BackEnd - Node.JS

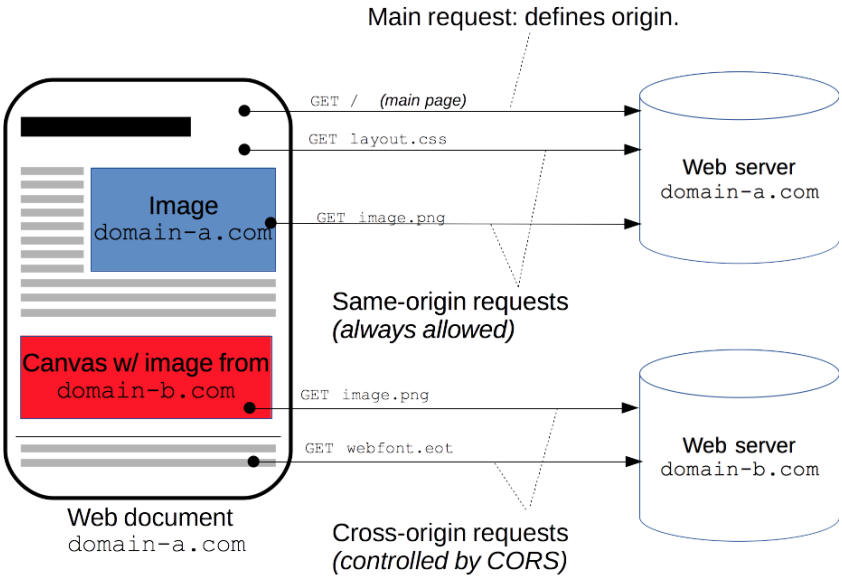
1. Ajustando el puerto de escucha de Node.Js en forma automática con process.enc.PORT
2. Haciendo Routes para definir diferentes carpetas para cada servicio.
3. Implementando un Crud con persistencia de datos.
4. Repasando las sentencias SQL a utilizar.
5. Creando el Front End para el Crud
6. Conectando el Front y el Back con JavaScript y Json
7. Creando un web service para:
8. Nuevo dato
9. Modificar un dato
10. Listar datos
11. Eliminar un dato
12. Buscar un dato

Control de acceso HTTP (CORS)

El Intercambio de Recursos de Origen Cruzado (CORS (en-US)) es un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un user agent (en-US) obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio) al que pertenece. Un agente crea una petición HTTP de origen cruzado cuando solicita un recurso desde un dominio distinto, un protocolo o un puerto diferente al del documento que lo generó.

Un ejemplo de solicitud de origen cruzado: el código JavaScript frontend de una aplicación web que es localizada en `http://domain-a.com` utiliza XMLHttpRequest para cargar el recurso `http://api.domain-b.com/data.json`.

Por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script. Por ejemplo, XMLHttpRequest y la API Fetch siguen la política de mismo-origen. Ésto significa que una aplicación que utilice esas APIs XMLHttpRequest sólo puede hacer solicitudes HTTP a su propio dominio, a menos que se utilicen cabeceras CORS.



El W3C Grupo de Trabajo de Aplicaciones Web recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, por sus siglas en inglés). CORS da controles de acceso a dominios cruzados para servidores web y transferencia segura de datos en dominios cruzados entre navegadores y servidores Web. Los exploradores modernos utilizan CORS en un contenedor API (como XMLHttpRequest o Fetch) para ayudar a mitigar los riesgos de solicitudes HTTP de origen cruzado.

¿Quién debería leer este artículo?

Todo el mundo, de verdad.

Más específicamente, este artículo está dirigido a administradores web, desarrolladores de servidores y desarrolladores de interfaz. Los exploradores modernos manejan los componentes

sobre el intercambio de origen cruzado del lado del cliente. Incluyendo cabeceras y políticas de ejecución. Pero, este nuevo estándar determina que los servidores tienen que manejar las nuevas solicitudes y las cabeceras de las respuestas. Se recomienda, como lectura suplementaria, otro artículo para desarrolladores de servidor que discute el intercambio de origen cruzado desde una perspectiva de servidor (con fragmentos de código PHP).

¿Qué peticiones utiliza CORS?

Este estándar de intercambio de origen cruzado es utilizado para habilitar solicitudes HTTP de sitios cruzados para:

- Invocaciones de las APIs XMLHttpRequest o Fetch en una manera de sitio cruzado, como se discutió arriba.
- Fuentes Web (para usos de fuente en dominios cruzados @font-face dentro de CSS), para que los servidores puedan mostrar fuentes TrueType que sólo puedan ser cargadas por sitios cruzados y usadas por sitios web que lo tengan permitido.
- Texturas WebGL.
- Imágenes dibujadas en patrones usando drawImage.
- Hojas de estilo (para acceso CSSOM).
- Scripts (para excepciones inmutadas).

Este artículo es una discusión general sobre Intercambio de Recursos de Origin Cruzado e incluye una discusión sobre las cabeceras HTTP.

Resumen

El estándar de Intercambio de Recursos de Origen Cruzado trabaja añadiendo nuevas cabeceras HTTP que permiten a los servidores describir el conjunto de orígenes que tienen permiso para leer la información usando un explorador web. Adicionalmente, para métodos de solicitud HTTP que causan efectos secundarios en datos del usuario (y en particular, para otros métodos HTTP distintos a GET, o para la utilización de POST con algunos tipos MIME), la especificación sugiere que los exploradores "verifiquen" la solicitud, solicitando métodos soportados desde el servidor con un método de solicitud HTTP OPTIONS, y luego, con la "aprobación" del servidor, enviar la verdadera solicitud con el método de solicitud HTTP verdadero. Los servidores pueden también notificar a los clientes cuando sus "credenciales" (incluyendo Cookies y datos de autenticación HTTP) deben ser enviados con solicitudes.

Las secciones siguientes discuten escenarios, así como el análisis de las cabeceras HTTP usados.

Ejemplos de escenarios de control de accesos

Aquí, presentamos tres escenarios que ilustran cómo funciona el Intercambio de Recursos de Origen Cruzado. Todos estos ejemplos utilizan el objeto XMLHttpRequest, que puede ser utilizado para hacer invocaciones de sitios cruzados en cualquier explorador soportado.

