

Bootcamp Java Developer

Fase 2 - Java Web Developer
Módulo 19

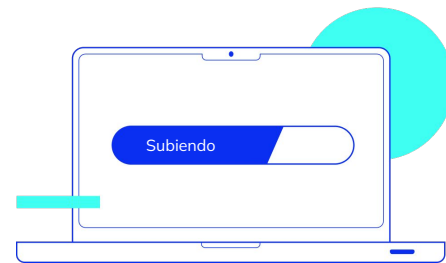
Manejo de archivos

Eventos de progreso

Los eventos de progreso existen para las transferencias de descarga y carga.

Los **eventos de descarga** se activan en el objeto `XMLHttpRequest`.

Los **eventos de carga** se disparan en el objeto `XMLHttpRequest.upload` que es una interfaz del objeto `XMLHttpRequestUpload`.



API XMLHttpRequestUpload

Un objeto **XMLHttpRequestUpload** define un conjunto de propiedades de registro de **controlador de eventos para supervisar el progreso de una carga de cuerpo de solicitud HTTP**.

En navegadores que implementan la especificación **XMLHttpRequest** Nivel 2, cada objeto **XMLHttpRequest** tiene una propiedad de carga que hace referencia a un objeto de este tipo.




Para supervisar el progreso de la carga de la solicitud, simplemente se deben establecer estas propiedades en las funciones apropiadas del controlador de eventos o llamar a los métodos **EventTarget**.

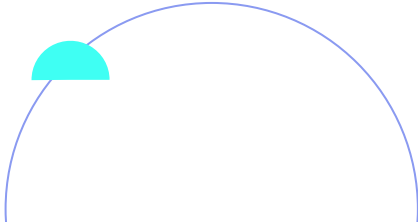
Los controladores de eventos de progreso de carga definidos aquí son exactamente los mismos que los controladores de eventos de progreso de descarga definidos en **XMLHttpRequest**, excepto que no hay una propiedad **on-state statechange** en este objeto.

API FormData

La interfaz **FormData** proporciona una forma de construir fácilmente un conjunto de pares clave/valor que representan campos de formulario y sus valores, que luego pueden enviarse fácilmente utilizando el método **XMLHttpRequest.send()**. Utiliza el mismo formato que un formulario usaría si el tipo de codificación se configura como **"multipart/form-data"**.



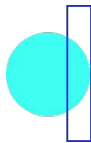
Un objeto que implementa **FormData** se puede usar directamente en una estructura **for ... of**, en lugar de **entries()**: **for(var p of myFormData)** es equivalente a **for(var p of myFormData.entries ())**.



Esta API además de poder construirse desde un formulario:

```
var formData = new FormData(form)
```

Sirve para obtener una representación en objeto JavaScript de un envío de formulario, como pares clave/valor. Podemos leer y escribir el **FormData** a través de sus métodos, como **append** o **get**. **FormData** puede contener tanto datos binarios como no binarios



append

El método **append()** de la interfaz **FormData** agrega un nuevo valor a una clave existente dentro de un objeto **FormData**, o agrega la clave si aún no existe.

La diferencia entre **FormData.set** y **append()** es que si la clave especificada ya existe, **FormData.set** sobrescribirá todos los valores existentes con el nuevo, mientras que **append()** agrega el nuevo valor al final del conjunto existente de valores.

Descarga

Aunque **XMLHttpRequest** se usa, con más frecuencia, para enviar y recibir datos de texto, también se puede utilizar para enviar y recibir contenido binario. Hay varios métodos bien probados para forzar la respuesta de un **XMLHttpRequest** en el envío de datos binarios.

Estos implican la utilización del método **overrideMimeType()** en el objeto **XMLHttpRequest** y es una solución viable.

```
var xhr = new XMLHttpRequest();  
xhr.open("GET", url);  
// recupera datos sin procesar como una cadena binaria  
xhr.overrideMimeType("text/plain;  
charset=x-user-defined");
```

Sin embargo, hay técnicas más modernas porque, ahora, el atributo **responseType** es compatible con varios tipos de contenido adicionales. Esto hace que enviar y recibir datos binarios sea mucho más fácil.



responseType

La propiedad **responseType** del objeto **XMLHttpRequest** puede ser configurado para cambiar el tipo de respuesta esperada del servidor. Los valores posibles son la cadena vacía (predeterminada), **"arraybuffer"**, **"blob"**, **"document"**, **"json"** y **"text"**.

La propiedad de respuesta contendrá el cuerpo de entidad de acuerdo con **responseType**, como **ArrayBuffer**, **Blob**, **Document**, **JSON** o **string**. Esto es nulo si la solicitud no está completa o no fue exitosa.



