Tabla de contenidos

8. Bugs and errors

Language

El concepto de bindings y propiedades en JS es muy vago, muchos errores derivan al propio lenguaje, ya que no se advierten hasta que se corre el programa.

Strict Mode

JavaScript puede ser un poco más estricto si se habilita el Strict Mode. Esto se hace agregando “use strict” al inicio de un archivo o el cuerpo de una función. Por ejemplo, si se olvida de agregar let antes de un binding, el modo estricto esto reporta un error. Otro ejemplo es que el binding this tiene valor undefined en funciones que no son llamadas métodos.

```
“use strict”
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // forgot new
// → TypeError: Cannot set property 'name' of undefined
```

Además, el modo estricto no permite darle a una función múltiples parámetros con el mismo nombre y remueve ciertos problemáticos features del lenguaje completamente (con el statement with).

Types

Cuando se conocen los tipos de un programa, es posible que la computadora los chequee por uno, señalando así los errores antes de que se corra el programa. JS no controla los tipos. Hay muchos dialectos de JS que agregan tipos al lenguaje y los chequean. El más popular se llama TypeScript.

Testing

Automated testing es el proceso de escribir un programa que testea otro programa. Le toma sólo unos pocos segundos verificar que el programa se comporta apropiadamente para todas las situaciones que se escribió el test.

Los test normalmente toman la forma de pequeños programas etiquetados que verifican algún aspecto del código.

Debugging

Una vez que se advierte que algo está mal con el programa porque se comporta de forma no esperada o produce errores, el siguiente paso es encontrar qué es lo que causa el problema.

Los browsesr vienen con la capacidad de poner un breakpoint en una línea específica del código. Cuando la ejecución del programa alcanza una línea con un breakpoint, este se pausa, y se pueden inspeccionar los valores de los bindings en ese punto.

Otra forma de setear un breakpoint es incluyendo la declaración debugger en el programa. Si las herramientas de desarrollador están activas en el browser, el programa se va a pausar cuando llegue a esa declaración.

Error propagation

En muchas situaciones, sobre todo cuando los errores son comunes y el que los llama debería estarlos explícitamente teniendo en cuenta, devolver un valor especial es una buena forma de indicar un error.

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"));
```

Exceptions

Las excepciones son un mecanismo que hace posible que el código que se encuentra con un problema, pueda arrojar una excepción. Una excepción puede ser cualquier valor. Salta no sólo la función actual, sino que también quienes la llamaron, hasta la primer llamada que empezó la ejecución actual. Esto se conoce como unwinding the stack.

Una vez que se atrapa una excepción, se puede hacer algo con ella para enfrentar el problema y luego continuar ejecutando el programa.

La palabra reservada throw se usa para arrojar una excepción. Para atraparla, se envuelve la porción de código en un bloque try, seguido de la palabra reservada catch. Cuando el código dentro del bloque try causa una excepción, se evalúa el bloque catch, con el nombre entre paréntesis vinculado al valor de la excepción.

Se utiliza el constructor Error para crear el valor de la excepción. Esto está estandarizado por JS como el constructor que crea un objeto con la propiedad message. En la mayoría de los entornos de JS, las instancias de este constructor también juntan información sobre el call stack que existe cuando la excepción fue creada, lo que se conoce como stack trace.

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L") {
    return "a house";
  } else {
    return "two angry bears";


  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

Cleaning up after exceptions

Un bloque final (finally) dice “no importa que pase, ejecuta este código luego de intentar ejecutar el código en el bloque try”.

Incluso si el código finally se ejecuta cuando se arroja una excepción en el bloque try, este no interfiere con la excepción. Luego de que el bloque finally se ejecuta, el resto del stack continúa unwinding.

Selective catching

Cuando una excepción llega hasta el final del stack sin ser atrapada, ésta se controla por el entorno. En los browsers, normalmente una descripción del error es escrita en la consola de JS. En Node.js, se aborta todo el proceso cuando una excepción no controlada ocurre.

JavaScript no provee de un soporte directo para catching selectivo de excepciones.

Como regla general, no se hace blanket-catch de excepciones.

Si se quiere atrapar un tipo específico de excepciones, se lo puede hacer chequeando el bloque catch viendo si la excepción es la que se está buscando, sino se vuelve a arrojar.

Se puede definir un nuevo tipo de error y usar instanceof para identificarlo:

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

La clase InputError extiende de Error. Ahora es catch puede encargarse de lo siguiente de forma más cuidadosa, Esto atrapa instancias de InputError únicamente y deja pasar las excepciones que no están relacionadas:

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

Assertions

Los assertions son verificadores dentro del programa que se aseguran que algo sea de la forma que se supone que es. Se utilizan para manejar situaciones que pueden surgir en una operación normal pero para encontrar errores del programador.

Esto vuela el programa ni bien se hace mal uso del mismo, pero no es recomendable tratar de escribir assertions para cada posible entrada errónea.

10. Modules

Un módulo es una porción de programa que especifica en qué otras piezas se basa y qué funcionalidad provee a otros módulos para utilizar (interfaz).

Las interfaces de los módulos se parecen a las de los objetos. Hacen que una parte del módulo sea visible y mantienen el resto privado.

Las relaciones entre los módulos se denominan dependencias. Cada módulo necesita su propio scope privado, separar código en distintos archivos no es suficiente ya que siguen compartiendo variables globales.

Packages

Una de las ventajas de modularizar un programa es que los módulos se pueden ejecutar independientemente y reutilizar en distintos programas.

Un package es un pedazo de código que puede ser distribuido (copiado e instalado). Contiene uno o más módulos además de información sobre los packages de los cuales depende. Usualmente también vienen con su propia documentación. Cuando se encuentra un bug o se agrega una nueva funcionalidad, se actualiza el paquete, entonces lo programas que dependen de él pueden obtener la nueva versión.

NPM (Node Package Manager) es dos cosas, por un lado un servicio online en el cual se pueden subir o descargar paquetes de JavaScript (https://npmjs.org). Y por otro lado es un programa que viene en bundle con Node.js el cual ayuda a instalar y manejar estos paquetes. La mayoría del código en NPM tiene licencia libre.

Improvised modules

Hasta el 2015, JS no tenía ningún sistema de módulos incluido. Se utilizaban funciones para crear local scopes y objetos para representar la interfaz del módulo. Estos módulos proveen de aislamiento hasta cierto punto pero no declaran dependencias. Actualmente esto es obsoleto.

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

Evaluating data as code

Hay muchas formas de tomar datos (un string de código) y ejecutarlo como parte del programa actual.

La más obvia es con el operador eval, que ejecuta el string en el scope actual, sin embargo es mala idea ya que rompe con algunas propiedades del scope.

Otra forma es utilizando el constructor Function, el cual tiene dos argumentos: un string con los argumentos separados por comas y un string con el cuerpo de la función.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

CommonJS

En enfoque más utilizado para los módulos en JS se llama CommonJS modules, es el sistema utilizado por Node.js y la mayoría de los paquetes en NPM.

El concepto principal en los módulos de CommonJS es una función llamada require. Cuando se llama a esta función con el nombre de un módulo del cual se tiene dependencia, se asegura que dicho módulo sea cargado y devuelve su interfaz.

Al cargarse, se envuelve el código del módulo en una función, los módulos tienen automáticamente su propio scope. Sólo se necesita llamar a require para acceder a las dependencias y poner la interfaz en el objeto asociado al exports.

Es decir, se crea en un archivo JS el módulo y se lo exporta, luego si otro archivo depende de este, se debe hacer un require del nombre del archivo, accediendo así a la interfaz que se exportó.

La función require se puede definir de la siguiente forma:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

Para evitar cargar el mismo módulo muchas veces, require almacena un caché de los módulos ya cargados. Cuando se la llama, primero se fija si el módulo ya está cargado, si no lo está, lo carga. Esto incluye leer el código del módulo, envolverlo en una función y llamarlo.

Por defecto los módulos de CommonJS devuelven un objeto de interfaz, pero esto se puede reemplazar por cualquier valor o función sobreescribiendo module.exports.

Al definir require, exports y module como parámetros para la wrapper function generada (y pasando los valores apropiados cuando se llama), el loader se asegura de que estas variables estén disponibles en el scope del módulo.

Cuando el string que se le pasa a require comienza con ./ o ../ se interpreta como relativo, es decir, es el path y el nombre de un archivo de extensión js (por ejemplo "./format-date" hace referencia al archivo format-date.js que se encuentra en el mismo directorio).

Cuando no es relativo, Node.js busca entre los paquetes instalados con ese nombre.

ECMAScript Modules

A partir de 2015 se introduce un nuevo sistema de módulos denominado ES modules. El concepto principal de dependencias e interfaces se mantiene, pero difiere en algunos detalles. En lugar de llamar a una función para acceder a una dependencia, se utiliza la palabra reservada import.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

A su vez, export se utiliza para exportar cosas, puede ser tanto una clase, como una función, una variable o una constante.

La interfaz de un módulo ES no es un solo valor sino un conjunto de variables. Cuando se importa desde otro módulo, se importa el binding, no un valor, es decir, el módulo exportado puede cambiar el valor del binding en cualquier momento y los módulos que lo importaron verían su nuevo valor.

Si hay un binding con el nombre default se trata como el valor a exportar por defecto. Si se importa un binding sin llaves (como ordinal en el ejemplo anterior) se obtiene el default binding. Los módulos pueden igualmente exportar otros binding bajo otros nombres además del default.

export default ["Winter", "Spring", "Summer", "Autumn"];

Se pueden cambiar los nombres de los binding importados con la palabra as

