
Notas de Curso

# Introducción al desarrollo Web con NodeJS y Bootstrap

## Muro Colaborativo

**M. en C. Manuel Ignacio Castillo López**

**M. en D.M. Ana Libia Eslava Cervantes**

**Dra. Selene Marisol Martínez Ramírez**

**Dr. Gustavo De la Cruz Martínez**

**23 de agosto del 2024**

**Tipo de Proyecto**

**Desarrollo**

**Financiamiento (PAPIME)**

# Título: Introducción al desarrollo Web con NodeJS y Bootstrap

## Subtítulo: Muro Colaborativo

Autores:

M. en C. Manuel Ignacio Castillo López, Facultad de Ciencias

M. en D.M. Ana Libia Eslava Cervantes, ICAT

Dra. Selene Marisol Martínez Ramírez, ICAT

Dr. Gustavo De la Cruz Martínez, ICAT

**Resumen**

Las presentes notas de curso desarrollan el contenido del *Curso de Introducción al desarrollo Web con NodeJS y Bootstrap* impartido en el ICAT como Actividad de Educación Contínua. La estructura del curso es de “arriba a abajo”, los primeros temas son generales respecto al desarrollo de aplicaciones Web, conforme progresan presentan contenido más específico hasta los últimos temas donde se presenta el uso de las tecnologías basadas en JavaScript y CSS, NodeJS y Bootstrap, para la construcción de aplicaciones. Se asume que el lector tiene conocimientos básicos de diseño de software y su implementación (particularmente con lenguajes Orientados a Objetos). El documento se organiza de la siguiente manera: el primer capítulo presenta una introducción general a las aplicaciones Web, el segundo presenta herramientas frecuentemente utilizadas para el desarrollo de aplicaciones Web, el tercer capítulo expone actividades de procesos de software para el desarrollo de aplicaciones Web. El cuarto capítulo introduce el lenguaje de programación JavaScript y los entornos de ejecución con los que se emplea en la Web, el quinto capítulo trata sobre el desarrollo de servidores Web para aplicaciones y el sexto el desarrollo de vistas dinámicas para navegadores Web.

## Índice

**Introducción..... 10**

1. Servidores y funciones remotas (back-end)..... 11

1.1. Rol de un Servidor en una Red de computadoras..... 11

1.2. World Wide Web – Una aplicación de las redes IP..... 11

1.3. Aplicaciones Web (Web apps)..... 12

1.4. Diseño y arquitectura general de una aplicación Web..... 13

1.5. Funciones remotas..... 14

1.5.1. Verbos HTTP/HTTPS..... 16

1.5.2. Códigos de respuesta HTTP/HTTPS..... 17

1.5.2.1. Códigos informativos 1XX..... 17

1.5.2.2. Códigos de éxito 2XX..... 17

1.5.2.3. Códigos de redireccionamiento 3XX..... 18

1.5.2.4. Códigos de error del cliente 4XX..... 18

1.5.2.5. Códigos de error del servidor 5XX..... 18

2. Arquitectura Modelo – Vista – Controlador (MVC)..... 18

2.1. Estructura y diseño de software..... 18

2.2. Arquitectura MVC..... 19

3. Navegadores Web y componentes generales del contenido Web (front-end)..... 20

3.1. Navegadores Web como plataforma de ejecución..... 20

3.2. Document Object Model (DOM)..... 21

3.3. Hojas de estilo..... 21

3.4. Entorno de ejecución JavaScript..... 22

4. Entornos de desarrollo integrados (IDE)..... 22

4.1. Introducción a los IDE..... 22

4.2. IDEs para el desarrollo de aplicaciones Web..... 23

5. Herramientas de trabajo colaborativo y repositorios de software..... 23
5.1. Repositorios de software..... 23
5.2. Desarrollo de software en equipo..... 24
5.3. Git..... 25
5.3.1. Inicializando un repositorio con git..... 25
5.3.2. Obteniendo un repositorio existente.....26
5.3.3. Guardando estados del proyecto. Creando versiones.....26
5.3.4. Recorriendo la historia de cambios.....27
En la mayoría de proyectos basta con emplear los primeros ocho dígitos de la firma hexadecimal del commit deseado, solo en proyectos significativamente grandes puede ser necesario usar más dígitos para distinguir unívocamente cada cambio del proyecto (Git arroja un error cuando se provee una firma que coincide con otra) (Chacon y Straub, 2020).....28
5.3.5. Ramas, creando subversiones de trabajo.....28
5.3.6. Empujando al servidor remoto..... 30
5.3.7. Desarrollo paralelo e integraciones.....31
5.3.7.1. Mezcla de ramas.....31
Ejemplo de mezcla de ramas.....32
5.3.7.2. Rebase de ramas.....32
5.3.7.3. Resolviendo conflictos de integración.....33
5.3.8. Etiquetas y lanzamientos..... 34
6. Desarrollo iterativo-incremental..... 35
6.1. Metodologías de desarrollo de software.....35
6.2. Desarrollo de software iterativo-incremental..... 35
6.3. Metodologías Ágiles..... 35
7. Desarrollo guiado por pruebas.....36
7.1. Estrategias de validación de software..... 36

7.2. Automatización en procesos de software..... 37
Ejemplo de automatización de pruebas de un servidor Web..... 37
7.3. Desarrollo guiado por pruebas..... 37
8. Desarrollo centrado en el usuario..... 39
8.1. Orígenes del Diseño de Interacción..... 39
8.2. Diseño Centrado en el Usuario..... 39
8.3. Usuarios tipo..... 39
8.4. Proceso típico de DCU..... 40
9. Programación defensiva..... 41
9.1. Manejo de entradas de datos..... 41
9.2. Programación defensiva..... 41
10. Calidad de código..... 42
10.1. Prioridades en procesos Ágiles..... 42
10.2. Calidad de código..... 43
10.2.1. Código limpio..... 43
11. Introducción a JavaScript..... 45
11.1. Netscape y la especificación ECMAScript..... 45
11.2. Características de JavaScript..... 46
12. JavaScript – Sintaxis y Operadores básicos..... 48
12.1. Declaración de variables..... 48
12.1.1. var..... 49
12.1.2. let..... 49
12.1.3. const..... 50
12.2. Tipos de datos y sus operadores..... 50
12.2.1. Objetos..... 50
12.2.1.1. Acceso..... 50

Ejemplo de uso del operador de acceso....50
Ejemplo de uso de un objeto como arreglo asociativo.... 51
12.2.1.2. Arreglos....51
Ejemplos de manipulación de arreglos....52
12.2.1.3. Funciones.... 52
12.2.1.4. Quitando miembros de un objeto....53
Ejemplo de remoción de miembros de un objeto....53
12.2.2. Tipos numéricos.... 53
12.2.3. Cadenas de caracteres.... 53
12.2.4. Booleanos.... 54
12.2.5. undefined....54
12.2.6. Preguntando por el tipo de un valor.... 54
12.3. Valores de verdad....54
12.3.1. Operadores lógicos.... 54
12.4. Paso de valores.... 55
Ejemplo de manipulación de un valor apuntado por dos variables diferentes.... 55
12.5. Operadores de control de flujo.... 55
12.5.1. Control de flujo secuencial.... 55
Ejemplo de bloque switch....56
12.5.2. Ciclos....56
Ejemplo de iteración de estructuras con FOR....57
12.6. Precedencia de operadores....57
13. Manejo de errores en JavaScript.... 58
13.1. Excepciones.... 58
Ejemplo de disparo de una excepción....59
13.2. Bloques try – catch.... 59
Ejemplo de mecanismo de recuperación para una excepción específica....60

14. Funciones de primer orden y anónimas.....60
14.1. Declaración de funciones.....60
Ejemplo de función con parámetros opcionales.....61
Ejemplo de función de aridad arbitraria.....62
14.2. Funciones anónimas.....62
Ejemplo de función anónima.....62
14.3. Funciones de primer orden.....64
14.4. Alcances y cerraduras.....64
Ejemplo de función que produce funciones.....65
15. Convenciones de codificación de JavaScript.....66
15.1. JavaScript Object Notation – JSON.....66
15.2. Estilo de codificación preferido en JavaScript.....67
15.2.1. Nomenclatura de archivos y directorios.....67
15.2.2. Nomenclatura de identificadores en JavaScript.....67
15.2.3. Sangría (indentación) y separación de bloques de código.....67
15.2.4. Comentarios.....69
Ejemplo de uso de comentarios multi-línea.....70
15.2.5. Paréntesis, corchetes y bloques de código.....70
15.2.6. Sentencias.....71
16. Inspectores de código y pruebas automáticas de software.....71
16.1. Actividades de control de calidad en procesos Ágiles.....71
16.2. Inspectores de código automáticos.....72
16.3. Pruebas automáticas de software.....72
17. Servicios y bibliotecas que integran a NodeJS como plataforma de ejecución...73
17.1. Introducción a NodeJS.....73
17.1.1. Configuración del entorno de ejecución.....73
17.1.2. Características de NodeJS.....74

17.2. Diseñando con el Ciclo de Eventos..... 74
17.3. Módulos..... 76
Ejemplo de módulo con NodeJS..... 76
17.3.1. Importando dependencias..... 76
17.4. Node Package Manager – npm..... 77
17.4.1. Express..... 77
17.4.2. Creando un repositorio de proyecto de Aplicación Web..... 77
17.4.3. Separando dependencias del producto de las de desarrollo..... 79
17.5. Bibliotecas y servicios de NodeJS..... 80
17.5.1. Emisores de eventos..... 80
17.5.2. Promesas ECMAScript v6..... 81
18. Definición de rutas (end-points)..... 82
18.1. Configuración general de una aplicación Web con NodeJS..... 82
18.2. Definición de ruteadores..... 83
Ejemplo de ruteador general con Express..... 84
18.2.1. Parámetros en la ruta..... 85
Ejemplo de definición de rutas parametrizadas con Express..... 85
18.3. Construcción de respuestas de mensajes HTTP..... 86
Ejemplo de generación de respuesta a una petición Web con Express..... 86
18.4. Definición de respuestas de error..... 87
Ejemplo de manejo de errores al procesar peticiones con middlewares..... 88
18.5. WebSockets..... 88
Ejemplo de ruteador WebSockets para una vista Web (en navegador)..... 90
19. Definición de Controladores y Modelos..... 91
19.1. Modelos..... 91
Ejemplo de un modelo simple de usuario..... 91
19.2. Controladores..... 92

Ejemplo de un controlador simple..... 92
19.3. Middleware..... 93
Ejemplo simplificado de middleware que comprueba sesiones de usuario..... 94
20. Elementos DOM..... 94
20.1. Representación de HTML/XHTML en tiempo de ejecución..... 94
20.2. Manipulación de vistas Web en tiempo de ejecución..... 96
Ejemplo de manipulación de una vista con JavaScript en el navegador.....96
Ejemplo de manipulación de nodos DOM con JavaScript en el navegador.....97
20.3. Personalizando la presentación de vistas Web..... 98
21. Bibliotecas para el desarrollo de vistas Web..... 100
21.1. Bibliotecas para vistas del lado del cliente..... 100
21.2. JQuery – JavaScript para manipulación de DOM.....100
21.2.1. Importando JQuery como dependencia de las vistas Web..... 101
21.2.2. Selectores – Objetos JQuery..... 102
Ejemplo de manipulación de presentación de vistas con JQuery..... 104
21.3. Bibliotecas para vistas del lado del servidor.....104
21.4. Plantillas de JavaScript embebido.....105
21.4.1. Definiendo vistas en archivos .ejs..... 105
Ejemplo de vista ejs.....106
21.4.2. Creando plantillas para las vistas.....107
Ejemplo de plantilla para crear un pie de página estandarizado..... 107
21.4.3. Creando vistas dinámicas..... 108
Ejemplo de vista dinámica con contenido que depende del estado de un modelo.. 108
22. Peticiones AJAX.....109
22.1. JavaScript asíncrono con XML..... 109
22.2. Conexiones abiertas y Peticiones discretas..... 109
22.3. Usando AJAX a través de JQuery..... 110

Ejemplo de uso de peticiones AJAX con JQuery para manejar formularios..... 111
23. Bootstrap y hojas de estilo CSS..... 113
23.1. Jerarquía de aplicación de selectores de estilos..... 113
23.1.1. Incluyendo hojas de estilo en las vistas..... 114
23.2. Bootstrap..... 114
23.2.1. Integrando Bootstrap en una aplicación NodeJS..... 115
23.2.2. Personalizando vistas con Bootstrap..... 116
Ejemplo de uso de Bootstrap para uniformar y estilizar la presentación de vistas... 116
**Agradecimientos..... 117**
**Referencias Bibliográficas..... 118**

## Introducción

El grupo ESIE ha desarrollado varios proyectos en el marco de los proyectos del Aula del Futuro y la Red de Aulas del Futuro, algunos de los cuales se basan en tecnologías Web. Estos proyectos son desarrollados con estudiantes, principalmente como parte de sus actividades de Servicio Social y egreso de licenciatura. En particular, el proyecto del *Muro Colaborativo* ha desarrollado una aplicación Web que permite a profesores crear espacios virtuales en los que otros usuarios puedan colaborar en el desarrollo, organización o presentación de ideas por medio de un tablero similar a un *collage* como se muestra en la figura 1.

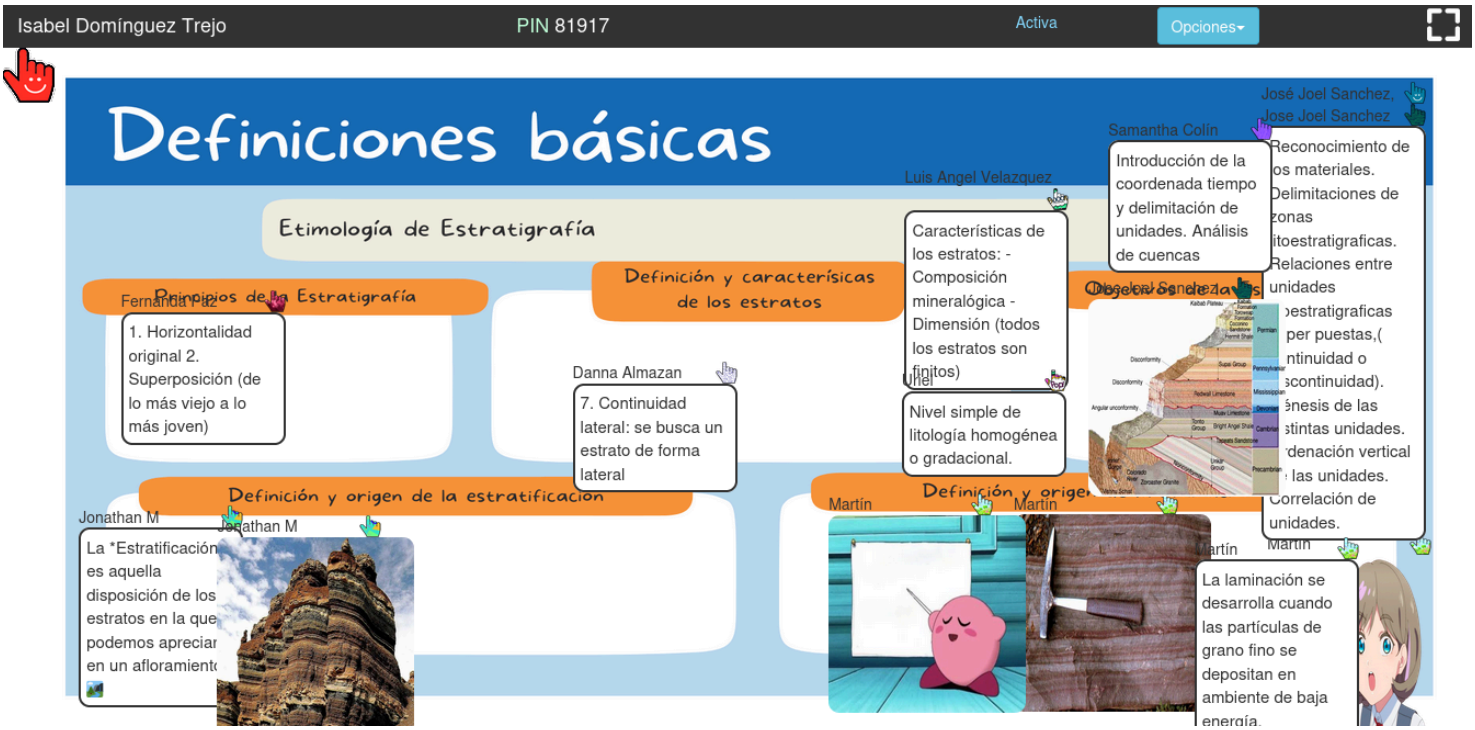


Figura 1. Instancia del *Muro Colaborativo* como herramienta didáctica.

El *Muro Colaborativo* ha evolucionado continuamente buscando ofrecer herramientas y modos de uso competitivos como herramienta didáctica. Buscando mejorar el ritmo de trabajo y reducir el esfuerzo que deben invertir los estudiantes para familiarizarse con el proyecto, el grupo ESIE ha decidido crear un curso introductorio a las tecnologías en las que se basa el *Muro Colaborativo*.

Este documento presenta las notas correspondientes al contenido de dicho curso. Presenta conceptos y prácticas básicas para el desarrollo de aplicaciones Web con NodeJS y Bootstrap. La presentación del contenido sigue el esquema “arriba hacia abajo”, comenzando por los aspectos más generales del desarrollo Web, concretando los temas hasta presentar el uso de las herramientas de trabajo y codificación para el desarrollo de esta clase de aplicaciones.

El contenido de este curso asume que los estudiantes poseen habilidades básicas de programación y modelado de procesos, así como conocimientos básicos en torno a diseño y documentación de software con algún lenguaje funcional u orientado a objetos (Programación Orientada a Objetos, POO).

# 1. Servidores y funciones remotas (back-end)

## 1.1. Rol de un Servidor en una Red de computadoras

En el contexto de redes de computadoras un servidor es un integrante de la red que se encarga de manejar peticiones realizadas por otros miembros de la red. A estos miembros solicitantes se les llama *clientes* (Kurose y Ross, 2013).

Existen muchas clases de servidores. Una taxonomía común es considerar diferentes aplicaciones de las redes de computadoras, en particular del Internet. Una de las principales aplicaciones del Internet es la *Web*. En la jerga común, Internet y Web se usan de forma indiferente pese a que la Web es un subconjunto del Internet: no todos los mensajes en el Internet pueden ser interpretados por un *cliente* o *servidor Web*, pero todas las peticiones *Web* son parte del Internet (Kurose y Ross, 2013).

## 1.2. *World Wide Web* – Una aplicación de las redes IP

Los mensajes *Web* se caracterizan por emplear el protocolo HTTP, o su variante que admite la transferencia de datos segura mediante llaves públicas y privadas HTTPS. Bajo este protocolo dos miembros de la Web se comunican mediante mensajes cuyo formato se apega a la especificación de HTTP. Esta especificación establece claramente cómo son los mensajes que espera recibir un *servidor* y cómo son los

mensajes con los que responde. De esta manera cualquier *cliente* o *servidor* desarrollados de manera independiente *pueden* interpretar los mensajes de otros *clientes* y *servidores* sin importar su origen (Kurose y Ross, 2013).

### 1.3. Aplicaciones Web (Web apps)

Por su parte una *aplicación Web* es una aplicación distribuida que se compone por las siguientes partes (pero no se limita a solo estás) (Kurose y Ross, 2013):

- Interfaces de usuario o *Vistas* que se representan como Documentos HTML. HTML es una lenguaje de marcado similar a XML, en el que la información del documento se organiza mediante etiquetas que modelan un árbol de datos: la etiqueta raíz del documento representa la raíz del árbol, cada etiqueta anidada dentro de otra es un sub-árbol de la etiqueta en que se anida. La figura 2 muestra un documento HTML y su representación como vista.
    - Las Vistas suelen incluir *hojas de estilo*, que incluyen información sobre *cómo* deben visualizarse los elementos definidos en el Documento HTML.

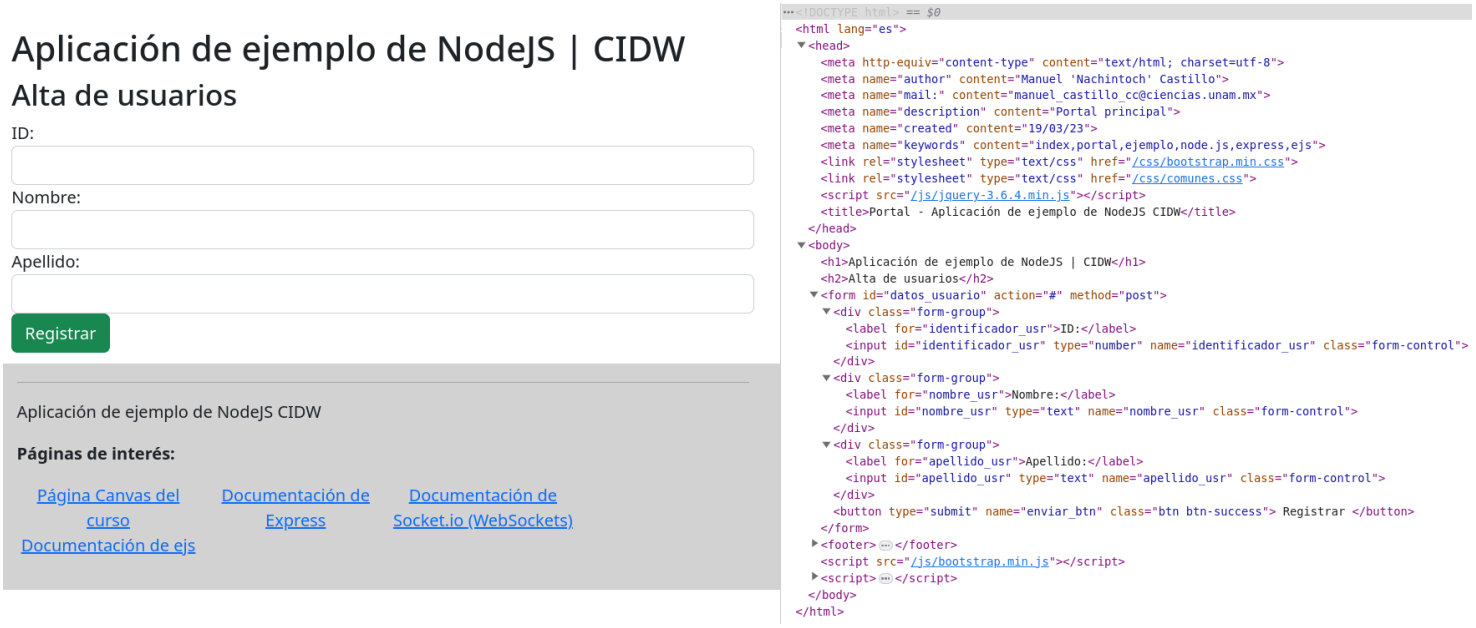


Figura 2. A la izquierda aparece una vista Web y a la derecha su fuente en HTML. Note que algunos de los elementos HTML están colapsados.

    - También suelen incluir *scripts* que permiten interactuar con el contenido del Documento HTML, de manera que pueda hacerse *dinámico e interactivo*.

- Un Servidor Web que utiliza rutas y distintos tipos de mensajes HTTP para accionar las distintas funcionalidades que ofrece.

### 1.4. Diseño y arquitectura general de una aplicación Web

La naturaleza distribuida de una aplicación Web requiere que el diseñador considere con atención especial a qué parte o componente de la aplicación le debe corresponder cada responsabilidad. Para esto existen varios patrones de diseño para la Web, estos suelen ser implementados por medio de marcos de trabajo (*frameworks*) y bibliotecas para esta clase de aplicaciones (Bass et al., 2013; Fowler, 2003).

Un patrón ampliamente utilizado para el desarrollo de aplicaciones Web es la arquitectura Modelo-Vista-Controlador (MVC), en la que se contemplan 3 roles primarios (Fowler, 2003):

- Modelos que representan la información que maneja la aplicación.

  Usualmente estos modelos son *clases* (en el sentido de POO) que representan colecciones (tablas) definidas en una base de datos.

- Las Vistas son Interfaces Gráficas de Usuario que permiten interactuar con los componentes que conforman la aplicación.

- Por último los Controladores son componentes que ofrecen servicios o funciones que usualmente son explotados por medio de las Vistas, su ejecución suele modificar Modelos.

De esta manera en una aplicación Web los Modelos y Controladores se distribuyen entre los componentes remotos de la aplicación que ofrecen servicios a los usuarios, cómo un sistema de correo electrónico, una base de datos, o el mismo servidor de la aplicación. Estos componentes remotos (usualmente llamados *back-end*) pueden ser

distribuidos y/o replicados en múltiples sistemas anfitriones (Fowler, 2003; Kurose y Ross, 2013).

Por su parte las Vistas se ejecutan del lado del cliente, en el navegador Web que se utilice para usar la aplicación, a este entorno se le llama *front-end* (Fowler, 2003; Kurose y Ross, 2013).

### 1.5. Funciones remotas

Las funciones más significativas de un servidor Web se concentran en sus Controladores. Sin embargo los controladores no suelen estar expuestos a la Web por si mismos, sino que sus funciones son solicitadas por componentes del servidor que sí están expuestos a los mensajes de los clientes (Fowler, 2003; Kurose y Ross, 2013).

Usualmente se utiliza un *ruteador*, un programa que se encarga de decidir qué componente del servidor corresponde a cada petición Web. Recordemos que la Web se basa en el protocolo HTTP/HTTPS para intercambiar información, estas peticiones tienen el siguiente formato (Kurose y Ross, 2013):

- **Línea de petición**. <verbo HTTP> <URL> <versión HTTP>.

  La línea de petición describe la información más básica del intercambio de datos vía HTTP.

  - El verbo HTTP describe la acción que deseamos se lleve a cabo por parte del servidor, como puede ser subir contenido nuevo o borrar contenido existente.
  - La URL indica al servidor el recurso al que queremos tener acceso o sobre el que queremos que realice alguna acción.
  - La versión de HTTP/HTTPS le indica al servidor la versión del protocolo que soporta el cliente (y la versión del protocolo bajo la que espera recibir una respuesta).

- **Líneas de encabezado**. Incluyen meta-información respecto a la petición. Entre otros puede incluir: si se desea mantener la conexión abierta, el lenguaje en el

que se espera recibir el contenido (inglés, español, alemán…), el navegador y sistema operativo que ejecuta el cliente.

- **Cuerpo de la entidad**. Segmento opcional de un mensaje HTTP. Incluye recursos asociados a la petición, como puede ser una cookie, texto introducido en un formulario o un archivo. No todas las consultas HTTP necesitan de este segmento y por eso es opcional. Considere por ejemplo una petición para obtener un portal o *home page* de una página Web por primera vez, usualmente esta petición no incluye información de sesión ni enviar un formulario.

