



**Universidad de Valladolid**

**Escuela de Ingeniería Informática**

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática  
(Mención de Computación)

**Análisis de Servicios y  
Mecanismos de Seguridad  
desde el Diseño en *front-ends*  
de desarrollo Web**

Autor:  
**D. Raúl Rodríguez Carracedo**



**Universidad de Valladolid**

**Escuela de Ingeniería Informática**

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática  
(Mención de Computación)

**Análisis de Servicios y  
Mecanismos de seguridad  
desde el Diseño en *front-ends*  
de desarrollo web**

Autor:

**D. Raúl Rodríguez Carracedo**

Tutores:

**D. Amador Aparicio de la Fuente**

**D. Valentín Cardeñoso Payo**

# Tabla de contenidos

Tabla de contenidos .....	1
1 INTRODUCCIÓN .....	5
1.1 Planteamiento del problema.....	5
1.2 Objetivos .....	5
1.3 Método y Plan de trabajo .....	6
1.4 Descripción del documento .....	7
2 DESARROLLO WEB.....	8
2.1 Patrones de Arquitectura.....	8
2.2 MVC.....	8
2.3 MVVM .....	9
2.3.1 Otros patrones .....	10
2.3.1.1 Modular Architecture .....	10
2.3.1.2 Component Architecture.....	10
2.3.1.3 Dumb-Smart Components.....	11
2.3.1.4 State Management.....	11
2.3.1.5 Unidirectional Architecture.....	11
2.3.1.6 Micro Frontends .....	11
2.3.2 Patrones de diseño .....	11
2.3.2.1 Patrones estructurales .....	12
2.3.2.1.1 Patrón de fachada .....	12
2.3.2.2 Patrones de creación .....	13
2.3.2.2.1 Patrón de fábrica ( <i>Factory Method</i> ).....	13
2.3.2.2.2 Patrón Singleton .....	13
2.3.2.3 Patrones de comportamiento .....	13
2.3.2.3.1 Patrón de estrategia ( <i>Strategy</i> ) .....	13
2.3.2.3.2 Patrón de observador .....	14
2.4 Evolución del desarrollo front-end .....	14
2.5 Frameworks y herramientas .....	19
2.5.1 Frameworks y librerías de desarrollo.....	19
2.5.1.1 React .....	19
2.5.1.2 Angular.....	20
2.5.1.3 Vue .....	21
2.5.1.4 Svelte.....	22
2.5.1.5 React Native .....	23
2.5.1.6 Electron.....	24
2.5.2 Herramientas .....	25

2.5.2.1 npm.....	25
2.5.2.2 JavaScript .....	26
2.5.2.3 TypeScript.....	27
3 ANÁLISIS DE AMENAZAS.....	28
3.1 Amenazas Aplicaciones web.....	28
3.1.1 OWASP .....	28
3.1.2 OWASP Top 10.....	29
3.1.3 OWASP Top 10 - 2021.....	31
3.1.3.1 A01:2021-Broken Access Control.....	32
3.1.3.2 A02:2021-Cryptographic Failures .....	33
3.1.3.3 A03:2021-Injection .....	34
3.1.3.4 A04:2021-Insecure Design .....	35
3.1.3.5 A05:2021-Security Misconfiguration .....	36
3.1.3.6 A06:2021-Vulnerable and Outdated Components .....	36
3.1.3.7 A07:2021-Identification and Authentication Failures .....	37
3.1.3.8 A08:2021-Software and Data Integrity Failures .....	38
3.1.3.9 A09:2021-Security Logging and Monitoring Failures .....	39
3.1.3.10 A10:2021-Server-Side Request Forgery.....	40
3.1.3.11 A11:2021 – Next Steps .....	41
3.1.3.11.1 Errores de código .....	41
3.1.3.11.2 Denegación de servicios.....	41
3.1.3.11.3 Errores de gestión de memoria.....	41
3.1.4 Otras vulnerabilidades.....	42
3.2 Amenazas JavaScript.....	42
3.2.1 Riesgos y problemas.....	43
3.2.1.1 Ejecución de código malicioso .....	43
3.2.1.2 Robo de datos del usuario.....	43
3.2.1.3 Actividad del usuario no intencionada .....	43
3.2.1.4 Explotación de vulnerabilidades en el código fuente .....	43
3.2.1.5 Vulnerabilidades más comunes.....	43
3.2.1.5.1 Cross-Site Scripting (XSS).....	43
3.2.1.5.2 Cross-Site Request Forgery (CSRF).....	44
3.2.1.5.3 Vulnerabilidades del código fuente .....	45
3.2.1.6 Buenas prácticas.....	47
3.3 Amenazas librerías / frameworks modernos.....	47
3.3.1 Cross-Site Scripting (XSS) .....	48
3.3.2 Broken Access Control.....	48
3.3.3 SQL Injection .....	48
3.3.4 XML External Entity Attack (XXE) .....	49
3.3.5 Vulnerabilidades del código fuente.....	49
3.3.6 Buenas prácticas.....	49
3.3.7 Amenazas React.....	50

