Actividad 12: Creación y exploración de aplicaciones web con Express

Objetivo: El objetivo de esta actividad es que los estudiantes se familiaricen con la creación de aplicaciones web utilizando el framework Express para Node.js. Al final de la actividad, los estudiantes habrán instalado y configurado una aplicación Express básica, utilizado plantillas para renderizar contenido dinámico, creado rutas estáticas y dinámicas, manejado solicitudes y respuestas HTTP, integrado middleware, y utilizado herramientas de depuración.

Observación: Utiliza la lectura 7 de los preliminares para utilizar el manejo de HTTP y API REST.

Construcción de aplicaciones web con Express

Express es el framework web más popular para JavaScript y ha sido el estándar de facto durante muchos años. Es un framework muy minimalista, fácil de aprender y que proporciona mucha flexibilidad para construir aplicaciones web.
En este lectura, comenzaremos con la aplicación más básica de "hello world" para construir una aplicación REST API sólida y bien probada. Exploraremos en detalle todos los componentes críticos de Express, incluidos solicitud y respuesta, middleware y enrutamiento. También aprenderemos a usar el middleware más común de Express y a construir tu propio middleware.

Familiarizándose con la biblioteca Express

Express se define en su propio sitio web (https://expressjs.com/) de la siguiente manera:

Fast, unopinionated, minimalist web framework for Node.js

Entonces, la buena noticia es que tenemos mucha libertad para construir nuestra aplicación. La mala noticia es que debemos tomar muchas decisiones, y debemos tener cuidado de no cometer errores. Express es muy minimalista en comparación con otros frameworks web, por lo que tenemos que agregar bibliotecas de terceros o construir nuestras propias abstracciones cuando sea necesario. Express tiene una comunidad muy activa, por lo que podemos encontrar muchas bibliotecas para resolver problemas comunes.

Además, la documentación oficial es de gran calidad y hay muchos recursos para aprender más sobre Express, lo que convierte a Express en una gran opción para principiantes.
Como Express es un framework sin opiniones, cuando sigas un tutorial o un curso, descubrirás que a veces el código no es consistente y no sigue los mismos patrones. Esto se debe a que tienes mucha libertad en Express, y con el tiempo desarrollarás tus propios patrones y tu propia forma de construir aplicaciones que mejor se adapten a ti.

En esta lectura, usaremos Express versión 4.18.3, pero cualquier versión de Express 4.x también debería estar bien. Usaremos Node.js versión 20.11.0. Ambas son las últimas versiones disponibles en el momento de escribir esto.

Instalando Express (puedes obviar esta sección)

Para instalar Express, debemos ejecutar el siguiente comando en un nuevo proyecto de Node.js:

npm install express@4

No necesitas ninguna configuración adicional, solo instálalo y estarás listo para comenzar.

Comencemos con un ejemplo simple, una aplicación Hello World. Crea un nuevo archivo llamado helloWorld.js y agrega el siguiente código:

```
import express from 'express'
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World desde Express!')
})
app.listen(port, () => {
  console.log( App Hello World escuchando desde el puerto ${port})
})
```

Muy simple. Vamos a desglosarlo:

1. Importamos la biblioteca Express y creamos una instancia de la aplicación Express.
2. Definimos una ruta para la ruta raíz / y enviamos una respuesta con el texto ¡Hello worlds desde Express!.
3. Iniciamos el servidor y escuchamos en el puerto 3000.

Para ejecutar la aplicación, utilizamos el siguiente comando:

node helloWorld.js

Analiza la salida. Ahora, si abres tu navegador y vas a http://localhost:3000, deberías ver el texto ¡Hello world desde Express!.

Usando el generador

Express tiene una herramienta de línea de comandos para generar una aplicación básica. Para usarla, debemos ejecutar el siguiente comando:

```
npx express-generator@4
```

Esto generará una nueva aplicación con muchos archivos y carpetas. Te recomiendo que crees una nueva carpeta y ejecutes el comando allí, para que no desordenes tu proyecto actual.