La respuesta de un mensaje HTTP tiene un formato muy similar al mensaje de petición, salvo la denominación de la primera línea "*estado"* que posee una estructura similar a la línea de petición. En lugar de la URL del recurso solicitado aparece un código de respuesta. Estos son códigos numéricos entre 100 y 599; cada centena tiene un significado estándar y nos ayudan a determinar la manera en que el servidor procesa la petición (o si no pudo hacerlo) (Kurose y Ross, 2013).

Las figuras 3 y 4 muestran mensajes HTTP de ejemplo, la figura 3 muestra un mensaje de petición y 4 muestra su respuesta.

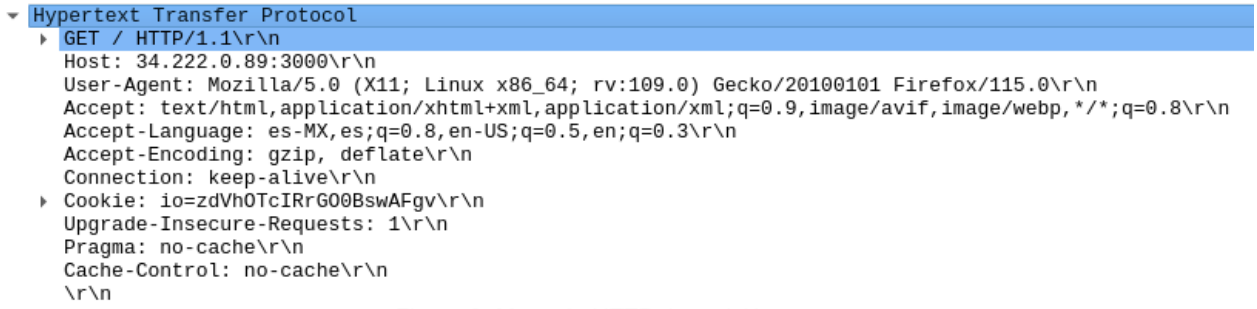


Figura 3. Mensaje HTTP de petición.


Figura 4. Mensaje HTTP de respuesta.

### 1.5.1. Verbos HTTP/HTTPS

- **GET** – Indica que se desea obtener un recurso, obtener la “página Web” en la URL de la petición (Kurose y Ross, 2013).
- **POST** – También indica que se desea obtener un recurso, pero el contenido depende de información provista por el usuario (Kurose y Ross, 2013).
- **HEAD** – De forma similar a **GET**, indica que se desea obtener un recurso, pero la respuesta a una petición **HEAD** únicamente contiene el encabezado del mensaje HTTP/HTTPS, dejando de lado el recurso solicitado. Suele usarse con propósitos de documentación o depuración (Kurose y Ross, 2013).
- **PUT** – Solicita la publicación de un recurso en el servidor, usualmente alojándolo en la URL indicada en la petición (Kurose y Ross, 2013).
- **DELETE** – Solicita la eliminación de un recurso en el servidor (Kurose y Ross, 2013).

Las peticiones **GET** suelen ser las más empleadas y aunque no especifican la entrega de datos por parte del cliente, es posible enviar datos al servidor por medio de la URL, ya que en su forma general (figura 5) permiten agregar *fragmentos* de información o *consultas* que incluyen una serie de claves y sus valores asociados (Kurose y Ross, 2013).

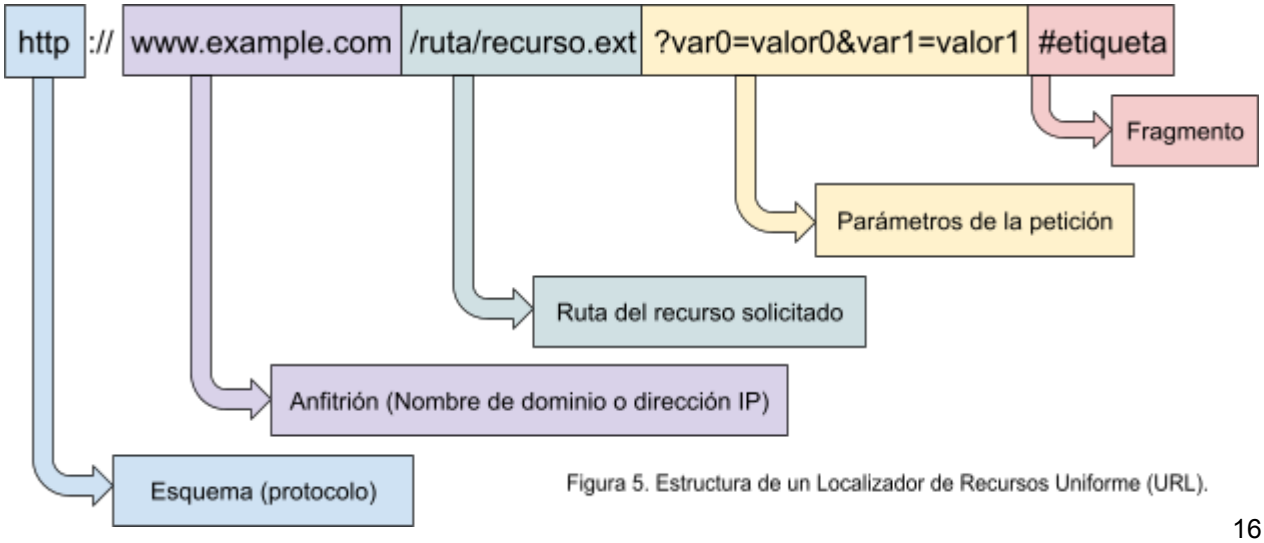


Figura 5. Estructura de un Localizador de Recursos Uniforme (URL).

Se envían datos en una petición **GET** cuando el recurso solicitado varía dependiendo de los datos del usuario. No confunda esto con la intención de una petición **POST**, estas indican que el recurso deseado depende o requiere de información del usuario para poder responder (cómo en un inicio de sesión). Cuando se envían datos en una petición **GET** usualmente se hace para *afinar* la búsqueda o distinción de aspectos específicos del recurso deseado (como son los parámetros de un buscador Web) (Kurose y Ross, 2013).

También es importante considerar la diferencia entre **POST** y **PUT**. **PUT** no se asocia con la obtención de un recurso sino con la publicación de uno, por lo que **PUT** resulta inapropiado para subir información a un servidor esperando una respuesta específica. Por esto el envío de formularios suele realizarse con el verbo **POST**, pues el usuario espera ver una vista de confirmación. Los resultados de una petición **PUT** pueden reflejarse en la misma vista en la que el usuario solicitó la publicación (Kurose y Ross, 2013).

### 1.5.2. Códigos de respuesta HTTP/HTTPS

#### 1.5.2.1. Códigos informativos 1XX

Se usan para notificar al cliente de algún evento por parte del servidor que requiere una acción del cliente. Por ejemplo, ignorar una respuesta (100) o cambiar el protocolo HTTP por HTTPS (101, también puede solicitar cambiar por otro protocolo diferente) (Kurose y Ross, 2013).

#### 1.5.2.2. Códigos de éxito 2XX

Indican que la petición es procesada con éxito. Puede indicar que la acción ha concluido exitosamente (200), que ha sido aceptada y está encolada para ocurrir en el futuro (201), o que el servidor espera que el cliente vacíe los datos provistos en la vista que solicitó la consulta (205), en general indica que la petición es atendida (Kurose y Ross, 2013).

#### 1.5.2.3. Códigos de redireccionamiento 3XX

Se usan para enviar las peticiones a una URL diferente, usualmente como resultado de migrar funciones en el servidor. Esto permite que los clientes que utilicen las URL antiguas aún puedan acceder a las funciones del servidor (Kurose y Ross, 2013).

#### 1.5.2.4. Códigos de error del cliente 4XX

Indica que el servidor no puede procesar la consulta del cliente. Por ejemplo puede indicar que el mensaje de consulta HTTP está mal-formado (400), que se intenta acceder a un recurso protegido (401), que se intenta obtener un recurso que no existe (404), o bien que el servidor cuenta con una tetera y se niega a preparar café con ella (418) (Kurose y Ross, 2013).

#### 1.5.2.5. Códigos de error del servidor 5XX

Indica que al procesar la consulta el servidor encontró un problema y no puede continuar. Puede indicar un error genérico (500), que el recurso solicitado corresponde con una función contemplada en la especificación del servidor pero que aún no ha sido implementada (501) o que el servidor está funcionando pero no disponible (503) (Kurose y Ross, 2013).

## 2. Arquitectura Modelo – Vista – Controlador (MVC)

### 2.1. Estructura y diseño de software

Al desarrollar software se debe considerar que no opera de manera aislada sino que debe contar con interfaces que permitan proporcionar datos de entrada y/o conocer los resultados de su operación. Incluso los sistemas digitales más simples requieren interfaces de entrada y salida, un reloj digital por ejemplo necesita una interfaz de entrada que permita ajustar la hora y una interfaz de salida que permita conocer la hora actual.

El diseño de estas interfaces establece mecanismos para proveer las entradas que necesita el sistema y poder recuperar los resultados de su operación de manera que los usuarios u otros mecanismos puedan interactuar con el software sin introducir

restricciones adicionales a la operación (Clements et al., 2011; Fowler, 2003; Martin y Martin, 2007).

Para alcanzar estos objetivos contamos con patrones y recomendaciones para el diseño de interfaces de software. La organización de responsabilidades tiene un rol positivo en el diseño de interfaces de software por lo que se agrupan componentes *cohesivamente*, las partes del software se agrupan por funcionalidades y responsabilidades similares o directamente relacionadas (Clements et al., 2011; Fowler, 2003; Martin y Martin, 2007).

Esta organización ayuda a identificar la manera en que los distintos componentes del software interactúan. Estudiar y refinar este diseño puede ayudar a identificar riesgos que pueden ocurrir durante la implementación y considerar cambios al diseño o contemplar medidas en caso de que se presente alguna situación de riesgo. También ayuda a identificar aspectos del diseño que son propensos a cambios y considerar puntos de flexibilidad que permitan cambiar o extender los componentes necesarios (Clements et al., 2011; Fowler, 2003; Martin y Martin, 2007).

## 2.2. Arquitectura MVC

En el capítulo anterior se presentó la arquitectura MVC con sus tres componentes, los cuales aparecen en la figura 6 (Fowler, 2003):

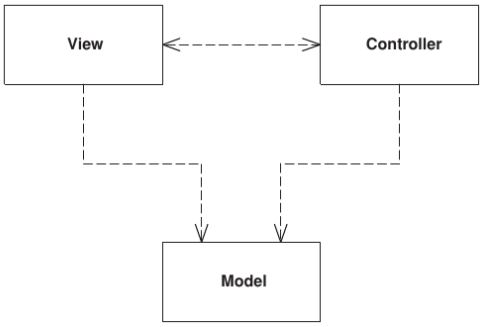

Figura 6: Modelo de la arquitectura MVC (Fowler, 2003)

- Modelos que representan la información que maneja el software.
- Controladores que atienden las acciones del usuario y actualizan las Vistas y Modelos conforme a los resultados de las operaciones.
- Vistas que le permiten al usuario interactuar con el software.

Esta arquitectura separa la *presentación* del *modelo* y el *controlador* de la *vista*. La separación de la *presentación* del *modelo* es una de las heurísticas fundamentales del

diseño de software puesto que tienen propósitos muy diferentes, separarlos ayuda a diseñar mecanismos aptos para los diferentes propósitos que persiguen los componentes de la presentación y los componentes del modelo (Fowler, 2003).

Por su parte los usuarios necesitan acceder a información consistente del modelo sin importar la vista desde la que se les presente esa información. La separación de estos elementos también permite diseñar múltiples vistas que permiten visualizar los mismos modelos. Otro uso de esta estrategia de diseño es ofrecer varios tipos de interfaces: podemos tener interfaces basadas en Web, en líneas de comando, por medio de API (del inglés *Application Programming Interface*, Interfaz de Programación de Aplicaciones), etc. Esto implica una dependencia: la presentación (vista) depende del modelo, pero el modelo es independiente de la presentación (Fowler, 2003).

El comportamiento de componentes no gráficos es distinto del comportamiento de componentes gráficos, por lo que otra ventaja de separar la presentación del modelo es que pueden ser probados de forma separada: los modelos pueden ser probados mediante revisiones de código o mediante pruebas automatizadas, mientras que la presentación puede ser probada mediante enfoques centrados en el usuario (Fowler, 2003).

## 3. Navegadores Web y componentes generales del contenido Web (front-end)

### 3.1. Navegadores Web como plataforma de ejecución

Los navegadores Web ofrecen diversos servicios para una interacción enriquecida mediante tres componentes primarios que procesan y despliegan contenido al usuario, permitiendo también modificar el contenido en cualquier momento para habilitar vistas dinámicas además de interactivas (Dean, 2019).

Estos tres componentes son: el Modelo de Objetos del Documento (DOM por sus siglas en inglés), hojas de estilo en cascada (CSS por sus siglas en inglés) y un entorno de ejecución *JavaScript* (Dean, 2019). Estos componentes se describen en las siguientes secciones.

### 3.2. Document Object Model (DOM)

Es un API que representa el contenido de un documento HTML. Puede interpretarse como un árbol (como estructura de datos) que tiene como nodo raíz la etiqueta *html* y dos subárboles: *header* y *body* (Dean, 2019). La figura 7 muestra un ejemplo de la representación DOM de un documento HTML.

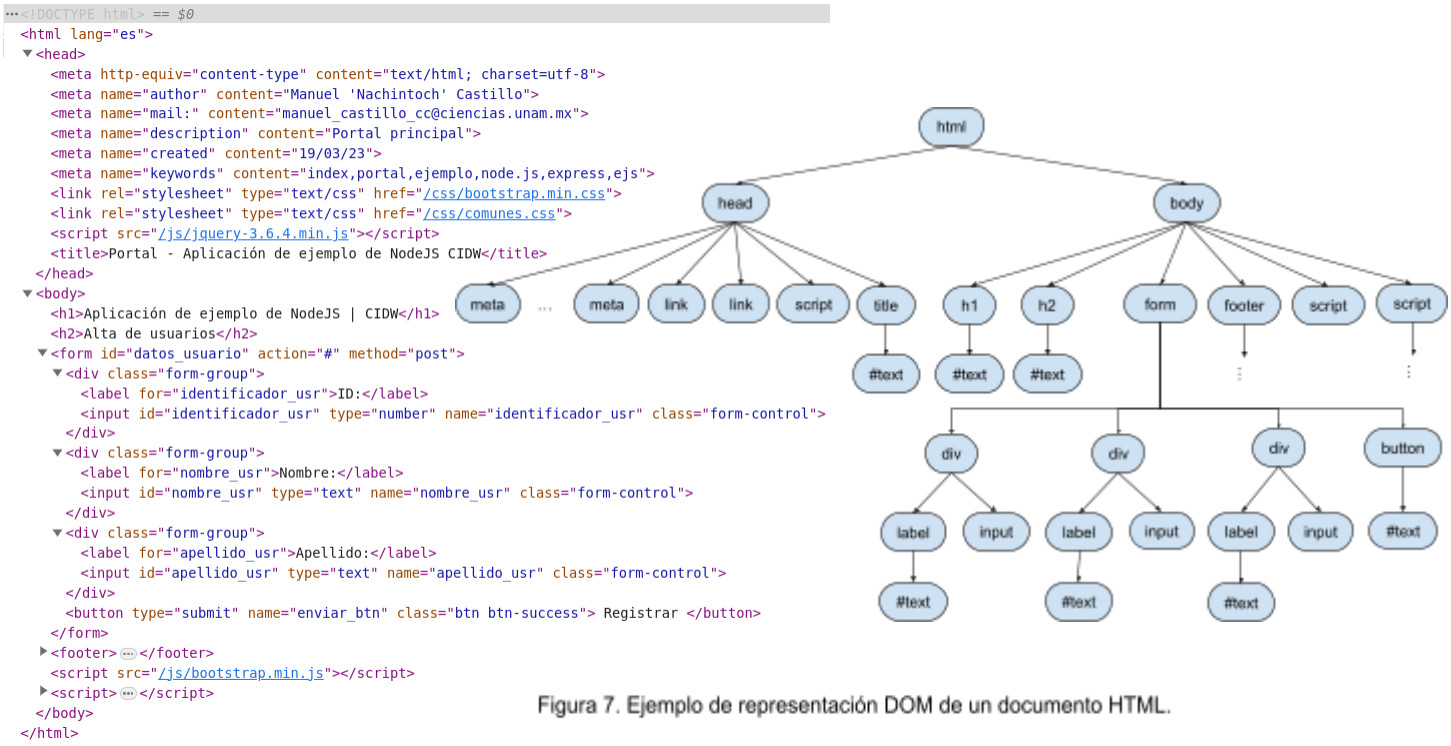


Figura 7. Ejemplo de representación DOM de un documento HTML.

La especificación DOM permite que diferentes navegadores puedan interpretar y mostrar cualquier documento *html* que se le presente (Dean, 2019).

### 3.3. Hojas de estilo

Las hojas de estilo contienen una serie de reglas que especifican la presentación de los elementos DOM como partes de una vista. Estas reglas se agrupan en clases que pueden definirse con un nombre arbitrario, por el tipo de nodo DOM sobre el que deben tener efecto (como `p`, `div` o `h1`) o por identificador del nodo DOM sobre el que deben tener efecto. También es posible crear clases que se deben aplicar sobre varias clases de elementos DOM con determinadas propiedades (Dean, 2019).

### 3.4. Entorno de ejecución JavaScript

La representación DOM del documento HTML y las clases de estilo que tienen efecto sobre ellas pueden ser alteradas de forma dinámica con JavaScript, un lenguaje multi-paradigma de tipos débiles con funciones de primera clase que hereda amplias características de lenguajes funcionales. Pese a su nombre JavaScript es más similar a *Lisp* y *Scheme* que a *Java* o *C* (Crockford, 2008).

Existen entornos que envuelven a JavaScript para desarrollar vistas dinámicas para la Web, estos entornos producen archivos JavaScript que los navegadores pueden ejecutar. Existen muchas críticas hacia JavaScript, su lanzamiento original se considera prematuro y existen muchas características negativas en el lenguaje, como la existencia de dos operadores de equivalencia (uno de los cuales tiene un comportamiento no reflexivo ni transitivo), comparte un espacio de memoria global para todos los scripts en ejecución (lo que puede resultar en conflictos entre ellos), la capacidad de terminar sentencias con final de línea o punto y coma (puede generar inconsistencias) o la existencia de múltiples valores *vacios* (`undefined` y `null`) (Crockford, 2008).

Pese a los problemas y críticas en torno a JavaScript, su rápida adopción desde su lanzamiento y su uso continuo son evidencia de su utilidad. Mientras que otras tecnologías como los Java Applets o Adobe Flash llegaron a poseer una enorme relevancia para el desarrollo Web, JavaScript es una de las pocas que sigue vigente y es el lenguaje universal para el desarrollo de interfaces Web dinámicas (Crockford, 2008).

## 4. Entornos de desarrollo integrados (IDE)

### 4.1. Introducción a los IDE

Los Entornos de Desarrollo Integrados o IDE por sus siglas en inglés (*Integrated Development Environment*) son herramientas de desarrollo de software que *integran* una serie de funciones y servicios para la codificación de software. Su función más básica es la de un editor de texto, normalmente integran: un compilador, una consola de comandos y/o REPL (Bucle de Lectura-Evaluación-Impresión; *Read-Evaluate-Print*

*Loop*), un depurador (debugger), un manejador de control de versiones y herramientas de refactorización (DelBono, 2016; Sommerville, 2011).

La mayoría de los IDE ofrecen capacidades de extensión mediante las que pueden aumentar sus capacidades, cómo el soporte de plataformas de desarrollo adicionales, gestión de máquinas virtuales, gestión de dependencias del proyecto, herramientas de calidad de código, herramientas de Integración Continua y Lanzamiento Continuo; entre otras (Sommerville, 2011).

### 4.2. IDEs para el desarrollo de aplicaciones Web

Para el desarrollo Web se recomienda utilizar un IDE con soporte explícito para HTML, CSS y Javascript, como son (DelBono, 2016):

- *Visual Studio Code* – IDE ligero desarrollado por Microsoft. Está basado en Electron, que es una plataforma de desarrollo de aplicaciones con NodeJS, por lo que VS Code está predispuesto para el desarrollo de aplicaciones en esta plataforma.

  https://code.visualstudio.com/

- *WebStorm* – IDE que permite automatizar las tareas de codificación desarrollado por JetBrains, al igual que *IntelliJ* (IDE para Java y Kotlin), en el que está basado *Android Studio*, por lo que *WebStorm* comparte las funcionalidades e interfaces de estos IDE pero con soporte especializado para las tecnologías Web, está orientado al desarrollo de aplicaciones Web y soporta JavaScript de manera nativa.

  https://www.jetbrains.com/webstorm/

## 5. Herramientas de trabajo colaborativo y repositorios de software

### 5.1. Repositorios de software

Un repositorio de software es un compendio de artefactos de software. A su vez los artefactos de software son aquellos que forman parte de un proyecto de software, como

es el código fuente, documentación, métricas, registros de incidentes, registros de un manejador de versiones, etc (Luzgin y Kholod, 2020).

Existen varias herramientas en línea para gestionar repositorios de software que permiten acceso constante y remoto a los miembros del equipo de trabajo, además de incorporar diversas herramientas comúnmente utilizadas principalmente en metodologías Ágiles, como herramientas para Integración Continua y Lanzamiento Continuo, o asignación de roles y niveles de acceso para los miembros del equipo (Sommerville, 2011).

### 5.2. Desarrollo de software en equipo

Los repositorios de software en línea impulsan el desarrollo de software en equipo. Permiten que varios desarrolladores construyan un producto de software sin necesidad de tener que compartir el mismo entorno de desarrollo, el mismo espacio físico o de tiempo (Sommerville, 2011).

Desarrollar software en equipo permite fragmentar las responsabilidades y repartirlas entre varias personas. Esto permite a los miembros ocupar distintos roles conforme a sus preferencias y experiencia. En metodologías Ágiles se contempla que un mismo integrante del equipo puede ejercer varios roles, esto fomenta la comunicación entre integrantes del equipo puesto que participarán en varios de sus aspectos en lugar de especializarse en alguno de ellos (Martin y Martin, 2007).

En estas metodologías también se prefiere que el equipo de trabajo sea lo más pequeño posible, lo que depende del tipo de proyecto, sus requerimientos y los recursos con los que cuente. Se busca que los miembros del equipo ejerzan roles en los que tengan experiencia suficiente (o en los que puedan ser capacitados oportunamente) y en los que se sientan cómodos ejerciendo (Martin y Martin, 2007).

El equipo debe tener comunicación constante. Se debe dar seguimiento puntual al progreso y problemas. Además debe haber canales de comunicación permanentes (Martin y Martin, 2007).

### 5.3. Git

Git es uno de los manejadores de versiones más populares. La funcionalidad básica para descubrir cambios de Git fue diseñada por Linus Torvalds, autor original del kernel Linux (Chacon y Straub, 2020; Torvalds, 2005a, 2005b, 2005c, 2005d).

#### 5.3.1. Inicializando un repositorio con git

El comando para inicializar un repositorio de git es (Chacon y Straub, 2020):

```
git init
```

Al inicializar el repositorio también suele crearse un archivo oculto llamado `.gitignore`. Este archivo puede contener nombres y patrones de archivos y directorios que son ignorados por git (Chacon y Straub, 2020).

Para terminar la preparación del repositorio local se define la ruta de al menos un repositorio remoto. Esto puede hacerse de muchas formas diferentes, lo más usual es dar de alta un proyecto en un servicio de hospedaje de proyectos basados en Git (como Github, Gitlab, Bitbucket, entre otros) (Chacon y Straub, 2020).

Una vez que cuente con la URL del repositorio remoto, en cualquier directorio del repositorio local se usa el siguiente comando para definir el remoto:

```
git remote add origin <URL del repositorio remoto>
```

Este comando contiene varias partes (Chacon y Straub, 2020):

- `git` - Comando principal de git.
- `remote` - Indica que la operación está relacionada a un repositorio remoto.
- `add` - Indica que se agrega un repositorio remoto.
- `origin` - Es el nombre del repositorio remoto. No es necesario que se llame *`origin`*, usualmente el único o principal repositorio remoto se suele llamar así. Es una buena práctica seguir convenciones y modismos ampliamente aceptados o establecidos por las organizaciones a las que pertenecen los equipos de trabajo.

### 5.3.2. Obteniendo un repositorio existente

Si el proyecto ya ha sido creado anteriormente (sea por uno de nuestros compañeros de equipo o por uno mismo), es posible clonarlo en uno o más entornos de trabajo usando el siguiente comando (Chacon y Straub, 2020):

```
git clone <URL del repositorio remoto> <directorio para el proyecto>
```

### 5.3.3. Guardando estados del proyecto. Creando versiones

Una vez que se han realizado cambios en el repositorio se debe hacer un *commit*. Un commit es como hacer una fotografía del estado actual del proyecto. Se identifican con un número hexadecimal generado automáticamente y un comentario que el desarrollador debe introducir manualmente (Chacon y Straub, 2020).

Antes de hacer un commit es recomendado conocer los cambios del repositorio para evitar introducir cambios indeseados. Para conocer los cambios desde el último commit se usa el comando (Chacon y Straub, 2020):

```
git status
```

Esto muestra una lista de archivos con seguimiento y sin seguimiento para el siguiente commit. Únicamente los cambios en archivos con seguimiento se agregan al commit. Para seguir los cambios de un archivo se emplea el comando (Chacon y Straub, 2020):

```
git add <ruta a un archivo o directorio a seguir>
```

También es posible seguir todos los cambios en un solo comando (excluyendo los archivos ignorados según `.gitignore`) (Chacon y Straub, 2020):

```
git add .
```

Puede usar `git status` para verificar que ha seguido todos los cambios que se desean agregar al siguiente commit. Si detecta que ha seguido algún archivo que no desea incluir en el commit, puede quitarle el seguimiento con (Chacon y Straub, 2020):

```
git rm --cached <archivo a quitar seguimiento>
```

Una vez que se verifique que el estado del repositorio local es adecuado para hacer commit, se aplica el commit con el comando (Chacon y Straub, 2020):

```
git commit -m “<mensaje del commit>”
```

El mensaje del commit debe ser corto y concreto, indicando en pocas palabras los cambios que se incluyen en el commit. No es recomendado acumular demasiados cambios entre commits. Si en algún momento desea volver a alguna versión anterior y los cambios entre cada commit son demasiados, puede tener dificultades para recuperar el estado deseado del repositorio. Las integraciones de cambios también pueden ser más complejas cuando los commits varían mucho entre sí (Chacon y Straub, 2020).

