la cadena
- `stripLow()` elimina los caracteres de control ASCII, que normalmente son invisibles

Forzar la conversión a un formato:

- `toBoolean()` convierte la cadena de entrada en un valor booleano. Todo excepto `'0'`, `'falso'` y `"` devuelve verdadero. En modo estricto, solo `'1'` y `'true'` devuelven verdadero
- `toDate()` convierte la cadena de entrada en una fecha, o nulo si la entrada no es una fecha
- `toFloat()` convierte la cadena de entrada en un flotante, o NaN si la entrada no es un flotante
- `toInt()` convierte la cadena de entrada en un número entero, o NaN si la entrada no es un número entero

Al igual que con los validadores personalizados, puede crear un desinfectante personalizado.

En la función de devolución de llamada, simplemente devuelve el valor desinfectado:

```
const sanitizeValue = valor => {
  //desinfectar...
}

app.post('/formulario', [
  check('valor').customSanitizer(valor => {
    devolver sanitizeValue(valor)
  }),
], (requerido, res) => {
  valor constante = req.cuerpo.valor
})
```


Manejo de formularios

Cómo procesar formularios usando Express

Este es un ejemplo de un formulario HTML:

```
<método de formulario="POST" action="/enviar-formulario">
  < tipo de entrada="texto" nombre="nombre de
    usuario" /> < tipo de entrada="enviar" />
</formulario>
```

Cuando el usuario presione el botón de enviar, el navegador realizará automáticamente una solicitud POST a la URL / submit-form en el mismo origen de la página, enviando los datos que contiene, codificados como application/x-www-form-urlencoded . En este caso, los datos del formulario contienen el

valor del campo de entrada de nombre de usuario .

Los formularios también pueden enviar datos mediante el método GET , pero la gran mayoría de los formularios que build usará POST .

Los datos del formulario se enviarán en el cuerpo de la solicitud POST.

Para extraerlo, utilizará el middleware `express.urlencoded()` , proporcionado por Express:

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

app.use(express.urlencoded())
```

Ahora necesita crear un punto final POST en la ruta /submit-form , y cualquier información estará disponible en `Request.body` :

```
app.post('/enviar-formulario', (req, res) => { const
  nombre de usuario = req.body.username
  //...
  res.end()
})
```

No olvide validar los datos antes de usarlos, usando `express-validator` .

Subida de archivos en formularios

Cómo administrar el almacenamiento y manejo de archivos cargados a través de formularios, en Express

Este es un ejemplo de un formulario HTML que permite a un usuario cargar un archivo:

```
<método de formulario="POST" action="/enviar-formulario">
  < tipo de entrada="archivo" nombre="documento" /
  > < tipo de entrada="enviar" />
</formulario>
```

Cuando el usuario presione el botón enviar, el navegador realizará automáticamente una solicitud POST a la URL /enviar-formulario en el mismo origen de la página, enviando los datos que contiene, no codificados como aplicación/x-www-form-urlencoded como una forma normal, pero como multipart/form-data .

Del lado del servidor, el manejo de datos de varias partes puede ser complicado y propenso a errores, por lo que vamos a utilizar una biblioteca de utilidades llamada **formidable**. [Aquí está el repositorio de GitHub](#), tiene mas de 4000 estrellas y bien mantenido

Puedes instalarlo usando:

```
npm instalar formidable
```

Luego, en su archivo Node.js, inclúyalo:

```
const express = require('express') const app
= express() const formidable =
require('formidable')
```

Ahora, en el extremo POST de la ruta /submit-form , instanciamos un nuevo formulario Formidable usando formidable.IncomingFrom() :

```
app.post('/enviar-formulario', (req, res) => {
  nuevo formidable.IncomingFrom() })
```

Después de hacerlo, necesitamos analizar el formulario. Podemos hacerlo sincrónicamente proporcionando una devolución de llamada, lo que significa que todos los archivos se procesan y, una vez que se hace formidable, los hace disponible:

```
app.post('/enviar-formulario', (req, res) => {
  nuevo formidable.IncomingFrom().parse(req, (err, campos, archivos) => {
    si (err) {
```

```

    consola.error('Error', err)
    tirar error

  } consola.log('Campos', campos)
  consola.log('Archivos', archivos)
  archivos.map(archivo =>
    { consola.log(archivo) }) }) })

```

O puede usar eventos en lugar de una devolución de llamada, para recibir una notificación cuando se analiza cada archivo y otros eventos, como finalizar el procesamiento, recibir un campo que no es de archivo o se produjo un error:

```

app.post('/enviar-formulario', (req, res) => {
  nuevo formidable.IncomingFrom().parse(req) .on('field',
    (name, field) => {
      console.log('Campo', nombre,
        campo) }) .on('archivo', (nombre, archivo) =>
    { console.log ('Archivo cargado ', nombre,
      archivo) }) .on('abortado' , () => {

      console.error('Solicitud cancelada por el usuario') }) .on('error',
    (err) => {

      consola.error('Error', err) throw err

    }) .on('fin', () =>
    { res.fin() })

  })

```

Independientemente de la forma que elija, obtendrá uno o más objetos `Formidable.File`, que le brindan información sobre el archivo cargado. Estos son algunos de los métodos a los que puede llamar:

- tamaño del archivo , el tamaño del archivo en
- archivo file.path bytes en la ruta de la que se escribe este
- Nombre del archivo ,
- el tipo MIME del archivo file.type ,

La ruta predeterminada es la carpeta temporal y se puede modificar si escucha el `archivoBegin` evento:

```

app.post('/enviar-formulario', (req, res) => {
  nuevo formidable.IncomingFrom().parse(req) .on('fileBegin',
    (nombre, archivo) => {
      form.on('fileBegin', (nombre, archivo) => {
        archivo.ruta = __dirname + '/cargas/' + archivo.nombre })
    })

```

```
    }) .on('archivo', (nombre, archivo) =>
      { console.log ('Archivo cargado ', nombre, archivo) })

    //...
  })
```

Un servidor Express HTTPS con un certificado autofirmado

Cómo crear un certificado HTTPS autofirmado para Node.js para probar aplicaciones localmente

Para poder servir un sitio en HTTPS desde localhost, debe crear una cuenta autofirmada certificado.

Un certificado autofirmado será suficiente para establecer una conexión HTTPS segura, aunque los navegadores se quejarán de que el certificado está autofirmado y, como tal, no es de confianza. Es genial para propósitos de desarrollo.

Para crear el certificado debe tener **OpenSSL** instalado en su sistema.

Es posible que ya lo tenga instalado, solo pruebe escribiendo openssl en su terminal.

Si no, en una Mac puede instalarlo usando `brew install openssl` si usa [Homebrew](#). De lo contrario, busque en Google "cómo instalar openssl en".

Una vez que OpenSSL esté instalado, ejecute este comando:

```
openssl req -nodes -new -x509 -keyout server.key -out servidor.cert
```

Le hará algunas preguntas. El primero es el nombre del país:

```
Generación de una clave privada RSA de 1024 bits
```

```
.....+++++
```

```
.....+++++
```

```
escribiendo una nueva clave privada en 'server.key'
```

```
-----
```

Se le pedirá que ingrese información que se incorporará a su solicitud de certificado.

Lo que está a punto de ingresar es lo que se llama un nombre distinguido o un DN.

Hay bastantes campos, pero puede dejar algunos en blanco. Para algunos campos habrá un valor predeterminado. Si ingresa '.', el campo se dejará en blanco.

```
-----
```

Nombre del país (código de 2 letras) [AU]:

Entonces su estado o provincia:

```
Estado o Provincia Nombre (nombre completo) [Algún-Estado]:
```

tu ciudad:

Nombre de la localidad (p. ej., ciudad) []:

y el nombre de su organización:

Nombre de la organización (p. ej., empresa) [Internet Widgits Pty Ltd]:

Nombre de la unidad organizativa (p. ej., sección) []:

Puede dejar todos estos vacíos.

Solo recuerda configurar esto en localhost :

Nombre común (por ejemplo, servidor FQDN o SU nombre) []: localhost

y para agregar su dirección de correo electrónico:

Dirección de correo electrónico []:

¡Eso es todo! Ahora tiene 2 archivos en la carpeta donde ejecutó este comando:

- server.cert es el archivo de certificado autofirmado
- server.key es la clave privada del certificado

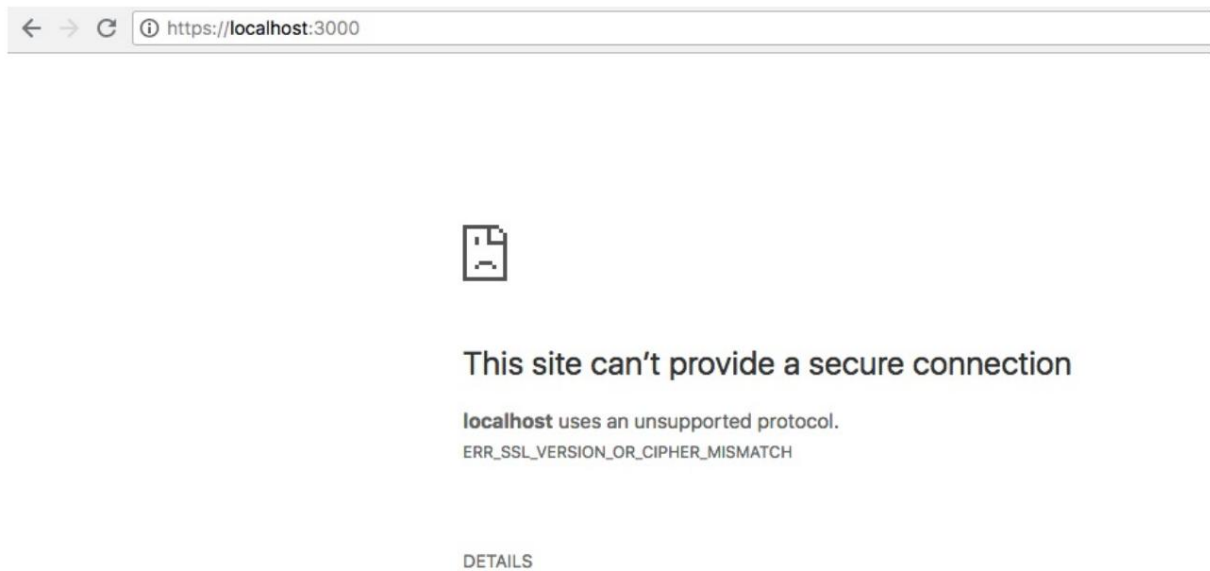
Se necesitarán ambos archivos para establecer la conexión HTTPS y, dependiendo de cómo vaya a configurar su servidor, el proceso para usarlos será diferente.

Esos archivos deben colocarse en un lugar accesible para la aplicación, luego debe configurar el servidor para usarlos.

Este es un ejemplo usando el módulo central https y Express:

```
const https = require('https') const app =  
express()  
  
app.get('/', (req, res) => { res.send('¡Hola  
  HTTPS!')  
})  
  
https.createServer({}, aplicación).escuchar(3000, () => {  
  console.log('Escuchando...') })
```

sin agregar el certificado, si me conecto a https://localhost:3000 esto es lo que hace el navegador
Mostrará:



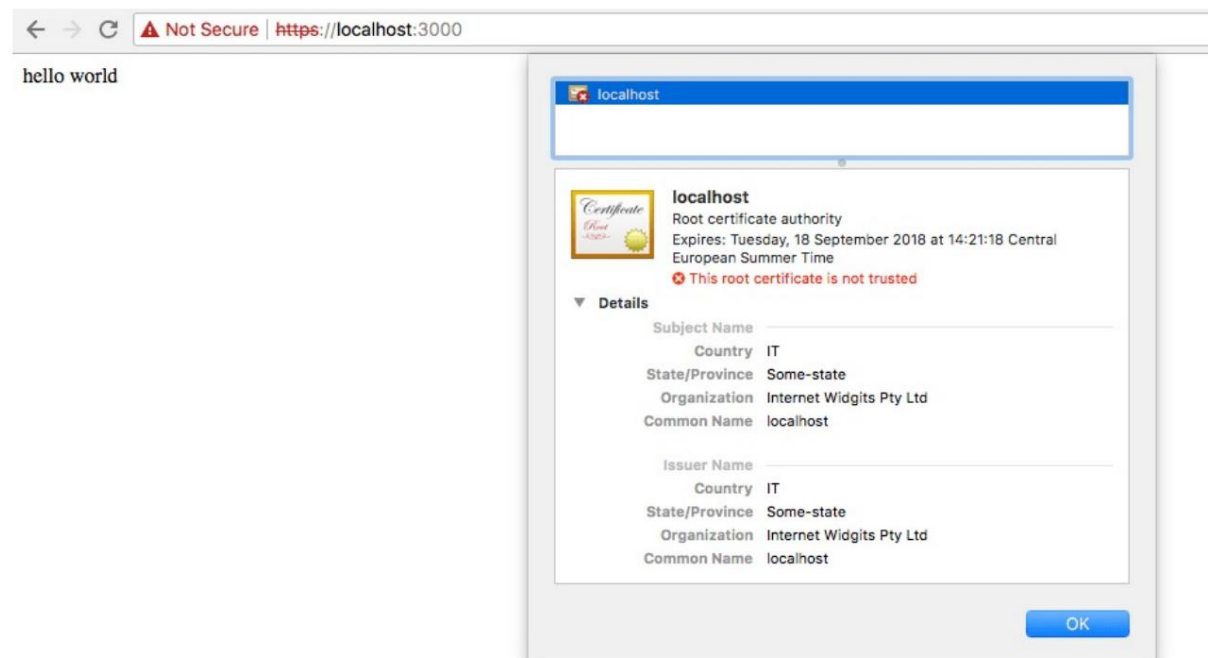
Con el certificado en su lugar:

```
const fs = require('fs')

//...

https.createServer({ clave:
  fs.readFileSync('servidor.clave'), certificado:
  fs.readFileSync('servidor.cert')
}, aplicación).escuchar(3000, () => {
  console.log('Escuchando...') })
```

Chrome nos dirá que el certificado no es válido, ya que está autofirmado, y nos pedirá confirmación para continuar, pero la conexión HTTPS funcionará:



Configurar Let's Encrypt para Express

Cómo configurar HTTPS usando la popular solución gratuita Let's Encrypt

Si ejecuta una aplicación Node.js en su propio VPS, debe administrar la obtención de un SSL certificado.

Hoy en día, el estándar para hacer esto es usar [Let's Encrypt](#) y [Certbot](#), una herramienta de [EFF](#), también conocida como Electronic Frontier Foundation, la principal organización sin fines de lucro centrada en la privacidad, la libertad de expresión y, en general, las libertades civiles en el mundo digital.

Estos son los pasos que seguiremos:

- [Instalar Certbot](#)
- [Genere el certificado SSL usando Certbot](#)
- [Permitir que Express sirva archivos estáticos](#)
- [Confirmar el dominio](#)
- [Obtener el certificado](#)
- [Configurar la renovación](#)

Instalar Certbot

Esas instrucciones asumen que está usando Ubuntu, Debian o cualquier otra distribución de Linux que usa apt-get :

```
sudo add-apt repository ppa:certbot/certbot sudo apt-get  
update sudo apt-get install certbot
```

También puede instalar Certbot en una Mac para probar:

```
brew install certbot
```

pero deberá vincularlo a un nombre de dominio real para que sea útil.

Genere el certificado SSL usando Certbot

Ahora que Certbot está instalado, puede invocarlo para generar el certificado. Debe ejecutar esto como raíz:


```
certbot certonly --manual
```

o llame a sudo

```
sudo certbot certonly --manual
```

El instalador te preguntará el dominio de tu sitio web.

Este es el proceso en detalle.

me pide el correo

```
¿ sudo certbot certonly --manual
Contraseña: XXXXXXXXXXXXXXXXXX
Guardando el registro de depuración en /var/log/letsencrypt/letsencrypt.log
Complementos seleccionados: Manual del autenticador, Instalador Ninguno
Ingrese la dirección de correo electrónico (utilizada para la renovación urgente y los avisos de seguridad)
(Ingrese 'c' para cancelar): flavio@flaviocopes.com
```

Pide aceptar los ToS:

Lea los Términos de servicio en
<https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf>. Debe aceptar para registrarse
en el servidor ACME en <https://acme-v02.api.letsencrypt.org/directory>

(A)aceptar/(C)cancelar: A

Pide compartir la dirección de correo electrónico.

¿Estaría dispuesto a compartir su dirección de correo electrónico con Electronic Frontier Foundation,
socio fundador del proyecto Let's Encrypt y la organización sin fines de lucro que desarrolla Certbot? Nos
gustaría enviarle un correo electrónico sobre nuestro trabajo encriptando la web, noticias de EFF,
campañas y formas de apoyar la libertad digital.

.....

(Y)es/(N)o: Y

Y finalmente podemos ingresar el dominio donde queremos usar el certificado SSL:

Ingrese su(s) nombre(s) de dominio (separados por comas y/o espacios) (Ingrese 'c' para cancelar):
copesflavio.com

Pregunta si está bien registrar su IP:

Obtención de un nuevo certificado
Realizando los siguientes retos:

reto http-01 para copesflavio.com

NOTA: La IP de esta máquina se registrará públicamente como si hubiera solicitado este certificado. Si está ejecutando certbot en modo manual en una máquina que no es su servidor, asegúrese de estar de acuerdo con eso.

¿Estás de acuerdo con que se registre tu IP?

(Y)es/(N)o: y

¡Y finalmente llegamos a la fase de verificación!

Cree un archivo que contenga solo estos datos:

TS_oZ2-ji23jrj3j2irj3iroj_U51u1o0x7rrDY2E.1DzOo_voCOsrpddP_2kpoek2opeko2pke-UAPb21sW1c

Y haz que esté disponible en tu servidor web en esta URL:

`http://copesflavio.com/.well-known/acme-challenge/TS_oZ2-ji23jrj3j2irj3iroj_U51u1o0x7rrDY2E`

Ahora dejemos a Certbot solo por un par de minutos.

Necesitamos verificar que somos dueños del dominio creando un archivo llamado TS_oZ2-ji23jrj3j2irj3iroj_U51u1o0x7rrDY2E en la carpeta .well-known/acme-challenge/ . ¡Presta atención!

La cadena extraña que acabo de pegar cambia cada vez.

Deberá crear la carpeta y el archivo, ya que no existen de forma predeterminada.

En este archivo necesitas poner el contenido que imprimió Certbot:

TS_oZ2-ji23jrj3j2irj3iroj_U51u1o0x7rrDY2E.1DzOo_voCOsrpddP_2kpoek2opeko2pke-UAPb21sW1c

En cuanto al nombre del archivo, esta cadena es única cada vez que ejecuta Certbot.

Permitir que Express sirva archivos estáticos

Para servir ese archivo desde Express, debe habilitar el servicio de archivos estáticos. Puede crear una carpeta estática y agregar allí la subcarpeta .well-known , luego configurar Express de esta manera:

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

//...
```

```
app.use(express.static(__dirname + '/static', { dotfiles: 'allow' } ))

//...
```