```
import {days as dayNames} from "date-names";

console.log(dayNames.length);
// → 7
```

Otra diferencia importante es que el módulo ES realiza el import antes de ejecutar el script. Esto implica que los import no deben estar dentro de funciones o bloques, y los nombres de las dependencias deben ser string entre comillas, no expresiones arbitrarias.

Building y Bundling

Cargar en un solo archivo resulta mucho más rápido que cargar muchos archivos pequeños. Por ello, los desarrolladores utilizan herramientas transforman el programa en un solo gran archivo antes de subirlo a la web. Estas herramientas se conocen como bundlers.

Otras herramientas llamadas minifiers, disminuyen el tamaño de los archivos JS eliminando comentarios y espacios en blanco, renombrando bindings y reemplazando porciones de código por otro equivalente de menor tamaño.

Module design

Un aspecto del diseño modular es la facilidad de uso, para una o varias personas. Otra ventaja es la facilidad con la que algo puede estar compuesto por otro código.

11. Asynchronous Programming

Programación sincrónica o bloqueante y asincrónica o no bloqueante

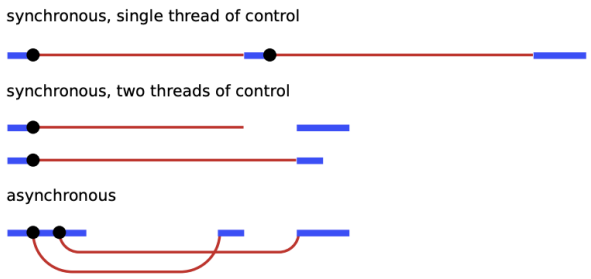
Un código síncrono es aquel código donde cada instrucción espera a la anterior para ejecutarse mientras que un código asíncrono no espera a las instrucciones diferidas y continúa con su ejecución. Por lo general la asincronía permite tener una mejor respuesta en las aplicaciones y reduce el tiempo de espera del cliente.

Asynchronicity

En un modelo sincrónico de programación, las cosas suceden de una a la vez. Cuando se llama a una función que requiere mucho tiempo de ejecución, esto detiene al programa hasta que esta función puede devolver un resultado.

Un modelo asincrónico permite que muchas cosas sucedan a la vez. Cuando se comienza una acción, el programa continúa ejecutándose. Cuando la acción concluye, el programa es informado y tiene acceso al resultado.

En el siguiente diagrama, las líneas gruesas representan el tiempo de ejecución real del programa mientras que las líneas delgadas son el tiempo de espera. En el modelo sincrónico el tiempo de espera es parte del timeline. En el modelo asincrónico, el tiempo se desdobla ya que el programa que había comenzado la acción continúa ejecutándose, a la espera de ser notificado por el resultado.



La espera de acciones para que finalicen es implícita en el modo sincrónico, pero es explícita y controlada en el modo asincrónico.

Callbacks

A las funciones que requieren mucho tiempo, se les agrega un argumento extra, una callback function. La acción se inicia y cuando termina se llama a la función callback con el resultado.

Realizar muchas acciones asincrónicas seguida implica que se deben pasar contínuamente nuevas funciones como argumento. Esto funciona pero es complejo de leer y manejar.