Los fragmentos de JavaScript incluidos en estas secciones (y las instancias ejecutadas del código servidor que correctamente maneja las solicitudes de sitios cruzados) pueden ser encontrados "en acción" aquí, y pueden ser trabajados en exploradores que soportan XMLHttpRequest de sitios cruzados. Una discusión de Intercambio de Recursos de Origen Cruzado desde una perspectiva de servidor (incluyendo fragmentos de código PHP) puede ser encontrada aquí.

Solicitudes simples

Una solicitud de sitio cruzado es aquella que cumple las siguientes condiciones:

- Los únicos métodos aceptados son:
- GET
- HEAD
- POST.
- Aparte de las cabeceras establecidas automáticamente por el agente usuario (ej. Connection, User-Agent,etc.), las únicas cabeceras que están permitidas para establecer manualmente son:
- Accept
- Accept-Language
- Content-Language
- Content-Type
- Los únicos valores permitidos de la cabecera Content-Type son:
- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

Nota: Estos son los mismos tipos de solicitud de sitios cruzados que un contenido web ya puede emitir, y ninguna respuesta de datos es liberada a menos que el servidor envíe
la cabecera apropiada. Por lo tanto, los sitios que prevengan solicitudes falsas de sitios cruzados no tienen nada nuevo que temer sobre el control de acceso HTTP.

Por ejemplo, suponga que el contenido web en el dominio http://foo.example desea invocar contenido en el dominio http://bar.other. Código de este tipo puede ser utilizado dentro de JavaScript desplegado en foo.example:

```
var invocation = new XMLHttpRequest();
```

```
var url = 'http://bar.other/resources/public-data/';



function callOtherDomain() {

  if(invocation) {

    invocation.open('GET', url, true);

    invocation.onreadystatechange = handler;

    invocation.send();

  }

}
```

Dejándonos ver lo que el explorador enviará al servidor en este caso, y veamos como responde el servidor:

```
GET /resources/public-data/ HTTP/1.1

Host: bar.other

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Connection: keep-alive

Referer: http://foo.example/examples/access-control/simpleXSInvocation.html
```

```
Origin: http://foo.example




HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 00:23:53 GMT

Server: Apache/2.0.61

Access-Control-Allow-Origin: *

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: application/xml



[XML Data]
```

Las líneas 1 - 10 son las cabeceras enviadas por Firefox 3.5. Observe que la cabecera de solicitud HTTP principal aquí es la cabecera `Origin:` en la línea 10 de arriba, lo que muestra que la invocación proviene del contenido en el dominio `http://foo.example`.

Las líneas 13 - 22 muestran la respuesta HTTP del servidor en el dominio `http://bar.other`. En respuesta, el servidor envía de vuelta una cabecera `Access-Control-Allow-Origin:`, mostrado arriba en la línea 16. El uso de la cabecera `Origin:` y `Access-Control-Allow-Origin:` muestran el protocolo de control de acceso en su uso más simple. En este caso, el servidor responde con un `Access-Control-Allow-Origin: *` lo que significa que el recurso puede ser accedido por cualquier dominio en una forma de sitio cruzado. Si el dueño del recurso en `http://bar.other` deseara restringir el acceso al recurso solamente para `http://foo.example`, debería devolver lo siguiente:

```
Access-Control-Allow-Origin: http://foo.example
```

Note que ahora, ningún otro dominio aparte de http://foo.example (identificado por la cabecera ORIGIN: en la solicitud, como en la línea 10 arriba) puede acceder al recurso en una forma de sitio cruzado. La cabecera Access-Control-Allow-Origin debe contener el valor que fue enviado en la solicitud del encabezado Origin.

Solicitudes Verificadas

A diferencia de las solicitudes simples (discutidas arriba), las solicitudes "verificadas" envían primero una solicitud HTTP por el método OPTIONS al recurso en el otro dominio, para determinar si es seguro enviar la verdadera solicitud. Las solicitudes de sitios cruzados son verificadas así ya que pueden tener implicaciones en la información de usuario. En particular, una solicitud es verificada sí:

- Usa métodos distintos a GET, HEAD o POST. También, si POST es utilizado para enviar solicitudes de información con Content-Type distinto a application/x-www-form-urlencoded, multipart/form-data, o text/plain, ej. si la solicitud POST envía una carga XML al servidor utilizando application/xml or text/xml, entonces la solicitud es verificada.
- Se establecen encabezados personalizados (ej. la solicitud usa un encabezado como X-PINGOTHER)

Nota: Empezando en Gecko 2.0, las codificaciones de datos text/plain, application/x-www-form-urlencoded, y multipart/form-data pueden ser enviadas en sitios cruzados sin verificación. Anteriormente, solo text/plain podía ser enviado sin verificación.
Un ejemplo de este tipo de invocación puede ser:

```
var invocation = new XMLHttpRequest();

var url = 'http://bar.other/resources/post-here/';

var body = '<?xml version="1.0"?><person><name>Arun</name></person>';



function callOtherDomain(){

  if(invocation)

    {

      invocation.open('POST', url, true);

      invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
```

```
        invocation.setRequestHeader('Content-Type', 'application/xml');

        invocation.onreadystatechange = handler;

        invocation.send(body);

    }

}



.....
```

En el ejemplo de arriba, la línea 3 crea un cuerpo XML para enviar con la solicitud `POST` en la línea 8. También, en la línea 9, se establece una cabecera HTTP de solicitud "personalizado" (no estándar `X-PINGOTHER: pingpong`). Dichas cabeceras no son parte del protocolo HTTP/1.1, pero son útiles generalmente en aplicaciones web. Dado que la solicitud (`POST`) usa un Content-Type `application/xml`, y dado que se establece una cabecera personalizada, la solicitud es verificada.