La salida de la ejecución debería ser algo como lo siguiente:

```
  create : public/
    create : public/javascripts/
    create : public/images/
    create : public/stylesheets/
```


```
   create : public/stylesheets/style.css
   create : routes/
   create : routes/index.js
   create : routes/users.js
   create : views/
   create : views/error.jade
   create : views/index.jade
   create : views/layout.jade
   create : app.js
   create : package.json
   create : bin/
   create : bin/www
   install dependencies:
     $ npm install
   run the app:
     $ DEBUG=generated:* npm start
```

Luego, el siguiente paso es instalar las dependencias:

```
npm install
```

Finalmente, podemos iniciar la aplicación:

```
npm start
```

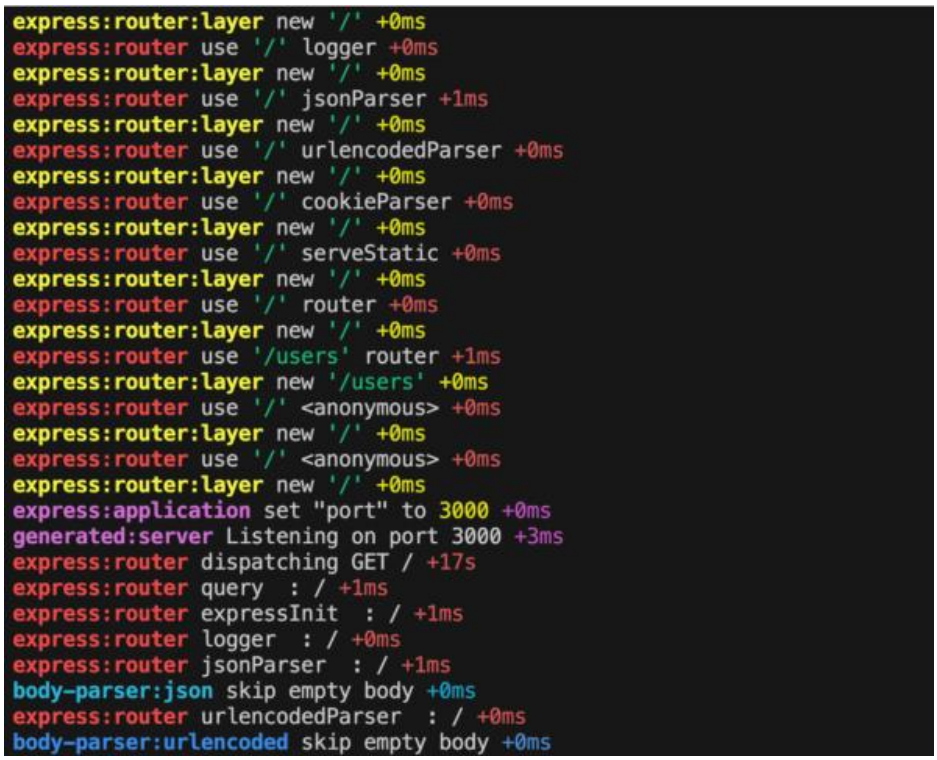
Si todo está bien, deberías ver la siguiente salida:

> generated@0.0.0 start > node ./bin/www
GET / 304 125.395 ms - -
GET /stylesheets/style.css 304 1.265 ms - -
GET / 304 11.043 ms - -
GET /stylesheets/style.css 304 0.396 ms - -
GET /ws 404 11.822 ms - 1322

Si accedes a http://localhost:3000 en el navegador, deberías ver la aplicación Express generada usando el express-generator
Siéntete libre de explorar el código generado, ten en cuenta que también la ruta http://localhost:3000/users está funcionando y si intentas cualquier otra ruta, obtendrás un error 404, como por ejemplo con http://localhost:3000/invented.

Depuración

Ahora, déjame mostrarte otra cosa interesante que incluye el generador de Express y que usaremos en un proyecto más adelante. Si inicias la aplicación con el comando DEBUG=* npm start o estableces DEBUG=* && npm start (si usas Windows) en la terminal, la salida será más detallada y verás mucha información sobre las solicitudes y respuestas:



Esto se debe a que Express y muchas otras dependencias utilizan la biblioteca debug (https://www.npmjs.com/package/debug) para registrar información.
Al usar la variable de entorno DEBUG=*, le estamos diciendo a la biblioteca debug que imprima la información relacionada con todos los espacios de nombres. Pero podemos ser más selectivos y limitar el alcance para Express, por ejemplo, utilizando la variable de entorno DEBUG=express:* npm start.

Entendiendo los motores de plantillas

En la lectura 7 de los preliminares aprendimos la diferencia entre aplicaciones de una sola página (SPAs) y el renderizado en el servidor.
Express proporciona una forma de renderizar páginas HTML utilizando motores de plantillas. Esta es la característica clave para construir aplicaciones renderizadas en el servidor con Express.