```
import {bigOak} from "./crow-tech";

bigOak.readStorage("food caches", caches => {
  let firstCache = caches[0];
  bigOak.readStorage(firstCache, info => {
    console.log(info);
  });
});
```

La asincronicidad es contagiosa, cualquier función que llame a una función que es asincrónica, debe ser entonces asincrónica en sí misma y manejarlo con una callback o algún mecanismo similar.

Promises

En lugar de llamar una función en algún punto del futuro, se puede devolver un objeto que representa este evento futuro. Una promesa es una acción asincrónica que se completará en algún punto y devolverá un valor.

La forma más sencilla de crear una promesa es llamando a Promise.resolve. Esta función se asegura de que el valor que se le da está envuelto en una promesa. Si ya es una promesa, simplemente lo devuelve, sino se obtiene una nueva promesa que termina con el valor como resultado.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

Para obtener el resultado de una promise, se utiliza el método then, esto registra una callback que será llamada cuando la promesa se resuelva y produzca un resultado. Se pueden agregar múltiples callbacks a una promesa.

El método then devuelve otra promesa, la cual resuelve al valor que la función handler devuelve o si eso devuelve otra promesa, espera por esa promesa y después resuelve.

El valor de una promesa es un valor que puede estar ahí o puede aparecer en un futuro. Para crear una promesa se puede usar el constructor Promise, el cual espera una función como argumento, la cual resolverá la promesa.

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

Failure

Es difícil asegurarse de que las fallas se reportan correctamente en las callbacks. Una convención utilizada es que el primer argumento de la callback se utiliza para indicar que la acción falló y el segundo contiene el valor producido por la acción cuando es satisfactorio.

En las promesas es más sencillo, pueden ser “resolved” o “rejected” por lo que si algún elemento en la cadena falla, el resultado final será marcado como rejected. Las promesas tienen un método catch que registra el handler a llamar cuando la promesa es rechazada (similar al then para el flujo normal). Then también acepta un rejection handler como segundo argumento.

La función que se le pasa al constructor Promise recibe un segundo argumento, junto a la función resolve, el cual se puede utilizar para rechazar a la nueva promesa.

Las cadenas de valores de promesas creados por llamar a then y catch se pueden ver como una tubería a través de la cual valores de promesas o fallos fluyen. Como esas cadenas son creadas por handlers registrados, cada link tiene un succesful, rejected o ambos handler.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Handler 1"))
  .catch(reason => {
    console.log("Caught failure " + reason);
    return "nothing";
  })
  .then(value => console.log("Handler 2", value));
// → Caught failure Error: Fail
// → Handler 2 nothing
```

Networks are hard

Incluso cuando un request y su response son entregados satisfactoriamente, el response puede estar indicando una falla. Por ello, send y defineRequestType siguen la convención donde el primer argumento que se pasa a la callback es la razón de la falla, si hay alguna, y el segundo el resultado real.

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;
    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
```

```
      }, 250);
    }
    attempt(1);
  });
}
```

Las promesas pueden ser resueltas o rechazadas sólo una vez. Hay que crear un loop asincrónico para los reintentos. La función attempt realiza un solo intento de enviar un request, también setea un timeout que si dentro de 250 milisegundo no recibió respuesta, o bien comienda el siguiente intento o si este fue el tercero, rechaza la promesa.

Para aislarse de las callback, se define algo que envuelva a defineRequesType y permita manejar mejor la callback. Promise.resolve se utiliza para convertir el valor devuelto por handler en una promesa, si ya no lo es.

```
function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
                           callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
        .then(response => callback(null, response),
              failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}
```

Collections of promises

Cuando se trabaja con colecciones de promesas, resulta útil la función Promise.all, devuelve una promesa que espera a que todas las otras promesas en el arreglo se resuelvan y luego resuelve un arreglo con los valores que todas las promesas produjeron. Si una promesa es rechazada, el valor de Promise.all también es rechazado.

```
requestType("ping", () => "pong");

function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

La función que está mapeada sobre el set de neighbors para transformarlos en request promises, agrega handlers que hacen que las request satisfactorias devuelvan true y las rechazadas false.

Filter se utiliza para remover los elementos del arreglo neighbors que corresponden a un valor falso.

Async functions

JavaScript permite crear código pseudo-sincrónico para describir computación asincrónica. Una función async es una función que implícitamente devuelve una promesa y que puede en su cuerpo await (esperar) por otras promesas en una forma que parece sincrónica.

```
async function findInStorage(nest, name) {
  let local = await storage(nest, name);
  if (local != null) return local;

  let sources = network(nest).filter(n => n != nest.name);
  while (sources.length > 0) {
    let source = sources[Math.floor(Math.random() *

                                    sources.length)];
    sources = sources.filter(n => n != source);
    try {
      let found = await routeRequest(nest, source, "storage",
                                     name);
      if (found != null) return found;
    } catch (_) {}
  }
  throw new Error("Not found");
}
```

Las funciones o métodos async se marcan usando dicha palabra antes del nombre de función. Cuando es llamado, devuelve una promesa, tan pronto como el cuerpo devuelva algo, la promesa se resuelve. Si arroja una excepción, la promesa se rechaza.

Dentro de la función async, la palabra await se puede poner en frente de una expresión para esperar a que una promesa se resuelva y recién ahí continuar con la ejecución del cuerpo.

13. JavaScript and the browser

Protocolos de red

Los protocolos de red describen cómo va a ser la comunicación en dicha red. Existen protocolos para enviar mails, compartir archivos, comunicarse entre computadoras, etc.

Por ejemplo, el protocolo HTTP (Hypertext Transfer Protocol) es un protocolo para obtener recursos con un nombre.

El protocolo TCP (Transmition Control Protocol) es un protocolo de comunicación el cual especifica que una computadora servidor debe estar escuchando en un puerto a la espera de request de computadoras cliente. Es un protocolo seguro ya que ofrece ACK.