Veamos este intercambio completo entre un cliente y un servidor:

```
OPTIONS /resources/post-here/ HTTP/1.1

Host: bar.other

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Connection: keep-alive

Origin: http://foo.example
```

```
Access-Control-Request-Method: POST

Access-Control-Request-Headers: X-PINGOTHER



HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:15:39 GMT

Server: Apache/2.0.61 (Unix)

Access-Control-Allow-Origin: http://foo.example

Access-Control-Allow-Methods: POST, GET, OPTIONS

Access-Control-Allow-Headers: X-PINGOTHER

Access-Control-Max-Age: 1728000

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 0

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/plain



POST /resources/post-here/ HTTP/1.1

Host: bar.other
```

```
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Connection: keep-alive

X-PINGOTHER: pingpong

Content-Type: text/xml; charset=UTF-8

Referer: http://foo.example/examples/preflightInvocation.html

Content-Length: 55

Origin: http://foo.example

Pragma: no-cache

Cache-Control: no-cache



<?xml version="1.0"?><person><name>Arun</name></person>




HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:15:40 GMT
```

```
Server: Apache/2.0.61 (Unix)

Access-Control-Allow-Origin: http://foo.example

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 235

Keep-Alive: timeout=2, max=99

Connection: Keep-Alive

Content-Type: text/plain



[Some GZIP'd payload]
```

Las líneas 1 - 12 arriba representan la solicitud verificada con los métodos `OPTIONS`. Firefox 3.1 determina lo que se necesita para enviar esto basándose en los parámetros de la solicitud que los fragmentos de JavaScript que se usaron arriba, para que el servidor pueda responder si es aceptable enviar la solicitud con los parámetros de la solicitud real. OPTIONS es un método HTTP/1.1 que se utiliza para determinar información adicional de los servidores, y es un método idempotente, esto significa que no puede ser utilizado para cambiar el recurso. Observe que, junto con la solicitud OPTIONS, se envían otras dos cabeceras de solicitud (líneas 11 y 12 respectivamente):

```
Access-Control-Request-Method: POST

Access-Control-Request-Headers: X-PINGOTHER
```

La cabecera `Access-Control-Request-Method` notifica al servidor como parte de una solicitud verificada que cuándo se envíe la solicitud real, esta será enviada con un método de solicitud `POST`. La cabecera `Access-Control-Request-Headers` notifica al servidor que cuando la solicitud real sea enviada, será enviada con un encabezado `X-PINGOTHER` personalizado. Ahora, el servidor tiene la oportunidad para determinar si desea aceptar la solicitud bajo estas circunstancias.

Las líneas 15 - 27 de arriba corresponden con la respuesta que devuelve el servidor indicando que el método de la petición (POST) y la cabecera `X-PINGOTHER` son aceptadas. En particular, echemos un vistazo a las líneas 18-21:

```
Access-Control-Allow-Origin: http://foo.example

Access-Control-Allow-Methods: POST, GET, OPTIONS

Access-Control-Allow-Headers: X-PINGOTHER

Access-Control-Max-Age: 1728000
```

El servidor responde con Access-Control-Allow-Methods y dice que POST, GET, y OPTIONS son métodos viables para consultar el recurso en cuestión. Observe que esta cabecera es similar al HTTP/1.1 Allow: encabezado de respuesta, pero usado estrictamente dentro del contexto del control de acceso. El servidor también envía Access-Control-Allow-Headers con un valor de X-PINGOTHER, confirmando que es una cabecera permitida para ser usado en la solicitud real. Como Access-Control-Allow-Methods, Access-Control-Allow-Headers es una lista separada por comas de cabeceras aceptables. Finalmente, Access-Control-Max-Age da el valor en segundos de cuánto tarda la respuesta de la solicitud verificada en ser capturada sin enviar otra solicitud verificada. En este caso, 1728000 segundos son 20 días.

Solicitudes con credenciales

La capacidad más interesante expuesta tanto por XMLHttpRequest y Access Control es la habilidad para hacer solicitudes "con credenciales" que estén al tanto de Cookies HTTP e información de Autenticación HTTP. Por defecto, en las invocaciones XMLHttpRequest de un sitio curzado, los exploradores no enviarán credenciales. Una bandera específica tiene que ser establecida en el objeto XMLHttpRequest cuando este es invocado.

En este ejemplo, el contenido cargado originalmente desde http://foo.example hace una solicitud GET simple a un recurso en http://bar.other que establece Cookies. El contenido en foo.example puede contener un JavaScript como este:

```
var invocation = new XMLHttpRequest();

var url = 'http://bar.other/resources/credentialed-content/';



function callOtherDomain(){

  if(invocation) {

    invocation.open('GET', url, true);
```

```
    invocation.withCredentials = true;

    invocation.onreadystatechange = handler;

    invocation.send();

  }

}
```

La línea 7 muestra la bandera en XMLHttpRequest que tiene que ser establecida para poder hacer la invocación con Cookies, es decir, el valor booleano withCredentials. Por defecto, la invocación es hecha sin Cookies. Dado que esta es una simple solicitud GET, no es verificada, pero el explorador rechazará cualquier respuesta que no tiene el encabezado Access-Control-Allow-Credentials: true,y no hará disponible la respuesta para invocar contenido web.

A continuación se proporciona una muestra de intercambio entre un cliente y un servidor:

```
GET /resources/access-control-with-credentials/ HTTP/1.1

Host: bar.other

User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre)
Gecko/20081130 Minefield/3.1b3pre

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Connection: keep-alive

Referer: http://foo.example/examples/credential.html

Origin: http://foo.example

Cookie: pageAccess=2
```

```
HTTP/1.1 200 OK

Date: Mon, 01 Dec 2008 01:34:52 GMT

Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod_ssl/2.0.61 OpenSSL/0.9.7e mod_fastcgi/2.4.2 DAV/2 SVN/1.4.2

X-Powered-By: PHP/5.2.6

Access-Control-Allow-Origin: http://foo.example

Access-Control-Allow-Credentials: true

Cache-Control: no-cache

Pragma: no-cache

Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT

Vary: Accept-Encoding, Origin

Content-Encoding: gzip

Content-Length: 106

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/plain
```

```
[text/plain payload]
```

Pese a que la línea 11 contiene la Cookie destinada para el contenido en `http://bar.other`, si bar.other no responde con `Access-Control-Allow-Credentials: true` (línea 19) la respuesta será ignorada y no puesta a disposición para el contenido web. Nota Importante: cuando se responde a una solicitud con credeciales, el servidor debe especificar un dominio, y no puede usar comodines. El ejemplo de arriba fallará si la cabecera fuera un comodín como: `Access-Control-Allow-Origin: *`. Dado que `Access-Control-Allow-Origin` menciona explícitamente `http://foo.example`, el contenido de credenciales competente es devuelto al contenido web invocador. Observe que, en la línea 22, se establece una cookie adicional.

Todos estos ejemplos pueden verse funcionando aquí. La siguiente sección se refiere a las verdaderas cabeceras HTTP.

Las cabeceras HTTP de respuesta

Esta sección lista las cabeceras HTTP de respuesta que los servidores envían de vuelta para las solicitudes de acceso de control definidas por la especificación del Intercambio de Recursos de Origen Cruzado. La sección anterior da un resumen de estos en acción.