### 5.3.4. Recorriendo la historia de cambios

Es posible recorrer la historia de cambios en el proyecto, para ello se usa el histórico de commits realizados. Es posible emplear el identificador de un commit del que se quiere recuperar el estado del repositorio. Los servicios de hospedaje de repositorios de software suelen ofrecer interfaces gráficas para conocer el histórico de los repositorios que hospedan (figura 8). También es posible consultar el histórico del repositorio usando el comando (Chacon y Straub, 2020):

```
git log
```

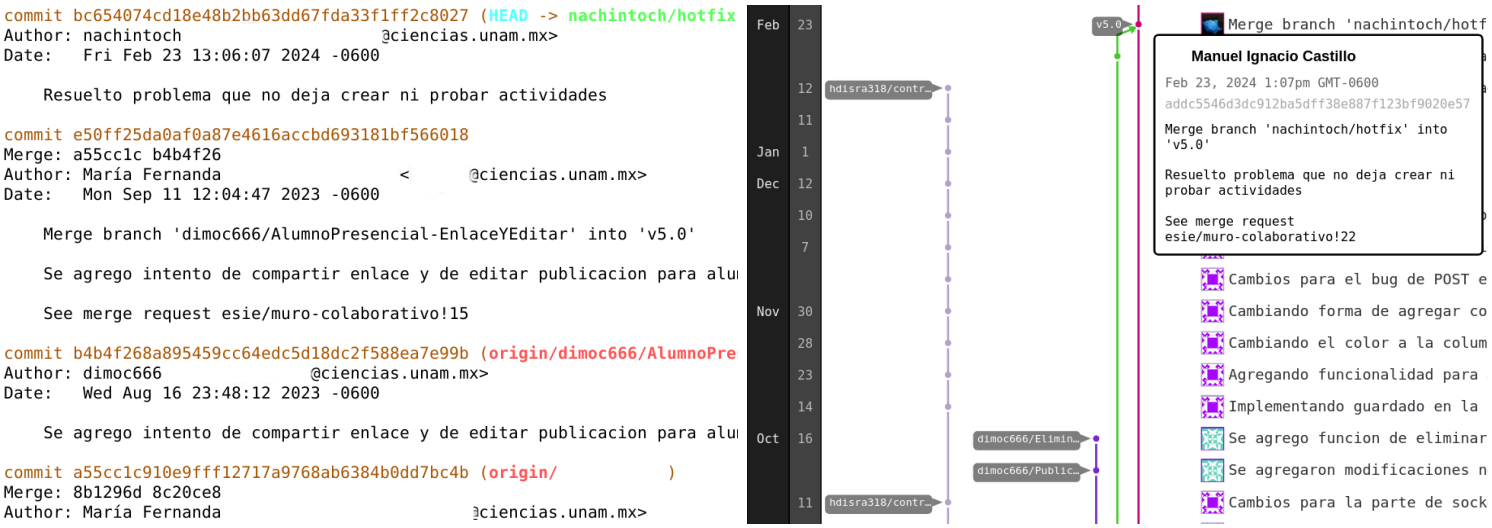


Figura 8. A la izquierda aparece el histórico de cambios como lo muestra el comando "git log". A la derecha aparece el histórico de cambios como lo muestra un repositorio Git con interfaz gráfica.

Una vez que cuente con el identificador del commit al que desea volver, se recupera con (Chacon y Straub, 2020):

```
git checkout <identificador del commit>
```

En la mayoría de proyectos basta con emplear los primeros ocho dígitos de la firma hexadecimal del commit deseado, solo en proyectos significativamente grandes puede ser necesario usar más dígitos para distinguir unívocamente cada cambio del proyecto (Git arroja un error cuando se provee una firma que coincide con otra) (Chacon y Straub, 2020).

### 5.3.5. Ramas, creando subversiones de trabajo

Los commits representan estados del proyecto pero son una forma limitada de manejo de versiones. Una forma más flexible de manejar versiones son las ramas. En Git todo repositorio tiene al menos una rama (usualmente llamada *main*, antes *master*). Todo commit pertenece a una rama, las ramas del proyecto crecen conforme evoluciona la historia de cambios (Chacon y Straub, 2020).

Para listar las ramas existentes en el proyecto se usa el comando (Chacon y Straub, 2020):

```
git branch
```

Para listar ramas tanto locales como remotas el comando es (Chacon y Straub, 2020):

```
git branch -a
```

En ocasiones el repositorio local no tendrá información de ramas nuevas dadas de alta en el remoto. Para conocerlas se debe usar el comando (Chacon y Straub, 2020):

```
git fetch
```

Para obtener los cambios de la versión remota de la rama en que estemos trabajando se usa (Chacon y Straub, 2020):

```
git pull
```

Para cambiar a alguna de las ramas existentes en el repositorio se emplea el comando (Chacon y Straub, 2020):

```
git checkout <nombre rama>
```

Para crear una rama nueva a partir del commit en el que esté el repositorio se hace (Chacon y Straub, 2020):

```
git checkout -b <nombre rama>
```

Este comando crea una rama con el nombre que se le indique y además mueve el repositorio a ella. Si se crea una rama cuando tenemos cambios no seguidos en git, entonces la rama nueva arrastra los cambios, más tarde deberán ser registrados con `git add` (Chacon y Straub, 2020).

Es importante tomar en cuenta que hay una convención para nombrar ramas de trabajo. El nombre de una rama se forma con el nombre de usuario del desarrollador quien la crea, una diagonal y después el nombre de la rama. Por ejemplo (Chacon y Straub, 2020):

```
git checkout -b <nombre de usuario>/<nombre rama>
```

Usualmente las ramas se usan para desarrollar características específicas de un producto de software. Por ejemplo, si se trata de una aplicación *calculadora*, puede haber una rama con el código de las funciones aritméticas, otra con las funciones científicas y otra más para la interfaz de usuario. El nombre de la rama debe describir la funcionalidad que implementa, por ejemplo:

```
nachintoch/funciones-cientificas
```

Note que el nombre de la rama no usa caracteres especiales (como acentos). Como en nombres de archivo, aunque es posible usarlos, es mejor evitar condiciones que pueden propiciar conflictos de codificación.

### 5.3.6. Empujando al servidor remoto

Las ramas se sincronizan entre el repositorio local y repositorios remotos. Para identificarse entre ellas se usan flujos (streams). El nombre de la rama local y su equivalente remota no tiene porque ser el mismo aunque casi siempre es así por conveniencia y practicidad. Para empujar una rama y reflejar su estado en el servidor remoto debe indicar el flujo (stream) al menos la primera vez que se publican los cambios de dicha rama (Chacon y Straub, 2020).

```
git push -u <nombre repositorio remoto> <nombre rama remota>
```

El modificador `-u` indica que lo que sigue es la configuración del *up-stream* (flujo de subida), se indica el remoto al que se desea empujar y la rama destino en el repositorio remoto. Si no existe la rama de destino se crea en ese momento (Chacon y Straub, 2020).

Si los cambios a empujar tienen como destino el mismo repositorio remoto no es necesario cambiar la rama destino y/o el servidor remoto puede omitir el modificador `-u` y sus parámetros, `push` usará la última configuración establecida (Chacon y Straub, 2020).

```
git push
```

Si al navegar por la historia de cambios (commits) desea empujar un commit que no es el siguiente consecutivo del último empujado (por ejemplo para recuperar una versión anterior del producto), es necesario incluir el modificador `--force` (o su equivalente abreviado `-f`) para confirmar que en efecto queremos sobrescribir el historial de la rama en cuestión (Chacon y Straub, 2020).

```
git push -f
```

*`--force`* ayuda a reducir riesgos de alterar el proyecto por accidente. Algunos servicios de hospedaje de repositorios Git ofrecen la capacidad de impedir forzar cambios en el historial de ciertas ramas, ya que la historia del proyecto solo debería sobreescribirse en situaciones excepcionales (Chacon y Straub, 2020).

### 5.3.7. Desarrollo paralelo e integraciones

El desarrollo de software suele ocurrir en equipo. Por convención cada desarrollador tiene su propia rama en la que acumula cambios específicos a una característica del producto. Otras ramas más generales pueden acumular la suma de cambios correspondientes a múltiples características y otras acumulan el total de cambios que representan versiones candidatas a lanzamiento del producto. Cuando cada integrante del equipo termina sus cambios debe integrarlos a la rama que acumulan los resultados de la característica correspondiente. Hay dos técnicas para integrar cambios usando Git: mezcla (merge) y rebase (rebase) (Chacon y Straub, 2020).

#### 5.3.7.1. Mezcla de ramas

Para mezclar dos ramas se usa el comando:

```
git merge <rama objetivo>
```

Esto mezcla la rama actual con la rama que se indique en el comando (Chacon y Straub, 2020).

Al mezclar dos ramas Git busca las diferencias entre los archivos de la rama actual contra los archivos de la rama destino y mezcla los cambios (Chacon y Straub, 2020).

***Ejemplo de mezcla de ramas***

Suponga que la rama original es *rama1*, y la rama que derivó de *rama1* y que desea mezclar es *rama2*. No es posible integrar dos ramas que no comparten una historia de cambios, pero tampoco es indispensable que la rama a integrar sea un descendiente directo, puede haber múltiples ramas intermedias en la jerarquía de su descendencia.

Suponga que los archivos contenidos en estas dos ramas son los que aparecen en la tabla 1:

| **rama1** | **rama2** |
|---|---|
| archivoA | arhivoA |
| archivoB | archivoB |
| archivoC | archivoC |
| <No existe en rama1> | archivoD |

Tabla 1. Archivos en las ramas hipotéticas *rama1* y *rama2*.

Suponga que el archivoA solo ha sido modificado en la *rama1*, el archivoB no ha sido modificado en ninguna de las dos ramas, el archivoC solo ha sido modificado en la *rama2* y el archivoD es nuevo y solo existe en la *rama2*.

Al terminar de trabajar en *rama2* y hacer desde *rama1*

```
git merge rama2
```

El resultado es que en *rama1* los archivos A y B no cambian, el archivoC toma los cambios de *rama2* y el archivoD es creado en la *rama1*.

#### 5.3.7.2. Rebase de ramas

El rebase es muy similar a la mezcla con la única diferencia de que en lugar de aplicar únicamente los cambios del último commit de la rama indicada a la rama actual, aplica todos los commits de la rama indicada desde que se ramificó de la actual. En caso de

que la rama indicada derive de otras ramas, el rebase considerará todos los cambios desde que se ramificó la primera rama de la que deriva la de destino (Chacon y Straub, 2020).

Por cada cambio nuevo en la rama indicada, se comparan las diferencias con la rama actual y se hace una mezcla por cambio. Para hacer un rebase el comando es (Chacon y Straub, 2020):

```
git rebase <rama objetivo>
```

Rebase ofrece cambios más limpios que un merge, donde un montón de cambios aparecen de un momento a otro. Por otro lado en ramas con muchos commits de diferencia hacer un rebase puede ser una tarea tediosa, puesto que hay que hacer una mezcla por cada cambio en el historial (Chacon y Straub, 2020).

#### 5.3.7.3. Resolviendo conflictos de integración

En ciertos casos ocurrirán conflictos al momento de hacer una mezcla o un rebase. Los conflictos se producen cuando se ha editado el mismo archivo en ambas ramas antes de hacer la integración. Si al hacer una integración Git indica que hay un conflicto, lo primero que debe hacer es usar el comando (Chacon y Straub, 2020).

```
git status
```

Git mostrará los archivos que presentan conflictos, deben ser resueltos manualmente ya que el manejador no puede saber cómo deben combinarse los cambios de ambas ramas. Los archivos con conflictos tendrán señalado la porción del texto que conflictúa con los siguientes caracteres (Chacon y Straub, 2020):

```
<<<<<<<<<<
(código en rama destino)
==========
(código en rama a integrar)
>>>>>>>>>>
```

Se debe elegir uno de los dos bloques de código, una mezcla de ambos o introducir un texto diferente, como mejor se resuelva el conflicto. Cuando termine de revisar los conflictos se registran los cambios de la integración (Chacon y Straub, 2020):

```
git add <archivos a los que se le han resuelto conflictos>
```

Finalmente se registra un commit para indicar que hemos resuelto los conflictos, mezclado las ramas (Chacon y Straub, 2020):

```
git commit -m “<mensaje del commit>”
```

Si necesita deshacer todos los cambios alrededor de la integración y suspenderla, puede usar (Chacon y Straub, 2020):

```
git merge --abort
```

### 5.3.8. Etiquetas y lanzamientos

Después de desarrollar características del producto en diversas ramas y resolver conflictos de integraciones, eventualmente el equipo de trabajo debe contar con una versión de lanzamiento del producto. Git permite distinguir commits que representan el estado o versiones especiales del producto, como lanzamiento, candidata a lanzamiento, lanzamientos anteriores o cualquier otra distinción por medio de etiquetas (Chacon y Straub, 2020).

Para crear una etiqueta se usa el comando (Chacon y Straub, 2020):

```
git tag -a <nombre clave de la versión> -m “<Descripción de la versión>”
```

Por ejemplo:

```
git tag -a v1.0 -m “Primera versión pública”
```

Para listar las etiquetas existentes se emplea (Chacon y Straub, 2020):

```
git tag
```

Puede elegir una etiqueta para trabajar a partir de su estado usando *checkout* (Chacon y Straub, 2020):

```
git checkout <nombre clave de una versión existente>
```

# 6. Desarrollo iterativo-incremental

## 6.1. Metodologías de desarrollo de software

Una metodología de desarrollo de software es una estrategia concreta que se usa para construir un producto de software. Existen muchas metodologías, los modelos “clásicos” (basados en planes) de desarrollo de software plantean el proceso por etapas: diseño, implementación, validación y lanzamiento. Estas etapas ocurren de forma secuencial, una tras otra, de forma que se obtiene el diseño del producto, se codifica, se valida que el producto resultante se apegue al diseño y se procede a su lanzamiento (Sommerville, 2011).

## 6.2. Desarrollo de software iterativo-incremental

La estructura lineal de los modelos “clásicos” de desarrollo de software los hace poco aptos al cambio. Por ejemplo durante el desarrollo de una red social pueden cambiar los patrones de interacción que se hubieran definido al inicio del proyecto, es posible que para cuando sea lanzado los usuarios consideren esta interacción como algo obsoleto y existan dificultades para la adopción del producto (Bourque et al., 2014; Sommerville, 2011).

Para prevenir este problema se han propuesto los modelos de desarrollo iterativo-incrementales. En cada iteración se construye un conjunto de características del producto de manera parcial: se eligen qué características trabajar y qué alcance de ellas se va a desarrollar. En el siguiente periodo pueden completarse todas o algunas de las características previamente desarrolladas, implementar todas o algunas de las características aún no implementadas, o alguna combinación de ellas (Sommerville, 2011).

## 6.3. Metodologías Ágiles

Las metodologías Ágiles incluyen varias de las metodologías de desarrollo de software más aceptadas y populares hoy en día (principalmente para el desarrollo de aplicaciones), como *Scrum* o *Extreme Programming*. Se basan en procesos iterativo-incrementales pero además enfatizan el rol del cliente, usuarios y otros

interesados como miembros del equipo de trabajo. En estas metodologías se busca contar con una participación activa del cliente y/o usuarios, o en su defecto representantes de ellos, de manera que participen en actividades de diseño y validación, buscando construir un producto que sea apto según sus necesidades e intereses (Sommerville, 2011).

En metodologías Ágiles se prefiere que los incrementos ocurran en el menor periodo de tiempo posible, se recomiendan periodos de entre 2 semanas y 2 meses. El proceso de desarrollo ocurre de manera completa en cada incremento pero el tiempo que se le dedican a las actividades de cada etapa depende del avance del proyecto y las prioridades de cada incremento (Sommerville, 2011).

Por ejemplo, al inicio de los proyectos las actividades de diseño son las más relevantes, pero también pueden ocurrir actividades de implementación para preparar prototipos, y de lanzamiento para presentar propuestas a los interesados (Sommerville, 2011).

Las metodologías Ágiles poseen una naturaleza centrada en el usuario, fomentando el desarrollo de requerimientos “no funcionales”, como el no requerir de condiciones o entrenamiento especializado para usar la aplicación y la facilidad de uso que habilitan una mejor adopción por el público (Sommerville, 2011).

## 7. Desarrollo guiado por pruebas

### 7.1. Estrategias de validación de software

La calidad de un producto de software depende de su apego a su especificación, es decir el apego a sus requerimientos y que estos reflejen adecuadamente las necesidades y expectativas de los clientes, usuarios u otros interesados relevantes. Una manera de validar que el producto que se construye se apega a su especificación es validando las precondiciones y postcondiciones de las funciones que implementan sus características y funcionalidades, verificando que bajo el contexto de uso esperado, el software se comporta de la manera esperada (Sommerville, 2011).

## 7.2. Automatización en procesos de software

Así como los compiladores transforman código que se asemeja a un lenguaje natural en código que puede ser ejecutado por la plataforma de ejecución destino, existen varias herramientas que ayudan a automatizar actividades que forman parte de los procesos de software. Los IDE por ejemplo suelen incluir mecanismos para automatizar tareas de *refactorización*, como mover un archivo de un directorio a otro ajustando referencias, paquetes o nombres de clase como pueda ser necesario (Sommerville, 2011).

A esta automatización de procesos de software se le llama *Ingeniería de Software Asistida por Computadora*. En el caso de la validación de software, validar entradas y salidas para verificar el comportamiento del producto resulta apto para ser automatizado, únicamente se requiere definir un programa que no forma parte del producto final, sino que toma elementos o *unidades* del producto final y verifica que dadas las entradas esperadas, el producto de software que se construye o modifica produce los resultados esperados (Martin y Martin, 2007).

**Ejemplo de automatización de pruebas de un servidor Web**

En el caso de un servidor Web puede enviar una petición HTTP con los parámetros adecuados y verificar que la respuesta sea la esperada (por ejemplo un código 200). También es posible verificar que produzca los cambios esperados en el servidor (como escribir un archivo, del que podríamos verificar su presencia en el disco del servidor, su tamaño y firma SHA).

## 7.3. Desarrollo guiado por pruebas

El desarrollo guiado por pruebas (TDD por sus siglas en inglés *Test Driven Development*) aprovecha la automatización de validaciones de precondiciones y postcondiciones de las funciones que implementan las características de un producto de software. En un ejercicio típico de metodologías Ágiles, los entregables de software desarrollados durante cada tarea de codificación son integrados al final del incremento en que ocurren, en combinación con el TDD estos cambios en el producto en construcción son verificados antes de acumularlos en la siguiente versión candidata a lanzamiento (Martin y Martin, 2007; Sommerville, 2011).

El TDD parte de definir pruebas para las características del producto a implementar. La meta de los desarrolladores es que sus entregables de software satisfagan las pruebas que les corresponden al final del periodo de implementación. En un proceso Ágil las pruebas se definen (o revisan) al inicio de cada incremento (por ejemplo al seleccionar *historias de usuarios*), de acuerdo con los objetivos de dicho incremento (Martin y Martin, 2007; Sommerville, 2011).

Las pruebas se definen en un repositorio (o sub-repositorio) ajeno al que contiene el código del producto. Las pruebas son parte del proyecto, no del producto; validan su comportamiento. El repositorio de pruebas suele ejecutarse en un entorno de ejecución que incluye un marco de trabajo especializado, el cual automatiza las pruebas de forma que solo es necesario definirlas como funciones o métodos en uno o varios scripts (o clases). Este entorno suele ser llamado “Entorno de pruebas”.

El repositorio del producto se trata como una dependencia del repositorio de pruebas: *importa* el producto para probar sus funcionalidades, proveyendo entradas de manera automatizada para verificar su comportamiento a través de las salidas directas o indirectas que produzca (por ejemplo si la función del producto devuelve un número, genera un flujo de datos, escribe un archivo o modifica un registro en Base de Datos, se comprueba que los datos producidos cumplan con las características esperadas según el diseño de la función y el producto de software en general) (Martin y Martin, 2007; Sommerville, 2011).

Una ventaja adicional del TDD al aplicarlo con metodologías Ágiles es que las pruebas ofrecen un medio accesible para los desarrolladores para conocer las funciones del producto, adquirir una idea general de su implementación, modo de uso y operación. En metodologías Ágiles se busca producir la menor cantidad necesaria de documentación en formatos tradicionales, las pruebas son un recurso valioso de documentación no tradicional (Martin y Martin, 2007).

## 8. Desarrollo centrado en el usuario

### 8.1. Orígenes del *Diseño de Interacción*

En la década de 1940 durante la segunda guerra mundial, incrementó la importancia de las fuerzas aéreas resultando en un cambio de enfoque: los pilotos se convirtieron en recursos más valiosos que los aviones que operan. Como resultado los diseñadores de aeronaves comenzaron a crear cabinas e instrumentos que se adaptan a las propiedades mentales y físicas de los pilotos, es decir se adaptan a sus *factores humanos*. Esta filosofía de diseño se hizo popular en otras industrias, impulsadas con nuevas capacidades de automatización que permiten un mayor rango de usuarios con menor entrenamiento (Bernsen y Dybkjaer, 2009; Pratt y Nunes, 2012).

### 8.2. Diseño Centrado en el Usuario

El Diseño Centrado en el Usuario (DCU) es una estrategia de desarrollo de productos que enfatiza la manera en que los usuarios realizan actividades relacionadas con el producto, sus necesidades, habilidades y en general *factores humanos* en torno al producto, de manera que resulte apto para los usuarios. El DCU sostiene que involucrar usuarios en el desarrollo y evaluación de un producto es crucial, por lo que en metodologías que adoptan DCU el usuario está presente a lo largo del proyecto, sea de forma física, mediante representantes o por medio de modelos. El DCU se aplica en el Diseño de Interacción (*Interaction Design* IxD), del que la Interacción Humano-Computadora (IHC) es un enfoque que aborda la interacción con productos de software (Bernsen y Dybkjaer, 2009; Dix et al., 2004; Nielsen, 1994; Pratt y Nunes, 2012).

### 8.3. Usuarios tipo

Los usuarios tipo son un subconjunto de los interesados de un proyecto cuyas características corresponden con los *factores humanos* contemplados para los usuarios en la especificación del producto. Estas características incluyen pero no se limitan a: edad, ocupación, intereses, habilidades, capacidades físicas, género y complexión física. Al interactuar a lo largo del proyecto con estos usuarios se obtiene un mejor

entendimiento de las actividades en las que el producto busca tener algún impacto, permitiendo verificar que el producto cumple sus objetivos respecto a las actividades del usuario (Dix et al., 2004).

### 8.4. Proceso típico de DCU

Varios enfoques de DCU sugieren actividades específicas que deben ser completadas en ciertas etapas del ciclo de vida del producto. El proceso típico de DCU puede describirse mediante las siguientes cuatro fases (Bernsen y Dybkjaer, 2009; Dix et al., 2004):

1. **Análisis**. En etapas tempranas del proyecto, deben alcanzarse acuerdos con los interesados y establecer la visión del proyecto. Se incluyen tareas de usabilidad en el plan del proyecto, debe ser ejecutado por un grupo multidisciplinario que permita abordar estas tareas con pericia suficiente. Se realizan estudios del contexto del proyecto y perfiles de usuarios para obtener una documentación de escenarios de uso y requerimientos de desempeño del usuario.

2. **Diseño**. Se presentan conceptos de diseño y modelos del flujo de navegación con los que se crean prototipos de baja fidelidad que permiten explorar y ajustar el diseño propuesto. Después se producen prototipos de alta fidelidad que también deben ser evaluados para afinar el diseño. Se prefiere realizar las evaluaciones de los prototipos mediante pruebas con usuarios. El diseño resultante se documenta en una especificación de diseño, acompañada de estándares y recomendaciones generadas para el proyecto.

3. **Implementación**. Deben realizarse evaluaciones heurísticas del producto conforme evoluciona, y realizar pruebas de usabilidad tan pronto como sea posible. Estas evaluaciones sirven para realizar un ajuste continuo del producto.

4. **Lanzamiento**. Se obtiene retroalimentación de usuarios por medio de cuestionarios y estudios de campo para medir el grado con el que se alcanzan los objetivos en torno a factores humanos.

# 9. Programación defensiva

## 9.1. Manejo de entradas de datos

Al codificar una función de software se esperan ciertas *precondiciones* establecidas por el diseño del producto, en específico establecidas por el diseño de la función en cuestión. Estas *precondiciones* son las características que deben tener los datos que reciba la función y el estado general del producto para que la función produzca los resultados esperados, como muchos otros tipos de productos el software solo puede garantizar su comportamiento bajo condiciones específicas contempladas en su diseño.

En lenguajes fuertemente tipados (como C y Java) existe una forma implícita de validación de las *precondiciones* dado que el ambiente de compilación se encarga de verificar que cuando se usa una función, los tipos de los parámetros que se le proveen correspondan la definición de la función. Por ejemplo, si se intenta codificar una llamada con texto a una función que recibe datos numéricos, el compilador detectará este problema y ni siquiera compilará el producto (Gabbrielli y Martini, 2010).

En el caso del desarrollo de interfaces, sean para componentes automáticos o interfaces de usuario, también puede ocurrir esta clase de problemas. Un usuario puede proveer datos no esperados, como escribir el nombre de un número en lugar de su valor numérico. Esta clase de problemas pueden deberse a una amplia serie de factores que pueden ir desde negligencia del usuario, hasta ambigüedades en la especificación de las interfaces de software, pasando por errores en el diseño o en la codificación.

## 9.2. Programación defensiva

La *programación defensiva* es una técnica de diseño de software en la que se desconfía de las entradas que se proporcionan a una función, programa o la manera en que se usan. Por ejemplo en POO una estrategia de programación defensiva es inicializar los atributos de clase en los constructores, de esta manera se mitigan problemas en ejecución derivados de leer una variable sin asignar (McConnell, 2004).

Una de las aplicaciones más populares de Programación defensiva es validar entradas de usuario o sistema. Suelen agregarse funciones auxiliares que se limitan a tomar las entradas de una función y verificar que tengan el formato y rango de valores esperados. Si no los tuvieran se recomienda disparar una excepción, pues el flujo normal de ejecución contempla que los parámetros son válidos, por lo que es más apropiado arrojar una excepción e iniciar un mecanismo de recuperación cuando esto no ocurre (McConnell, 2004).

## 10. Calidad de código

### 10.1. Prioridades en procesos Ágiles

Los procesos Ágiles enfatizan las necesidades y expectativas de los interesados, incorporándolos al equipo de trabajo como supervisores del diseño y la validación del producto. El producto se construye en incrementos breves que permiten realizar ajustes al diseño o a los requerimientos conforme el producto evoluciona (Martin y Martin, 2007; Sommerville, 2011).

El ritmo de trabajo esperado en un proceso Ágil busca la optimización de todas las tareas que no están directamente involucradas con la construcción del producto. Una actividad que suele consumir mucho tiempo para producir sus productos, y estos al final no suelen ser empleados plenamente, es la documentación (Martin y Martin, 2007; Sommerville, 2011).