La web

La World Wide Web es un conjunto de protocolos y formatos que permiten acceder a páginas web a través de browsers. Todos los documentos en la web son nombrados por un URL (Uniform Resource Locator), el cual tiene la siguiente forma:

```
  http://eloquentjavascript.net/13_browser.html
 |      |                      |              |
protocol       server              path
```

Cuando un dispositivo se conecta a internet, se le asigna un número de IP el cual se utiliza para rutear los mensajes, sin embargo, para recordarlo más fácilmente se crearon los nombres de dominio (el que figura como server).

HTML

Hypertext Markup Language es el formato de documento utilizado por las páginas web. Contiene tanto texto como etiquetas (tags) que le dan estructura al texto.

Los documentos HTML comienzan con la etiqueta <!doctype html> la cual le indica al browser que interprete el documento como HTML moderno.

Los documentos HTML también tienen un head y un body. El head contiene información sobre el documento (como por ejemplo el título y el encoding) y el body contiene el documento en sí.

Algunos tag contienen información extra, se les denomina atributos. La mayoría de las etiquetas deben abrirse y cerrarse y contienen texto en el medio, sin embargo no todas lo hacen, por ejemplo <meta charset="utf-8">. HTML es muy tolerante a errores, a modo de que si hay etiquetas faltantes o que no fueron cerradas, el browser se encarga de reconstruirlas, de acuerdo a ciertos estándares.

HTML y JavaScript

Es posible incluir JavaScript en un documento HTML utilizando la etiqueta <script>, sin embargo, incluir grandes códigos JS en el documento HTML no es conveniente, por lo tanto se puede utilizar el atributo src para referenciar un archivo JS, por ejemplo:

```
<script src="code/hello.js"></script>
```

El browser busca el archivo e incluye el código en el documento. Las etiquetas de script siempre deben cerrarse, ya que sino todo el resto del documento es interpretado como un script.

También se pueden agregar módulos ES agregándole a la etiqueta script el atributo type="module".

Algunos atributos pueden contener también código JS, por ejemplo la etiqueta <button> tiene un atributo onClick, el cual ejecuta el código cuando se hace click en el botón.

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

In the sandbox

Por cuestiones de seguridad, los browsers limitan las cosas que JS puede hacer en un dispositivo, no se pueden ver archivos ni modificar nada que no esté relacionado a la página web en sí.

Aislar el entorno de un programa de esta forma se llama **sandboxing** (la idea surge de que el programa está jugando sin riesgo dentro de una caja de arena). Es complicado ya que se debe permitir que el programa pueda ejecutarse correctamente pero sin la libertad suficiente para que sea peligroso. Por ello los desarrolladores de browsers siempre deben actualizarlos.

Compatibilidad y browser wars

Se denomina browser wars a un período en el cual los browser más populares desarrollaban nuevos features y los programas comenzaban a utilizarlos, sin importar la compatibilidad con los otros browser. De este modo, existían un montón de distintas plataformas para el desarrollo web y todas con distintos bugs.

Actualmente la incompatibilidad es mucho menor y la cantidad de bugs se disminuyó considerablemente.

14. The Document Object Model

El DOM (Document Object Model) es una API para documentos HTML y XML. Define una estructura lógica de documentos y la forma en la cual los documentos son accedidos y manipulados.

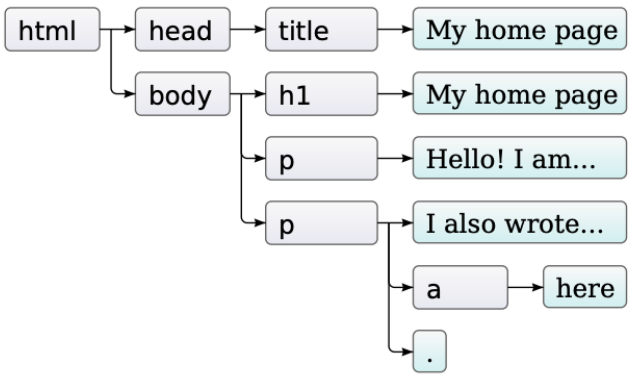
Estructura del documento

Un documento HTML se puede definir como un conjunto de etiquetas anidadas. Por cada etiqueta hay un objeto con el cual se puede interactuar. Esta representación se denomina DOM. La variable global document da acceso a estos objetos. La propiedad documentElement refiere al objeto que representa la etiqueta html>. Ya que todos los documentos HTML tienen un head y un body, también tiene las propiedades head y body, apuntando a esos elementos.

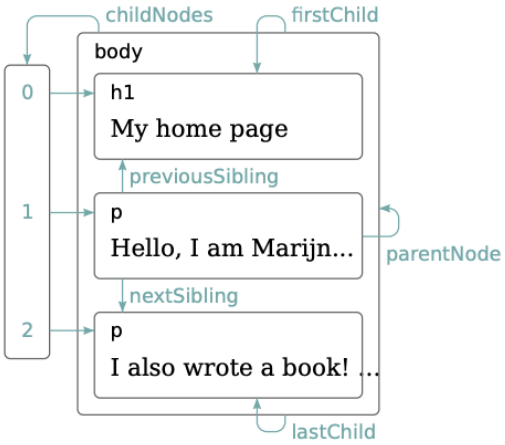
```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

Árboles

Los elementos del DOM se pueden representar como un árbol donde los nodos son elementos que representan etiquetas HTML, las hojas son el contenido y la raiz es el document.documentElement. Cada nodo objeto tiene del DOM tiene una propiedad nodeType que contiene un código que identifica el tipo de nodo.



Los nodos del DOM contienen links a los nodos cercanos. Técnicamente se podría mover a lo largo del árbol utilizando las propiedades de parentNode y childeNodes. También previousSibling y nextSibling se mueven a los nodos adyacentes. Así mismo la propiedad children devuelve sólo elementos del tipo 1 (no texto).



Encontrar elementos

No es eficiente recorrer todo el árbol en busca de un elemento, puedo buscar por tag y obtener un atributo de ese tag. El método getElementByTagName devuelve un array con todos los resultados, al agregar [0] utiliza solo el primer elemento.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

Todos los elementos tienen el método getElementByTagName, pero si quiero encontrar uno en particular, se le puede dar un atributo id y luego buscar con la propiedad getElementById. Una tercera forma similar es utilizar el método getElementByClass, que en lugar de buscar por clase lo hace por id.

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

Cambiar el documento

Casi todo en la estructura de datos del DOM se puede cambiar. Los nodos tienen métodos tales como remove, appendChild (pone el nodo al final de la lista de hijos), insertBefore o replaceChild.