Access-Control-Allow-Origin

Un recurso devuelto puede tener una cabecera `Access-Control-Allow-Origin`, con la siguiente sintaxis:

```
Access-Control-Allow-Origin: <origin> | *
```

El parámetro `origin` específica una URI que puede tener acceso al recurso. El explorador debe asegurar esto. Para solicitudes sin credenciales, el servidor debe especificar "*" como un comodín permitiendo, de este modo, el acceso al recurso a cualquier origen.

Por ejemplo, para permitir a http://mozilla.com acceder al recurso, usted puede especificar:

```
Access-Control-Allow-Origin: http://mozilla.com
```

Si el servidor especifica un host de origen en vez de "*", entonces se debe incluir `Origin` en el encabezado de respuesta `Vary` para indicar a los clientes que las respuestas del servidor difieren basándose en el valor del encabezado de respuesta `Origin`.

Access-Control-Expose-Headers

Requires Gecko 2.0(Firefox 4 / Thunderbird 3.3 / SeaMonkey 2.1)

Esta cabecera permite una whitelist de cabeceras del servidor que los exploradores tienen permitido acceder. Por ejemplo:

```
Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header
```

Esto permite a las cabeceras `X-My-Custom-Header` y `X-Another-Custom-Header` ser expuestos al explorador.

Access-Control-Max-Age

Esta cabecera indica durante cuánto tiempo los resultados de la solicitud verificada pueden ser capturados. Para un ejemplo de solicitudes verificadas, vea los ejemplos de arriba.

```
Access-Control-Max-Age: <delta-seconds>
```

El parámetro `delta-seconds` indica el número de segundos en que los resultados pueden ser capturados.

Access-Control-Allow-Credentials

Indica si la respuesta puede ser expuesta cuando la bandera `credentials` es verdadera. Cuando se usa como parte de una respuesta para una solicitud verficada, este indica si la solicitud verdadera puede realizarse usando credenciales. Note que las solicitudes `GET` simples no son verificadas, y por lo que si una solicitud es hecha para un recurso con credenciales, si la cabecera no es devuelta con el recurso, la respuesta es ignorada por el explorador y no devuelta al contenido web.

```
Access-Control-Allow-Credentials: true | false
```

Las Solicitudes con credenciales son discutidas arriba.

Access-Control-Allow-Methods

Específica el método o los métodos permitidos cuando se asigna un recurso. Es usado en respuesta a la solicitud verificada. Las condiciones sobre cuándo una solicitud es verificada se discuten arriba.

```
Access-Control-Allow-Methods: <method>[, <method>]*
```

Un ejemplo de una solicitud verificada se muestra arriba, incluyendo un ejemplo donde se envía este encabezado al explorador.

Access-Control-Allow-Headers

Usado en respuesta a una solicitud verificada para indicar qué encabezado HTTP puede ser usado cuando se realiza la solicitud real.

```
Access-Control-Allow-Headers: <field-name>[, <field-name>]*
```

Los encabezados HTTP de solicitud

Esta sección lista las cabeceras que los clietnes deben utilizar cuando realizan solicitudes HTTP para usar la característica de intercambio de origen cruzado. Note que estas cabeceras son establecidas cuando se realizan realizan invocaciones a los servidores. Los desarrolladores usan la capacidad de sitios cruzados `XMLHttpRequest` para no tener que establecer ninguna solicitud de intercambio de origen cruzado programada.

Origin

Indica el origen de las solicitudes de acceso a sitios cruzados o solicitudes verificadas.

```
Origin: <origin>
```

El origen es una URI indicando al servidor dónde se ha iniciado la solicitud. Este no incluye ninguna información de recorrido, sólo el nombre del servidor.

Nota: El `origin` puede ser un string vacío; esto es útil, por ejemplo, si el recurso es un `data` URL.

Observe que en cualquier solicitud de acceso de control, la cabecera `ORIGIN` siempre se envía.

Access-Control-Request-Method

Se usa cuando se emite una solicitud verificada, para indicarle al servidor qué método HTTP será usado cuando se haga la solicitud real.

```
Access-Control-Request-Method: <method>
```

Ejemplos de esta utilización pueden ser encontrados arriba.

Access-Control-Request-Headers

Usada cuando se emite una solicitud verificada para indicarle al servidor qué cabecera HTTP será usada cuando se haga la solicitud real.

```
Access-Control-Request-Headers: <field-name>[, <field-name>]*
```

RUTAS / ROUTING CON NODEJS Y EXPRESS

Para añadir más páginas a nuestro sitio, necesitamos más rutas. Podemos hacerlo usando Express Router, que ya viene integrado en Express. Esta entrada será un tanto bestia ya que el manejo de las rutas será una parte importante de todas las aplicaciones que hagamos de ahora en adelante.

Esta es la manera de crear las rutas básicas para sitios web, y también la forma en la que finalmente construiremos nuestras APIs RESTful que utilizará una app frontend en Angular. Así que, sin más dilación procedemos a enrutar (que palabra más fea señores, usaremos routing a partir de ahora).

EXPRESS ROUTER

¿Qué es exactamente Express Router? Puedes considerarlo como una mini aplicación de express sin tantas funcionalidades, solo routing. No incorpora vistas o configuraciones, pero nos proporciona una routing API con funciones como .use(), .get(), .param(), y route(). Echaremos un vistazo al significado de todo esto.

Hay varias formas de usar routing. Ya usamos uno de sus métodos cuando creamos la página de inicio en la anterior entrada usando 'app.get('/', …). Veremos otros métodos para hacer más secciones de nuestro sitio y comentaremos el por qué y cuando usarlos.

EJEMPLO DE LAS CARACTERÍSTICAS DE UNA APLICACIÓN

Estas son las principales características que añadiremos a nuestra aplicación actual:

Rutas básicas (ya hemos tenemos hecha la página principal)
Rutas de otras secciones del sitio (parte del Admin/Administrador con sus sub-rutas)
Middleware para registros de peticiones (log request) en consola
Ruta con Parámetros (http://localhost:1337/usuarios/darthvader)
Ruta Middleware para Parámetros, para validar parámetros específicos
Rutas de inicio de sesión (login routes) haciendo GET y POST
Validar un parámetro que se pasa a una ruta concreta

¿Qué es el Middleware? El Middleware se invoca ente la petición de un usuario y la respuesta final. Es todo lo que ocurre desde que sale una solicitud en lado del cliente hasta que llega a nuestra lógica de la ruta en el servidor. Volveremos sobre este concepto cuando tengamos que registrar los datos de cada petición por la consola/terminal (log data). Un usuario solicitará la página, lo registramos en la consola (middleware) y después enviaremos la respuesta con la página solicitada. Próximamente, más sobre el middleware.