En metodologías “clásicas” de software (guiadas por plan) se contemplan actividades exhaustivas para la generación de documentación. En esta clase de procesos se suele generar mucha documentación que describe en varios niveles de abstracción el producto desde varios puntos de vista, por ejemplo: describiéndolo desde el punto de vista del usuario (como un manual de usuario), describiendo sus interfaces (como API), describiendo sus requerimientos (Especificación de Requerimientos de Software); entre otros (Martin y Martin, 2007; Sommerville, 2011).

En metodologías Ágiles se prefiere generar la mínima cantidad de documentación necesaria, priorizando la documentación de alto nivel. Puesto que la documentación es

un subproducto valioso de un proyecto de software, aún en metodologías Ágiles se busca obtener documentación completa del producto. La propuesta de estas metodologías es recurrir a formatos de documentación informal. Una de las estrategias recomendadas es obtener documentación directamente de los artefactos de software (Martin y Martin, 2007; Sommerville, 2011).

## 10.2. Calidad de código

Una forma de obtener documentación directamente de los artefactos en el repositorio del proyecto es empleando una estructura clara del código del producto de software en construcción. Puesto que el código debe reflejar la especificación del producto, el código en sí mismo es una forma de la especificación del producto. Si la estructura del código es clara puede ofrecer una vía para entender el diseño del producto (Martin, 2008).

### 10.2.1. Código limpio

El *código limpio* es una propuesta que busca generar código de alta calidad, *fácil* de escalar y mantener. Se vale de prácticas ágiles como la simplicidad, abstracción y la refactorización (vea la sección 16.1). Algunos de los principios del *código limpio* son los siguientes (Martin, 2008):

- **Nombres significativos**. Las variables, funciones y clases en el código fuente de un software deben tener nombres claros y significativos expresados en muy pocas palabras (una sola de preferencia). El nombre de entidades de software debe revelar la función o propósito de la entidad que nombran.

- **Principio de única responsabilidad**. Cada componente de un software debe tener un único propósito, responder a una sola responsabilidad en la especificación del producto. Es común tener componentes de alto nivel cuyas funciones son generales, estas funciones generales son implementadas por componentes con responsabilidades más específicas.

  Puede contarse con varios niveles jerárquicos de responsabilidades para especificar el papel que tiene cada componente de un software.

- **Mejora continua**. Cada vez que se edita cualquier parte del código fuente de un software debe revisarse su integridad y hacer mejoras alrededor del código modificado. Cuando el diseño no es comprensible por medio del código que se modifica, es necesario revisar tanto el diseño del componente del que forma parte, como la implementación para evitar que el componente sea difícil de mantener y se convierta en un factor de riesgo. Para esto suele aplicarse la práctica de *refactorización*, que consiste en transformar código sin modificar su comportamiento, pero con una mejor calidad que en su forma original.

- **Funciones cortas**. Tanto funciones como scripts o clases deben ser lo más pequeñas posibles. Se recomienda fragmentarlas en partes con responsabilidades más específicas cuando se determina que su responsabilidad puede dividirse o cuando el artefacto es demasiado largo.

- **Comentarios significativos**. Los comentarios en el código deben ser breves y su único propósito debe ser el de explicitar aspectos de la operación o el diseño del software que no se aprecian a partir de los nombres de los artefactos involucrados. Los comentarios tautológicos (como `// devuelve verdadero` seguido por `return true`), así como el *código muerto* (código comentado) son innecesarios y hacen el código difícil de leer.

- **Respetar la encapsulación en POO**. El principio de encapsulación nos ayuda a definir programas POO que reducen problemas de codificación. Por ejemplo las variables privadas nos aseguran que nadie fuera de la clase en que están definidas pueden depender de ellas, por lo que le otorga control a la clase en la que se definen el comportamiento de esas variables. Podemos exponer sus valores y compartirlos con otros componentes del software, pero siempre bajo los términos de la clase a la que pertenecen.

- **Organización del repositorio**. Es importante organizar adecuadamente los proyectos. Por ejemplo separar el código de pruebas del código del producto principal, o separar el código de la base de datos del código del producto

principal. Seguir las convenciones de codificación y organización que se usan en la plataforma para la que estemos desarrollando es una práctica que asiste la organización de proyectos de software.

La calidad de código ha resultado de gran utilidad para incrementar la productividad en proyectos de software, existe evidencia de que el código de mala calidad incentiva la generación de más código de mala calidad, seguir prácticas de *código limpio* ayuda a generar productos de calidad en tiempo y forma (Martin, 2008).

## 11. Introducción a JavaScript

### 11.1. Netscape y la especificación ECMAScript

A mediados de la década de 1990 *Netscape Communications Corporation* publicó una nueva versión de su navegador Web *Netscape navigator*, uno de los primeros navegadores Web en el mercado y uno de los más populares en ese momento. Esta nueva versión de Netscape implementa dos características que incrementan la capacidad de interacción con el contenido Web por medio de *Java Applets* y un entorno de ejecución capaz de interactuar con la página Web que se visualiza, *JavaScript* (Hamerly et al., 1999).

Poco tiempo después JavaScript fue adoptado como una especificación estandarizada llamada *ECMAScript*. Esta estandarización permite una adopción uniforme de este entorno de ejecución en otros navegadores, como *Microsoft Internet Explorer* y *Opera*. Las capacidades de interacción de JavaScript resultaron más generales y amigables con el usuario que los Java Applets, convirtiéndose en una de las bases de lo que hoy conocemos como la Web 2.0 y las aplicaciones Web (Flanagan, 2011).

Hoy en día ECMAScript está especificado en la norma internacional ISO/IEC 22275:2018. JavaScript se ha convertido en el “lenguaje de la Web”, las aplicaciones Web implementan una parte significativa de sus características mediante código JavaScript que es ejecutado por los navegadores Web (Crockford, 2008).

### 11.2. Características de JavaScript

JavaScript es un lenguaje multiparadigma, principalmente Orientado a Objetos y Funcional. JavaScript es débilmente tipado y cuenta con funciones de primer orden. Tiene una sintaxis similar a la de *Java* o *C*, aunque opera de forma más similar a *Lisp* o *Scheme*. Existen varias características de JavaScript que prefieren evitarse puesto que suelen ser el origen de problemas en los productos que se construyen con él. Este tipo de características son por las que JavaScript suele tener una mala reputación, no entender las características problemáticas puede introducir defectos (Crockford, 2008).

Las características problemáticas de JavaScript son el resultado de un corto periodo de desarrollo que no permitieron evolucionar dichas características, su integración en el navegador *Netscape* y la rápida adopción de las primeras versiones se hubieran beneficiado de mayores periodos de desarrollo y prueba. Algunas de estas características a evitar son las siguientes (Crockford, 2008):

- JavaScript soporta bloques de comentarios de una sola línea con `//` y también comentarios multi-línea con `/* … */`. Sin embargo en algunas situaciones es posible que necesitemos usar los símbolos `*/` como parte de una expresión regular, por lo que los comentarios multi-línea son propensos a errores. Por esto se prefiere evitar usar bloques de comentarios con `/* … */` (Crockford, 2008). **Ejemplo de comentario multi-línea con errores**:

  ```
  /*
      var coincidenciasRegex = texto.match(/a*/)
  */
  ```

  En el ejemplo anterior puede entenderse que el contenido del comentario es la declaración y asignación de la variable `coincidenciasRegex` con el arreglo que devuelve la función `String.match(regexp)`[1].

  Sin embargo la expresión regular que se pasa como parámetro de `match` contiene los símbolos `*/`, lo que termina el comentario de manera prematura e interpretando la instrucción `)\n*/`, que no es válida en JavaScript.

[1] https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String/match

- La definición de bloques de código no necesariamente delimita alcances. Las variables de un script JavaScript pueden ser alcanzables inclusive desde otro script o programa.

- JavaScript deduce la ausencia de punto y coma (`;`) como delimitador de sentencia al final de una línea de código. Esto resulta en una limitación para separar sentencias en renglones, pues JavaScript puede intentar interpretar varias líneas como sentencias diferentes aunque sean parte de una sola. Se prefiere escribir código JavaScript sin punto y coma, por lo que es necesario que al final de una línea se concluya una sentencia o se separe de manera que el intérprete no inserte puntos y comas (por ejemplo iniciando un bloque de código o iniciando la siguiente línea con un operador de acceso).

  **Ejemplo de separación de sentencias problemática**

  La siguiente línea será interpretada como dos sentencias individuales:

  ```
  let a = 3
      + 5
  ```

  En lugar de asignar el valor 8 en `a`, asignará el valor 3 en `a` y evaluará el valor 5. La línea anterior puede separarse de la siguiente manera para evitar problemas de interpretación, note que el operador de asignación indica al intérprete que la sentencia está incompleta:

  ```
  let a =
      3 + 5
  ```

  Si el último caracter fuese el operador `+`, el intérprete también detectaría que la sentencia está incompleta por lo que podría ser una estrategia para separar una sentencia en varios renglones. Sin embargo sería menos claro para otros programadores que leer una asignación incompleta.

- En JavaScript existen dos valores vacíos: *`undefined`* y *`null`*. `undefined` es un tipo de datos especial que indica un valor no definido, mientras que `null` representa

una dirección de memoria no asignada. Pese a que ambos son valores vacíos, no son equivalentes, por lo que es necesario distinguirlos por separado.

- Existen dos grupos de operadores de equivalencia: los operadores de similitud `==` y `!=` y los operadores de equivalencia `===` y `!==`. Los operadores de similitud tratan de convertir los tipos de datos de los operandos para determinar si representan el mismo valor. Por su parte, los operadores de equivalencia comparan valores y tipos de datos.

  **Ejemplo de diferencia de evaluación entre equivalencia y similitud**

  `"1" == 1` es evaluado como verdadero pues por similitud entre el texto `"1"` y el entero `1` son operandos que representan el mismo valor. En cambio la equivalencia `"1" !== 1` es evaluada como verdadera puesto que el texto `"1"` y el entero `1` son valores diferentes.

  En general se recomienda usar los operadores de equivalencia y no los de similitud.

- Las operaciones numéricas con operandos no-numéricos `NaN` (*Not a Number*) tiene comportamientos inconsistentes, en particular que no es posible determinar si un valor `NaN` es equivalente a `NaN`: `NaN !== NaN` produce verdadero.

  Esto ocurre ya que JavaScript usa tipos débiles, el intérprete siempre intenta ejecutar instrucciones aún cuando los parámetros de una función no tengan el tipo de datos adecuado.

## 12. JavaScript – Sintaxis y Operadores básicos

### 12.1. Declaración de variables

Los nombres de variables deben empezar con una letra que puede estar seguida por cualquier combinación de: letras, números y guiones bajos. Como en muchos lenguajes existen palabras reservadas que no pueden utilizarse como nombres de variables (Crockford, 2008).

Es importante considerar que los nombres de variable de JavaScript son sensibles a mayúsculas: `param`, `Param` y `paRam` son tres variables diferentes (Crockford, 2008).

En JavaScript las variables se declaran mediante alguna de las siguientes tres palabras reservadas (Crockford, 2008).

### 12.1.1. var

Es el operador clásico de JavaScript para definir variables. Es posible re-definir una variable previamente declarada con `var`, en cuyo caso el valor de la variable es el último asignado. El alcance asociado a una variable declarada con `var` es el de la función inmediata en la que está definida. Cuando se declara una variable en el nivel más externo de un *script*, define variables globales (accesibles no solo al script en cuestión sino a todo el entorno de ejecución) *(Crockford, 2008)*.

Al declarar con `var` las variables se preparan en memoria, siendo inicializadas con `undefined` cuando el intérprete de JavaScript lee un *script* (y la variable no ha sido declarada previamente). Puesto que como muchos otros lenguajes interpretados la interpretación y ejecución de código JavaScript ocurren en fases separadas, es posible acceder al valor de estas variables antes de que sean declaradas en el flujo del *script* (Crockford, 2008).

**Ejemplo de acceso al valor de una variable no declarada en el flujo de ejecución**

El siguiente bloque de código JavaScript puede ser ejecutado sin generar errores pese a que lee una variable antes de que sea declarada en el flujo de la ejecución:

```
console.log(porDeclarar) // muestra el mensaje undefined en la consola
var porDeclarar // variable declarada, en la interpretación se prepara con undefinded
porDeclarar = 0 // variable asignada explícitamente durante la ejecución
```

### 12.1.2. let

Es un operador introducido en la versión 6 de la especificación ECMAScript que responde a los problemas que introducen las variables `var`. `Let` también permite definir variables que cumplen con las reglas de nombrado del lenguaje, sin embargo es más

estricto. `let` no permite definir una variable más de una vez (intentarlo produce un error). El alcance de sus variables está delimitado por el alcance del bloque de código inmediato como en muchas otras plataformas de ejecución (Crockford, 2008).

Las variables declaradas con `let` no se inicializan ni preparan con la interpretación de un script, intentar acceder a una variable `let` antes de que sea asignada o declarada resulta en un error (como en muchas otras plataformas de ejecución) (Crockford, 2008).

#### 12.1.3. const

`const` es un operador que permite declarar constantes en un programa JavaScript. Su comportamiento es análogo al de una variable declarada con `let`, salvo porque las variables `let` pueden volver a ser asignadas tantas veces como se desee, mientras que las variables `const` únicamente pueden ser asignadas una sola vez, por lo que se usan para definir constantes (Crockford, 2008).

### 12.2. Tipos de datos y sus operadores

#### 12.2.1. Objetos

En JavaScript todas las variables son instancias de un tipo principal: objetos. Los tipos de datos primitivos (como números o booleanos) son instancias de objetos, por lo que poseen métodos. Puede definir un objeto nulo utilizando `null` (Crockford, 2008).

##### 12.2.1.1. Acceso

Para acceder a los miembros de un objeto se usa el operador *punto*. En el caso de un arreglo se usan paréntesis cuadrados para acceder a sus localidades (Crockford, 2008).

***Ejemplo de uso del operador de acceso***

```
objeto.toString()
```

En el caso de un arreglo:

```
arreglo[0]
```

También es posible acceder a los miembros de un objeto paréntesis cuadrados, indicando el nombre del miembro al que se desea tener acceso. Esto es especialmente

útil cuando los miembros de un objeto tienen nombres que no cumplen con las reglas de nombramiento de JavaScript, como en `objeto[“apellido-paterno”]` (note el guión medio, es invalido en un nombre de variable JavaScript) (Crockford, 2008).

Tanto al acceder a arreglos como objetos, si intentamos acceder a un miembro que no existe, obtendremos el valor `undefined`. El usar objetos como arreglos es análogo a los *arreglos asociativos* de PHP, por lo que los objetos de JavaScript también se usan como *tablas hash* o diccionarios (Crockford, 2008).

***Ejemplo de uso de un objeto como arreglo asociativo***

```
let coloresHex = {
  rojo : ‘#FF0000’,
  verde : ‘#00FF00’,
  azul : ‘#0000FF’
}
coloresHex[‘rojo’] // devuelve ‘#FF0000’
```

También es posible incluir miembros al vuelo en un objeto. Basta con hacer referencia a un atributo aún no definido y asignarle algún valor (Crockford, 2008):

```
objeto.miembroNuevo = “algún valor”
```

#### 12.2.1.2. Arreglos

Los arreglos en JavaScript son un tipo especial de objeto en el que se agregan miembros identificados por enteros. Además de ser similares a una *tabla hash* o diccionario, cuentan con métodos de pila (*stack*), como *push* para agregar miembros y *pop* para quitarlos. También cuentan con un atributo *length* que indica el número de elementos actualmente en el arreglo. Los arreglos de JavaScript se indexan a partir de cero (Crockford, 2008).

La forma preferida de crear arreglos es utilizando literales y no constructores (Crockford, 2008):

```
var arreglo = [1, 2, 3, 4, 5]
```

La forma preferida de agregar elementos en un arreglo es tratarlo como una pila y emplear el método *`push`*, que inserta elementos al final del arreglo (Crockford, 2008).

***Ejemplos de manipulación de arreglos***

```
arreglo.push(nuevoValor)
```

Si se agrega el valor `nuevoValor = 6` mediante la instrucción anterior, el estado resultante de `arreglo` es:

```
[1, 2, 3, 4, 5, 6]
```

De forma similar se eliminan elementos de un arreglo tratándolo como una pila empleando el método *`pop`*, el cuál elimina el último elemento del arreglo. Retomando el ejemplo anterior, al aplicar la operación `pop()` después de insertar `nuevoValor = 6`, vuelve a su estado original (pues se elimina el elemento antes insertado al final):

```
arreglo.pop() // arreglo vuelve a ser [1, 2, 3, 4, 5]
```

Para borrar elementos que no estén al final del arreglo, se usa el método *`splice`* que recibe dos parámetros obligatorios: el índice del arreglo a partir del que se quiere cambiar el contenido del arreglo y el número de elementos que desea eliminar (o reemplazar, también puede proporcionar parámetros adicionales que son insertados a partir del índice del primer parámetro) (Crockford, 2008).

```
arreglo.splice(1, 3) // arreglo ahora es [1, 5]
```

*`splice`* devuelve un arreglo con los elementos eliminados o reemplazados del arreglo en que se haya llamado. Actualiza los índices y el tamaño del arreglo al quitar/reemplazar los elementos indicados (Crockford, 2008). En el ejemplo anterior, el arreglo de devuelve la operación `splice` es:

```
[2, 3, 4]
```

#### 12.2.1.3. Funciones

Las funciones son un tipo de objeto especial que se puede llamar a ejecutar. Al ser objetos tenemos la libertad de tratarlas como valores: podemos almacenarlas en variables y pasarlas como parámetros. En los siguientes capítulos veremos funciones con mayor detalle (Crockford, 2008).

#### 12.2.1.4. Quitando miembros de un objeto

En JavaScript es posible quitar miembros de un objeto dinámicamente, de forma similar a en que se asignan al vuelo. Para esto se usa el operador *delete* seguido por un acceso al miembro de un objeto que se desea eliminar (Crockford, 2008).

***Ejemplo de remoción de miembros de un objeto***

```
delete objeto.miembroNuevo // miembroNuevo se quita del objeto
```

Tome en cuenta que si el objeto hereda propiedades de algún prototipo, y se quita un miembro homónimo con otro definido en el prototipo (es decir, el objeto en cuestión oculta un miembro heredado de su prototipo), el miembro permanecerá en el objeto con el valor que tiene en el prototipo (Crockford, 2008).

### 12.2.2. Tipos numéricos

En JavaScript existe un único tipo numérico, flotantes de 64 bits. El valor pseudo-numérico especial `NaN` (*Not a Number*) resulta de operaciones que no pueden producir un resultado con los parámetros dados (Crockford, 2008).

Como en muchos otros lenguajes se cuenta con operadores aritméticos `+` (suma), `-` (resta), `*` (multiplicación) y `/` (división). También se disponen de variantes del operador de asignación que permite auto-operar una variable con otra o algún valor, como `a += 2`. El operador `isNaN(number)` nos ayuda a determinar si un valor numérico es `NaN` o no (Crockford, 2008).

JavaScript cuenta con un objeto predefinido `Math` que contiene una serie de funciones para trabajar con números, como `Math.abs(number)` que devuelve el valor absoluto o `Math.floor(number)` que devuelve el menor entero más cercano en la recta real al número dado (Crockford, 2008).

### 12.2.3. Cadenas de caracteres

En JavaScript es posible definir cadenas usando comillas dobles (`“Hola mundo”`) o simples (`‘Hola mundo’`) indistintamente. En general se recomienda definir cadenas con comillas simples, puesto que en documentos HTML/XHTML las comillas dobles se usan

para definir el valor de un atributo de un elemento del documento (como en `<a href="URL" >`) (Crockford, 2008).

Una cadena puede contener cero o más caracteres. El carácter de escape es `\` (diagonal invertida). Las cadenas de JavaScript se conforman de caracteres Unicode de 16-bits, pero no maneja un tipo de dato *caracter* como tal. En JavaScript un *caracter* es una cadena de longitud 1 (Crockford, 2008).

### 12.2.4. Booleanos

En JavaScript se representan booleanos con las literales `true` (verdadero) y `false` (falso), que representan los valores de verdad verdadero y falso respectivamente (Crockford, 2008).

### 12.2.5. undefined

El tipo de datos *`undefined`* representa valores no definidos. Es diferente de `NaN` y `null` (`NaN` es un número y `null` un objeto) (Crockford, 2008).

### 12.2.6. Preguntando por el tipo de un valor

El operador unario *`typeof`* permite conocer en forma de cadena de texto el tipo de datos que tiene algún valor o variable (Crockford, 2008).

```
typeof 1 === "number" // devuelve true
```

## 12.3. Valores de verdad

En JavaScript se sigue la filosofía de evaluar como falso los valores vacíos o equivalentes a cero. Estos son (Crockford, 2008): `false` (falso booleano), `null` (apuntador de objeto nulo), `undefined`, `''` (cadena vacía), `0` (el número cero) y `NaN`.

Cualquier otro valor se evaluará como verdadero (Crockford, 2008).

### 12.3.1. Operadores lógicos

En JavaScript contamos con las operaciones lógicas de conjunción y disyunción. Una conjunción se distingue por usar el operador `&&`, que devuelve el valor de su primer

operando si su valor lógico se reduce a falso. En otro caso devuelve el valor de su segundo operando (Crockford, 2008).

```
“false” && 1 // produce 1
```

Por su parte, la disyunción se caracteriza por el operador `||`, en el que se busca si el primer operando es verdadero, de ser así devuelve su valor. En otro caso devuelve el valor de verdad del segundo operando (Crockford, 2008).

```
0 || “” // produce “”
```

Note que los operadores `&&` y `||` no necesariamente devuelven verdadero o falso, el valor de verdad de ambas expresiones depende de los valores de verdad a los que se reducen sus resultados. Retomando los ejemplos anteriores, `“false” && 1` se puede reducir a verdadero puesto que devuelve `1`, mientras que `0 || “”` se puede reducir a falso puesto que devuelve la cadena vacía (vea la sección 12.3).

## 12.4. Paso de valores

En JavaScript el paso es por referencias, no se copian valores. Esto significa que si asigna una variable con otra, el valor de la primera variable será el mismo objeto al que apunte la segunda, por lo que si modifica los atributos del objeto en la segunda variable, se verán reflejados en la primera (puesto que apuntan al mismo valor) (Crockford, 2008).

### Ejemplo de manipulación de un valor apuntado por dos variables diferentes

```
let arreglo = []
let x = arreglo
arreglo.push(1)
x[0] === 1 // true
```

## 12.5. Operadores de control de flujo

### 12.5.1. Control de flujo secuencial

En JavaScript se definen bloques IF – THEN – ELSE de la misma forma que en lenguajes similares a C o Java: los bloques de código se definen entre corchetes `{ }` y la cláusula ELSE es opcional (Crockford, 2008).

```
if (condicion) {
  hazAlgo()
} else {
  hazOtraCosa()
}
```

También se cuenta con bloques `switch` que permiten realizar acciones dependiendo de varios posibles valores de una condición. Al igual que en Java y otros lenguajes se usa la sentencia `break` al final de un caso (`case`) para evitar que se ejecuten los siguientes salvo que esa sea nuestra intención. Es posible agregar una cláusula `default` para cubrir cualquier caso no especificado (Crockford, 2008).

**Ejemplo de bloque switch**

```
swtich (condicion) {
case valor0:
  hazAlgo()
case valor1: // como no hay break entre valor0 y valor1, valor0 ejecuta ambos casos
  tambienHazEsto()
  break
case valor2: // este caso es excluyente respecto a todos los demás
  hazOtraCosa()
  break
default: // este caso ocurre cuando la condición se reduce a un valor no especificado
  hazLoQueNingunOtro()
}
```

Por último tenemos el operador ternario que devuelve un valor dependiendo del valor de verdad de una condición (Crockford, 2008):

```
condicion ? valorSiVerdadero() : valorSiFalso()
```

### 12.5.2. Ciclos

En JavaScript se cuenta con las estructuras de ejecución cíclica WHILE, FOR y DO – WHILE. WHILE y DO – WHILE son muy similares, con la diferencia de que WHILE verifica su condición de permanencia *antes* de ejecutar su bloque de código, mientras que DO – WHILE *primero* ejecuta su bloque de código y al final evalúa la condición de permanencia (Crockford, 2008).

```
while (condicion) {
  hazVariasVecesONinguna()
}
```

```
do {
  hazAlMenosUnaVez()
while (condicion)
```

Los ciclos FOR permiten definir una variable que asiste en la iteración de estructuras (Crockford, 2008).

JavaScript también soporta otro tipo de ciclo FOR que permite extraer todos los miembros de un objeto y trabajar secuencialmente con cada uno de ellos. Por la forma en que JavaScript maneja herencia usando prototipos en lugar de clases, es recomendado usar el método `hasOwnProperty(var)` presente en todos los objetos, esclarece si un miembro del objeto pertenece al objeto, o si pertenece a un prototipo del que hereda (Crockford, 2008).

**Ejemplo de iteración de estructuras con FOR**

Podemos iterar los miembros de un objeto de la siguiente manera:

```
for (miembro in unObjeto) {
  if (unObjeto.hasOwnProperty(miembro)) {
    hazAlgoCon(miembro)
  }
}
```

Los miembros de un arreglo pueden iterarse con:

```
for (var i = 0; i < arreglo.length; i++) {
  hazAlgoCon(arreglo[i])
}
```

## 12.6. Precedencia de operadores

La tabla 2 exhibe los operadores de JavaScript, ordenados por precedencia de mayor a menor (de arriba a abajo). Los operadores en el mismo renglón tienen la misma precedencia, en caso de aparecer en una misma instrucción el orden de ejecución es el orden de lectura (tiene precedencia el primero que aparezca) (Crockford, 2008).

| Operadores | Sintaxis de los operadores en JavaScript |
|---|---|
| Operadores de acceso y llamada | `. [ ] ( )` |
| Operadores unarios | `delete` `new` `typeof` `+ -` (signos numéricos) `!` (negación) |
| Multiplicación, división y módulo | `*` `/` `%` |
| Suma, concatenación y resta | `+` `-` |
| Desigualdad | `>=` `<=` `>` `<` |
| Equivalencia | `===` `!==` |
| Conjunción lógica | `&&` |
| Disyunción lógica | `\|\|` |
| Operador ternario | `? :` |

Tabla 2. Operadores de JavaScript ordenados por precedencia de mayor a menor.

Recuerde que la precedencia de operadores es el orden en el que se ejecutan en una instrucción, por ejemplo en la expresión aritmética `3 + 5 * 2`, el operador `*` tiene mayor precedencia por lo que primero se hace el producto y al final la suma (con el resultado del producto).