```
<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

Crear nodos

Se pueden crear, agregar y reemplazar. Para crear un nodo de texto se utiliza el método document.createTextNode. Para crear un nodo de un elemento se utiliza el método document.createElement, este método recibe un tag y devuelve un nodo vacío del tipo dado.

Atributos

Algunos atributos de los elementos pueden ser accedidos a través de una propiedad del mismo nombre en el objeto DOM del elemento. Sin embargo, HTML permite setear atributos propios en los nodos, para ello se utilizan los métodos getAttribute y setAttribute.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

Layout

Para cada documento, los browser pueden crear su propio layout, que luego le va a dar a cada elemento su posición y tamaño de acuerdo a su tipo. Este layout luego se utiliza para dibujar el documento. La unidad básica de medida en los browser es el pixel y existen métodos para calcular el ancho, altura y posición exacta de los elementos.

Un programa que alterna constantemente entre leer la información de layout del DOM y cambiar el DOM, forza a mucha computación de layout y por consecuencia será muy lento.

Styling

Se puede cambiar el estilo asociado a un elemento. El atributo style debe contener una o más declaraciones, las cuales son propiedades junto con su valor. JavaScript puede manipular el estilo de un elemento directamente, por medio de la propiedad style.

```
<p id="para" style="color: purple">
  Nice text
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Cascading styles

El sistema de estilos para HTML se denomina CSS (Cascade Styling Sheet). Un style sheet es un conjunto de reglas para el estilo del documento, también pueden estar dentro de la etiqueta <style>.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

Se denomina en cascada ya que múltiples reglas se combinan para definir el estilo. Cuando más de una regla se aplica a un elemento, la más recientemente leída obtiene preferencia. También se pueden agregar estilos a clases o id específicos (el punto representa clase y el numeral id). A su vez, si una regla tiene mayor nivel de especificación, tiene mayor precedencia (una regla para p.a es más importante que una para solo p o solo .a).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

Query Selectors

El método querySelectorAll está definido tanto en el objeto documento como en los nodos, este toma un selector string y devuelve una nodeList con todos los elementos que coincidan. A diferencia de métodos como getElementById, el objeto que devuelve no es dinámico, es decir que no va a cambiar si se cambian los valores del documento.

```
<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>
```

15. Handling Events

Para manejar lo que ocurre cuando el usuario presiona una tecla o hace un click, el sistema notifica al código cuando ocurre un evento. Los Browser permiten registrar funciones como handlers para eventos específicos.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

El binging window hace referencia a un objeto que viene provisto por el browser. Representa la ventana del browser que contiene el documento. El método addEventListener llama la acción del segundo argumento cuando el primer argumento ocurre.

Events and DOM nodes

Cada Event Handler es registrado en un contexto. Métodos como addEventListener pueden ser encontrados en los elementos del DOM y otro tipo de objetos. Los event listener son llamados solo cuando el evento ocurre en el contexto del objeto en el cual están registrados.

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

También se le puede dar a un nodo el atributo onClick, pero un nodo sólo puede tener un atributo onClick, por lo cual solo se puede registrar un handler por nodo de esa forma. El método addEventListener permite agregar cualquier cantidad de handlers.

El método removeEventListener utiliza argumentos similarmente a addEventListener, pero elimina el handler. Recibe el mismo valor de la función que fue dada al crear el handler.

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
```

Event objects

El evento object posee información adicional sobre el evento, por ejemplo, qué con qué botón del mouse se hizo click. La información guardada en cada evento object difiere según el tipo de evento.

Propagation

Para la mayoría de los eventos, si un handler está registrado en un nodo que tiene hijos, también recibe los eventos que le ocurren a los hijos. Por ejemplo, si se hace click a un botón dentro de un párrafo, el event handler del párrafo también ve ese evento. Si ambos tienen un handler definido, el más específico se ejecuta primero. Los eventos se propagan hacia afuera, desde el nodo en el que ocurren hacia los padres.

Si en algún punto un event handler llama al método stopPropagation, el resto de los handlers de más arriba no recibirán el evento.

La mayoría de los event object tienen una propiedad target que hace referencia al nodo donde se origina. Esta propiedad se utiliza para asegurarse de que no se está activando un handler por accidente, de un evento que se propagó.

También se puede utilizar target para crear una red para un evento específico. Por ejemplo, si se tiene un nodo con muchos botones dentro, se puede hacer un click handler solo en el padre y luego utilizar target para saber cuál de los hijos fue el que se clickeó.

Default actions

Muchos eventos tienen una acción por defecto asociada a ellos. Para la mayoría de los eventos, el event handler de JS es llamado antes que el comportamiento por defecto. Si el handler no quiere que se ejecute el comportamiento por defecto, llama al método preventDefault.

No es recomendable cambiar ciertos comportamientos por defecto que los usuarios están acostumbrados a utilizar.

Key events

Cuando se presiona una tecla, el browser dispara un evento "keydown" y cuando se suelta se obtiene un evento "keyup".

Si una tecla se mantiene apretada, el evento keydown se dispara cada vez que la tecla se repite, es necesario tener cuidado con esto.

Las teclas modificadoras como shift, control, alt y meta (command en Mac) generan eventos igual que las teclas normales, sin embargo, cuando se buscan combinaciones de teclas, se puede saber si estas teclas están siendo presionadas con las propiedades shiftKey, ctrlKey, altKey, and metaKey.

El nodo del DOM donde el key event se origina depende del elemento que tiene foco cuando se presiona la tecla. La mayoría de los nodos no pueden tener foco a no ser que se les de el atributo tabindex. Cuando nada en particular tiene foco, document.body actúa como target.

Cuando el usuario está tipeando un texto, no se suelen disparar eventos. Para notificar que se está tipeando, elementos como imput y textarea disparan eventos cuando el usuario cambia su contenido.

Pointer events

Actualmente existen dos formas de apuntar a algo, haciendo click (con cualquier dispositivo) o tocando la pantalla. Estas dos formas disparan distintos eventos.

Mouse clicks

Cuando se hace click con el mouse se disparan los eventos “mousedown” y “mouseup” en el elemento del DOM que se está señalando. Después del evento “mouseup”, se dispara el evento “click” del elemento del DOM más específico. Si se hace doble click, se dispara el evento “dblclick”.