Como hemos hecho hasta ahora, tendremos nuestras rutas en el archivo server.js. No necesitaremos hacer cambio alguno en nuestro package.json puesto que ya viene todo instalado con Express.

RUTAS BÁSICAS / BASIC ROUTES

Ya definimos nuestra ruta básica en la página de inicio. Express nos deja definir las rutas con nuestro objeto app. También podemos manejar métodos HTTP como GET, POST, PUT/PATCH, y DELETE.

Está es la forma más simple de definir rutas, pero a medida que nuestra aplicación crezca, necesitaremos más organización para nuestras rutas. Tan solo imagínate una app que tenga una parte de administrador y otra parte para el usuario, con muchas rutas cada una. Express router nos ayuda a regular todo esto.

Para las rutas siguientes, no enviaremos vistas al navegador, solo mensajes. Así será más sencillo ya que quiero centrarme en los aspectos del routing.

EXPRESS.ROUTER()
express.Router() actúa como una mini aplicación. Puedes crear una instancia (como hicimos con Express) y luego definir rutas con ella. Vamos a ver un ejemplo.

Debajo nuestro app.get() route dentro del server.js, añade lo siguiente. Vamos a 1. llamar una instancia del router 2. aplicarle rutas a esa instancia 3. y luego añadir estas rutas a nuestra app principal.

```
1  //creamos las rutas para la parte de admin
2
3  //instanciamos router
4  var adminRouter = express.Router();
5
6  //página principal del admin, panel de administración/dashboard (http://localhost:133
7  adminRouter.get('/', function(req, res) {
8   res.send('¡Soy el panel de administración!');
9  });
10
11 //users page (http://localhost:1337/admin/users)
12 adminRouter.get('/users', function (req, res) {
13  res.send('¡Muestro todos los usuarios!');
14 });
15
16 //posts page (http://localhost:1337/admin/users)
17 adminRouter.get('/posts', function(req, res) {
18  res.send('¡Muestro todos los posts!');
19 });
20
21 //aplicamos las rutas a nuestra aplicación, app
22 app.use('/admin', adminRouter);
```

//creamos las rutas para la parte de admin

//instanciamos router
var adminRouter = express.Router();

//página principal del admin, panel de administración/dashboard (http://localhost:1337/admin)
adminRouter.get('/', function(req, res) {
 res.send('¡Soy el panel de administración!');
});

//users page (http://localhost:1337/admin/users)
adminRouter.get('/users', function (req, res) {
 res.send('¡Muestro todos los usuarios!');
});

//posts page (http://localhost:1337/admin/users)
adminRouter.get('/posts', function(req, res) {
 res.send('¡Muestro todos los posts!');
});

//aplicamos las rutas a nuestra aplicación, app
app.use('/admin', adminRouter);
Llamaremos a una instancia de express.Router() y lo asignamos a la variable adminRouter, le aplicamos las rutas, y luego le decimos a nuestra app que use esas rutas.

Ahora podemos acceder al panel de administración en http://localhost:1337/admin y a las sub-páginas en http://localhost:1337/admin/users y http://localhost:1337/admin/posts.

Observa como podemos establecer una raíz/root por defecto usando estas rutas que acabamos de definir. Si hubiéramos cambiado la línea 22 por app.use('/app', router), entonces nuestras rutas serían http://localhost:1337/app/ y http://localhost:1337/app/users.

Esto es muy potente porque podemos creas varias express.Router() y luego aplicarlas a nuestra app. Podríamos tener un Router para las rutas básicas, rutas de autentificación, etc.

Usar Router nos permite hacer nuestras aplicaciones más modulares y flexibles que nunca mediante la creación de instancias de Router y aplicándolas como corresponde. Ahora vamos a ver como usar middleware para manejar peticiones.

ROUTE MIDDLEWARE (ROUTER.USE())
Route middleware en Express es una forma de hacer algo antes de que una petición se procese. Este algo podrían ser cosas como comprobar si un usuario se ha autentificado (logueado con su cuenta por ejemplo), en definitiva, cualquier cosa que gustemos hacer antes de mandarle información al usuario.

El middleware registra un mensaje en nuestra consola cada vez que se realiza una petición / solicitud. Haremos una demostración de como crear middleware usando Express Router. Simplemente añadiremos el middleware al adminRouter que creamos en el último ejemplo. Asegúrate de ponerlo después de declarar tu adminRouter y antes de definir las rutas del admin, usuarios y posts.

Te habrás fijado en el argumento 'next'. Es el único modo que tiene Express de saber que la función se ha completado y puede proceder con la siguiente parte del middleware o continuar con el enrutamiento (routing).

```
1 //middleware que nos dirá qué ocurre en cada petición
2 adminRouter.use(function(req, res, next){
3
4  //registra cada petición en la consola
5  console.log(req.method, req.url);
6
7  //continúamos haciendo lo que sea que estábamos haciendo y vamos a la ruta
8  next();
9 });
```

//middleware que nos dirá qué ocurre en cada petición
adminRouter.use(function(req, res, next){

 //registra cada petición en la consola
 console.log(req.method, req.url);

```
 //continúamos haciendo lo que sea que estábamos haciendo y vamos a la ruta
 next();
});
```

adminRouter.use() se utiliza para definir el middleware. Ahora esto se aplicará a todas las peticiones que entran a nuestra aplicación por la instancia de Router. Vamos al navegador y entramos en por ejemplo en http://localhost:1337/admin/users y veremos la petición (request) en nuestra consola.

Express Router petición en consola
Express Router petición en consola

Es muy importante el orden en que coloques tus middleware y las rutas. Todo ocurre en el orden en el que están. Significa que si colocas tus middleware después de una ruta, la ruta procedería antes que el middleware y la petición finalizará ahí mismo. Tu middleware no se ejecutaría en este caso.

Ten en mente que puedes usar el route middleware para muchas cosas. Por ejemplo, puedes usarlo para comprobar si un usuario se ha logueado durante la sesión antes de dejarlo continuar.

ESTRUCTURANDO RUTAS

Al usar Router(), somos capaces de dividir en partes de nuestro sitio. Significa que puedes crear un basicRouter para rutas de navegación básica del sitio. También podrías crear un adminRouter para las rutas como administrador que estarían protegidas por algún tipo de autentificación.