## 13. Manejo de errores en JavaScript

### 13.1. Excepciones

El manejo de excepciones en JavaScript es muy similar a como se hace en otros lenguajes: si se dispara una excepción dentro de un bloque de código protegido (`try – catch`), entonces la excepción disparada será manejada en el bloque `catch`. Si la excepción es lanzada fuera de un bloque `try – catch`, termina la ejecución de la función actual y la excepción se propaga por el *stack* de ejecución hasta ser atrapada por un bloque `catch` o terminar la ejecución del programa (Crockford, 2008).

Puesto que en JavaScript se definen objetos con base en prototipos y no en clases, no existe alguna clase *Exception* que defina a todas las excepciones posibles. En su lugar se crean objetos que deben tener al menos los siguientes dos miembros (Crockford, 2008):

- *name* – Representa el nombre de la excepción (como puede ser "ArrayIndexOutOfBoundsException").
- message – Descripción legible para humanos de la excepción (por ejemplo "El índice -1 no es válido en un arreglo").

Se usa la cláusula *throw* para disparar una excepción (Crockford, 2008).

**Ejemplo de disparo de una excepción**

```
throw {
  name: "ArrayIndexOutOfBoundsException",
  message: "-1 no es un índice válido en un arreglo"
}
```

Note que el objeto que representa la excepción puede tener otros atributos, es posible agregar información respecto al problema que originó la excepción o mecanismos de recuperación (Crockford, 2008).

### 13.2. Bloques try – catch

Los bloques try – catch en JavaScript atrapan cualquier excepción que sea disparada por las funciones que se llaman en él, incluyendo excepciones originadas por funciones llamadas directa o indirectamente dentro del bloque try – catch. Puesto que las excepciones no poseen alguna jerarquía de tipos como en otros lenguajes, en JavaScript se atrapan todas las excepciones sin importar la razón que las haya causado (Crockford, 2008).

```
try {
  <instrucciones protegidas>
} catch (<nombre de variable del objeto excepción>) {
  <bloque de recuperación de la excepción>
}
```

Si únicamente se desea manipular un tipo específico de excepciones, puede revisar su nombre, mensaje o cualquier otro atributo que debería existir en la excepción esperada: si coincide con las características esperadas entonces se ejecuta el resto del bloque de recuperación, de lo contrario puede relanzar la excepción para que sea atrapada por otro bloque más externo (o produzca el fin de la ejecución) (Crockford, 2008).

**Ejemplo de mecanismo de recuperación para una excepción específica**

```
try {
  <instrucciones protegidas>
} catch (excepcion) {
  if (excepcion.atributo && excepcion.atributo === 'valor esperado') {
    <bloque de recuperación de la excepción>
  } else {
    throw excepcion
  }
}
```

## 14. Funciones de primer orden y anónimas

### 14.1. Declaración de funciones

La sintaxis básica para definir una función es la siguiente (Crockford, 2008):

```
function <nombreFuncion>(<parametros separados por coma>) {
  <cuerpo de la función>
}
```

En JavaScript todas las funciones devuelven algún valor. Cuando no se incluye una cláusula `return` al final de una función, por defecto devuelve `undefined`. Si se incluye un `return` pero no se especifica un valor a devolver, también devuelve `undefined` (Crockford, 2008).

Existe una excepción a esta regla cuando se invoca una función usando el operador `new`, en este caso la función opera como constructor y devuelve la referencia a un objeto creado (`this` desde el punto de vista del constructor que se define) (Crockford, 2008).

Las funciones que se diseñan para ser usadas con el operador `new` deben tener la primera letra en su nombre en mayúscula por convención. Por ejemplo (Crockford, 2008):

```
var fecha = new Date()
```

Es posible definir atributos con valores por omisión u opcionales. Por convención se definen al final en la lista de parámetros con una asignación al valor por omisión. Cuando se llame a la función puede o no darse un valor alternativo a estos parámetros. También es posible definir una función cuyos parámetros son totalmente opcionales (Crockford, 2008).

**Ejemplo de función con parámetros opcionales**

```
function ejemploParamsOps(paramObligatorio, paramOp1 = null, paramOp2 = 0) {
  let primerParamOpConValor =  paramOp1 || paramOp2
  if (primerParamOpConValor) {
    console.log("Al menos uno de los parámetros opcionales es no-falso")
    return primerParamOpConValor
  } else {
    return paramObligatorio
  }
}
```

Al llamar a una función con parámetros opcionales es posible especificar los parámetros a los que se les desea asignar algún valor. Para ello se debe especificar el nombre de los parámetros opcionales, asignándoles el valor deseado. Tomando la función `ejemploParamsOps` del ejemplo anterior, se le puede proveer valor únicamente al parámetro `paramOp2` (y por supuesto, al (los) parámetro(s) obligatorio(s)) (Crockford, 2008):

```
ejemploParamsOps("Valor obligatorio", paramOp2="segundo valor opcional")
```

Siempre que se llama a una función se le pasa un parámetro especial llamado `arguments`. Este es un arreglo con todos los valores que se le hayan pasado a la función, incluyendo parámetros no nombrados en la firma de la función, es decir, parámetros adicionales a los que espera la función. Esto permite definir funciones de aridad arbitraria, funciones que pueden recibir cualquier número de parámetros (Crockford, 2008).

**Ejemplo de función de aridad arbitraria**

La siguiente función suma cualquier grupo de números que se le pase como parámetros :

```
function sumaCualesquiera() {
  let i, sum = 0
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i]
  }
  return sum
}
```

Puede invocar la función `sumaCualesquiera` con cualquier grupo de números, como:

- `sumaCualesquiera(1,1)` → `2`
- `sumaCualesquiera(1,2,3,4,5)` → `15`
- `sumaCualesquiera(0.5, 1.5, 8)` → `10.0`

## 14.2. Funciones anónimas

También es posible definir una función sin nombre o *anónima* de la siguiente manera (Crockford, 2008):

```
function(<parámetros separados por coma>) {
  <cuerpo de la función>
}
```

**Ejemplo de función anónima**

La siguiente función obtiene el valor mínimo en un arreglo numérico y lo devuelve:

```
function(arreglo) {
  let i, mayor = Number.MAX_SAFE_INTEGER
  for(i = 0; i < arreglo.length; i++) {
    if (arreglo[i] < mayor) {
      mayor = arreglo[i]
    }
  }
  return mayor
}
```

Note que la función anterior se puede definir sin parámetros y usar la variable `arguments` en lugar de `arreglo`.

Si se desea invocar una función anónima tan pronto como se define, puede pasarle parámetros justo después de su definición (Crockford, 2008):

```
function(<parámetros separados por coma>) {
  <cuerpo de la función>
} (<parámetros>)
```

Lo anterior es válido aún cuando los parámetros proporcionados son vacíos como:

```
function(<parámetros separados por coma>) {
  <cuerpo de la función>
} () // lista vacía de parámetros
```

Existe una segunda sintaxis para definir funciones anónimas llamada *funciones flecha*, se caracteriza por usar los símbolos `=>` para abreviar la definición de una función sin nombre de la siguiente manera (Mozilla Corportation, 2023b):

```
(<parámetros separados por coma>) => {
  <cuerpo de la función>
}
```

La función anónima del ejemplo anterior puede ser transformada en una función flecha simplemente quitando la palabra reservada `function` al inicio de la expresión y agregando los símbolos `=>` entre los parámetros de la función y el corchete que inicia la expresión del cuerpo de la función:

```
(arreglo) => {
  let i, mayor = Number.MAX_SAFE_INTEGER
  for(i = 0; i < arreglo.length; i++) {
    if (arreglo[i] < mayor) {
      mayor = arreglo[i]
    }
  }
  return mayor
}
```

Las *funciones flecha* también pueden ser invocadas inmediatamente al ser definidas, pero es necesario envolverlas entre paréntesis antes de poner los paréntesis con los parámetros con que se desea ejecutar (Mozilla Corportation, 2023b).

```
((<parámetros separados por coma>) => {
  <cuerpo de la función>
}) (<parámetros>)
```

En algunos contextos como el anterior las *funciones flecha* pueden tener problemas de interpretación, por lo que se recomienda solo usarlas en contextos específicos donde son permitidas o recomendadas.

### 14.3. Funciones de primer orden

JavaScript es un lenguaje de primer orden, lo que significa que sus funciones son valores. Es posible pasarlas como parámetros, almacenarlas en variables o como miembros de objetos (en cuyo caso se consideran métodos) (Crockford, 2008).

```
let variable = function(<parámetros>) {
  <cuerpo de la función>
}
```

Para llamar a una función en una variable o miembro de una clase, se llama a la variable que contiene la función seguida por la lista de los parámetros con los que se desea llamar a la función.

```
variable(<parámetros>)
```

### 14.4. Alcances y cerraduras

Las variables definidas en un programa JavaScript pueden ser leídas por otros programas en el mismo entorno de ejecución cuando son definidas con `var`. Esto puede generar conflictos con valores manejados por otros programas (Crockford, 2008).

Por su parte las funciones definen un alcance local aún con `var`. Se puede aprovechar esta propiedad de las funciones para definir estructuras con alcances locales como cerraduras de la siguiente manera (Crockford, 2008):

```
let objeto = function() {
  var atributoLocalDelObjeto
  return {
    metodoLocalDelObjeto: function (<parametros>) {
      <cuerpo del método local>
    }
  }
} ()
```

Note que la variable *`objeto`* se asigna con lo que devuelve una función anónima (observe los paréntesis al final que invocan a la función). La función devuelve un objeto

que posee atributos y métodos locales que no pueden ser accedidos fuera de él (Crockford, 2008).

Note que aunque los atributos “privados” del objeto se definen fuera del objeto que devuelve la función (como aparece en el ejemplo `atributoLocalDelObjeto`), se definen en la *cerradura* de la función y puesto que el objeto también está definido *dentro de la cerradura*, pueden ser accedidos y manipulados por los métodos del objeto como miembros de clase, pero no por otros objetos o *scripts*. (Crockford, 2008).

Esta estrategia es contraria a manipular el *prototipo* de un objeto, puesto que los miembros de un prototipo son públicos y tienen efecto en todas las instancias del objeto (Crockford, 2008).

Las funciones y cerraduras permiten crear *módulos*, que son una colección de objetos y funciones que ocultan sus mecanismos, eliminando los problemas que produce el espacio de memoria global (Crockford, 2008).

Un ejemplo del uso de estas características es la producción de funciones. Es posible crear funciones que producen otras funciones, estas pueden usarse para atender un caso particular de algún problema, siendo configurada con algún valor indicado al producirla o pueden almacenar resultados anteriores utilizando *memoización* (Crockford, 2008).

**Ejemplo de función que produce funciones**

La siguiente función produce funciones que elevan un número a una potencia específica indicada al producirla y además recuerda el resultado de cada potencia calculada para no repetir operaciones (Crockford, 2008):

```
function elevaYRecuerda(a) {
  var resultados = {}
  return function(n) {
    if (!resultados[a]) {
      resultados[a] = {}
    }
    if (!resultados[a][n])
      resultados[a][n] = Math.pow(n, a)
    }
```

```
    return resultados[a][n]
  }
}
```

## 15. Convenciones de codificación de JavaScript

### 15.1. JavaScript Object Notation – JSON

JSON es el nombre que se le da a *literales de objetos*, estas son definiciones de objetos al vuelo en un programa JavaScript. Por ejemplo (Crockford, 2008):

```
let objetoVacio = {}
```

El ejemplo anterior define un objeto que no tiene ningún miembro, más que los miembros definidos en el prototipo global de objetos. Podemos incluir miembros en un objeto tratándolo como un diccionario en el que tenemos `clave : valor`, separados por dos puntos. La clave o nombre de los miembros puede ser cualquier cadena, incluso la cadena vacía. Por ejemplo (Crockford, 2008):

```
let persona = {
  “nombre” : “Homero”,
  “apellido” : “Simpson”
}
```

Cuando los nombres de los miembros del objeto cumplen con las reglas de nombrado de variables de JavaScript se pueden omitir las comillas en los nombres de los miembros. En el ejemplo anterior los miembros *nombre* y *apellido* cumplen con dichas reglas, por lo que podrían definirse sin comillas. Si en su lugar apareciera `apellido-paterno`, el guión medio no es válido en un nombre de JavaScript, por lo que en ese caso sería necesario definirlo entre comillas como miembro de un objeto (Crockford, 2008).

Los valores que asociamos a los miembros de una clase pueden ser cualquier valor de JavaScript, incluyendo arreglos literales como `[1, 2, 3]`, funciones u otros objetos (Crockford, 2008).

## 15.2. Estilo de codificación preferido en JavaScript

### 15.2.1. Nomenclatura de archivos y directorios

Los nombres de archivos y directorios que forman parte de un repositorio de código JavaScript deben apegarse al esquema `snake_case`. Se prefiere usar letras minúsculas y evitar guiones en medida de lo posible. Esta regla tiene la intención de abstraer si un servidor Web hace distinción entre mayúsculas y minúsculas en las URL de recursos (Crockford, 2019).

### 15.2.2. Nomenclatura de identificadores en JavaScript

Los nombres de variables, funciones u otros identificadores de elementos de un programa JavaScript siguen la convención de nombrado *camelCase* (Crockford, 2019):

```
let unaVariable
function unaFuncion() { … }
```

Existe una excepción pues las constantes se prefieren escribir en mayúscula, separando palabras con guiones bajos (Crockford, 2019):

```
const UNA_CONSTANTE
```

La nomenclatura en JavaScript sigue las convenciones de nombrado en Java. Como en muchos otros lenguajes se prefieren usar nombres alfanuméricos (no incluir símbolos en el nombre) y utilizar únicamente caracteres ASCII-compatibles, otros ***pueden*** generar problemas (Crockford, 2019).

```
let contraseña  // No use caracteres no ASCII-compatibles
let password    // alternativa a “contraseña”
let contrasegna // otra posible alternativa
let contrasena  // es mejor que usar ñ, á u otros caracteres no disponibles en ASCII
```

### 15.2.3. Sangría (*indentación*) y separación de bloques de código

Como en muchos otros lenguajes, se prefiere utilizar espacios en lugar de tabuladores para el sangrado (*indentación*). En JavaScript es común utilizar sangrías de 2 o 4 espacios (no tabulaciones), ambos son aceptables pero debe ser consistente en un mismo repositorio: si el proyecto usa sangrías de 2 espacios, entonces todos los scripts en el repositorio deben apegarse a esa convención.

La demarcación bloques de código con `{ … }` emplea la misma convención de codificación que Java, en la que el corchete que inicia el bloque `{` se escribe en la misma línea que la definición del bloque de código y el corchete que lo cierra `}` debe aparecer como el único caracter visible en la línea donde se escriba, con el mismo sangrado que la línea donde inicia el bloque que cierra (Crockford, 2019):

```
function () {
    …
  switch () {
      …
  }
    …
}
```

Cuando se define un objeto, arreglo o función empleando varias líneas, se prefiere dejar el paréntesis o corchete que inicia la definición de miembros o parámetros como el último elemento de la primera línea de la definición, las siguientes líneas listan los miembros o parámetros, la última línea de la definición del objeto, arreglo o función contiene el paréntesis que cierra la definición (Crockford, 2019).

Las líneas de miembros o parámetros deben tener un nivel de sangría mayor que la primera línea de la definición, mientras que la línea final con el cierre de la definición debe tener el mismo nivel de sangría que la primera línea (Crockford, 2019):

```
function funcionLarga (
    param1, param2, param3
) {
  let objeto = {
    miembro1: valor1,
    miembro2: valor2,
    miembro3: [
      valor3_1, valor3_2, valor3_3
    ]
  }
}
```

A menos que un operador unario sea una palabra, como `typeof`, el resto de los operadores unarios deben aparecer sin separación del valor o variable sobre la que tienen efecto (Crockford, 2019):

```
typeof 5 // operador unario que es una palabra, se separa del valor
```

```
-5 // operador unario que es un símbolo, sin espacio con el valor
```

Los operadores no-unarios deben separarse de sus operandos con un espacio, excepto los operadores de acceso (punto y corchetes) (Crockford, 2019):

```
2 + 2 // operador no-unario, se separa con un espacio de sus operandos
arreglo[0] // operador no-unario de acceso, sin espacio con sus operandos
```

Después de una coma (por ejemplo en una lista de parámetros o miembros de un arreglo) debe aparecer un espacio en blanco o un salto de línea, el uso de espacios o saltos de línea en una misma lista debe ser consistente (Crockford, 2019).

```
Math.max(1,
  2,
  3)
[1, 2, 3, 4, 5]
```

En una cláusula `swtich` no se sangran los encabezados de sus casos, pero se sangra el contenido de los casos (Crockford, 2019):

```
switch (condicion) {
case expresion1: // encabezado de caso, sin sangrar
    <bloque con sangrado>
case expresion2: // encabezado de caso, sin sangrar
  <bloque con sangrado>
default: // encabezado de caso, sin sangrar
  <bloque con sangrado>
}
```

### 15.2.4. Comentarios

Los comentarios deben apegarse a principios de código limpio: no agregar comentarios obvios, tautológicos o redundantes. Evite comentarios como (Crockford, 2019; Martin, 2008):

```
// devuelve true
function returnsTrue() {
  return true
}
```

No solo el nombre de la función clarifica su comportamiento, sino que el cuerpo de la función permite entenderlo fácilmente. Use comentarios para ilustrar el diseño de mecanismos complejos, exhibiendo decisiones de diseño que el código no transmita directamente (Martin, 2008).

Por otro lado los comentarios de bloque en JavaScript comparten características con la sintaxis de expresiones regulares, por lo que se prefiere evitarlos y usar siempre comentarios de una sola línea (vea la sección 11.2), aún si es necesario iniciar varios bloques de comentarios para hacer una descripción completa (Crockford, 2019, 2008).

**Ejemplo de uso de comentarios multi-línea**

```
// Memoiza resultados en el objeto “resultados”
// Note que cada vez que se llama a la función con un parámetro ‘a’ diferente,
// la función maneja una memoria en el objeto “resultados”
// que depende del parámetro ‘a’
function elevaYRecuerda(a) {
  var resultados = {}
  return function(n) {
    if (!resultados[a]) {
      resultados[a] = {}
    }
    if (!resultados[a][n])
      resultados[a][n] = Math.pow(n, a)
    }
    return resultados[a][n]
  }
}
```

### 15.2.5. Paréntesis, corchetes y bloques de código

Se prefiere separar palabras clave como `if`, `while` o `switch` del paréntesis que le sigue por un (y solo un) espacio en blanco (Crockford, 2019):

```
if (condición) { … }
for (<inicialización>; <condición>; <incremento>) { ... }
```

En el caso de llamadas de funciones se prefiere que el nombre de la función y los paréntesis con sus argumentos aparezcan juntos, sin separar (Crockford, 2019):

```
string.trim()
number.valueOf()
```

Al crear objetos y arreglos se prefiere usar literales en lugar de constructores (Crockford, 2008).

```
let objeto = {
  miembro : ‘valor’
}
let arreglo = [
  1, 2, 3
```

```
]
```

### 15.2.6. Sentencias

En cada línea de código debe haber únicamente una sola sentencia. JavaScript inserta punto y coma (`;`) al final de una línea automáticamente cuando lo determine necesario. Por esto en algunos repositorios se prefiere evitar usar punto y coma del todo, mientras que en otros se prefiere terminar todas las sentencias con punto y coma. Cualquiera de las dos convenciones es aceptable, pero debe ser consistente en todo el repositorio de un proyecto (Crockford, 2008).

La extensión de una línea de código no debería ser mayor a 80 o 120 caracteres como se prefiere en muchos entornos de codificación (Crockford, 2019).

## 16. Inspectores de código y pruebas automáticas de software

### 16.1. Actividades de control de calidad en procesos Ágiles

En un proyecto Ágil es posible prescindir de roles explícitos de Control de Calidad si se agregan las tareas de este rol a las tareas de codificación. En un proyecto Ágil estas tareas incluyen (Martin y Martin, 2007):

- **Refactorización**. Consiste en modificar el código sin alterar su comportamiento buscando que tenga una estructura clara, eficiente, escalable y mantenible. Cada vez que se trabaja con una porción del código, se recomienda refactorarlo para mejorarlo.

- **Integración continua.** Al integrar los cambios producidos durante una iteración, es importante revisar los cambios para validar que cumplen con su especificación en el diseño, de manera que la versión del producto en construcción implemente sus características de forma adecuada.

- **Reuniones al inicio o final de las iteraciones**. Las reuniones de inspección de las iteraciones buscan discutir el progreso y dificultades que hayan tenido los integrantes del equipo, estas discusiones permiten esclarecer la interpretación de

las características del producto y además pueden incluir actividades de revisión de código.

### 16.2. Inspectores de código automáticos

Existen herramientas que ayudan a comprobar la calidad de un código de manera automática. Se basan en las reglas y convenciones de codificación del lenguaje para el que están diseñadas. Estas herramientas se denominan *linters*. Los *linters* asisten en la unificación del estilo de codificación y la aplicación de convenciones de manera uniforme en un repositorio de software (DelBono, 2016).

De esta manera las revisiones se pueden enfocar en la funcionalidad que implementa el código, además el código debería ser legible por todos los desarrolladores que usen el mismo lenguaje y plataforma de trabajo (DelBono, 2016).

En JavaScript se cuenta con el linter ESLint, que está diseñado para trabajar con NodeJS.

https://eslint.org/

### 16.3. Pruebas automáticas de software

En varias metodologías Ágiles los requerimientos se definen a partir de descripciones de alto nivel proporcionadas por los clientes o usuarios. En etapas tempranas del diseño del producto basta con contar con una descripción general de los casos de uso (particularmente desde el punto de vista del usuario) y entender cómo realiza sus actividades para definir funcionalidades en torno a ellas (Martin y Martin, 2007).

Estas descripciones de requerimientos de alto nivel pueden emplearse para definir *pruebas de aceptación* en colaboración con el cliente o usuarios. Estas pruebas de aceptación suelen implementarse en código en un repositorio o sub-repositorio del proyecto, este código no forma parte del producto del proyecto, pero valida que las características que se integran en el satisfacen su especificación (Martin y Martin, 2007).

La definición de estas pruebas suele ser llevada a cabo por analistas del negocio y por roles relacionados al control de calidad. La estructura de este entorno de pruebas se configura de manera que las pruebas son automatizadas, permitiendo que los desarrolladores puedan validar sus productos de trabajo fácilmente al ejecutar las pruebas definidas en el proyecto (Martin y Martin, 2007).

Si los productos de trabajo de los desarrolladores satisfacen las pruebas de las características que deben desarrollar, además de las pruebas que ya terminaban con éxito anteriormente, entonces estos productos de trabajo pueden ser integrados en el producto del proyecto en construcción (o modificación) (Martin y Martin, 2007).

# 17. Servicios y bibliotecas que integran a NodeJS como plataforma de ejecución

## 17.1. Introducción a NodeJS

NodeJS es una plataforma de ejecución basada en JavaScript, toma la filosofía de ejecutar tareas en un único hilo para implementar un servidor sin bloqueo, sin hilos que deban esperarse unos a los otros. La filosofía de NodeJS es que las peticiones hechas a un servidor Web hecho en JavaScript no pueden interactuar ni generar inconsistencias entre sí (DelBono, 2016).

NodeJS fue adoptado rápidamente por una amplia comunidad que incluye entusiastas y varias de las más grandes empresas de tecnología digital, como Microsoft e IBM. Como resultado NodeJS cuenta con un enorme catálogo de bibliotecas, extensiones, soporte y recursos para desarrollar servidores Web (DelBono, 2016).

### 17.1.1. Configuración del entorno de ejecución

Dependiendo del entorno de desarrollo que emplee, puede obtener NodeJS en su sitio Web en variantes para Microsoft Windows, Apple MacOS y el código fuente para su compilación en otras plataformas. Muchas distribuciones Linux cuentan con paquetes para instalar y usar NodeJS sin necesidad de compilarlo.

Una vez que haya instalado NodeJS puede usar el comando `node` para iniciar una terminal interactiva donde puede interpretar código JavaScript utilizando componentes

de NodeJS. También puede llamar `node` con un script `*.js` como argumento para ejecutarlo. Esta última es la estrategia preferida para iniciar aplicaciones Web hechas con NodeJS.

### 17.1.2. Características de NodeJS

A partir del diseño de NodeJS se han desarrollado varias características que en parte heredan del entorno de ejecución de JavaScript (DelBono, 2016).

#### Arquitectura basada en eventos

Las funciones en un servidor hecho con NodeJS suelen estructurarse para ser llamadas por eventos en lugar de ser llamadas directamente. *El flujo de ejecución de un servidor NodeJS depende de acciones externas y de esperar por respuestas*.

#### Un solo hilo de ejecución

Los servidores NodeJS se ejecutan en un único hilo que no depende de otros hilos en marcha.

#### Entrada y Salida sin bloqueo

La arquitectura basada en eventos de NodeJS es general. Esto incluye eventos de entrada y salida. Por ejemplo, mientras el servidor “espera” respuestas por red o de un archivo, NodeJS no implementa una *espera ociosa* en la que el servidor espera en un bucle a que la operación termine de ejecutarse, sino que encola un evento que espera la finalización de la operación, manteniendo el servidor disponible.

## 17.2. Diseñando con el Ciclo de Eventos

Cómo hemos visto NodeJS funciona con un solo hilo de ejecución que encola actividades que toman tiempo (como Entrada/Salida) y las ejecuta en cuanto es posible. Esto significa que debemos tener precaución al definir instrucciones que se ejecutan en el servidor. Debemos emplear herramientas que nos permitan delegar operaciones de segundo plano cuando sea necesario y limitar el servidor a inspeccionar y construir respuestas de peticiones Web (DelBono, 2016).

Para esto NodeJS utiliza un Ciclo de Eventos que permite encolar tareas mediante un grupo de hilos preparado por el Sistema Operativo anfitrión. Al codificar un servidor con NodeJS no es necesario manipular o interactuar directamente este grupo de hilos, en su lugar NodeJS emplea *cerraduras* (closures) que implementan la acción a ejecutar en el Ciclo de Eventos (DelBono, 2016).

En un servidor NodeJS encontraremos la mayor parte del código escrito en *closures* (DelBono, 2016). La figura 9 representa el Ciclo de Eventos de NodeJS.