Elegir un motor de plantillas

Lo primero que debemos hacer es elegir un motor de plantillas. Hay muchas opciones disponibles. La opción más popular históricamente fue Jade (https://www.npmjs.com/package/jade), pero se renombró a Pug (https://www.npmjs.com/package/pug). Puedes encontrar muchos tutoriales y ejemplos buscando con ambos nombres.

Puedes elegie Embedded JavaScript templating (ejs) (https://www.npmjs.com/package/ejs) por su simplicidad y porque está bien documentado. Con el tiempo, te familiarizarás más con los motores de plantillas y podrás elegir el que mejor se adapte a tus necesidades.

Renderizando una plantilla

Entonces, volviendo a nuestro ejemplo de hola mundo, vamos a crear un nuevo archivo llamado helloWorldTemplate.js y agregar el siguiente código:

```
import express from 'express'
const app = express()
const port = 3000
app.set('view engine', 'ejs')
app.get('/', (req, res) => {
  res.render('index', {
    title: 'Este un app de Express',
    subtitle: 'usando EJS como plantilla'})
})
app.listen(port, () => {
  console.log(App corriendo en http://localhost:${port})
})
```

Ahora, tenemos que crear una nueva carpeta llamada views, dentro de la cual crearemos un nuevo archivo llamado index.ejs con el siguiente contenido:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><%= title %></title>
</head>
<body>
  <h1><%= title %></h1>
  <h2><%= subtitle %></h2>
</body>
</html>
```

Como puedes ver, el motor de plantillas utiliza las etiquetas <%= y %> para interpolar los valores. En este caso, estamos pasando la variable title a la plantilla.
Finalmente, tenemos que instalar la dependencia ejs:

```
npm install ejs@3
```

Luego, iniciamos la aplicación de la siguiente manera:

```
node helloWorldTemplate.js
```

Si accedes a http://localhost:3000 en el navegador, que es lo que ves?
Además, si accedes a view-source:http://localhost:3000/, verás el HTML sin procesar que Express está enviando al navegador:


```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Este es un app de Express/title>
</head>
<body>
 <h1>Este es un app de Express</h1>
  <h2>usando EJS como plantilla</h2>
</body>
</html>
```

Como puedes ver, el motor de plantillas está interpolando los valores y generando el HTML para nosotros.

Entendiendo el proceso

Ahora, entendamos lo que realmente está sucediendo en el código:

```
app.set('view engine', 'ejs')
```

La línea anterior le dice a Express que vamos a usar ejs como nuestro motor de plantillas, por lo que ahora podemos usar res.render para renderizar la plantilla.

```
res.render('index', {
   title: 'Este es un app de Express',
   subtitle: 'usando EJS como plantilla'
})
```

En el código anterior, el método res.render recibe dos parámetros. El primero es el nombre de la plantilla, en este caso, index (views/index.ejs), y el segundo es los datos que queremos interpolar en la plantilla.

Luego, el motor de plantillas reemplazará las etiquetas <%= title %> y <%= subtitle %> con los valores que estamos pasando en el segundo parámetro del método res.render.

En aplicaciones del mundo real, los datos que pasamos a la plantilla serán dinámicos. Por ejemplo, los datos que obtenemos de una base de datos o de una API externa. Pero por ahora, vamos a usar datos estáticos para mantener el ejemplo simple.

Dominando las solicitudes

En la misma lectura 7, aprendimos toda la teoría sobre las solicitudes y respuestas HTTP. Aquí, cubriremos cómo manejarlas con Express.
En esta sección, nos enfocaremos en este fragmento de código:

```
import express from 'express'
```


```
const app = express()
app.method(route, handler)
```

Aquí tenemos tres elementos para entender:

- method, que es el método HTTP que queremos manejar, por ejemplo, GET, POST, PUT, DELETE, y así sucesivamente.
- route, que es la ruta que queremos manejar, por ejemplo, /, /users, o /users/:id.
- handler, que es la función de callback que se ejecutará cuando method y route coincidan.

Métodos HTTP

Express proporciona un método para cada método HTTP. Hay muchos por ahí (get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search, and connect). Los más comunes son get, post, put, y delete, por lo que nos centraremos en ellos:

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

Si deseas gestionar todos los métodos HTTP en la misma ruta, puedes usar el método all:

```
app.all('/', () => {})
```

Rutas

Las rutas son muy flexibles y pueden ser dinámicas. Podemos definirlas de diferentes maneras, incluidas las expresiones regulares.

Rutas estáticas

Las rutas estáticas son la forma más común de definir rutas. Se utilizan para manejar solicitudes a una ruta específica, por ejemplo, /, /users, o /user/me:

```
app.get('/', () => {})
app.get('/users', () => {})
app.get('/user/me', () => {})
```

Parámetros dinámicos

Los parámetros dinámicos se utilizan para manejar las solicitudes a una ruta específica. Podemos usar el carácter : para definir un parámetro dinámico, como /users/:id o /users/:id/profile:

```
app.get('/users/:id', () => {})
```

En este caso, :id es un parámetro dinámico, por lo que coincidirá con cualquier valor, incluidas /users/1, /users/peter, /users/jane-doe, y así sucesivamente.
Incluso puedes combinar parámetros estáticos y dinámicos, como /users/:id/profile:

```
app.get('/users/:id/profile', () => {})
```

El ejemplo anterior resolverá solicitudes a /users/1/profile, /users/kapu/profile, /users/kapumota/profile, y así sucesivamente.

Este patrón es bastante común en aplicaciones de transporte, por ejemplo, donde puedes tener una ruta como /users/:id/rides/:rideId para obtener los detalles de un viaje específico, o cuando reservas boletos para un vuelo utilizando una ruta como /flights/from/:originCity/to/:destinationCity. Express proporcionará los valores de los parámetros dinámicos en el objeto req.params:

```
app.get('/users/:id', (req, res) => {
    const { originCity, destinationCity } = req.params
    res.send(` Tu estan volando desde ${originCity} a ${destinationCity} `)
})
```

Parámetros opcionales

Los parámetros opcionales se utilizan para manejar las solicitudes a una ruta específica, pero el parámetro es opcional. Esto se puede hacer con el carácter ?, como /invoice/:id?:

```
 app.get('/invoice/:id?', (req, res) => {
    const id = parseInt(req.params.id)
    if (id) {
        res.send(` Estas buscando la factura con id ${id} `)
    } else {
        res.send(` Estás buscando todas las facturas `)
    }
})
```

En este caso, el parámetro :id es opcional, por lo que coincidirá con /invoice, /invoice/167, /invoice/G123S8123SD123MJ, y así sucesivamente.

Expresiones regulares

Podemos utilizar expresiones regulares para definir rutas. Esto es muy útil cuando queremos hacer coincidir la ruta utilizando patrones predecibles; por ejemplo, /.*fly$/ identificará cualquier texto que termine en fly:

```
app.get(/.*fly$/, (req, res) => {
  res.send(`Combina con cualquier ruta que finalice con volar `)
})
```

La ruta anterior coincidirá con /butterfly, /dragonfly, /fly, /mcfly, y así sucesivamente. Creemos un ejemplo menos exótico:

```
 app.get('/msg/:id/:action(edit|delete)', (req, res, next) => {
  res.send(`Tu solicitas la accion ${req.params.action} para el mensaje
${req.params.id}`);
});
```

En este caso, la ruta coincidirá con /msg/1/edit, /msg/1/delete, y así sucesivamente.

Nota: Si no estás familiarizado con las expresiones regulares, no te preocupes: puedes usar las otras opciones para definir tus rutas. Pero si quieres explorar más a fondo las expresiones regulares, te recomiendo que pruebes Regular Expressions 101 (https://regex101.com/).

Parámetros de consulta

Hemos aprendimos sobre las diferentes partes de la URL y vimos que los parámetros de consulta son esas partes que comienzan con ?. Estos se utilizan para enviar información adicional al servidor. Por ejemplo, en la URL /films?category=scifi&director=George+Lucas estamos enviando dos parámetros de consulta, category y director.
Podemos capturar los parámetros de consulta en el objeto req.query para usarlos en nuestras rutas:

```
app.get('/films', (req, res) => {
    const { category, director } = req.query
    res.send(`Estas buscando una pelicula por categoria ${category} y director
${director}`)
})
```

Es importante tener en cuenta que los parámetros de consulta son opcionales, lo que significa que las solicitudes pueden no tener ningún parámetro de consulta en absoluto. En ese caso, el objeto req.query estaría vacío.

Nota: Los fragmentos de URL (es decir, /mypath#fragment) no forman parte de las solicitudes y no serán incluidos como tales por el navegador, por lo que no podemos capturarlos. Consulta https://github.com/expressjs/express/issues/1083 para más información.

La importancia del orden

Las rutas se registran en el orden en que las defines, lo que permite a Express evitar conflictos entre rutas. Veamos un ejemplo:

```
app.get('/users/:id', () => {
    res.send(`Estás buscando el usuario con id ${req.params.id} `)
})
app.get('/users/me', () => {
    res.send(`Estás buscando el usuario actual `)
})
```

Si intentas acceder a /users/me, recibirás el mensaje `Estás buscando el usuario con id` porque la ruta /users/:id se registró primero, por lo que coincidirá con /users/me y el valor me se almacenará en la propiedad req.params.id.

Puedes resolver este problema cambiando el orden de las rutas:

```
app.get('/users/me', () => {})
```

```
app.get('/users/:id',
() => {})
```

En proyectos grandes, esto puede convertirse en un problema si no tienes una buena estrategia para definir las rutas. Esta es también una buena razón para incluir pruebas automatizadas en tu proyecto para evitar configuraciones accidentales de rutas.

Manejadores

Los manejadores (handlers) son las funciones que se ejecutan cuando una solicitud coincide con una ruta. Mientras que un manejador es una función simple con tres parámetros (req, res, y next), tiene la gran responsabilidad de manejar la respuesta a la solicitud o delegar la solicitud a otros manejadores:

```
app.get('/', (req, res, next) => {
    res.send("Hello World")
})
```

Veamos los parámetros del manejador con más detalle.

request

El objeto de solicitud (req) contiene toda la información sobre la solicitud, incluidos los parámetros, IP, encabezados, cuerpo, y así sucesivamente. Si utilizas otras bibliotecas que amplían las capacidades de Express, es muy probable que encuentres más propiedades en este objeto.
Puedes encontrar más información sobre el objeto de solicitud en la documentación de Express (https://expressjs.com/en/4x/api.html#req).

response

El objeto de respuesta (res) contiene todos los métodos para manejar la respuesta de la solicitud, incluidos métodos simples como send o json hasta métodos más complejos como download o redirect.

next

La función next (next) se utiliza para delegar la solicitud al siguiente manejador. Esto es útil cuando quieres dividir la lógica del manejador en múltiples funciones o delegar la gestión de errores.

Dominando las respuestas

Las respuestas son la forma en que el servidor se comunica de vuelta con el cliente después de una solicitud, por lo que es crucial entender cómo manejarlas. En esta sección, aprenderemos sobre cómo agregar encabezados, códigos de estado, redirecciones, envío de datos y envío de archivos.

Descubrirás los métodos disponibles cuando comiences a construir aplicaciones más complejas.
Puedes encontrar más información sobre el objeto de respuesta en la documentación de Express (https://expressjs.com/en/4x/api.html#res).

Gestión de encabezados

Los encabezados (headers) se utilizan para enviar información adicional sobre la respuesta. Express maneja los encabezados utilizando el método set, que recibe dos parámetros, el nombre del encabezado y el valor del encabezado:

```
app.get('/', (req, res, next) => {
    res.set('Content-Type', 'text/html')
    res.send("<h1>Hello World</h1>")
})
```

En el ejemplo anterior, estamos configurando el encabezado Content-Type a text/html para que el navegador sepa que la respuesta es un documento HTML y lo renderice como HTML.

Múltiples encabezados

También puedes usar el método set para establecer múltiples encabezados al mismo tiempo pasando un objeto como primer parámetro:

```
 app.get('/', (req, res, next) => {
    res.set({
        'Content-Type': 'text/html',
        'x-powered-by': 'Unicornio y arco iris'
    })
    res.send("<h1>Hello World</h1>")
})
```

En el código anterior, estamos configurando dos encabezados, Content-Type y x-powered-by.

Eliminando encabezados

Puedes eliminar un encabezado utilizando el método removeHeader, que recibe el nombre del encabezado como primer parámetro:

```
 app.get('/', (req, res, next) => {
    res.set({
        'Content-Type': 'text/html',
        'x-powered-by': 'Unicornios y arco iris'
    })
    res.removeHeader('x-powered-by')
    res.send("<h1>Hello World</h1>")
})
```

En el ejemplo anterior, estamos eliminando el encabezado x-powered-by que acabamos de agregar en la declaración anterior.

Códigos de estado

Un código de estado es un número que representa el estado de la respuesta. Se utiliza para comunicar el estado de la solicitud al cliente. Es importante utilizar el código de estado correcto, ya que es parte del protocolo HTTP.
Puedes manejar los códigos de estado utilizando el método status, que recibe el código de estado como primer parámetro:

```
app.get('/', (req, res, next) => {
    res.status(200)
res.send("<h1>Hello World</h1>")
})
```

En el ejemplo anterior, estamos configurando el código de estado a 200, lo que significa que la solicitud fue exitosa. Por defecto, Express configurará el código de estado a 200 si no lo configuras.

Encadenando métodos

Puedes encadenar el método status con otros métodos, como set o send:

```
 app.get('/', (req, res, next) => {
    res.status(200).set('Content-Type', 'text/html').send("<h1>Hello World</h1>")
})
```

Enviando solo códigos de estado
Si solo deseas enviar el código de estado, puedes usar el método sendStatus, que recibe el código de estado como primer parámetro:

```
app.get('/', (req, res, next) => {
    res.sendStatus(500)
})
```

En el ejemplo anterior, estamos enviando el código de estado 500, lo que significa que la solicitud no fue exitosa.

Redirecciones
Puedes redirigir la solicitud a otra URL utilizando el método redirect, que recibe la URL como primer parámetro:

```
 app.get('/', (req, res, next) => {
    res.redirect('https://example.com/')
})
```

En el ejemplo anterior, estamos redirigiendo la solicitud a https://example.com.
El código de estado predeterminado para las redirecciones es 302, pero puedes cambiarlo especificando el código de estado como primer parámetro:

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


```
    res.redirect(301, 'https://example.com/')
})
```

El método redirect también acepta URLs relativas, por lo que puedes redirigir a otra ruta en tu aplicación:

```
app.get('/', (req, res, next) => {
    res.redirect('/about')
})
```

Incluso puedes redirigir a un nivel superior en la URL:

```
app.get('/about/me', (req, res, next) => {
    res.redirect('..')
})
```

En este caso, la solicitud será redirigida a /about, similar a cuando haces cd.. en la terminal.
También es posible redirigir a la URL del referente utilizando el método back. Si el encabezado referrer no está presente en la solicitud, entonces la solicitud será redirigida a /:

```
app.get('/', (req, res, next) => {
    res.redirect('back')
})
```

Enviando datos

Al comienzo vimos cómo usar res.render para renderizar una plantilla, pero hay otras formas de enviar datos al cliente. La forma más común de enviar datos es utilizando el método send, que recibe los datos como un parámetro. Esto puede ser cualquier tipo de datos, incluidos buffers:

```
app.get('/', (req, res, next) => {
    res.send("Hello World")
})
```

Usando res.send()

El método send convertirá los datos en una cadena y configurará el encabezado Content-Type en text/html a menos que especifiques lo contrario utilizando res.set(). También incluirá Content-Length.
Si utilizas buffers, el encabezado Content-Type se configurará en application/octet-stream y Content-Length se configurará en la longitud del buffer, pero puedes cambiar esto utilizando res.set():

```
app.get('/', (req, res, next) => {
    res.set('Content-Type', 'text/html')
    res.send(Buffer.from('<p>Hello World</p>'))
```

```
})
```

Usando res.json()

Si deseas enviar datos JSON, puedes utilizar el método json directamente, que recibe los datos como primer parámetro. Configurará el encabezado Content-Type en application/json y realizará la serialización de los datos por ti:

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

Esta es la forma más común de enviar datos JSON, pero también puedes usar el método send y configurar el encabezado Content-Type en application/json, realizando tú mismo la serialización de los datos:

```
app.get('/', (req, res, next) => {
    res.set('Content-Type', 'application/json')
    res.send(JSON.stringify({message: 'Hello World'}))
})
```

Esto puede ser muy útil si deseas utilizar una biblioteca de stringificación diferente, como fast-json-stringify (https://www.npmjs.com/package/fast-json-stringify).

Enviando archivos

Puedes enviar archivos al cliente utilizando el método sendFile, que recibe la ruta al archivo como primer parámetro:

```
app.get('/report', (req, res, next) => {
    res.sendFile('/path/to/file.txt')
})
```

En el ejemplo anterior, estamos enviando el archivo /path/to/file.txt al cliente. Este método permite una gran flexibilidad, incluido un callback para gestionar posibles errores. Consulta la documentación (http://expressjs.com/en/4x/api.html#res.sendFile) para obtener más información.

Otra forma de enviar archivos es utilizando el método res.download(), que recibe la ruta al archivo como primer parámetro:

```
app.get('/report', (req, res, next) => {
    res.attachment('/path/to/file.txt')
})
```

Este método configurará el encabezado Content-Disposition en attachment y el encabezado Content-Type en application/octet-stream a menos que especifiques lo contrario utilizando res.set(). Este método permite una gran flexibilidad, incluido un callback para gestionar posibles errores.

Puedes consultar la documentación (http://expressjs.com/en/4x/api.html#res.download) para obtener más información.

Ejercicios teóricos

Conceptos básicos de Express:

Pregunta: ¿Qué es Express en el contexto de Node.js y cuáles son sus principales ventajas al desarrollar aplicaciones web?

Generación de aplicaciones con express-generator:

Pregunta: Explica qué es `express-generator` y cuáles son los beneficios de utilizar esta herramienta al iniciar un nuevo proyecto con Express.

Motores de plantillas en Express:

Pregunta: ¿Qué es un motor de plantillas en Express y por qué es importante? Menciona al menos dos motores de plantillas populares compatibles con Express.

Manejo de solicitudes y respuestas HTTP:

Pregunta: Describe los componentes clave del manejo de solicitudes y respuestas HTTP en Express, incluyendo los objetos `req`, `res` y la función `next`.

Importancia del orden en las rutas:

Pregunta: ¿Por qué es crucial el orden en que se definen las rutas en una aplicación Express? Proporciona un ejemplo de cómo el orden puede afectar el comportamiento de la aplicación.

1. Creación de una aplicación Hello World con Express

Objetivo: Familiarizarse con la instalación y configuración básica de Express creando una aplicación simple que responda con "Hello World desde Express!".

Instrucciones:

a. Inicializar un nuevo proyecto: Crea una nueva carpeta para tu proyecto y ejecuta npm init para inicializar un archivo package.json.
b. Instalar Express: Ejecuta el comando de instalación de Express utilizando npm install express@4.
c. Crear el archivo principal: Crea un archivo llamado helloWorld.js y configura una aplicación Express que escuche en el puerto 3000 y responda con el mensaje especificado cuando se acceda a la ruta raíz /.
d. Ejecutar la aplicación: Inicia la aplicación utilizando node helloWorld.js y verifica que al acceder a http://localhost:3000 en el navegador se muestre el mensaje esperado.

2. Generación de una aplicación con express-generator

Objetivo: Utilizar `express-generator` para crear una estructura de aplicación Express y explorar los archivos y carpetas generados.

Instrucciones:

a. Instalar express-generator: Asegúrate de tener instalado express-generator ejecutando npx express-generator@4.
b. Generar la aplicación: Dentro de una nueva carpeta, ejecuta el comando para generar la estructura básica de la aplicación.
c. Instalar dependencias: Navega a la carpeta generada y ejecuta npm install para instalar las dependencias necesarias.
d. Iniciar la aplicación: Inicia la aplicación utilizando npm start y accede a http://localhost:3000 para ver la aplicación generada. Explora las diferentes rutas y archivos generados, como routes/index.js y views/index.jade (o views/index.ejs si se ha configurado).

3. Configuración y uso de un motor de plantillas (EJS)

Objetivo: Configurar un motor de plantillas EJS en una aplicación Express y renderizar una página dinámica con datos interpolados.

Instrucciones:

a. Instalar EJS: Ejecuta npm install ejs@3 para agregar EJS como dependencia.
b. Configurar el motor de plantillas: En tu archivo principal de Express (por ejemplo, helloWorldTemplate.js), configura EJS como el motor de plantillas utilizando app.set('view engine', 'ejs').
c. Crear una vista EJS: Dentro de la carpeta views, crea un archivo index.ejs que contenga etiquetas para interpolar variables como <%= title %> y <%= subtitle %>.
d. Renderizar la vista: Define una ruta en Express que renderice index.ejs pasando un objeto con title y subtitle.
e. Ejecutar y verificar: Inicia la aplicación y accede a http://localhost:3000 para ver la página renderizada con los valores interpolados.

4. Definición de rutas estáticas y dinámicas en express

Objetivo: Practicar la creación de rutas estáticas y dinámicas en una aplicación Express, incluyendo el manejo de parámetros opcionales y expresiones regulares.

Instrucciones:

a. Crear rutas estáticas: Define rutas como `/`, `/users`, y `/about` que respondan con mensajes o rendericen vistas específicas.

b. Implementar rutas dinámicas: Crea rutas que incluyan parámetros dinámicos, por ejemplo, `/users/:id`, y maneja las solicitudes extrayendo los parámetros desde `req.params`.
c. Añadir parámetros opcionales: Implementa una ruta con parámetros opcionales, como `/invoice/:id?`, y maneja casos donde el parámetro puede estar presente o ausente.