Enrutar nuestra aplicación es un método que nos permite dividir las piezas que la componen. Nos proporciona la flexibilidad que necesitamos para aplicaciones complejas o APIs. También podemos mantener nuestra aplicación limpia y organizada ya que podemos trasladar cada router definido a su propio archivo individual y luego simplemente coger este archivo cuando lo llamamos con app.use(), de esta manera:

```
app.use('/', basicRoutes);
app.use('/admin', adminRoutes);
app.use('/api', apiRoutes);
```

RUTAS CON PARÁMETROS (/HELLO/:NAME)

Vamos a ver como añadir parámetros a las rutas, digamos que queremos tener una ruta llamada /admin/users/:name donde pasamos el nombre de una persona a la URL, y la aplicación lanza un

‘¡Hola :name!’, añade esto debajo de donde definimos la ruta /admin/users:

```
---------------------------------------------------------------------------------------------
//ruta con parámetros (http://localhost:1337/admin/users/:name)
adminRouter.get('/users/:name', function(req, res){
 res.send('hola ' + req.params.name + '!');
});
```

Ahora visitamos por ejemplo http://localhost:1337/admin/users/darthvader y veremos que nuestro navegador nos lanza un ‘hola darth vader!’, req.params guarda todos los datos que provienen de la petición (request) del usuario. Muy sencillo.

Express Router con parámetros

En el futuro, podríamos usar esto para coger todos los datos de usuario que coincidan con el nombre darthvader. Podríamos hacer un panel de administración para gestionar nuestros usuarios.

Pongamos como ejemplo que queremos validar el nombre de alguna forma. Quizás queremos asegurarnos de que no es una palabra ofensiva o políticamente incorrecta. Haríamos esta validación dentro del middleware. Usaremos un middleware especial para esto.

MIDDLEWARE PARA PARÁMETROS (.PARAM())

Usaremos Express .param() middleware. Esto crea un middleware que se ejecutará para cierto parámetro de la ruta. En nuestro caso, estamos usando :name en nuestra ruta de saludo. Una vez más, asegúrate de que el middleware está colocado antes que la ruta:

```
//middleware para validar :name
adminRouter.param('name', function(req, res, next, name){
 //haz aquí la validación de name
 //blah blah blah, validación
 //mostramos en consola para saber si funciona
 console.log('haciendo validaciones de ' + name);

 //una vez hecha la validación guardamos el nuevo objeto en la petición
 req.name = name;
 //pasamos al siguiente asunto
 next();
});

//ruta con parámetros (http://localhost:1337/admin/hello/:name)
adminRouter.get('/hello/:name', function(req, res){
 res.send('hola ' + req.name + '!');
});
```

Ahora cuando vamos a la ruta /hello/:name, nuestro middleware entrará en funcionamiento. Podemos hacer las validaciones y luego pasarle la nueva variable a nuestra ruta .get almacenándola dentro de req (request). Después accedemos a ella cambiando req.params.name por req.name.

Si ponemos en el navegador http://localhost:1337/admin/hello/darthvader veremos nuestra petición mostrarse en la consola.

Express Router Parámetros Middleware

Middleware para parámetros se puede usar para validar datos que se envían a tu aplicación. Si te da por hacer una API RESTful, puedes validar también un token y asegurarte de que el usuario es capaz de acceder a su información. Todo lo que hemos trabajado en Node hasta ahora dará lugar a la API RESTful que comentamos en la primera entrada cuando hablamos del modelo cliente-servidor.

Ahora pasamos a ver la última característica de Express router, cómo usar app.route() para definir varias rutas de una sola vez.

RUTAS LOGIN (APP.ROUTE())

Con esto definimos las rutas de nuestra aplicación de forma similar a usar app.get, pero usando app.route. Es básicamente un shortcut/atajo para llamar al Express Router. En lugar de llamar express.Router(), podemos llamar app.route y empezar a aplicar nuestras rutas ahí.

Usar app.route() nos permite definir varias acciones en una sola ruta de login. Necesitaremos un GET route para mostrar el formulario de login o acceso y un POST route para procesar el formulario.

```
app.route('/login')
 //mostramos el formulario (GET http://localhost:1337/login)
 .get(function(req, res){
   res.send('este es el formulario de login');
 })

  //procesamos el formulario (POST http://localhost:1337/login)
 .post(function(req, res){
   res.send('procesando el formulario de login');
  });
```

Listo, ya hemos definido dos acciones diferentes en nuestra ruta /login. Limpio y sencillo. Esta vez hemos aplicado directamente la ruta a nuestro objeto app dentro del archivo server.js, pero también podemos definirlas en el objeto adminRouter que teníamos antes.

Este es un buen método para crear rutas, ya que es limpio y facilita ver qué rutas hay y dónde se aplican. Pronto estaremos haciendo una API RESTful y una de las cosas que debemos hacer es usar los diferentes verbos de una petición HTTP para las acciones o funciones de nuestra aplicación. GET /login dará lugar al formulario de acceso (login) mientras que POST /login procesará los datos de login.

RESUMEN
Express Router nos da mucha flexibilidad a la hora de definir rutas. En resumen, podemos:

Utilizar express.Router() varias veces para definir grupos de rutas
Aplicar express.Router() a una sección o parte de nuestra web usando app.use()
Usar route middleware para procesar peticiones (requests)
Usar route middleware para validar parámetros usando .param()
Usar app.route() como un atajo de Router para definir varias peticiones en una ruta

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() (en-US) que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como Service Workers (en-US). Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

La especificación fetch difiere de JQuery.ajax() en dos formas principales:

- El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo sera rechazado ante un fallo de red o si algo impidió completar la solicitud.

- Por defecto, fetch no enviará ni recibirá cookies del servidor, resultando en peticiones no autenticadas si el sitio permite mantentener una sesión de usuario (para mandar cookies, credentials de la opción init deberan ser configuradas). Desde el 25 de agosto de 2017. La especificación cambió la politica por defecto de las credenciales a same-origin. Firefox cambió desde la versión 61.0b13.