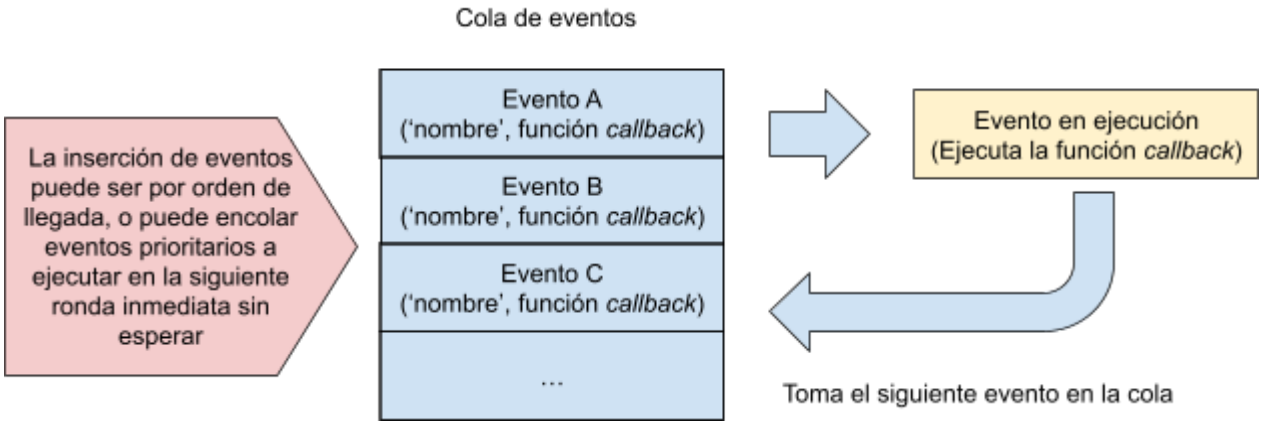


Figura 9. Representación del Ciclo de Eventos de NodeJS.

Estos *closures* o *eventos de respuesta* (callbacks) suelen definirse como funciones anónimas dentro de controladores. Se ejecutan de manera asíncrona y reciben dos parámetros (DelBono, 2016):

- `err` – Objeto con errores que hayan ocurrido al obtener los datos requeridos por esta función. Por ejemplo, si se había solicitado leer un archivo este objeto contendrá información respecto al porqué no se pudo recuperar el archivo o su contenido. Por convención el error es el primer parámetro de un *evento de respuesta*.
- `data` – Los datos requeridos por la función. Siguiendo el ejemplo anterior, si se había solicitado leer un archivo `data` contendrá la información del archivo.

### 17.3. Módulos

Para permitir un mayor control sobre los alcances y dominios entre dependencias y componentes de un servidor NodeJS (pese a la estructura de alcances potencialmente globales de JavaScript), se utiliza el objeto **module.exports** en el que se definen los atributos y funciones "públicas" que se desean exponer de un script a otros componentes (o dependencias). El contenido de `module.exports` que declara un script recibe el nombre de *módulo* (DelBono, 2016).

Las dependencias y componentes de un servidor NodeJS se agrupan y distribuyen en módulos (DelBono, 2016).

**Ejemplo de módulo con NodeJS**

```
module.exports = {
  atributoEjemplo: 42,
  funcionEjemplo: (<parametros>) => {
    <contenido de la función>
  },
  <puede definir más atributos/funciones como parte del módulo>
}
```

Para emplear los miembros definidos en el módulo en otro script:

```
const moduloEjemplo = require('./modulo_ejemplo.js')
let resultado = moduloEjemplo.funcionEjemplo(<parametros>)
```

#### 17.3.1. Importando dependencias

Para traer dependencias a un script se usa la función `require` que toma una cadena con el nombre de una dependencia del proyecto o una ruta de archivo (absoluta o relativa). `require` devuelve el objeto **module.exports** definido en el script que se le indique como parámetro (DelBono, 2016).

Es importante usar notación de ruta siempre que queramos importar un componente y no una dependencia (administrada por npm). Si la cadena que se le provee a `require` no tiene formato de ruta, NodeJS intentará buscarla en el directorio `node_modules` que contiene las dependencias del proyecto (administradas por npm). Cuando se le pasa una ruta, interpreta el script en la ruta dada (por lo que es indispensable proveer rutas a scripts existentes) (DelBono, 2016).

Los scripts se comportan como *singleton*: para optimizar el desempeño se guarda la interpretación del script en un caché y cada vez que el script sea requerido por otros scripts del servidor NodeJS, recibirán la misma referencia. Esto significa que cualquier inicialización o variable en el módulo deben ser diseñados para ejecutarse una sola vez, y el acceso a las variables debería ser protegido con estrategias como *getters* y *setters* (métodos de acceso y modificación) (DelBono, 2016).

### 17.4. Node Package Manager[2] – npm

npm es el gestor de paquetes utilizado en proyectos NodeJS. Cuenta con un extenso catálogo de paquetes que contienen versiones específicas de dependencias. Para instalar un paquete se usa el comando (DelBono, 2016):

```
npm install <nombre del paquete>
```

Los paquetes instalados por npm son depositados en el directorio `node_modules`, donde `require` busca dependencias por nombre (DelBono, 2016).

#### 17.4.1. Express

Express.js es un marco de trabajo para desarrollar aplicaciones Web con NodeJS. Express abstrae algunos aspectos de NodeJS que permiten acelerar la implementación de este tipo de aplicaciones (DelBono, 2016).

#### 17.4.2. Creando un repositorio de proyecto de Aplicación Web

Las funcionalidades de gestión del proyecto de npm van más allá de la gestión de dependencias. También nos ayuda a inicializar el repositorio del proyecto y actualizar su descripción con las dependencias que se instalen. La configuración del proyecto se realiza en un archivo llamado `package.json` que describe entre otros aspectos: el nombre del proyecto, versión y dependencias (DelBono, 2016).

---

[2] Aunque en las primeras versiones de NPM se usaba el nombre "*Node Package Manager*" y ser el nombre con el que lo identifican muchos de sus usuarios, actualmente los gestores del proyecto insisten que NPM *no es un acrónimo*, sino una definición recursiva (NPM is Not an acronym).

Para crear una plantilla de proyecto NodeJS con su archivo `package.json` base, es recomendado primero crear un directorio vacío con el nombre tentativo de la aplicación/proyecto. Luego en el directorio del proyecto se usa el comando (DelBono, 2016):

```
npm init
```

npm hace una serie de preguntas para definir meta-datos del proyecto, como el nombre del paquete, versión y otros meta-datos en el archivo `package.json`. Lo siguiente que se hace es instalar Express en el repositorio recién creado (DelBono, 2016):

```
npm install express
```

Esto crea el directorio `node_modules` en el repositorio, en el que encontraremos los componentes de Express. También se actualiza el archivo `package.json` con el registro de la dependencia y versión instalada. El siguiente paso es distribuir el repositorio con el equipo de trabajo, para ello puede emplear un Manejador de Versiones como git.

El manejo de versiones de un proyecto NodeJS suele excluir el seguimiento del directorio `node_modules` puesto que de esta manera se pueden crear versiones del proyecto que emplean dependencias diferentes (al menos en su versión), o alternativas que eventualmente pueden convertirse en parte de la versión de lanzamiento. El seguimiento de versiones con Git en repositorio se inicializa con el comando:

```
git init
```

El siguiente paso es configurar el gestor de versiones para que ignore el directorio `node_modules`, también puede agregar patrones de nombres de archivos y directorios de cache, depuración y configuración, como `.cache`, `*.log`, `.npm`, `.env*` y `.DS_Store`. Esto evita que los registros de depuración o caché consuman recursos del repositorio de versiones y previene conflictos entre entornos de ejecución, prueba o de trabajo.

En el caso de git esto se hace escribiendo estos nombres o patrones de nombre en un archivo llamado `.gitignore`, colocando cada nombre o patrón en una línea del archivo.

Finalmente solo queda queda agregar el estado inicial del repositorio al seguimiento de estado actual del repositorio. Con git se hace:

```
git add .
```

Y se registra la versión inicial del proyecto:

```
git commit -m “Inicializando repositorio de aplicación Web NodeJS”
```

### 17.4.3. Separando dependencias del producto de las de desarrollo

En un proyecto de software en general es común que el repositorio incluya componentes del producto y componentes de desarrollo, como entornos de prueba, migraciones entre versiones de modelos y base de datos, entre otros. Es posible separar las dependencias del producto de las de desarrollo mediante el archivo de configuración `package.json`. npm también ofrece gestionar estos dos grupos de dependencias por separado.

Al usar el comando:

```
npm install <nombre del paquete>
```

npm asume que se agrega una dependencia del producto, por lo que de forma implícita se está agregando la bandera `--save-prod` al comando. Para explicitar que se agrega una dependencia de desarrollo, se debe usar la bandera `--save-dev` (DelBono, 2016):

```
npm install <nombre del paquete> --save-dev
```

Por ejemplo puede agregar la dependencia `jest`, que es un entorno de pruebas automatizadas para JavaScript:

```
npm install jest --save-dev
```

De esta manera `jest` es registrado como una dependencia de desarrollo y no forma parte del producto.

## 17.5. Bibliotecas y servicios de NodeJS

### 17.5.1. Emisores de eventos

Una manera de interactuar con el Ciclo de Eventos (vea la figura 9) para definir funciones que se encolan en el Ciclo, es usando *Emisores de Eventos*. Para ello se debe importar `events` de los componentes de NodeJS e indicar el nombre de un evento y su función asociada que será encolada para su ejecución en el Ciclo de Eventos (DelBono, 2016).

```
const EventEmitter = requiere('events').EventEmitter
let emitter = new EventEmitter()
emitter.on('nombre-del-evento', <función>)
```

Para encolar una operación asociada a `'nombre-del-evento'`, simplemente se solicita *emitir* un evento, con los parámetros que requiera la función (DelBono, 2016):

```
emitter.emit('nombre-del-evento', <parámetros de la función asociada al evento>)
```

Si se asocia el evento directamente con la función de respuesta, el comportamiento será síncrono, en cuando se llama a `emitter.emit` NodeJS ejecuta la función del evento. Si se espera que el evento sea llamado de forma consecutiva, es mejor configurarlo con una estructura asíncrona que evite apropiarse del tiempo de ejecución del Ciclo de Eventos. Para esto, se puede envolver la función de evento en la función `setImmediate()` (DelBono, 2016)

```
const EventEmitter = requiere('events').EventEmitter
let emitter = new EventEmitter()
emitter.on('nombre-del-evento', <parámetros del evento> => setImmediate(
    () => <función, sus parámetros se alcanzan desde la función externa>)
)
```

`setImmediate()` encola la función que tiene como parámetro en el Ciclo de Eventos, pero no como el siguiente evento a ejecutar, sino al final de la lista de todos los eventos por ejecutar. Esto permite la implementación asíncrona y evitar apoderarnos del tiempo de ejecución del servidor (DelBono, 2016).

Es importante no confundir `setImmediate()` con `nextTick()`, este último encola eventos para ser ejecutados tan pronto como termine de ejecutarse el evento actual en el Ciclo

de Eventos, por lo que `nextTick()` suele usarse principalmente con eventos de alta prioridad (vea la figura 9) (DelBono, 2016).

### 17.5.2. Promesas ECMAScript v6

En la sexta versión del estándar que gobierna la implementación de JavaScript se introdujo el objeto **Promise**. Estas Promesas comparten la filosofía de ejecución asíncrona de NodeJS, se ejecutan en un Ciclo de Eventos (vea la figura 9) y permiten encadenar resultados. Las Promesas usualmente se definen dentro de una función que lo único que hace es devolver la Promesa. El constructor de *Promise* toma una función que recibe dos parámetros, los cuáles son funciones provistas por el entorno de ejecución (DelBono, 2016):

- `resolve` – Se usa para indicarle al servidor el resultado de la ejecución de la promesa para ser devuelto al cliente Web (front-end).
- `reject` – Se usa para generar una respuesta de error cuando la ejecución de la promesa no se pueda completar. La respuesta de error se envía al cliente Web (front-end).

El alcance de función y cerraduras que definen las funciones de JavaScript motivan que la función contenedora de la Promesa sea quien recibe los parámetros que requiere la función a ejecutar asíncronamente la promesa, por lo que la función contenida en el constructor de *Promise* únicamente recibe los dos parámetros anteriores (DelBono, 2016).

```
function miFuncion(<parámetros>) {
  return new Promise((resolve, reject) => {
    <implementación de los mecanismos de "miFuncion" con los parámetros dados>
    if (error) {
      // error representa una validación sobre los resultados generados
      // o la disponibilidad de recursos para generarla (como acceso a la BD)
      reject(error)
      // pasa un objeto con el error a reject
    } else {
      resolve(data)
      // data representa el resultado de la promesa que se devuelve al cliente
    }
  })
```

```
}
```

También es posible encadenar una secuencia de promesas. Esto permite reusar código o definir componentes modulares. Para ello basta con usar el método `then` al invocar una promesa. `then` recibe como parámetro una función que recibe el resultado de la promesa sobre la que fue invocado. El resultado de la promesa es el objeto que se le pase a `resolve` (DelBono, 2016).

```
function otraFuncion(<parámetros>) {
  return new Promise((resolve, reject) => {
    miFuncion(<parámetros>).then(resultado => {
      // Si la ejecución de la promesa del ejemplo anterior no tiene problemas,
      // genera resultado con resolve
    }).catch(error => {    // si la promesa del ejemplo anterior termina en reject
      return reject(error) // se ejecuta este catch en lugar del then anterior
    })
  })
}
```

## 18. Definición de rutas (end-points)

### 18.1. Configuración general de una aplicación Web con NodeJS

Aunque es posible construir una aplicación Web utilizando el módulo `http` de NodeJS, existen varias bibliotecas y marcos de trabajo que ofrecen herramientas para construir aplicaciones ricas en funcionalidad con menor esfuerzo. Uno de los marcos de trabajo más populares para esto es *Express* (DelBono, 2016).

La implementación de una aplicación Web con Express comienza con la definición de un script de configuración general que indica: las rutas y verbos HTTP a los que responde la aplicación, los comportamientos en caso de errores y la escucha de peticiones por algún puerto específico de red (vea la figura 10) (DelBono, 2016).

El nombre por convención que se le da a este archivo de configuración general es `server.js` (DelBono, 2016). El siguiente es un ejemplo de archivo de configuración `server.js` mínimo:

```
const express = require('express')      // se importa el marco de trabajo Express
const app = express()                   // se crea una aplicación Express-NodeJS
app.get('/', (peticion, respuesta) => { // se define una ruta con verbo GET
  respuesta.send('Hola Mundo')          // se genera una respuesta a la petición
```

```
})
app.listen(8000, () => {                // se arranca el servidor
  console.log('Iniciada app de ejemplo que escucha por el puerto 8000')
})
```

Note en el ejemplo que esto no incluye la definición de mensajes de error (por ejemplo, cuando el cliente intenta consultar una ruta no definida -error 404-, o cuando el servidor no puede procesar una petición válida -error 500-), se limita a definir una única ruta `'/'`.

Si se desean definir más rutas, este archivo crecería rápidamente complicando su lectura y mantenimiento. Por esto es mejor definir rutas y respuestas a errores en scripts diferentes, además pueden ser agrupados en el repositorio de maneras convenientes.

### 18.2. Definición de ruteadores

Una aplicación Web ofrece funciones expuestas por diferentes rutas y verbos HTTP en el (los) servidor(es) de la aplicación como se ilustra en la figura 10. Mantener todas estas rutas en el archivo de configuración del servidor `server.js` es poco práctico, en primer lugar es posible que exista un gran número de rutas y muchas de estas rutas pueden estar asociadas a mecanismos relacionados que pueden agruparse en scripts de rutas cohesivas que permiten una fácil identificación de los Controladores que involucran.

Por ejemplo es posible agrupar todas las rutas que procesan altas, bajas y cambios en usuarios, separándolas de las rutas que exponen funcionalidades respecto a otros modelos o características de la aplicación.

`express` proporciona el objeto `Router` que permite definir rutas de la aplicación(DelBono, 2016).

#### Ejemplo de ruteador general con Express

Considere el siguiente ruteador general para una aplicación llamado `./rutas/web.js`

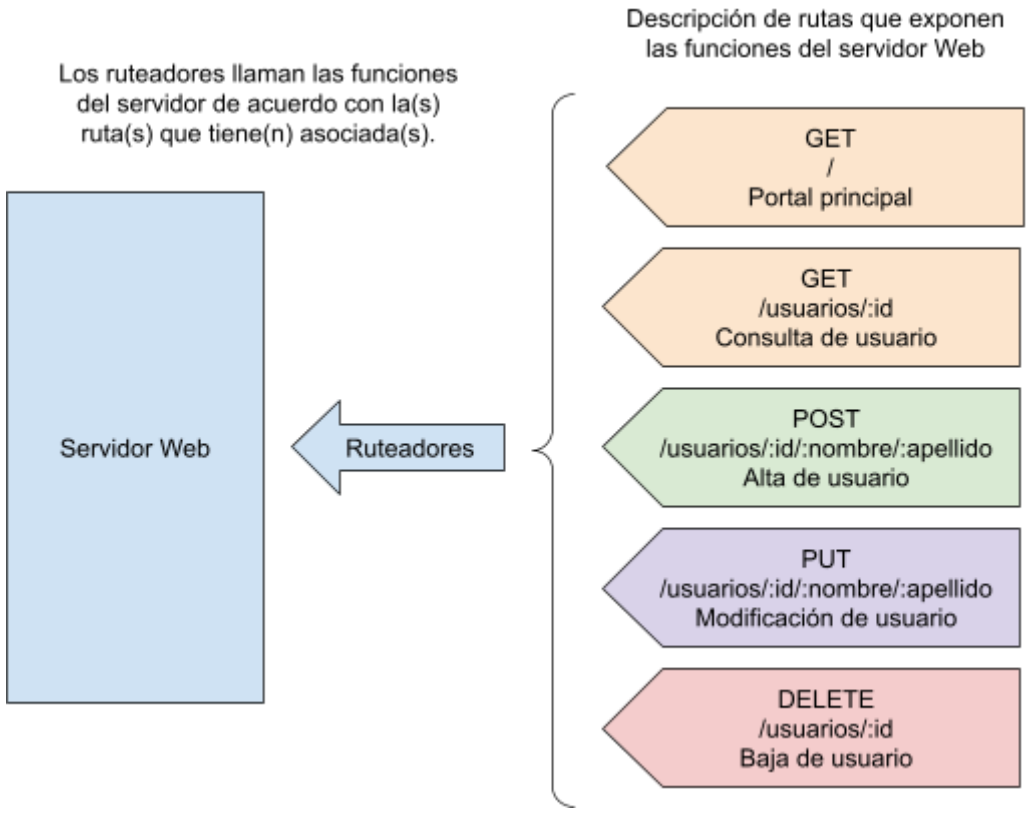


Figura 10. Ejemplo de rutas que exponen las funciones de un servidor Web.
Note que las rutas pueden usar la misma URL con verbos HTTP diferentes.

```
const express = require('express')
const router = express.Router()
// se importan los controladores necesarios, por ejemplo
const controladorUsuarios = require('../controladores/usuario.js')
// definición de rutas, note que se indica el verbo HTTP (.get, .post) y ruta
// En este caso el portal principal de la aplicación
router.get('/', (peticion, respuesta) => {
  respuesta.render('index.ejs', {})
})
// use los controladores para responder a las peticiones cómo sea necesario
router.get('/users', (peticion, respuesta) => {
  controladorUsuarios.mostrarTodos(peticion, respuesta)
})
router.post('/users/registrar', (peticion, respuesta) => {
  controladorUsuarios.registrarNuevo(peticion, respuesta)
})
...
module.exports = router
```

En el archivo de configuración general de la aplicación `server.js` solo es necesario importar este archivo de rutas y asociarlas con la aplicación express, indicando el prefijo

de rutas al que responde el ruteador. En este ejemplo `./rutas/web.js` es el ruteador general de toda la aplicación, por lo que su prefijo es vacío (`/`):

```
const express = require('express')
const rutas = require('./rutas/web.js')
const app = express()
app.use('/', rutas)
```

### 18.2.1. Parámetros en la ruta

Una URL (Localizador Uniforme de Recursos – *Uniform Resource Locator*) puede contener **parámetros** y/o un **fragmento** al final (consulte la estructura de una URI en la figura 5, sección 1.5.1). Los parámetros de la URL son una serie de claves-valores que aparecen después de la ruta del recurso en el mensaje HTTP antecedido por el caracter **?** y separados entre ellos por **&** (Kurose y Ross, 2013; Nottingham, 2020).

```
<protocolo>://<nombre de dominio>/<ruta>?param1=valor1&param2=valor2
```

Estos parámetros suelen emplearse para completar información para la respuesta de la ruta y verbo HTTP solicitados, por ejemplo pueden usarse para realizar búsquedas.

Al final de una URL con o sin parámetros también puede aparecer un *fragmento*, que es un texto antecedido por el caracter **#**, suele usarse para indicar algún comportamiento particular de la ruta solicitada, por ejemplo que la pantalla se desplace a una sección del contenido automáticamente (Nottingham, 2020).

```
<protocolo>://<autoridad – nombre de dominio>/<ruta>[?parámetros]#fragmento
```

En NodeJS es posible definir rutas con elementos variables más allá de los parámetros y fragmentos. La definición de rutas en el ruteador puede incluir nombres de recursos que inician con dos puntos "`:`". Este caracter no es válido como nombre de archivo (inodo) en muchos sistemas de archivos, por lo que encontrar una ruta como `/usuarios/:id` no tiene sentido como identificador de recurso (DelBono, 2016).

Los recursos con el prefijo "`:`" representan variables. El ruteador abstrae cualquier valor que se haya proporcionado en estos `:identificadores` con una variable homónima para procesar la petición. El primer parámetro que recibe la función asociada a una ruta,

usualmente llamado `request` o `peticion`, posee un atributo `params` que es un objeto que contiene el valor de los parámetros en la ruta (DelBono, 2016).

**Ejemplo de definición de rutas parametrizadas con Express**

```
app.get('/usuarios/:nombre/:apellido', (peticion, respuesta) => {
  let nombre = peticion.params.nombre
  let apellido = peticion.params.apellido
  respuesta.send(`Hola ${nombre} ${apellido}`)
})
```

La definición de parámetros dentro de la ruta es mucho más claro que el manejo de parámetros al final de la URL. Puesto que cualquier URL puede incluir parámetros y/o un fragmento, para saber si una petición utiliza parámetros en la URL (y cuales son), es necesario consultar la documentación o revisar con detalle las funciones que atienden la petición.

Al indicar parámetros explícitamente en el ruteador dentro de la ruta, es claro que son obligatorios y cuál es su nombre, los parámetros tradicionales en URL suelen reservarse para parámetros opcionales o que refinan el comportamiento de la función asociada a la ruta.

### 18.3. Construcción de respuestas de mensajes HTTP

Los métodos para responder una consulta, `response.json` y `response.send`, pueden ser encadenados con el método `response.status` para indicar el código de respuesta HTTP que se requiere responder (DelBono, 2016).

```
respuesta.status(200).json(datos)
respuesta.status(404).send('texto')
```

También puede agregar un encabezado al mensaje HTTP de respuesta a la petición con el método `set` (DelBono, 2016):

`respuesta.set(nombreEncabezado, valorEncabezado)`.

Cuando la respuesta a una petición crea un recurso, se debe indicar la URL del recurso creado. Para esto se emplea el método `respuesta.location(ruta)` (DelBono, 2016).

**Ejemplo de generación de respuesta a una petición Web con Express**

```
app.post('/usuarios', (peticion, respuesta) => {
  const usuario = crearUsuario(peticion.body)
  respuesta.location(`/ususarios/${usuario.id}`)
  respuesta.status(201)
})
```

### 18.4. Definición de respuestas de error

En el ruteador principal de la aplicación se definen respuestas a los errores que puede arrojar nuestra aplicación. Los dos errores que suelen ser personalizados en la mayoría de aplicaciones Web, son el error del cliente 404 – Recurso no encontrado, y el error genérico del servidor, error 500.

Puede personalizar la vista del error 404 utilizando la pila de operaciones por la que el servidor procesa la petición. Esta pila de operaciones es un concepto empleado en varios entornos Web, se ejemplifica en la figura 11. En NodeJS la siguiente operación en la pila se representa con un tercer parámetro en la función asociada a una ruta. Este tercer parámetro suele denominarse `next` o `siguiente` en español (DelBono, 2016).

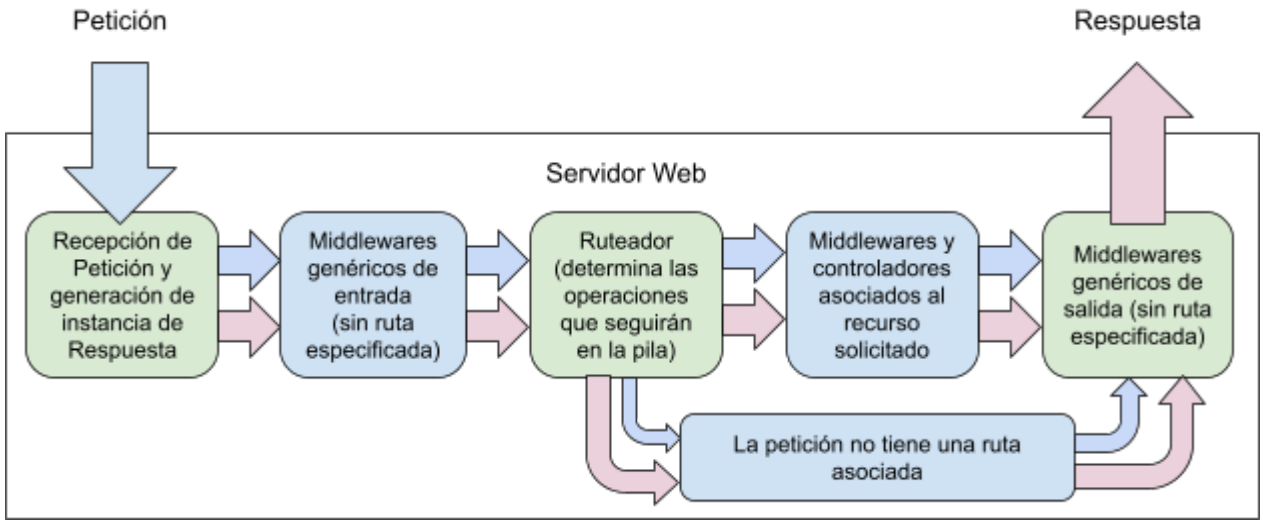


Figura 11. Esquema general de la pila de operaciones de un servidor Web.
Las flechas azules representan la petición, note que es propagada hasta el último evento involucrado en la generación de su respuesta.
Las flechas rojas representan la respuesta, note que se construye desde el primer evento que responde a la petición.

También es posible definir instrucciones en la pila de operaciones no asociadas a una ruta específica. Estas operaciones se denominan *middleware*, suelen usarse para

validar peticiones como se presenta en la sección 19.3. El orden de operaciones en la pila de operaciones se basa en el orden de definición de rutas en los ruteadores y en Express. Por lo anterior, cuando desee asociar un middleware de respuesta de error debe definirlo *al final* de todas las demás rutas, en otro caso el middleware tendrá efecto en todas las rutas que le siguen *(DelBono, 2016)*.