d. Utilizar expresiones regulares en rutas: Define una ruta que utilice una expresión regular para coincidir con patrones específicos, por ejemplo, rutas que terminen con `fly`.
e. Verificar el orden de las rutas: Experimenta cambiando el orden de las rutas definidas y observa cómo afecta a la coincidencia de rutas específicas y dinámicas.

5. Manejo de respuestas HTTP en Express

Objetivo: Comprender y practicar el manejo de diferentes tipos de respuestas HTTP, incluyendo la configuración de encabezados, códigos de estado, redirecciones y el envío de datos o archivos.

Instrucciones:

a. Configurar encabezados: Define rutas que configuren diferentes encabezados en las respuestas utilizando res.set().
b. Establecer códigos de estado: Implementa respuestas que utilicen diferentes códigos de estado HTTP, como 200 OK, 404 Not Found, y 500 Internal Server Error, utilizando res.status() o res.sendStatus().
c. Realizar redirecciones: Crea rutas que redirijan a otras URLs, ya sean externas (por ejemplo, https://example.com) o internas (otras rutas dentro de la aplicación), utilizando res.redirect().
d. Enviar datos en fiferentes formatos: Practica el envío de respuestas en formato de texto, JSON y archivos utilizando métodos como res.send(), res.json() y res.sendFile().
e.Probar las respuestas: Accede a las diferentes rutas definidas y utiliza herramientas como el navegador o curl para verificar que las respuestas se manejan correctamente según lo configurado.

6. Implementación de middleware para manejo de errores

Objetivo: Aprender a crear e integrar middleware personalizado para manejar errores en una aplicación Express.

Instrucciones:

a. Definir middleware de error: Crea una función middleware que capture errores y envíe una respuesta adecuada al cliente.
b. Integrar middleware en la aplicación: Asegúrate de que el middleware de error esté definido después de todas las rutas para que pueda capturar cualquier error que ocurra durante el manejo de las solicitudes.
c. Generar errores intencionalmente: Define rutas que generen errores intencionales para probar el funcionamiento del middleware de error.
d. Verificar la respuesta de errores: Accede a las rutas que generan errores y confirma que el middleware maneja y responde correctamente a los errores.

7. Uso de la herramienta debug para depuración en Express

Objetivo: Utilizar la herramienta debug para obtener información detallada sobre las solicitudes y respuestas en una aplicación Express.

Instrucciones:

a. Iniciar la aplicación con DEBUG: Ejecuta la aplicación Express utilizando la variable de entorno DEBUG=* o una específica como DEBUG=express:* para habilitar la salida de depuración.
b. Analizar la información de depuración: Observa la información detallada que se muestra en la terminal sobre las solicitudes entrantes, respuestas enviadas y otros eventos internos de Express.
c. Filtrar información de depuración: Experimenta utilizando diferentes patrones en la variable DEBUG para filtrar la información de depuración relevante.
d. Aplicar depuración en casos de errores: Utiliza la herramienta debug para identificar y resolver problemas en la aplicación mediante el análisis de la información proporcionada.

8. Envío de archivos estáticos y descargas en Express

Objetivo: Implementar rutas que sirvan archivos estáticos y permitan a los usuarios descargar archivos específicos desde la aplicación Express.

Instrucciones:

a. Servir archivos estáticos: Configura Express para servir archivos estáticos desde una carpeta específica utilizando express.static().
b. Enviar archivos específicos: Define rutas que utilicen res.sendFile() para enviar archivos específicos al cliente.
c. Implementar descargas de archivos: Crea rutas que permitan a los usuarios descargar archivos utilizando res.download(), configurando encabezados apropiados para la descarga.
d. Verificar el envío y descarga de archivos: Accede a las rutas definidas y confirma que los archivos se sirven correctamente y que las descargas funcionan como se espera.

9. Configuración de middleware para procesamiento de datos de solicitud

Objetivo: Integrar y utilizar middleware en Express para procesar datos enviados en las solicitudes, como cuerpos JSON o formularios.

Instrucciones:

a. Instalar middleware necesario: Añade y configura middleware como express.json() y express.urlencoded() para manejar datos JSON y formularios.
b. Definir rutas que reciban datos: Crea rutas que reciban datos enviados en el cuerpo de la solicitud y procesa estos datos dentro de los manejadores de rutas.
c. Validar y responder a los datos recibidos: Implementa lógica para validar los datos recibidos y envía respuestas adecuadas al cliente basadas en la validación.

d. Probar el envío de datos: Utiliza herramientas como Postman o curl para enviar solicitudes con diferentes tipos de datos y verifica que la aplicación los maneje correctamente.