La opción `dotfiles` es obligatoria , de lo contrario, el punto conocido no se , que es un archivo de puntos ya que comienza con un `.` hará visible. Esta es una medida de seguridad, porque los dotfiles pueden contener información confidencial y es mejor conservarlos de manera predeterminada.

Confirmar el dominio

Ahora ejecute la aplicación y asegúrese de que se pueda acceder al archivo desde la Internet pública, y vuelva a Certbot, que aún se está ejecutando, y presione ENTER para continuar con el script.

Obtener el certificado

¡Eso es todo! Si todo salió bien, Certbot creó el certificado y la clave privada y los puso a disposición en una carpeta en su computadora (y le dirá qué carpeta, por supuesto).

Ahora copie/pegue las rutas en su aplicación, para comenzar a usarlas para atender sus solicitudes:

```
const fs = require('fs') const
https = require('https') const app =
express()

app.get('/', (req, res) => { res.send('¡Hola
  HTTPS!')
})

https.createServer({
  clave: fs.readFileSync('/etc/letsencrypt/ruta/a/clave.pem'), cert: fs.readFileSync('/
etc/letsencrypt/ruta/a/cert.pem'), ca: fs.readFileSync( '/etc/letsencrypt/ruta/a/
cadena.pem')
}, aplicación).escuchar(443, () => {
  console.log('Escuchando...') })
```

Tenga en cuenta que hice que este servidor escuche en el puerto 443, por lo que debe ejecutarlo con permisos de root.

Además, el servidor se ejecuta exclusivamente en HTTPS, porque usé `https.createServer()` . También puede ejecutar un servidor HTTP junto con esto, ejecutando:

```
http.createServer(aplicación).listen(80, () => {
  console.log('Escuchando...') })

https.createServer({
```

```
clave: fs.readFileSync('/etc/letsencrypt/ruta/a/clave.pem'), cert:
fs.readFileSync('/etc/letsencrypt/ruta/a/cert.pem'), ca: fs.readFileSync('/etc/
letsencrypt/ruta/a/cadena.pem')
}, aplicación).escuchar(443, () => {
  console.log('Escuchando...') })
```

Configurar la renovación

El certificado SSL no será válido durante 90 días. Debe configurar un sistema automatizado para renovarlo.

¿Cómo? Usando un trabajo cron.

Un trabajo cron es una forma de ejecutar tareas en cada intervalo de tiempo. Puede ser cada semana, cada minuto, cada mes.

En nuestro caso, ejecutaremos el script de renovación dos veces al día, como se recomienda en Certbot documentación.

Primero averigüe la ruta absoluta de certbot en su sistema. Uso `type certbot` en macOS para obtenerlo, y en mi caso es `/usr/local/bin/certbot`.

Aquí está el script que necesitamos ejecutar:

```
certbot renovar
```

Esta es la entrada del trabajo cron:

```
0 */12 * * * raíz /usr/local/bin/certbot renovar >/dev/null 2>&1
```

Significa ejecutarlo cada 12 horas, todos los días: a las 00:00 y a las 12:00.

Consejo: generé esta línea usando <https://crontab-generator.org/>

Agregue este script a su crontab, usando el comando:

```
env EDITOR=pico crontab -e
```

Esto abre el pico editor (puedes elegir el que prefieras). Ingrese la línea, guarda y se instala el trabajo cron.

Una vez hecho esto, puede ver la lista de trabajos cron activos usando

```
crontab-l
```