**Ejemplo de manejo de errores al procesar peticiones con *middlewares***

En el siguiente ejemplo se define un *middleware* que define un error en la pila de operaciones, el cuál es disparado cuando no exista una operación en la pila asociada a la ruta de petición. El que no exista una ruta definida para la petición, significa que el cliente solicitó *un recurso que no existe*.

```
app.use((peticion, respuesta, siguiente) => {
  let error = new Error('No encontrado')
  error.status = 404
  siguiente(error)
})
```

El error 500 se puede definir mediante un *middleware* cuya función asociada posee un parámetro adicional en la primer posición de sus argumentos. Este parámetro contiene un error del servidor y el *middleware* es invocado por NodeJS cuando alguna operación en la pila de operaciones encuentra un error o excepción no controlada, este error o excepción se pasa al *middleware* como primer parámetro (DelBono, 2016):

```
app.use((error, peticion, respuesta, siguiente) => {
  if (respuesta.headersSent) return // error pudo ocurrir al construir una respuesta
  respuesta.status(error.status || 500)
  respuesta.json({
    message: error.message,
    error: error
  })
})
```

Note que en el ejemplo el error se envía como respuesta, por lo que será mostrado al usuario. Es recomendado definir al menos dos variantes del manejo de errores del servidor cuya ejecución depende del entorno de ejecución, de manera que en entornos de producción los errores no se muestren explícitamente a los usuarios, pero se muestren con detalle en entornos de prueba o desarrollo.

### 18.5. WebSockets

*WebSockets* es un estándar de comunicación Web (basado en los protocolos HTTP/HTTPS) en los que ambos participantes en la comunicación actúan tanto como cliente como servidor. Permite un mayor grado de interacción entre el servidor y los usuarios. Existen muchas bibliotecas para trabajar con WebSockets, en NodeJS se puede usar Socket.IO, disponible a través de npm (DelBono, 2016):

```
npm install socket.io
```

Puesto que con WebSockets ambas partes de una interacción HTTP fingen tanto como cliente y servidor es necesario definir rutas y peticiones tanto en el servidor como en las vistas. En el servidor se puede definir un ruteador para WebSockets, es conveniente agrupar los ruteadores del repositorio en un directorio común, por ejemplo:

```
<repositorio del proyecto>
    /rutas
        web.js
        sockets.js
```

Un ruteador de WebSockets en una aplicación NodeJS posee la siguiente estructura (DelBono, 2016):

```
const http = require('http').Server(app)
const io = require('socket.io')(http)
const controladorSockets = require('../controladores/controladorSockets.js')
io.on('connection', socket => {
  socket.on('nombre-evento', controladorSockets.nombreEvento(datos))
  // datos es el contenido del mensaje enviado por WebSocket
  // es una cadena de formato arbitrario, en muchos casos se usa JSON.
  // Puede definir más eventos con llamadas adicionales a socket.on
})
```

Del lado del cliente no suele ser conveniente definir un ruteador para todos los eventos de WebSockets, ya que podrían contener referencias a características que no son accesibles desde cualquiera de las vistas de la aplicación, pero el ruteador tendría que ser enviado al cliente Web en todas las peticiones que puedan involucrar WebSockets.

A diferencia del servidor, donde el ruteador dispone del resto de los componentes directamente, por cuestiones de desempeño de la aplicación en ejecución tanto en

clientes como en servidores y por cuestiones transferencia de datos por red, las vistas únicamente deben cargar dependencias indispensables.

Por lo anterior del lado del cliente es más conveniente definir scripts que procesan la interacción para cada vista con la que están relacionados, limitando también el tamaño de los scripts que carga cada vista.

**Ejemplo de ruteador WebSockets para una vista Web (en navegador)**

El siguiente script inicia la conexión de WebSockets y prepara una función que envía el evento `'nombre-evento'` al servidor (DelBono, 2016):

```
var socket = io.connect(urlServidor) // inicia la conexión con el servidor
// rutas de evento del servidor
socket.on('evento-enviado-servidor', procesaEventoServidor(datos))
...
// funciones que procesan eventos enviados por el servidor
function procesaEventoServidor(datos) {
  ...
}
// funciones que disparan eventos que se envían al servidor
function disparaEventoAlServidor(contenido) {
  let datos = {
    ejemplo: 'Soy un JSON',
    contenido: contenido
  }
  socket.emit('nombre-evento', datos)
}
```

Note que *io* no está definido en este script. Por medio del encabezado HTML/XHTML se definen las dependencias de las vistas, en este caso debe incluirse la referencia a Socket.IO. El orden en el que se definen las dependencias en vistas es importante, por lo que Socket.IO debe aparecer antes que el script anterior. De lo contrario *io* sería una variable no definida, la vista arrojaría errores y no se comportaría de la manera esperada (DelBono, 2016).

```
<html>
  <head>
    …
    <script src="/js/socket.io/socket.io.js" ></script>
    <script src="/js/ejemploWebSockets.js" ></script>
  </head>
  …
</html>
```

## 19. Definición de Controladores y Modelos

### 19.1. Modelos

Como se presenta en la sección 18.5 los componentes del servidor como el ruteador, modelos y controladores, son separados en directorios que agrupan componentes de manera cohesiva y a su vez pueden agruparse en subdirectorios conforme sea conveniente. Los modelos suelen manejarse como dependencias de los controladores, definiéndolos cómo objetos y aplicando el principio de encapsulación con prácticas de JavaScript, de manera que los controladores no puedan manipular los modelos arbitrariamente (Crockford, 2008; DelBono, 2016).

Recuerde que en JavaScript los atributos de un objeto no son públicos ni privados, sino que se emplean cerraduras para crear espacios de memoria reservados (vea la sección 14.4) (Crockford, 2008; DelBono, 2016).

**Ejemplo de un modelo simple de usuario**

```
module.exports = (idUsr, nombreUsr, apellidoUsr) => {
  let id = idUsr.valueOf()
  let nombre = nombreUsr.valueOf()
  let apellido = apellidoUsr.valueOf()
  return {
    obtenerId: () => {
      return id
    },
    obtenerNombre: () => {
      return nombre
    },
    obtenerApellido: () => {
      return apellido
    },
    toString: () => {
      return `${nombre} ${apellido}`
    }
  }
}
```

El modelo de usuario definido en el ejemplo anterior posee tres atributos: un identificador (`id`) un nombre de pila (`nombre`) y un apellido (`apellido`). Cuando se instancia este modelo, se copia el valor del identificador, nombre de pila y apellido provistos y devuelve la instancia del modelo.

Los valores de los atributos de la instancia no pueden ser modificados (el objeto no define métodos de modificación para ninguno de los atributos), únicamente pueden ser consultados por separado (usando los métodos de acceso o lectura para cada atributo) o en una representación textual (definida por el método `toString`).

## 19.2. Controladores

Los controladores son componentes de un servidor Web que reciben peticiones Web y construyen una respuesta de acuerdo con el contenido de cada petición que procesan.

### Ejemplo de un controlador simple

El siguiente controlador utiliza el modelo del ejemplo anterior para atender peticiones que procesan acciones sobre los usuarios que describe el modelo (DelBono, 2016; Fowler, 2003):

```
const modeloUsuario = require('../modelos/usuario.js')
var usuarios = {}
module.exports = {
  crearUsuario: (peticion, respuesta) => {
    if (usuarios[peticion.params.id]) {
      respuesta.status(409).json({
        name: 'ID de usuario repetido',
        message: `Ya existe el usuario con ID ${peticion.params.id}`
      })
      return
    }
    usuarios[peticion.params.id] = modeloUsuario(
      peticion.params.id, peticion.params.nombre, peticion.params.apellido
    )
    let urlNuevoUsuario = `/usuarios/${peticion.params.id}`
    respuesta.status(201).send(urlNuevoUsuario)
  },
  borrarUsuario: (peticion, respuesta) => {
    delete usuarios[peticion.params.id]
    let urlUsuarioBorrado = `/usuarios/${peticion.params.id}`
    respuesta.send(urlUsuarioBorrado)
  },
  obtenerUsuario: (peticion, respuesta) => {
    if (!usuarios[peticion.params.id]) {
      respuesta.status(404).json({
        name: 'Usuario inexistente',
        message: `No existe el usuario con ID ${peticion.params.id}`
      })
      return
    }
    let usr = usuarios[peticion.params.id]
```

```
    respuesta.send(usr.toString())
  }
}
```

El controlador se emplea en el ruteador para asociar las peticiones entrantes de acuerdo a la ruta a la que apunta cada petición (DelBono, 2016):

```
const controladorUsuarios = require('./controladores/usuario.js')
router.post('/usuarios/:id/:nombre/:apellido', (peticion, respuesta) => {
  controladorUsuarios.crearUsuario(peticion, respuesta)
})
router.delete('/usuarios/:id', (peticion, respuesta) => {
  controladorUsuarios.borrarUsuario(peticion, respuesta)
})
router.get('/usuarios/:id', (peticion, respuesta) => {
  controladorUsuarios.obtenerUsuario(peticion, respuesta)
})
```

### 19.3. Middleware

En el ejemplo anterior aparecen diferentes rutas asociadas a operaciones relacionadas, operaciones sobre el registro de usuarios. Esta es una situación común, particularmente respecto a la validación de las peticiones, el proceso de verificar que el contenido de la petición es adecuado conforme a las precondiciones del controlador asociado a la ruta.

A través del manejo de sesiones para las peticiones pueden distinguirse tipos de usuarios, por ejemplo usuarios sin sesión, que suelen tener acceso a funciones limitadas como consultar información general o crear una cuenta, y usuarios registrados que tienen acceso mayor o total a las funciones de la aplicación. Entre usuarios registrados también puede haber varios roles, como usuarios en un plan gratuito y distintos roles de usuarios con mayores privilegios por planes de suscripción, programas de recompensa o lealtad.

Para validar una sesión en cada petición entrante al ruteador, determinando si debe proceder al controlador respectivo o devolver un error 4XX puede usar *middlewares*. Por ejemplo, el error 401 se usa para indicar que una ruta requiere una sesión y el 403 indica que se conoce la identidad del usuario pero no posee permisos para acceder al contenido solicitado (vea la sección 1.5.2) (DelBono, 2016).

**Ejemplo simplificado de middleware que comprueba sesiones de usuario**

```
function compruebaPermisos(peticion, respuesta, siguiente) {
  // comprobamos permisos con la información en la petición
  // el resultado se almacena en la variable local “ok”
  if (ok) {
    siguiente()
  } else {
    respuesta.status(401)
  }
}
```

En los ruteadores se importan los middleware para asociarlos con rutas de la aplicación. Por ejemplo (DelBono, 2016):

```
const middlewareEjemplo = require(‘./middleware/middlewareEjemplo.js’)
const controladorEjemplo = require(‘./controladores/controladorEjemplo.js’)
…
router.get(‘/users/:id’, middlewareEjemplo.compruebaPermisos,
        (peticion, respuesta) => {
  // note que el middleware se indica al deifinir la ruta
  controladorEjemplo.obtenerUsuario(peticion, respuesta)
})
```

Si desea que el middleware tenga efecto sobre un grupo de rutas, se agregan las funciones sin asociarlas con una ruta antes que el resto de las rutas sobre las que se desea que tenga efecto (DelBono, 2016):

```
app.use(middlewareEjemplo.compruebaPermisos)
```

Note que lo anterior implica que es posible definir *middlewares* globales, que tienen efecto sobre todas las rutas que soporta el servidor, si se definen antes que cualquier otra ruta.

Otra aplicación de los *middlewares* es completar información en la petición o preprocesar la petición antes de que la reciba su controlador objetivo (DelBono, 2016).

## 20. Elementos DOM

### 20.1. Representación de HTML/XHTML en tiempo de ejecución

Como se presenta en la sección 3.2, cuando un navegador Web recibe un documento HTML/XHTML debe interpretarlo y obtener un modelo que le permita visualizarlo

conforme a sus atributos y dependencias, además de exponer este modelo al entorno JavaScript que permite manipular la vista al vuelo (sección 11.1). Como se muestra en la figura 7, los navegadores modelan estos documentos mediante un árbol en el que cada nodo es un elemento del documento HTML/XHTML. Este modelo se denomina *Document Object Model* (Modelo de Objeto del Documento – DOM) (Dean, 2019).

En DOM cada elemento del HTML/XHTML se representan con un nodo, los atributos de cada etiqueta son sub-nodos hoja de los nodos-etiqueta, así como el contenido textual y elementos comentarios. El elemento raíz del árbol DOM es un miembro distinguido llamado *document*, que suele tener un único descendiente directo llamado *html* (Dean, 2019).

*document* representa la totalidad del documento HTML/XHTML y posee atributos especiales, mientras que *html* representa el nodo raíz en el documento HTML/XHTML del que descienden todos los demás nodos en el documento (tales como *head*, *body*, *div*, *form*, *h1*, *span*, *input*, etc). Dada la sintaxis de HTML/XHTML el nodo *html* tiene exactamente dos descendientes: *head* y *body* (Dean, 2019).

*head* es el encabezado del documento HTML/XHTML, contiene meta-información del documento como: la codificación de su contenido (por ejemplo UTF-8), las hojas de estilo que definen su presentación y los scripts que gestionan el documento y su interacción con el usuario. Por su parte *body* es la etiqueta cuyos descendientes son el contenido del documento y representan una vista Web (Dean, 2019).

```
<!DOCTYPE html>
<html lang=”es” >
  <head >
    <meta charset=”utf-8” >
    <title >
      Texto que aparece en la pestaña o encabezado del navegador
    </title>
  </head>
  <body >
    <h1 >T&iacute;tulo que se muestra al usuario</h1>
    <p>P&aacute;rrafo de texto.</p>
    <!-- Comentario HTML/XHTML -->
  </body>
</html>
```

## 20.2. Manipulación de vistas Web en tiempo de ejecución

Mediante JavaScript es posible explorar el árbol DOM que modela un documento HTML/XHTML a partir de su nodo raíz (*document*), también es posible modificar su contenido, los cambios suelen tener efecto de manera inmediata en la interfaz del navegador Web (Dean, 2019).

### Ejemplo de manipulación de una vista con JavaScript en el navegador

El siguiente script recorre todos los nodos de un árbol DOM generado a partir de cualquier documento HTML/XHTML (note que la exploración en este ejemplo es de tipo *a profundidad* - *Depth First Search* DFS) (Crockford, 2008):

```
function recorreDom(nodo) {
  console.log(nodo.nodeName)
  nodo = nodo.firstChild
  while (nodo) {
    recorreDom(nodo)
    nodo = nodo.nextSibling
  }
}
recorreDom(document)
```

`document` es una variable global alcanzable por cualquier script que invoque un navegador Web. Los atributos que se muestran permiten recorrer el árbol DOM accediendo a los descendientes directos y posteriormente a sus vecinos (puesto que es DFS).

Los nodos que representan elementos del documento poseen el atributo `nodo.nodeName`, contiene el nombre de la etiqueta HTML/XHTML que representa el elemento DOM (como *head*, *body*, *div*, *form*, *h1*, *span*, *input*, etc). En el caso de nodos de contenido textual, el valor de `nodo.nodeName` es `"#text"`, mientras que los nodos que representan comentarios tienen como valor `"#comment"` (Crockford, 2008).

Los elementos DOM ofrecen capacidades de búsqueda a partir del valor de sus identificadores (`id="<identificador>"`), nombres (`name="<nombre>"`), clases (`class="<clase>"`) o el nombre de la etiqueta en el documento HTML/XHTML con el que corresponde el elemento (`<nombreEtiqueta atributos ... />`) (Dean, 2019).

El valor de los atributos `id` de los nodos debe ser único en el documento, por lo que la búsqueda por `id` devuelve un único elemento (o `null` si no se encuentra ningún elemento con el `id` proporcionado). El resto de los métodos de búsqueda devuelven una lista con todos los elementos que tienen el atributo proporcionado (que es vacía si no hay coincidencias). Estos métodos de búsqueda son (Dean, 2019):

- `elemento.getElementById(id)`
- `elemento.getElementsByName(nombre)`
- `elemento.getElementsByClassName(clase)`
- `elemento.getElementsByTagName(etiqueta)`

Los elementos DOM también contienen atributos que representan los atributos incluidos en las etiquetas HTML/XHTML.

**Ejemplo de manipulación de nodos DOM con JavaScript en el navegador**

Considere un campo de entrada de datos como la siguiente:

```
<input type=”text” id=”entradaTextual” name=”entradaTextual”
       maxLength=”150” />
```

En JavaScript es posible acceder a los atributos de un elemento utilizando el operador de acceso y el nombre del atributo. Tome en cuenta que se distinguen mayúsculas de minúsculas por lo que `maxlength` y `maxLength` son dos atributos diferentes y solo el último está definido en el elemento DOM que representa la etiqueta anterior:

```
var entradaTextual = document.getElementById(‘entradaTextual’)
console.log(entradaTextual.type)         // “text” (String)
console.log(entradaTextual.id)           // “entradaTextual” (String)
console.log(entradaTextual.name)         // “entradaTextual” (String)
console.log(entradaTextual.maxLength)    // 150 (Number)
console.log(entradaTextual.class)        // undefined (undefined)
```

También es posible acceder al contenido textual de un elemento DOM. Para ello se usa el atributo `elemento.innerText`. También puede asignar valores a estos atributos, el valor es reflejado automáticamente en el DOM y la vista Web. Si se asigna el atributo `elemento.class`, podrá ver que cambia la presentación del elemento en la vista (dependiendo de las clases CSS cargadas por la vista).

### 20.3. Personalizando la presentación de vistas Web

En la sección anterior se menciona el atributo `class` que suele tener una presencia significativa en documentos HTML/XHTML. Este atributo se relaciona con otro tipo de dependencias que tiene una vista Web: Hojas de Estilo. Las hojas de estilo suelen ser archivos CSS (Hojas de estilo en Cascada – *Cascading Style Sheet*) que definen cómo mostrar la vista en el navegador, definiendo su apariencia (Dean, 2019).

Las hojas de estilo se incluyen en las vistas como dependencias que se especifican con la etiqueta `link` en el encabezado (*head*) de un documento HTML/XHTML:

```
<!DOCTYPE html>
<html lang=”es” >
  <head >
    …
    <link rel=”stylesheet” href=”ruta_archivo.css” type=”text/css” >
  </head>
  <body>
    ...
  </body>
</html>
```

También es posible definir reglas de estilo dentro del documento HTML/XHTML. Sin embargo para habilitar modularidad y una mejor organización, es recomendado únicamente incluir estilos dentro de un HTML/XHTML cuando definen reglas específicas para el documento. Las reglas de estilo se definen en el encabezado del documento mediante la etiqueta `style` o usando el atributo `style` en los elementos de la vista que requieran alguna regla específica (Dean, 2019).

```
<!DOCTYPE html>
<html lang=”es” >
  <head >
  …
    <style >
      * {
        text-align: center;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    ...
    <h2 style=”text-decoration:underline;background-color:#0f42ba” >
      T&iacute;tulo secundario
```

```
    </h2>
  </body>
</html>
```

La definición de una regla se asocia con un *selector* (Dean, 2019):

- **Selector de tipo**. Son reglas que se aplican al tipo de elemento HTML/XHTML con el que deben ser asociadas. Por ejemplo:

  ```
  div {
      width: 50%;
      font-style: italic;
  }
  ```

- **Selector de identificador**. Estas reglas se aplican en el elemento que posea el identificador que define el selector. Estos selectores se caracterizan por iniciar con el carácter `#` seguido sin espacios por el valor esperado del `id`. Por ejemplo:

  ```
  #nombre-identificador {
      font-size: x-large;
      border: 1px solid black;
  }
  ```

- **Selector de clase**. Este tipo de selector es el que se usa con el atributo `class` que aparece en la sección 20.2. Las reglas que se definen con un selector de clase únicamente tienen efecto sobre los elementos HTML/XHTML que explícitamente indiquen la clase en su atributo `class`. Los elementos pueden aplicar varias clases CSS en su atributo `class`, para ello se escriben las clases separadas por un espacio. Por convención los nombres de clases de estilo se definen en minúscula y si incluyen varias palabras, se separan con un guión medio. Estos selectores se caracterizan por iniciar con el caracter `.` seguido por el nombre de la clase. Por ejemplo:

  ```
  .nombre-clase {
      color: blue;
      font-family: Impact;
  }
  ```

- **Selector universal**. Se aplica sobre todos los elementos del documento HTML/XHTML. Se define con el comodín `*`, por ejemplo:

```
* {
    line-height: 2;
    letter-spacing: 1px;
}
```

Existen otros tipos de selectores. Los selectores presentados pueden ser especificados para que tengan efecto sobre un conjunto más específico de elementos del documento HTML/XHTML. En un bloque de definición de estilos es posible definir varias reglas, separándolas por `;`. Por convención se prefieren sangrías con 4 espacios cada regla dentro del bloque y definir una sola regla por línea (Dean, 2019).

También es posible definir reglas dirigidas a varios tipos de elementos, definiendo el bloque de reglas con varios selectores separados por comas (Dean, 2019), por ejemplo:

```
.nombre-clase, div, span {
    background: #777799;
    cursor: progress;
}
```

## 21. Bibliotecas para el desarrollo de vistas Web

### 21.1. Bibliotecas para vistas del lado del cliente

La manipulación de elementos DOM mediante el API estándar que ofrecen los navegadores puede convertirse en una tarea tediosa, en particular debido a que dicho API puede tener diferencias entre navegadores. Existen varias bibliotecas que ayudan a abstraer el API de DOM para el desarrollo de vistas dinámicas con menor esfuerzo. Una de las bibliotecas más populares para este propósito es *JQuery*.

### 21.2. JQuery – JavaScript para manipulación de DOM

JQuery comenzó como una utilidad para buscar elementos DOM de manera más flexible que los métodos del API estándar `getElement[s]By[Id|Name|Class|TagName]`, de allí su nombre *JavaScript Query*. JQuery ha evolucionado para incorporar una serie de funcionalidades basadas en envolver elementos DOM en un objeto especial llamado *selector* que permite manipular el elemento sin importar el navegador, simplificando también la gestión de eventos y cambios de estado del documento HTML/XHTML (Resig, 2005).

Los selectores de JQuery emplean un caché que optimiza la exploración y manipulación del documento HTML/XHTML, cuando se usa apropiadamente este caché evita impactos negativos en el tiempo de ejecución e incluso puede mejorar el desempeño de las vistas Web.

### 21.2.1. Importando JQuery como dependencia de las vistas Web

Aunque en general la disponibilidad de los servidores de JQuery es confiable, es recomendado que cualquier dependencia de una aplicación Web sea instalada en los servidores que la ejecutan. Por lo anterior el primer paso para usar JQuery es descargarlo, prefiriendo la versión más reciente durante el desarrollo o mantenimiento de la aplicación Web en cuestión.

En la página de descarga de JQuery se hospedan variantes de cada versión lanzada, para usar JQuery como dependencia se recomienda usar la distribución *comprimida de producción*, pues son versiones de lanzamiento de tamaño reducido que mejoran el desempeño de la carga de las vistas en que se usan.

https://jquery.com/download/

Una vez que instale el archivo *jquery-x.x.x.[min.]js* en un directorio público del servidor Web, se importa en las vistas que lo requieran mediante el encabezado del documento HTML/XHTML, de manera similar a como se importan *WebSockets* en el tema 18.5:

```
<html>
  <head>
    …
    <script src=”/js/jquery/jquery-x.x.x.js” ></script>
    …
  </head>
  …
</html>
```

Una vez que el documento importe JQuery, todos los scripts que se definan en el resto del encabezado o en el cuerpo del documento HTML/XHTML pueden hacer uso de la variable global `$` que expone el API de JQuery, permitiendo explotar sus funcionalidades.

### 21.2.2. Selectores – Objetos JQuery

El uso de JQuery se basa en envolver elementos DOM en objetos *selectores*, recopilando los elementos DOM en un objeto similar a un arreglo. Por ejemplo puede envolver elementos DOM con los siguientes selectores (js.foundation, 2023a):

```
<script >
  var selectorId = $(‘#un-identificador’)
  var selectorTipo = $(‘h1’) // h1 es un tipo de elemento HTML/XHTML
  var selectorClase = $(‘.nombre-clase-css’)
  var selectorAtributo = $(‘[name=”usuario”]’)
  var selectorCombinado = $(‘div.clase-esperada-de-los-div’)
</script>
```

Los selectores anteriores producen los siguientes resultados (js.foundation, 2023a):

- `selectorId` – El selector por identificador se caracteriza por iniciar con el caracter **`#`**. El resultado contiene uno o ningún elemento DOM cuyo atributo id corresponde con el resto de la cadena provisto al identificador después del prefijo **`#`**.

  Este selector es equivalente a usar la función estándar `document.getElementById(id)` (vea la sección 20.2).

- `selectorTipo` – El selector de tipo se caracteriza por contener el nombre de una etiqueta en el documento HTML/XHTML. El resultado contiene todos los elementos DOM cuya etiqueta HTML/XHTML corresponde con la cadena del selector.

  Este selector es equivalente a usar la función estándar `document.getElementsByTagName(etiqueta)` (vea la sección 20.2).

- `selectorClase` – El selector de clase inicia con el caracter `.`. El resultado contiene los elementos DOM cuyo atributo `class` corresponde con el nombre de clase CSS indicada en el selector.

  Este selector es equivalente a usar la función estándar `document.getElementsByClassName(clase)` (vea la sección 20.2).

- `selectorAtributo` – El selector de atributo se expresa entre corchetes cuadrados como una relación `[atributo=valor]`. El resultado contiene todos los elementos DOM que contengan el atributo y valor especificados en el selector.

  No existe una función de búsqueda estándar en el API DOM equivalente a este selector, la más cercana es `document.getElementsByName(nombre)` pero solo cuando el atributo de selección es `name` (vea la sección 20.2).

- `selectorCombinado` – Es posible combinar selectores de JQuery. En el ejemplo anterior se combina el selector de tipo con el de clase, de manera que JQuery primero buscará los elementos con la etiqueta HTML/XHTML especificada (en este caso `<div />`) y los filtrará dependiendo si poseen la clase especificada en el selector o no (en este caso `'clase-esperada-de-los-div'`).

Existen otros tipos de selectores, variantes de algunos de los presentados (en particular existen muchas variantes del selector de atributo) y combinaciones de ellos. Es recomendado utilizar los selectores más específicos para cada caso de uso, evitando obtener elementos DOM inesperados y mejorando el desempeño de los selectores (js.foundation, 2023a).

Los objetos selectores devueltos permiten manipular los elementos DOM de manera individual o grupal. Es posible actualizar todos los elementos en el selector en una sola operación manipulando el selector como si fuera un único elemento DOM. Esto simplifica la manipulación del documento HTML/XHTML, en muchos casos es más eficiente que usar el API DOM estándar del navegador (js.foundation, 2023a).

Para manipular los atributos de los elementos DOM, los métodos de un selector JQuery ofrecen dos comportamientos:

- Si no reciben parámetros, se comportan como una consulta (*getter*) y devuelven el valor del atributo del elemento DOM con el que corresponde el método.