Una petición básica de fetch es realmente simple de realizar. Eche un vistazo al siguente código:

```
fetch('http://example.com/movies.json')

  .then(response => response.json())

  .then(data => console.log(data));
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo en la consola. El uso de fetch() más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto Promise conteniendo la respuesta, un objeto Response.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método json() (definido en el mixin de Body, el cual está implementado por los objetos Request y Response).

Nota: El mixin de Body tambien tiene metodos parecidos para extraer otros tipos de contenido del cuerpo. Vease Body para más información.
Las peticiones de Fetch son controladas por la directiva de connect-src de Content Security Policy en vez de la directiva de los recursos que se han devuelto.

Suministrando opciones de petición

El método `fetch()` puede aceptar opcionalmente un segundo parámetro, un objeto `init` que permite controlar un numero de diferentes ajustes:

Vea `fetch()` (en-US), para ver todas las opciones disponibles y más detalles.

```
// Ejemplo implementando el metodo POST:

async function postData(url = '', data = {}) {

  // Opciones por defecto estan marcadas con un *

  const response = await fetch(url, {

    method: 'POST', // *GET, POST, PUT, DELETE, etc.

    mode: 'cors', // no-cors, *cors, same-origin

    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached

    credentials: 'same-origin', // include, *same-origin, omit

    headers: {

      'Content-Type': 'application/json'

      // 'Content-Type': 'application/x-www-form-urlencoded',

    },

    redirect: 'follow', // manual, *follow, error

    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url

    body: JSON.stringify(data) // body data type must match "Content-Type" header

  });
```

```
  return response.json(); // parses JSON response into native JavaScript objects
}

postData('https://example.com/answer', { answer: 42 })
  .then(data => {
    console.log(data); // JSON data parsed by `data.json()` call
  });
```

Tenga en cuenta que `mode: "no-cors"` solo permite un conjunto limitado de encabezados en la solicitud:

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Content-Type` with a value of `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`

Comprobando que la petición es satisfactoria

Una petición promise fetch() (en-US) será rechazada con TypeError cuando se encuentre un error de red, aunque esto normalmente significa problemas de permisos o similares — por ejemplo, un 404 no constituye un error de red. Una forma precisa de comprobar que la petición `fetch()` es satisfactoria pasa por comprobar si la promesa ha sido resuelta, además de comprobar que la propiedad Response.ok tiene el valor `true` que indica que el estado de la petición HTTP es OK (código 200-299). El código sería algo así:

```
fetch('flores.jpg').then(function(response) {
  if(response.ok) {
    response.blob().then(function(miBlob) {
      var objectURL = URL.createObjectURL(miBlob);
      miImagen.src = objectURL;
```

```
    });

  } else {

    console.log('Respuesta de red OK pero respuesta HTTP no OK');

  }

})

.catch(function(error) {

  console.log('Hubo un problema con la petición Fetch:' + error.message);

});
```

Proporcionando tu propio objeto Request

En lugar de pasar la ruta al recurso que deseas solicitar a la llamada del método `fetch()`, puedes crear un objeto de petición utilizando el constructor Request() (en-US), y pasarlo como un argumento del método `fetch()`:

```
var myHeaders = new Headers();



var myInit = { method: 'GET',

               headers: myHeaders,

               mode: 'cors',

               cache: 'default' };



var myRequest = new Request('flowers.jpg', myInit);
```

```
fetch(myRequest)

.then(function(response) {

  return response.blob();

})

.then(function(myBlob) {

  var objectURL = URL.createObjectURL(myBlob);

  myImage.src = objectURL;

});
```

Request() acepta exactamente los mismos parámetros que el método fetch(). Puedes incluso pasar un objeto de petición existente para crear una copia del mismo:

```
var anotherRequest = new Request(myRequest, myInit);
```

Esto es muy útil ya que el cuerpo de las solicitudes y respuestas son de un sólo uso. Haciendo una copia como esta te permite utilizar la petición/respuesta de nuevo, y al mismo tiempo, si lo deseas, modificar las opciones de init. La copia debe estar hecha antes de la lectura del <body>, y leyendo el <body> en la copia, se marcará como leido en la petición original.

Nota: Existe también un método clone() (en-US) que crea una copia. Este tiene una semántica ligeramente distinta al otro método de copia — el primero fallará si el cuerpo de la petición anterior ya ha sido leído (lo mismo para copiar una respuesta), mientras que clone() no.

Enviar una petición con credenciales incluido

Para producir que los navegadores envien una petición con las credenciales incluidas, incluso para una llamada de origen cruzado, añadimos credentials: 'include' en el el objeto init que se pasa al método fetch().

```
fetch('https://example.com', {

  credentials: 'include'

})
```

Copy to Clipboard

Si sólo quieres enviar la credenciales si la URL de la petición está en el mismo origen desde donde se llamada el script, añade credentials: 'same-origin'.

```
// El script fué llamado desde el origen 'https://example.com'



fetch('https://example.com', {

  credentials: 'same-origin'

})
```

Sin embargo para asegurarte que el navegador no incluye las credenciales en la petición, usa credentials: 'omit'.

```
fetch('https://example.com', {

  credentials: 'omit'

})
```

Enviando datos JSON

Usa [fetch() (en-US)]() para enviar una petición POST con datos codificados en JSON .

```
var url = 'https://example.com/profile';

var data = {username: 'example'};



fetch(url, {

  method: 'POST', // or 'PUT'

  body: JSON.stringify(data), // data can be `string` or {object}!

  headers:{
```

```
    'Content-Type': 'application/json'

  }

}).then(res => res.json())

.catch(error => console.error('Error:', error))

.then(response => console.log('Success:', response));
```

Enviando un archivo

Los archivos pueden ser subido mediante el HTML de un elemento input `<input type="file" />`, FormData() (en-US) y fetch() (en-US).

```
var formData = new FormData();

var fileField = document.querySelector("input[type='file']");



formData.append('username', 'abc123');

formData.append('avatar', fileField.files[0]);



fetch('https://example.com/profile/avatar', {

  method: 'PUT',

  body: formData

})

.then(response => response.json())

.catch(error => console.error('Error:', error))

.then(response => console.log('Success:', response));
```

Copy to Clipboard

Cabeceras

La interfaz Headers te permite crear tus propios objetos de headers mediante el constructor Headers() (en-US). Un objeto headers es un simple multi-mapa de nombres y valores:

```
var content = "Hello World";

var myHeaders = new Headers();

myHeaders.append("Content-Type", "text/plain");

myHeaders.append("Content-Length", content.length.toString());

myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

Lo mismo se puede lograr pasando un "array de arrays" o un objeto literal al constructor:

```
myHeaders = new Headers({

  "Content-Type": "text/plain",

  "Content-Length": content.length.toString(),

  "X-Custom-Header": "ProcessThisImmediately",

});
```

Los contenidos pueden ser consultados o recuperados:

```
console.log(myHeaders.has("Content-Type")); // true

console.log(myHeaders.has("Set-Cookie")); // false

myHeaders.set("Content-Type", "text/html");

myHeaders.append("X-Custom-Header", "AnotherValue");



console.log(myHeaders.get("Content-Length")); // 11
```

```
console.log(myHeaders.getAll("X-Custom-Header")); // ["ProcessThisImmediately",
"AnotherValue"]



myHeaders.delete("X-Custom-Header");

console.log(myHeaders.getAll("X-Custom-Header")); // [ ]
```

Algunas de estas operaciones solo serán utiles en ServiceWorkers (en-US), pero estas disponen de una mejor API para manipular `headers`.