API Blob

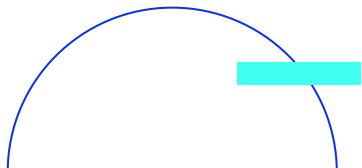
Un objeto **Blob** representa un **objeto tipo fichero de datos planos inmutables**. Los **Blobs** representan datos que no necesariamente se encuentran en un formato nativo de JavaScript.

La interfaz **File** está basada en un **Blob**, hereda y expande su funcionalidad para soportar archivos en el sistema del usuario 3.



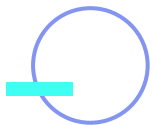
Lamentablemente, si bien los **blobs** sirven para almacenar los datos binarios que podemos pedir por AJAX, no se pueden usar para una posterior construcción del DOM ya que son objetos que tienen en su interior información que no podemos procesar.

Para extraer su contenido, podemos utilizar otra API, como por ejemplo: un método **slice()**, la API de **FileReader** o la API de URL para crear una URL en memoria del contenido que guarda nuestro **blob**.



API URL

La interfaz URL **representa a un objeto que provee métodos estáticos para crear objetos URL**. Esta API provee propiedades y métodos para usarlos de muchas otras maneras pero nosotros vamos a estar interesados en su método **createObjectURL** 4.



createObjectURL

El método estático **URL.createObjectURL()** crea un **DOMString** que contiene una URL que representa el objeto dado en el parámetro. La duración de la URL está vinculada al documento en la ventana en la que se creó. El nuevo objeto URL representa el objeto de archivo especificado o el objeto **Blob**, **File** o **MediaSource**.



Con esta URL generada por la API podemos empezar a construir DOM en forma dinámica:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", url);
xhr.responseType = "blob";
xhr.addEventListener("load",function(e){
    if (xhr.status == 200) {
        var url = URL.createObjectURL(xhr.response)
        console.log(url)
    }
})
```

API Drag & Drop

Las interfaces de arrastrar y soltar HTML permiten a las aplicaciones web arrastrar y soltar archivos en una página web. Este documento describe cómo una aplicación puede aceptar uno o más archivos que se arrastran desde el administrador de archivos de la plataforma subyacente y se eliminan en una página web.



Los pasos principales para arrastrar y soltar son **definir una zona de colocación** (*dropzone*: es decir, un elemento de destino para la eliminación del archivo) **y definir *handlers*** de eventos para los eventos de *drop* y *dragover*.

Tener en cuenta que:

Estos eventos tienen un comportamiento por defecto en el navegador, de la misma forma que vimos con las etiquetas **<a>** y **<form>**. Por este motivo, debemos cancelarlos para poder trabajar libremente con los listeners de cada evento:

```
//index.html  
<div id="dropzone">  
  <p>Arrastre sus archivos aquí</p>  
</div>
```

...



Continúa el ejemplo:

...

```
//index.js
var dropzone = document.querySelector("#dropzone")
dropzone.addEventListener("dragover", e=>{
  e.preventDefault()
  e.stopPropagation()
})
dropzone.addEventListener("drop", e=>{
  e.preventDefault()
  e.stopPropagation()
  console.log("Archivo soltado!")
})
```

Los objetos evento de los eventos de tipo **drag y drop** cuentan con una propiedad bastante particular:

dataTransfer

Se usa para **contener los datos que se arrastran durante una operación de arrastrar y soltar**.

Contiene pares **clave/valor**, donde la **clave** es el tipo de dato del elemento y el **valor** es la información de dicho elemento.

dataTransfer.files contiene una lista de todos los archivos locales disponibles en la transferencia de datos. Si la operación de arrastre no implica arrastrar archivos, esta propiedad es una lista vacía.

El evento de **drop** se activa cuando el usuario suelta el archivo (s). En el siguiente gestor de colocación, si el navegador admite la interfaz **DataTransferItemList**, el método **getAsFile()** se utiliza para acceder a cada archivo; de lo contrario, se utiliza la propiedad de archivos de la interfaz **DataTransfer**.

API File

La interfaz **File** proporciona información sobre archivos binarios y permite acceder a su información desde una aplicación web mediante JavaScript.

Los objetos **File** generalmente se recuperan de un objeto **FileList** devuelto como resultado de que un usuario haya seleccionado archivos usando el elemento `<input>`, desde un objeto **DataTransfer** de una operación de arrastrar y soltar, o desde la API `mozGetAsFile()` en un **HTMLCanvasElement**.

En Gecko, el código con privilegios puede crear objetos **File** que representen cualquier archivo **local sin interacción del usuario**.

Un objeto **File** es un tipo específico de **Blob** y se puede usar en cualquier contexto que pueda tener **Blob**. En particular, **FileReader**, `URL.createObjectURL()`, `createImageBitmap()` y `XMLHttpRequest.send()` aceptan **Blobs** y **Archivos**.



**¡Sigamos
trabajando!**