- Si recibe parámetros, se comporta como un modificador (*setter*) del atributo del elemento DOM.

**Ejemplo de manipulación de presentación de vistas con JQuery**

```
<script >
  // Se accede al tipo del atributo.
  // Si no tuviera un atributo type, devuelve undefined
  let tipo = selector.attr('type')
  // También puede usar attr() para leer las clases CSS actuales, almacenarlas
  // como cadena de texto, agregar otra separada con un espacio y finalmente usar
  // attr() nuevamente para agregar la clase nueva, o simplemente puede usar:
  selector.addClass('clase-css-nueva')
  // define una longitud máxima de contenido de 150 caracteres
  selector.attr('maxLength', 150)
</script>
```

Los selectores de JQuery también permiten explorar el árbol DOM que modela el documento HTML/XHTML y hacer modificaciones en él. Por ejemplo, la siguiente función recorre el árbol DOM de la misma manera que la presentada en el tema 20.2 usando JQuery:

```
function recorreDomJquery(selector) {
  console.log(selector.prop('tagName'))
  selector = selector.children().first()
  while (selector.length) {
    recorreDomJquery(selector)
    selector = selector.next()
  }
}
recorreDomJquery($(document))
```

Note que en esta función se construye un selector de manera directa proporcionando un elemento DOM a `$`, en este caso el nodo global `document`. Puede envolver cualquier elemento en un selector para operarlo como un objeto de JQuery.

```
$(elementoDOM)
```

## 21.3. Bibliotecas para vistas del lado del servidor

En la sección 19 se presenta cómo construir controladores y *middleware* que generan una respuesta a las peticiones del cliente con las funciones `Express.Response.send(string)` y `Express.Response.json(Object)`. Aunque es posible usar estas funciones para enviar texto que represente documentos HTML/XHTML con las vistas de la aplicación, existen bibliotecas que nos permiten organizar, generalizar y producir vistas de manera eficiente.

## 21.4. Plantillas de JavaScript embebido

**ejs** (***e****mbedded* ***J****avaScript* ***t****emplates*) es una biblioteca para generar vistas conforme al sistema de vistas de Express. Permite definir vistas en archivos independientes, agrupandolas como el resto de los componentes de una aplicación NodeJS/Express (Eernisse, 2023a).

Estas plantillas de vistas usan la extensión de nombre de archivo `.ejs`. La generación de vistas con *ejs* ofrece una forma de trabajo muy similar a la de otros marcos de trabajo para la Web (Eernisse, 2023a).

Para agregar *ejs* a un proyecto, primero se debe instalar con npm:

```
npm install ejs
```

Lo siguiente es configurar la aplicación NodeJS/Express para que genere vistas con *ejs*. Esto se hace modificando el archivo `server.js` con el siguiente contenido resaltado:

```
const ejs = require('ejs')
…
const app = express()
app.set('view engine', 'ejs')
app.set('views', './vistas')
…
```

En la primera línea se importa la dependencia *ejs*. Después de instanciar la aplicación, se indica que el motor de vistas es *ejs* y finalmente se especifica el directorio donde Express debe buscar las vistas. En este caso se usa el directorio "`vistas`", localizado en el directorio principal del proyecto (el mismo que contiene a `server.js`, tal como sugiere la ruta del directorio en la configuración del ejemplo) (Eernisse, 2023b).

### 21.4.1. Definiendo vistas en archivos .ejs

Los archivos de plantillas de vistas `.ejs` se interpretan como contenido HTML/XHTML salvo que se indique lo contrario mediante bloques de código JavaScript. Por ejemplo es posible generar una tabla desde código sin que sea necesario definir su estructura HTML/XHTML por completo. Más adelante en la sección 21.4.3 se presenta como generar vistas dinámicas en tiempo de ejecución (Eernisse, 2023b).

### Ejemplo de vista ejs

`./vistas/index.ejs`

```
<!DOCTYPE html>
<html lang="es" >
  <head >
    <meta charset="utf-8">
    <title>Portal - Aplicación de ejemplo de NodeJS CIDW</title>
  </head>
  <body>
    <div >
      <p>
        <%= '¡Mira mamá, estoy en la Web!' %>
      </p>
    </div>
  </body>
</html>
```

En la vista del ejemplo anterior se define una pantalla cuyo único contenido es un párrafo de texto. Sin embargo este texto no es una parte estática de la vista, está siendo generado como una cadena de JavaScript que se agrega como el contenido del párrafo al interpretar el archivo `.ejs`.

Para enviar una vista al usuario se emplea el método `Express.Response.render(string, object)`. Por ejemplo, para emplear la vista anterior como portal principal en una aplicación, se modifica la respuesta del ruteador asociada a la ruta principal `/` para devolver la vista del ejemplo (StrongLoop et al., 2017):

`./rutas/web.js`

```
...
const router = express.Router()
router.get('/', (peticion, respuesta) => {
    respuesta.render('index')
})
...
```

Note que no se incluye el nombre del directorio `vistas` ni la extensión del archivo `.ejs`, sino que únicamente se indica el nombre de archivo `index`. Esto es porque en la configuración establecida en el archivo `server.js` en la sección 21.4, se ha especificado que las vistas están contenidas en el directorio `./vistas`, NodeJS automáticamente

busca las vistas en él y no es necesario incluirlo en el nombre de la vista a importar, al igual que la extensión del archivo `.ejs` (StrongLoop et al., 2017).

Observe también que se omite el segundo parámetro de `render(string, object)`. Esto es porque el segundo parámetro se usa para pasar información a la vista, como se presenta en la sección 21.4.3. El contenido de la vista del ejemplo no depende del estado del servidor ni de la petición, no utiliza parámetros. En estos casos es posible omitir el segundo parámetro de la función `render`.

### 21.4.2. Creando plantillas para las vistas

La generación de vistas puede depender de varios archivos `.ejs` que pueden definir partes específicas de las vistas. Esto suele aprovecharse para definir *plantillas* de elementos comunes en las aplicaciones, por ejemplo, encabezados y pies de página que deben aparecer en todas las pantallas (Eernisse, 2023b).

De esta manera no es necesario reescribir código, eliminando también el riesgo de introducir inconsistencias al tener que realizar cambios uniformes en la presentación de las aplicaciones (Eernisse, 2023b).

**Ejemplo de plantilla para crear un pie de página estandarizado**

`./vistas/comunes/pie_pagina.ejs`

```
<footer >
  <hr>
  <div >
    <p>Aplicación de ejemplo de NodeJS CIDW</p>
  </div>
</footer>
```

Para emplearlo en una vista, por ejemplo el portal definido en la sección anterior, se usa el siguiente bloque resaltado dentro del archivo `.ejs`, en este caso `./vistas/index.ejs` del ejemplo anterior:

```
...
        <%= '¡Mira mamá, estoy en la Web!' %>
      </p>
    </div>
    <%- include('comunes/pie_pagina') %>
  </body>
```

```
</html>
```

### 21.4.3. Creando vistas dinámicas

La presentación de las vistas de las aplicaciones puede variar dependiendo del estado del servidor y/o de las peticiones del usuario. Mediante código JavaScript en las vistas *ejs* pueden usarse variables para parametrizar y personalizar su presentación (Eernisse, 2023b).

**Ejemplo de vista dinámica con contenido que depende del estado de un modelo**

Retomando los controladores de ejemplo desarrollados en la sección 19.2, se define una vista que muestra una tabla con los datos de los usuarios que maneja la aplicación del ejemplo:

./vistas/usuario.ejs

```
    <%- include(‘comunes/encabezado_html’) %>
    <title>
      Detalle del usuario <%= usuario.obtenerId() %>
      - Aplicación de ejemplo de NodeJS CIDW
    </title>
  </head>
  <body>
    <div >
      <table>
        <thead>
          <tr>
            <th>ID</th>
            <th>Nombre</th>
            <th>Apellido</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td><%= usuario.obtenerId() %></td>
            <td><%= usuario.obtenerNombre() %></td>
            <td><%= usuario.obtenerApellido() %></td>
          </tr>
        </tbody>
      </table>
    </div>
    <%- include('comunes/pie_pagina') %>
  </body>
</html>
```

Los datos son provistos a la vista mediante el controlador. Retomando nuevamente el ejemplo anterior, la respuesta que genera el controlador se construye de la siguiente manera:

./controladores/usuario.js

```
...
  obtenerUsuario: (peticion, respuesta) => {
    if (!usuarios[peticion.params.id]) {
      // envía el error 404 – Usuario no encontrado
    }
    let usr = usuarios[peticion.params.id]
    respuesta.render('usuario', {usuario: usr})
  }
}
```

## 22. Peticiones AJAX

### 22.1. JavaScript asíncrono con XML

AJAX (*Asynchronous JavaScript with eXtensible Markup Language*) es una combinación de tecnologías Web que permite realizar peticiones en segundo plano y actualizar el estado de una vista con el resultado de la petición. Las peticiones AJAX se basan en el objeto **XMLHttpRequest**, es un objeto global que permite hacer conexiones con un servidor y procesar la respuesta en la misma vista donde se generó la petición (Mozilla Corportation, 2022).

Pese al nombre de la petición *XMLHttpRequest* también puede usarse para realizar peticiones HTTPS, FTP, lectura de archivos, entre otros; además de aceptar documentos HTML, texto plano, JSON y otros (Mozilla Corportation, 2022).

### 22.2. Conexiones abiertas y Peticiones discretas

En la sección 18.5 se explica que es posible abrir conexiones de *WebSockets* que permiten tanto al servidor de la aplicación como a los navegadores de los usuarios que ejecuten una aplicación Web adquirir un canal bidireccional de comunicación en que el ambos actúan como cliente – servidor.

La desventaja de mantener canales de *WebSockets* abiertos es que requieren más recursos que otro tipo de conexiones Web. Aunque los servidores usualmente tienen la

capacidad de mantener miles o millones de conexiones, muchos usuarios dependen de entornos con recursos limitados.

Una porción significativa de los usuarios puede tener conexiones limitadas, sea en ancho de banda, en costo de tráfico o por tiempo de uso, por lo que es recomendado solo emplear *WebSockets* cuando es indispensable mantener una conexión abierta constantemente, por ejemplo si se requiere de una función de chat en vivo, o compartir el estado de recursos en tiempo real.

Para otro tipo de peticiones en segundo plano es mejor emplear peticiones AJAX. Para esto no es necesario manipular el objeto *XMLHttpRequest* directamente. Bibliotecas como JQuery ofrecen herramientas para manejar peticiones asíncronas altamente personalizables.

### 22.3. Usando AJAX a través de JQuery

En esta sección se presenta cómo usar AJAX mediante la aplicación de ejemplo, en la que peticiones asíncronas crean y eliminan usuarios, exponiendo estos mecanismos a los usuarios con un formulario y redireccionamientos.

Después de crear el proyecto con npm como se muestra en la sección 17.4.2, el primer paso es definir un directorio que pueda ser accedido públicamente desde la Web con las dependencias de las vistas, en este caso JQuery. Los recursos que conforman las dependencias y componentes de las vistas de una aplicación Express/NodeJS suelen colocarse en un directorio llamado `public`. Se configura en el directorio en el archivo `server.js` (StrongLoop et al., 2023):

```
...
const app = express()
...
app.use(express.static(__dirname + '/public'))
...
```

Luego se agrega JQuery al repositorio del proyecto como se hace en la sección 21.2.1, descargando el archivo https://code.jquery.com/jquery-3.6.4.min.js (o la versión de

lanzamiento más reciente disponible) en `./public/js/jquery-x.x.x.min.js` (con `x.x.x` el número de la versión descargada).

**Ejemplo de uso de peticiones AJAX con JQuery para manejar formularios**

En este ejemplo se incluye JQuery en el portal de la aplicación, `./vistas/index.ejs` junto con un formulario que permite crear usuarios usando la ruta `POST /usuarios/:id/:nombre/:apellido`. Comenzaremos agregando JQuery a la vista del portal:

`./vistas/index.ejs`

```
...
  <script src="/js/jquery-3.6.4.min.js" ></script>
  <title>Portal - Aplicación de ejemplo de NodeJS CIDW</title>
</head>
```

A continuación se construye el cuerpo del archivo para agregar el formulario que recibe los siguientes datos de los usuarios: *identificador*, *nombre* y *apellido*.

```
<body>
  <h1>Alta de usuarios</h1>
  <form id="datos_usuario" action="#" method="post">
    <div >
      <label for="identificador_usr" >ID:</label>
      <input id="identificador_usr" type="number" name="identificador_usr" >
    </div>
    <div >
      <label for="nombre_usr" >Nombre:</label>
      <input id="nombre_usr" type="text" name="nombre_usr" >
    </div>
    <div >
      <label for="apellido_usr" >Apellido:</label>
      <input id="apellido_usr" type="text" name="apellido_usr" >
    </div>
    <button type="submit" name="enviar_btn" >Registrar</button>
  </form>
  <%- include('comunes/pie_pagina') %>
```

Para enviar el formulario como una petición AJAX, se usa JQuery para definir la acción que corresponde al evento de enviar el formulario, sustituyendo el comportamiento por defecto del evento de envío del formulario. Este comportamiento se define dene dentro y al final de la vista mediante una etiqueta `<script>` (js.foundation, 2023b):

```
    <script >
```

```
      $('#datos_usuario').submit(function(eventoEnviar) {
        eventoEnviar.preventDefault()
        let idUsr = $('#identificador_usr')
        let nombreUsr = $('#nombre_usr')
        let apellidoUsr = $('#apellido_usr')
        let form = $(this)
        $.ajax({
          type: 'POST',
          url: `/usuarios/${idUsr.val()}/${nombreUsr.val()}/${apellidoUsr.val()}`,
          data: form.serialize(),
```

Si la petición obtiene un resultado exitoso, se vacía el formulario para permitir el registro de otro usuario:

```
          success: function(respuesta) {
            idUsr.val('')
            nombreUsr.val('')
            apellidoUsr.val('')
            alert(`Usuario registrado en: "${respuesta}"`)
          },
          error: function(ajax, mensaje) {
            alert('Ocurrió un problema al registrar al usuario.')
          }
        })
      })
    </script>
</body>
```

Ahora se construye la funcionalidad de borrado de usuarios. Se agrega JQuery y un formulario a la vista de detalle de usuarios que creada en la sección 21.4.3:

`./vistas/usuario.ejs`

```
...
  <script src="/js/jquery-3.6.4.min.js" ></script>
  <title>
    Detalle del usuario <%= usuario.obtenerId() %>
    - Aplicación de ejemplo de NodeJS CIDW
  </title>
</head>
...
```

De forma similar a como se hizo en la página del Portal, en el caso del borrado se agrega un botón debajo de la tabla con los datos del usuario que acciona el mecanismo de borrado. Este comportamiento se define nuevamente en una etiqueta `<script>` antes de terminar el cuerpo del HTML.

Este `<script>` muestra una confirmación para realizar el borrado: si se confirma que desea borrar al usuario, se envía la petición `DELETE /usuarios/:id`. El resultado se muestra en pantalla, de ser exitoso redirecciona al Portal principal, pues la ruta asociada al usuario que se acaba de borrar ya no es válida (ahora apunta a un recurso no existente). De lo contrario, permanece en la página.

```
...
    </tbody>
  </table>
  <button id="eliminar_usr_btn" type="button" name="eliminar_usr_btn">
      Borrar usuario
  </button>
  </div>
  <%- include('comunes/pie_pagina') %>
  <script >
    $('#eliminar_usr_btn').click(function(eventoClick) {
      if (!confirm('¿Borrar? Esta acción no puede deshacerse')) return
        let botonBorrar = $(this)
        botonBorrar.prop('disabled', true)
        $.ajax({
          type: 'DELETE',
          url: "/usuarios/<%= usuario.obtenerId() %>",
          success: function(respuesta) {
            alert(`El usuario "${respuesta}" ha sido eliminado.`)
            window.location.href = '/'
          },
          error: function(ajax, mensaje) {
              alert('Ocurrió un problema al borrar al usuario.')
              botonBorrar.prop('disabled', false)
          }
      })
    })
  </script>
</body>
```

## 23. Bootstrap y hojas de estilo CSS

### 23.1. Jerarquía de aplicación de selectores de estilos

En la sección 20.3 se presentan las Hojas de Estilo en Cascada y algunos de los principales selectores con los que se definen bloques de reglas de estilos, pero no se profundiza en cómo se aplican más allá de los selectores. El sustantivo *cascada* en el nombre de las hojas de estilo, refleja la estrategia de aplicación de las reglas. Esta estrategia se describe a continuación (Dean, 2019):

1. Las reglas definidas en el atributo `style` de un elemento del documento HTML/XHTML son las más prioritarias, sobreescriben cualquier regla que se aplique en el elemento conforme a los selectores en la vista.

2. La siguiente prioridad es para las reglas definidas en el encabezado del documento HTML/XHTML dentro de la etiqueta `<style />`

3. Por último tenemos las reglas definidas en archivos de estilo que cargue la vista como dependencias.

Si desea que una regla conserve su efecto pese a su posible sobreescritura por reglas más prioritarias, puede terminar la regla con la directiva `!important` (Mozilla Corportation, 2023a).

### 23.1.1. Incluyendo hojas de estilo en las vistas

En la sección 22.3 se modifica el archivo de configuración `server.js` del proyecto de ejemplo para definir un directorio con las dependencias y componentes de las vistas llamado `public`. Las hojas de estilo también se colocan en él como parte de las dependencias de las vistas. Por ejemplo, puede personalizar la presentación del pie de página con el siguiente archivo `./public/css/comunes.css`:

```
footer {
  background:lightgray;
}
```

Para incluirlo en una vista, se emplea la siguiente etiqueta en el encabezado de los documentos HTML/XHTML:

```
<link rel="stylesheet" type="text/css" href="/css/comunes.css">
```

## 23.2. Bootstrap

Bootstrap es un conjunto de bibliotecas que ayudan a estilizar la presentación de las vistas de una aplicación con menor esfuerzo. Ofrece marcos de trabajo para desarrollar aplicaciones o dependencias para documentos HTML/XHTML comprimidos de forma similar a JQuery (Otto et al., 2023).

### 23.2.1. Integrando Bootstrap en una aplicación NodeJS

Como se hace con JQuery, se descarga la versión con la que se desea trabajar (usualmente la más reciente) y se instala en `./public/css/bootstrap.min.css`

https://cdn.jsdelivr.net/npm/bootstrap/dist/css/bootstrap.min.css

Muchos componentes de Bootstrap utilizan JavaScript, por lo que también se deben importar las dependencias JavaScript de Bootstrap. En particular Bootstrap utiliza JQuery, por lo que es importante primero importar JQuery como dependencia de las vistas ***antes*** que Bootstrap. Los scripts de Bootstrap usualmente se instalan en `./public/js/bootstrap.min.js`

https://cdn.jsdelivr.net/npm/bootstrap/dist/js/bootstrap.min.js

Tras lo anterior es posible importar Bootstrap como dependencia en las vistas usando la etiqueta `link` en el encabezado del documento HTML/XHTML.

```
<html >
  <head >
    …
    <link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css">
    <script src="/js/jquery-x.x.x.min.js" ></script>
    …
  </head>
…
```

Los scripts de Bootstrap deben cargarse al final del cuerpo del documento HTML/XHTML, de esta forma todos sus componentes están definidos para cuando se ejecuta Bootstrap y las animaciones asociadas a los estilos serán enlazadas de forma adecuada (Otto et al., 2023):

```
...
        <script src="/js/bootstrap.min.js" ></script>
    </body>
</html>
```

### 23.2.2. Personalizando vistas con Bootstrap

**Ejemplo de uso de Bootstrap para uniformar y estilizar la presentación de vistas**

Para terminar la aplicación de ejemplo se da estilo a las vistas del Portal principal y de Detalle de usuario. Comenzando por el Portal principal `./vistas/index.ejs` se le da el estilo de Bootstrap al formulario para crear usuarios modificando las etiquetas de las entradas del formulario, especificando clases de Bootstrap:

```
...
<form id="datos_usuario" action="#" method="post">
  <div class="form-group" >
    <label for="identificador_usr" >...</label>
    <input id="identificador_usr" ... class="form-control" >
  </div>
  <div class="form-group" >
    <label for="nombre_usr" >...</label>
    <input id="nombre_usr" ... class="form-control" >
  </div>
  <div class="form-group" >
    <label for="apellido_usr" >...</label>
    <input id="apellido_usr" ... class="form-control" >
  </div>
  <button type="submit" ... class="btn btn-success" >...</button>
</form>
...
```

En el caso de la vista de Detalle de usuario `./vistas/usuario.ejs`, se le da estilo a la tabla que presenta la información del usuario y el botón que solicita el borrado del usuario:

```
...
<div >
  <table class="table table-bordered table-hover" >
    <thead class="thead-light" >
      ...
    </thread>
    ...
  </table>
  <button id="eliminar_usr_btn" ... class="btn btn-danger" >...</button>
```

**Agradecimientos**

**Trabajo realizado con el apoyo del Programa PAPIME PE109623 El Aula del Futuro del Colegio de Ciencias y Humanidades Plantel Sur y PAPIME PE309523 El Aula del Futuro del Colegio de Ciencias y Humanidades Plantel Oriente.**

## Referencias Bibliográficas


Bass, L., Clements, P., Kazman, R., 2013. Software architecture in practice, 3rd ed. ed, SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ.

Bernsen, N.O., Dybkjaer, L., 2009. Multimodal usability, Human-computer interaction series. Springer.

Bourque, P., Fairley, R.E., Society, I.C., 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0, 3rd ed. IEEE Computer Society Press, Washington, DC, USA.

Chacon, S., Straub, B., 2020. Pro Git, 2nd ed. Apress.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J., 2011. Documenting software architectures: views and beyond, 2nd ed. ed, SEI series in software engineering. Addison-Wesley, Upper Saddle River, NJ.

Crockford, D., 2019. Code Conventions for the JavaScript Programming Language [WWW Document]. URL https://www.crockford.com/code.html (accedido 28.04.23).

Crockford, D., 2008. JavaScript: the good parts, 1. ed. ed, Unearthing the excellence in JavaScript. O'Reilly, Beijing Köln.

Dean, J., 2019. Web programming with HTML5, CSS, and JavaScript. Jones & Bartlett Learning, Burlington, Massachusetts.

DelBono, E., 2016. Node.js Succinctly.

Dix, A., Finlay, J., Abowd, G.D., Beale, R., 2004. Human-computer interaction, 3rd ed. Pearson/Prentice-Hall, Harlow, England; New York.

Eernisse, M., 2023a. ejs [WWW Document]. npm. URL https://www.npmjs.com/package/ejs (accessed 3.18.23).

Eernisse, M., 2023b. EJS -- Embedded JavaScript templates [WWW Document]. URL https://ejs.co/ (accedido 04.05.23).

Flanagan, D., 2011. JavaScript: the definitive guide, 6th ed. ed. O'Reilly, Beijing ; Sebastopol, CA.

Fowler, M., 2003. Patterns of enterprise application architecture, The Addison-Wesley signature series. Addison-Wesley, Boston.

Gabbrielli, M., Martini, S., 2010. Programming languages: principles and paradigms, Undergraduate topics in computer science. Springer, London ; New York.

Hamerly, J., Paquin, T., Walton, S., 1999. Freeing the Source: The Story of Mozilla, in: DiBona, C., Ockman, S. (Eds.), Open Sources: Voices from the Open Source Revolution. O'Reilly, California.

js.foundation, 2023a. Selectors | jQuery API Documentation. URL http://api.jquery.com/category/selectors/ (accedido 17.03.23).

js.foundation, 2023b. jQuery.ajax() | jQuery API Documentation. URL https://api.jquery.com/Jquery.ajax/ (accedido 04.05.23).

Kurose, J.F., Ross, K.W., 2013. Computer networking: a top-down approach, 6th ed. ed. Pearson, Boston.

Luzgin, V.A., Kholod, I.I., 2020. Overview of Mining Software Repositories, in: 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). pp. 400–404.

Martin, R.C., 2008. Clean Code: A Handbook of Agile Software Craftsmanship, 1st ed. Prentice Hall PTR, USA.

Martin, R.C., Martin, M., 2007. Agile principles, patterns, and practices in C#, Robert C. Martin series. Prentice Hall, Upper Saddle River, NJ.

McConnell, S., 2004. Code complete, 2nd ed. ed. Microsoft Press, Redmond, Wash.

Mozilla Corportation, 2022. AJAX - Guía de Desarrollo Web | MDN [WWW Document]. URL https://developer.mozilla.org/es/docs/Web/Guide/AJAX (accedido 04.05.23).

Mozilla Corportation, 2023a. !important - CSS: Cascading Style Sheets | MDN [WWW Document]. URL https://developer.mozilla.org/en-US/docs/Web/CSS/important (accedido 04.05.23).

Mozilla Corportation. 2023b. Arrow function expressions—JavaScript | MDN [WWW Document]. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (accedido 03.06.24)

Nielsen, J., 1994. Usability Engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nottingham, M., 2020. URI Design and Ownership (Request for Comments No. RFC 8820). Internet Engineering Task Force. https://doi.org/10.17487/RFC8820

Otto, M., Thornton, J., Bootstrap, 2023. Get started with Bootstrap [WWW Document]. URL https://getbootstrap.com/docs/5.3/getting-started/introduction/ (accedido 04.05.23).

Pratt, A., Nunes, J., 2012. Interactive design: an introduction to the theory and application of user-centered design. Rockport Publishers, Beverly, MA.

Resig, J., 2005. Selectors in Javascript. URL https://johnresig.com/blog/selectors-in-javascript/ (accedido 17.03.23).

Sommerville, I., 2011. Software engineering. Addison-Wesley, Boston.

StrongLoop, IBM Corporation, OpenJS Foundation, 2023. Serving static files in Express [WWW Document]. URL https://expressjs.com/en/starter/static-files.html (accedido 04.05.23).

StrongLoop, IBM Corporation, OpenJS Foundation, 2017. Using template engines with Express [WWW Document]. URL http://expressjs.com/en/guide/using-template-engines.html (accedido 04.05.23).

Torvalds, L., 2005a. Re: Kernel SCM saga. [WWW Document]. URL https://marc.info/?l=linux-kernel&m=111300303827338&w=2 (accedido 02.05.23).

Torvalds, L., 2005b. Re: Kernel SCM saga. [WWW Document]. URL https://marc.info/?l=linux-kernel&m=111298705212803&w=2 (accedido 02.05.23).

Torvalds, L., 2005c. Re: Kernel SCM saga. [WWW Document]. URL https://marc.info/?l=linux-kernel&m=111289606218373&w=2 (accedido 02.05.23).

Torvalds, L., 2005d. Re: Kernel SCM saga. [WWW Document]. URL https://marc.info/?l=linux-kernel&m=111293537202443&w=2 (accedido 02.05.23).