Para obtener información precisa de dónde ocurrió el evento del mouse, se pueden ver los valores de las propiedades clientX y clientY que contienen coordenadas en relación a la esquina superior izquierda de la ventana, o sino pageX y pageY, que contienen coordenadas en relación a la esquina superior izquierda de todo el documento (varían cuando el documento fue scrolleado).

Mouse motion

Cada vez que el puntero del mouse se mueve, se dispara un evento “mousemove”. Este evento se puede utilizar para hacer tracking del movimiento del mouse. El handler del “mousemove” debe estar registrado en toda la ventana.

Touch events

Una pantalla táctil funciona distinto que un mouse, no tiene multiples botones, no se puede hacer tracking del dedo cuando no toca la pantalla y además permite que varios dedos toquen la pantalla simultáneamente.

Cuando un dedo comienza a tocar la pantalla, se obtiene un “touchstart” event, cuando se desplaza mientras toca la pantalla se dispara un “touchmove” event. Por último, cuando deja de tocar la pantalla, un “touchend” event.

Como las pantallas táctiles detectan múltiples dedos simultáneamente, los event objects tienen una propiedad touches que almacena un array con objetos de las coordenadas.

Comúnmente se utiliza el método preventDefault en los touch event handlers para evitar que se disparen eventos del mouse.

Scroll events

Cuando un elemento es scrolleado, se dispara un “scroll” event. Esto tiene varios usos, como saber qué parte de la ventana está viendo el usuario o mostrar datos de indicación o progreso.

Focus events

Cuando un evento obtiene foco, el browser dispara un “focus” event, cuando pierde el foco el elemento obtiene un “blur” event.

Estos dos eventos no se propagan, el handler de un elemento padre no es notificado cuando el hijo gana o pierde foco.

El elemento window recibe los eventos focus y blur cuando el usuario navega a través de los distintos tab del navegador.

Load event

Cuando una página termina de cargarse, el evento “load” se dispara en la ventana y los objetos del body. Esto se utiliza para controlar acciones que requieren que todo el documento esté cargado.

Los elementos como image y script que cargan archivos externos, también tienen eventos load que indican que el archivo referenciado ya fue cargado. Los eventos load no se propagan.

Cuando una página se cierra o lleva a otro link, se dispara el evento “beforeunload”, el cual se utiliza para prevenir que el usuario pierda datos por cerrar o salir de un sitio sin querer.

Events and the Event Loop

En el contexto de event loop, los event handler de los browser se comportan como otras notificaciones asincrónicas. Están planificados cuando el evento ocurre pero dejeb esperar a que los otros script terminen de ejecutarse antes de hacerlo ellos.

Timers

A veces es necesario cancelar una función que fue programada, para ello se almacena el valor que devuelve setTimeout y se llama a clearTimeout en él.

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

De forma similar se utilizan las funciones setInterval y clearInterval, que setean un timer para que se ejecute el codigo cada X cantidad de milisegundos.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

Debouncing

Algunos tipos de eventos tienen el potencial de dispararse rápidamente muchas veces seguidas. Cuando se manejan estos eventos, hay que tener cuidado de no hacer cosas que consuman mucho tiempo, sino navegar en el documento resulta lento.

Si se necesita hacer algo no trivial en ese handler, se puede utilizar setTimeout para asegurarse de que esa tarea no se realiza muy seguido. A esto se le llama debouncing.

18. HTTP and Forms

Hypertext Transfer Protocol es el mecanismo a través del cual se solicitan y proveen datos en la World Wide Web.

The protocol

Si se escribe un nombre de domino en la barra de dirección del browser, este intenta abrir una conexión TCP y envía algo así:

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

Luego el servidor responde con algo como:

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... the rest of the document
```

El browser toma el body de la respuesta, que es todo lo que está debajo de la línea en blanco, como el documento HTML. La información enviada por el cliente es el request y empieza con una línea así:

```
GET /18_http.html HTTP/1.1
```

La primer palabra es el método del request. Existen diversos métodos que se utilizan para distintas cosas. El servidor no está obligado a aceptar todos los request.

La parte que viene después del método es el path del recurso al cual se le aplica el request.

Luego del path, se indica la versión de HTTP que se está utilizando. Actualmente se utiliza la versión 2 que es compatible con la 1.1, el browser cambia automáticamente a la versión del servidor cuando hacen el request.

La respuesta del servidor empieza con la versión de HTTP seguido del estado de la respuesta, un status code de tres dígitos y un string con la descripción.

```
HTTP/1.1 200 OK
```

La primer línea de un request o response puede estar seguido de cualquier cantidad de headers. Estas son líneas de la forma name: value que especifican información extra del request o response, como en el ejemplo:

```
Content-Length: 65585
```

Luego de los headers, tanto el request como response pueden incluir una linea en blanco seguida del body, el cual contiene los datos enviados.

Browsers and HTTP

Las páginas HTML pueden incluir formularios (forms) que permiten al usuario ingresar diversa información y enviarla al servidor. Cuando se hace click en el botón “enviar”, el form es *submitted* su contenido se empaqueta y se envía como un HTTP request.

Cuando un <form> tiene un método GET o es omitido, sus datos se agregan como un string al final del URL como un query string:

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

El signo de pregunta indica que termina la parte del path del URL y comienza el query. Seguido por pares de nombre y valor, de los datos del form. El & se utiliza para separar los pares.

Dependiendo el tipo de encoding (en este caso URL encodign) algunos símbolos no pueden enviarse y son reemplazados por otra cosa (por ejemplo, %3F representa un ?). JavaScript cuanta con las funciones encodeURIComponent y decodeURIComponent.

Si el método del form es un POST en lugar de un GET, entonces el query string será agregado al body del HTTP request en lugar de ser agregado al URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

Fetch

La interfaz a traves de la cual el JS del browser puede realizar HTTP request se llama fetch. Es relativamente nueva y utiliza promesas.

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

Llamar a fetch, devuelve una promesa que devuelve un objeto Response que almacena información sobre la respuesta del servidor. El header es envuelto en un objeto Map-like que trata los keys como case insensitive. La respuesta de la promesa devuelta por fetch es exitosa incluso si la respuesta del servidor es un código de error. Podría ser rechazada si hay un error en la red o si no se encuentra el servidor.

El primer argumento de fetch es el URL del request, cuando este URL no comienza con el nombre de un protocolo, se toma como relativo. Cuando comienza con una barra inclinada, reemplaza el path actual, la parte luego del nombre del servidor. Cuando no, la parte del path actual se agrega en frente del URL relativo.

Para llegar al contenido de la respuesta, se utiliza el método text. Como esto puede tomar un momento, de nuevo devuelve una promesa.

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));

// → This is the content of data.txt
```

Un método similar, llamado json, se resuelve cuendo el valor que se obtiene se parsea a json o se rechaza cuando el valor no es compatible con json.