Todos los métodosde de `headers` lanzan un `TypeError` si un nombre de cabecera no es un nombre de cabecera HTTP válido. Las operaciones de mutación lanzarán un `TypeError` si hay un guarda inmutable (ver más abajo). Si no, fallan silenciosamente. Por ejemplo:

```
var myResponse = Response.error();

try {

  myResponse.headers.set("Origin", "http://mybank.com");

} catch(e) {

  console.log("Cannot pretend to be a bank!");

}
```

Un buen caso de uso para `headers` es comprobar cuando el tipo de contenido es correcto antes de que se procese:

```
fetch(myRequest).then(function(response) {

  var contentType = response.headers.get("content-type");

  if(contentType && contentType.indexOf("application/json") !== -1) {

    return response.json().then(function(json) {

      // process your JSON further
```

```
    });

  } else {

    console.log("Oops, we haven't got JSON!");

  }

});
```

Guarda (Guard)

Desde que las cabeceras pueden ser enviadas en peticiones y recibidas en respuestas, y tienen limitaciones sobre que información puede y debería ser mutable, los objeto headers tienen una propierdad de guarda. Este no está expuesto a la Web, pero puede afectar a que operaciones de mutación son permitidas sobre el objeto headers.

Los valores posibles de guarda (guard) son:

- `none`: valor por defecto.
- `request`: Guarda para el objeto headers obtenido de la petición (Request.headers).
- `request-no-cors`: Guarda para un objeto headers obtenido desde una petición creada con Request.mode (en-US) a `no-cors`.
- `response`: Guarda para una cabecera obetenida desde un respuesta (Response.headers (en-US)).
- `immutable`: Mayormente utilizado para ServiceWorkers, produce un objeto headers de solo lectura.

Nota: No se debería añadir o establecer una petición a un objeto headers guardado con la cabecera `Content-Length`. De igual manera, insertar `Set-Cookie` en la respuesta de la cabecera no esta permitido: ServiceWorkers no estan autorizados a establecer cookies a través de respuestas sintéticas.

Objetos Response

Cómo has visto anteriormente, las instancias de Response son devueltas cuando `fetch()` es resuelto.

Las propiedades de response que usarás son:

- Response.status — Entero (por defecto con valor 200) que contiene el código de estado de las respuesta.
- Response.statusText (en-US) — Cadena (con valor por defecto "OK"), el cual corresponde al mensaje del estado de código HTTP.

- Response.ok — Visto en uso anteriormente, es una clave para comprobar que el estado está dentro del rango 200-299 (ambos incluidos). Este devuelve un valor Boolean (en-US), siendo true si lo anterior se cumple y false en otro caso.

Estos pueden también creados programáticamente a través de JavaScript, pero esto solamente es realmete útil en ServiceWorkers (en-US), cuando pones un objeto response personalizado a una respuesta recibida usando un método respondWith() (en-US):

```
var myBody = new Blob();



addEventListener('fetch', function(event) {
  event.respondWith(
    new Response(myBody, {
      headers: { "Content-Type" : "text/plain" }
    })
  );
});
```

El constructor Response() toma dos argurmentos opcionales, un cuerpo para la respuesta y un objeto init (similar al que acepta Request() (en-US)).

Nota: El método estático error() (en-US) simplemente devuelve un error en la respuesta. De igual manera que redirect() (en-US) devuelve una respuesta que resulta en un redirección a una URL especificada. Estos son solo relevantes tambien a ServiceWorkers.

Body

Tanto las peticiones como las respuestas pueden contener datos body. Body es una instancia de cualquiera de los siguientes tipos:

- ArrayBuffer (en-US)
- ArrayBufferView (en-US) (Uint8Array y amigos)
- Blob/File
- string

Copy to Clipboard

- URLSearchParams
- FormData

El mixin de Body define los siguientes metodos para extraer un body (implementado por Request and Response). Todas ellas devuelven una promesa que es eventualmente resuelta con el contenido actual.

- arrayBuffer() (en-US)
- blob() (en-US)
- json()
- text() (en-US)
- formData()

Este hace uso de los datos no texttuales mucho mas facil que si fuera con XHR.

Las peticiones body pueden ser establecidas pasando el parametro body:

```
var form = new FormData(document.getElementById('login-form'));

fetch("/login", {
  method: "POST",
  body: form
});
```

Tanto peticiones y respuestas (y por extensión la function fetch()), intentaran inteligentemente determinar el tipo de contenido. Una petición tambien establecerá automáticamente la propiedad Context-Type de la cabecera si no es ha establecido una.

Detectar característica

Puedes comprobar si el navegador soporta la API de Fetch comprobando la existencia de Headers, Request, Response o fetch() (en-US) sobre el ámbito de Window o Worker. Por ejemplo:

```
if (self.fetch) {
    // run my fetch request here
} else {
```

```
    // do something with XMLHttpRequest?

}
```

```
function consulta(){
    fetch('http://localhost:3000/listado',{method:'GET'})
    .then(response => response.json())
    .then( function(datos){
            console.log(datos);
            let resultado1 =document.querySelector('#dato1');
            resultado1.innerHTML = '';
            for( var x in datos){
                resultado1.innerHTML +=
                "<div class='col-3 bg-secondary text-white'>"+
                 datos[x].pers_apellido + "</div>"+
                "<div class='col-3 bg-secondary text-white'>"+
                 datos[x].pers_nombre + "</div>"+
                "<div class='col-3 bg-secondary text-white'>"+
                 datos[x].pers_edad + "</div>"  +
                "<div class='col-3 bg-secondary text-white'>"+
                "<a href='http://localhost:3000/modifica/" +
                datos[x].pers_id + "'>" + 'Modificar' +  "</a>"


                + "</div>";

            }
```