3.3.7.1 Cross-Site Scripting (XSS) .....	50
3.3.7.2 Zip Slip .....	51
3.3.7.3 Server Side Rendering (SSR) .....	51
3.3.8 Amenazas Angular .....	52
3.3.8.1 Cross-Site Scripting (XSS) .....	52
3.3.9 Amenazas Vue.....	53
3.3.9.1 Cross-Site Scripting (XSS) .....	54
3.3.9.2 Server Side Rendering (SSR) .....	54
3.3.10 Amenazas Svelte .....	54
3.3.10.1 Cross-Site Scripting (XSS) .....	54
3.3.11 Amenazas React Native.....	55
3.3.11.1 Broken Access Control.....	56
3.3.11.2 Persistencia de datos .....	56
3.3.11.3 Deep Linking .....	56
3.3.12 Amenazas Electron .....	57
3.3.12.1 Cross-Site Scripting (XSS) .....	57
3.3.12.2 Vulnerabilidades del código fuente.....	57
4 SERVICIOS Y MECANISMOS .....	58
4.1 Servicios .....	58
4.1.1 Autenticación .....	58
4.1.2 Autorización .....	58
4.1.3 Integridad.....	59
4.1.4 Confidencialidad.....	59
4.1.5 Disponibilidad.....	59
4.2 Mecanismos.....	60
4.2.1 Autenticación .....	60
4.2.1.1 Usuario y contraseña .....	60
4.2.1.1.1 Validaciones de contraseñas .....	60
4.2.1.1.2 Recuperación de contraseñas .....	61
4.2.1.1.3 Política de cambio de contraseñas .....	62
4.2.1.1.4 Protección contra ataques automáticos .....	62
4.2.1.2 Autenticación multi-factor .....	62
4.2.1.3 Otros mecanismos de autenticación.....	63
4.2.1.3.1 OAuth .....	63
4.2.1.3.2 OIDC .....	64
4.2.1.3.3 FIDO.....	65
4.2.2 Autorización y Confidencialidad .....	65
4.2.2.1 Autorización basada en Roles y Permisos .....	65
4.2.2.1.1 Scopes .....	65

4.2.2.1.2 Roles .....	66
4.2.2.1.3 Permisos.....	66
4.2.2.2 Feature Flags .....	66
4.2.2.3 Buenas prácticas.....	67
4.2.3 Integridad.....	67
4.2.4 Disponibilidad.....	68
4.2.4.1 Monitorización y Trazabilidad .....	68
4.2.4.1.1 Pingdom .....	68
4.2.4.1.2 RayGun .....	69
4.2.4.1.3 Sentry .....	70
4.2.4.1.4 LogRocket .....	70
4.2.4.1.5 AppSignal .....	71
4.2.4.1.6 Firebase .....	71
4.2.4.1.7 Otras herramientas .....	71
4.2.4.2 Escaneo de vulnerabilidades.....	72
4.2.4.3 Testing .....	73
4.2.4.4 CI/CD .....	76
5 DECISIONES EN FASE DE DISEÑO.....	78
5.1 UX.....	78
5.2 Tipos de renderizado.....	80
5.2.1 Server Side Rendering (SSR) .....	80
5.2.2 Client Side Rendering (CSR).....	81
5.2.3 Static Side Generation (SSG).....	81
5.2.4 Conclusión .....	82
5.3 Híbridas vs Nativas .....	82
5.3.1 Ventajas y desventajas .....	82
5.3.2 Conclusiones .....	83
5.4 DevSecOps.....	85
5.4.1 Buenas prácticas.....	86
5.4.1.1 Shift left & shift right .....	86
5.4.1.2 Capacitar al equipo .....	86
5.4.1.3 Organizar al equipo.....	86
5.4.1.4 Analizar los resultados .....	86
5.4.1.5 Actualizarse .....	86
6 CONCLUSIONES.....	87
7 REFERENCIAS.....	88