Por defecto, fetch utiliza el método GET y no incluye un body en el request.

HTTP Sandboxing

Por protección, los browsers no permiten que los scripts realicen HTTP requests a otros dominios. Esto puede ser un problema cuando se quiere crear sitios web con conexión a otras páginas de forma legítima, por ellos los servidores pueden incluir este header para indicar que pueden ser accedidos por otros scripts:

```
Access-Control-Allow-Origin: *
```

Appreciating HTTP

La comunicación se construye en torno a recursos y métodos HTTP. En lugar de llamadas a procedimientos remotos, se realiza por ejemplo un PUT request. Del mismo modo se pueden obtener recursos ajenos con un método GET. Esto también ayuda a almacenar recursos en la caché del cliente.

Security and HTTPs

El protocolo HTTP secure, que se usa con las páginas que empiezan con https:// envuelve el tráfico de HTTP de forma tal que es complicado de leer o manipular. Antes de intercambiar datos, el cliente verifica que el servidor es quien dice ser, pidiéndole un certificado encriptado, emitido por una autoridad que el browser reconoce.

Form fields

Un formulario consiste en cualquier cantidad de campos input, agrupados dentro de un <form>. Una diversa cantidad de estilos de camps son contemplados por HTML. Muchos tipos de campos utilizan el tag <input> algunos de ellos son:

`text`	A single-line text field
`password`	Same as `text` but hides the text that is typed
`checkbox`	An on/off switch
`radio`	(Part of) a multiple-choice field
`file`	Allows the user to choose a file from their computer

Los campos del form pueden aparecer fuera de él, pero solo pueden ser submitted si están dentro de él.

Focus

A diferencia de otros elementos, los campos de un formulario si pueden tener foco del teclado. Cuando se les hace click o activa, se convierten en el receptor de la entrada por teclado. Se puede controlar el foco con los métodos focus y blur de JS. El método document.activeElement devuelve el elemento que tiene foco en el momento.

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

Se puede utilizar JS para darle foco a un elemento, pero a su vez HTML provee el atributo autofocus que genera el mismo efecto pero el browser es consciente de lo que se realiza.

También es posible dar un orden a los campos para que se avance sobre ellos con la tecla tab, utilizando el atributo tabindex. Por defecto, a la mayoría de los elementos de HTML no se les puede hacer foco, pero se les puede agregar dandole un número de tabindex o hacer quitárselo dandole valor -1.

Disabled fields

Todos los campos de un formulario se pueden desactivar con el atributo disabled. Un elemento desactivado no puede tener foco ni ser cambiado.

The form as a whole

El elemento form tiene una propiedad llamada elements que tiene colección array-like de todos los campos que están dentro de él.

El atributo name determina la forma en la cual el valor de ese campo va a ser identificado cuando el form sea submitted. También se puede usar como nombre del elemento cuando se lo acceda por medio del atributo elements del form.

```
<form action="example/submit.html">
  Name: <input type="text" name="name"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Log in</button>
</form>
<script>
  let form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

Hacer un submit manual significa que el browser navega a través de la pagina indicada en el atributo action del form, usando un request GET o POST. Pero antes de que eso ocurra, se dispara el evento del “submit”, esto se puede manejar con JS evitando el comportamiento por defecto llamando a preventDefault.

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

Text fields

Los campos creados con <textarea> o <input> que tienen el tipo text o password comparten una misma interface. Sus elementos DOM tienen una propiedad value que almacena su contenido como un string. Si se setea el valor de esta propiedad a otro string, cambia el contenido del campo.

Las propiedades selectionStart y selectionEnd dan información sobre el cursor y la selección de texto. Cuando nada está seleccionado, ambas tienen el mismo número. Estos valores también se pueden escribir.

La función replaceSelection reemplaza el texto seleccionado por otro. El evento change no se dispara cada vez que se detecta una entrada, para ello hay que registrar un handler para el evento input.

Checkboxes and radio buttons

Un checkbox es un campo binario. Su valor se puede extraer o cambiar con la propiedad checked, que almacena un valor booleano.

El tag <label> asocia un pedazo del documento con un campo de entrada, al hacer click en cualquier parte del label, se activa el campo, haciendole foco y marcando el valor si es un checkbox o radio button.

Un radio button es similar pero está implícitamente ligado a otros radio buttons con el mismo atributo name, por lo tanto sólo uno de ellos puede estar activo a la vez.

Select fields

Son conceptualmente similares a los radio buttons, pero la apariencia del <select> es determinada por el browser. Cuando se les da el atributo multiple, el usuario puede seleccionar cualquier cantidad de opciones. Cada tag <option> tiene un valor que puede ser definido con el atributo value. La propiedad value de un elemento select refleja la opción seleccionada.

Los tag <option> de un campo <select> pueden ser accedidos como un array-like de objetos a través de la propiedad options del campo. Cada opción tiene una propiedad llamada selected, que indica que esa opción está seleccionada actualmente.

File fields

Estos campos fueron diseñados originalmente para que el usuario cargue archivos desde su computadora, en browsers modernos proporcionan una forma de leer archivos desde programas JS.

Este campo suele verse como un botón con un texto como “seleccionar archivo” con información sobre el archivo seleccionado al lado. La propiedad files es un array-like de objetos que contiene los archivos seleccionados. También tienen una propiedad multiple que permite seleccionar varios archivos.

Los objetos en files tienen propiedades como name, size, type (text/plain o image/jpeg). Lo que no tienen es una propiedad con el contenido del archivo.

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

Para leer los archivos se debe crear un objeto FileReader, registrar un event handler “load” para el mismo y llamar a su método readAsText, dándole el archivo que se quiere leer.

FileReaders también dispara un evento “error” cuando la lectura del archivo falla por cualquier motivo.

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

Storing data client-side

El objeto localStorage puede ser utilizado para guardar datos que sobrevivan a las recargas de la página. Permite guardar strings bajo un nombre. Los valores se almacenan hasta que se sobreescriben, se remueven con removeItem o el usuario limpia los datos locales.

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

En un principio, se supone que los datos almacenados por un dominio, pueden ser leídos y sobreescritos sólo por ese dominio. Los browser ponen un límite a la cantidad de datos que se pueden almacenar en el localStorage por dominio.

Existe otro objeto similar, llamado sessionStorage, la diferencia es que el contenido de este es borrado cuando se termina la sesión, en la mayoría de los casos esto es cuando se cierra el browser.

20. Node.js

Node.js es un entorno de ejecución de JavaScript orientado a eventos asíncronos. Diseñado para realizar aplicaciones multiplataforma y cuenta con un solo thread de ejecución.

The Node Command

Se puede correr node desde la línea de comandos para ejecutar el programa:

```
$ node hello.js
Hello world
```

Si se corre node sin darle un archivo, provee de un prompt en el cual se puede ejecutar código JS y obtener el resultado inmediatamente.

El binding process, igual que el binding console, está disponible globalmente en Node. Provee de varias formas de inspeccionar y manipular el programa actual. El método exit termina el proceso y se le puede dar un exit status code, el cual le dice al programa que inició node si se completó satisfactoriamente o si encontró un error.

Todos los bindings standard globales de JS se encuentran presentes en el entorno de Node, tales como Array, Math y JSON.

Modules

Node pone algunos bindings adicionales en el scope global. El sistema de módulos CommonJS viene incluido en Node y se utiliza para cargar cualquier cosa, desde módulos incorporados hasta paquetes descargados y archivos que son parte del propio programa.

La extensión .js puede ser omitida y Node la agregará si dicho archivo existe.

Cuando un string que no se ve como un path relativo o absoluto se le da a un require, se asume que se refiere a un módulo incorporado o a un módulo instalado en el directorio node_modules.

Installing with NPM

El uso principal de NPM es instalar paquetes. Luego de ejecutar npm install, NPM habrá creado el directorio llamado node_modules. Por defecto, NPM instala paquetes en el directorio actual, en lugar de en un lugar central.

Package files

Es recomendable crear el archivo package.json para cada proyecto, ya sea manualmente o ejecutando el comando npm init. Contiene información sobre el proyecto como nombre, versión y lista de dependencias, entre otros.

Cuando se ejecuta npm install sin nombrar el paquete a instalar, NPM instalará todas las dependencias listadas en package.json. Cuando se instala un paquete específico que no está listado como dependencia, NPM lo agregará a package.json.

Versiones

El archivo package.json lista tanto la versión del propio programa como las versiones de las dependencias.

NPM demanda que sus paquetes sigan un esquema denominado semantic versioning. Esta consiste en tres números separados por puntos. Cada vez que se agrega funcionalidad, el número del medio es aumentado. Cada vez que se rompe la compatibilidad, de modo que el código existente que utiliza al paquete puede no funcionar con la nueva versión, el primer número se incrementa.

El caracter caret (*) indica que cualquier versión compatible con el número dado debe ser instalada.

The file system module

Exporta funciones para trabajar con archivos y directorios. Por ejemplo, la función llamada readFile lee un archivo y luego llama una callback con el contenido del archivo.

El segundo argumento de readFile indica el character encoding utilizado para decodificar el archivo en un string. Si no se pasa un encoding, Node va a asumir que se está interesado en el binario y devolverá un objeto Buffer.

Una función similar es writeFile, la cual se utiliza para escribir un archivo en disco.

La mayoría de estas funciones toman una función callback como último parámetro, al cual llaman ya sea con un error (el primer argumento) o con un resultado satisfactorio (el segundo).

Existe un objeto promises que se exporta del paquete fs, que contiene la mayoría de las funciones de fs pero utiliza promises en lugar de callbacks.

The HTTP module

Otro módulo central es el llamado http. Este provee funcionalidad para ejecutar servidores HTTP y hacer requests HTTP.

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

La función pasada como argumento a createServer es llamada cada vez que un cliente se conecta al servidor. Los binding request y response son objetos que representan los datos entrantes y salientes.

Entonces, cuando se abre esa página en el browser, envía un request. Para enviar algo de vuelta, se llama a un método en el objeto response. El primero, writeHead, escribirá los headers del response. Luego, el verdadero cuerpo del response (el documento en sí) se envía con response.write. Finalmente, response.end señala el final de la respuesta.

La llamada a server.listen causa que el servidor comience a esperar por conexiones en el puerto 8000.

Para actuar como un cliente HTTP, se puede utilizar la función request del módulo HTTP.

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code",

              response.statusCode);
});
requestStream.end();
```

El primer argumento del request configura la petición. El segundo argumento es la función que debe ser llamada cuando viene la respuesta.

Al igual que el objeto response en el servidor, el objeto que devuelve request permite enviar datos en el request con el método write y terminar el request con el método end.

Streams

Los writable streams son un concepto ampliamente utilizado en Node. Estos objetos tienen un método write que puede ser pasado como un string o un objeto Buffer para escribir algo en el stream. Su método end cierra el stream y opcionalmente toma un valor para escribir en el stream antes de cerrarlo.

Ambos, el request binding que fue pasado al callback del servidor HTTP y el response binding pasado al callback del cliente HTTP son streams legibles (un servidor lee requests y luego escribe responses). Leer desde un stream se hace con event handlers en lugar de métodos.

Los objetos que emiten eventos en Node tienen un método llamado on el cual es similar a addEventListener. Se le da el nombre de un evento y luego una función para ser llamada cuando el evento ocurra.

El siguiente código crea un servidor que lee el cuerpo del request y hace un stream de vuelta al cliente con el texto en mayúscula:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

El siguiente código, cuando corre con el servidor anterior activo, enviará un request al servidor y escribirá la respuesta que obtiene:

```
const {request} = require("http");
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

A file server

Se denomina file server a un servidor HTTP que permite acceso remoto a un sistema de archivos. Cuando se tratan archivos como recursos HTTP, los métodos HTTP tales como GET, PUT y DELETE (entre otros) pueden usarse para manejar los archivos.