# 1 INTRODUCCIÓN

La Ciberseguridad siempre ha sido uno de los principales problemas dentro de la Informática, y según van apareciendo nuevas tecnologías van apareciendo nuevas vulnerabilidades dentro de éstas. El mundo de las aplicaciones, en especial el de las aplicaciones web, no ha parado de crecer desde sus inicios y está constantemente cambiando, y uno de los aspectos más importantes de estos cambios son los datos, los datos que viajan por las aplicaciones web a día de hoy no son iguales que los de hace unos años, ahora, estos datos por lo general, son de mayor importancia para los usuarios, siendo así, uno de los principales activos a proteger dentro de estas aplicaciones, ya que, cuanto más sensibles sean los datos mayor es el número de atacantes interesados. Otro factor que aumenta el riesgo de las aplicaciones web es, el número de usuarios, ya que cuantos más usuarios tenga una aplicación mayor es la probabilidad de sufrir un ataque. Por otro lado, el medio por el que se transmiten los datos tampoco es el mismo, sino que ha ido cambiando según ha ido avanzando la tecnología, y por ello, hay que estar constantemente actualizado con las últimas novedades para poder detectar, identificar y solucionar los posibles problemas de seguridad emergentes.

## 1.1 Planteamiento del problema

El desarrollo de aplicaciones web cada día es más popular y solicitado por las grandes empresas, y su complejidad cada vez es mayor. Estas aplicaciones normalmente están compuestas por lo que denominamos la parte front-end, y la parte back-end; ambas partes serán definidas en profundidad más adelante, pero por el momento, nos sirve con saber que ambas partes requieren de algunos de los conocimientos provenientes de diferentes campos de la Informática, entre ellos, la Ciberseguridad. La Ciberseguridad es una de las partes más importantes en las aplicaciones web, ya que por lo general, estas están expuestas a todos los usuarios de internet, estos aspectos de seguridad simplemente se tenían en cuenta en la parte back-end de las aplicaciones debido a que la parte front-end apenas tenía grandes responsabilidades, pero, con el aumento del grado de complejidad de las mismas, la parte front-end ha ido adquiriendo mayor tamaño integrando funcionalidad que es crítica para el correcto funcionamiento de las aplicaciones, es por ello por lo que, a la hora de desarrollar la parte front-end de las aplicaciones web los aspectos relacionados con la seguridad se han vuelto bastante importantes siendo estos una de las principales preocupaciones de los desarrolladores a la hora de diseñar la parte front-end de las aplicaciones.

## 1.2 Objetivos

El principal objetivo de este trabajo es el de ofrecer una guía de desarrollo seguro para aplicaciones front-end, ofreciendo un análisis a alto nivel de todo el ecosistema actual que rodea al desarrollo de aplicaciones front-end. Un segundo objetivo es ofrecer al desarrollador de front-end un análisis de aquellos aspectos de seguridad que afectan directamente a la parte front-end de las aplicaciones web, o a cualquier tipo de aplicación que sea desarrollada utilizando tecnologías front-end. En tercer lugar, el trabajo expone cuales son los principales riesgos que hoy en día más nos afectan, y qué acciones podemos tomar para protegernos ante ellos, pasando por las medidas de seguridad preventivas como pueden ser buenas prácticas, mecanismos de seguridad, herramientas, etc. Finalmente, se detallarán aquellas decisiones que hay que tomar en la fase de diseño de las aplicaciones para asegurar que el resultado final sea una aplicación segura, estable, y fiable.

## 1.3 Método y Plan de trabajo

Para poder ofrecer una guía que sirva para el desarrollo seguro de aplicaciones front-end, es necesario partir desde un planteamiento del problema hasta un análisis y valoración de la solución. Por ello, el procedimiento a seguir es el siguiente: primero se realiza un estudio y análisis de los conceptos más importantes en el campo de las aplicaciones web y la Ciberseguridad, con estos conceptos ya fijados, detallar cuáles son las principales preocupaciones de seguridad a día de hoy en el campo de las aplicaciones web, y sobre esta base de conocimientos presentar las principales amenazas que afectan al desarrollo de aplicaciones front-end, así como las diferentes técnicas y herramientas de las que disponemos actualmente para combatir dichas amenazas, finalmente, listar un conjunto de recomendaciones a seguir en el diseño de las aplicaciones front-end.

A continuación, se muestra una imagen con la planificación de trabajo que se seguirá para el desarrollo de este libro, la división de tareas en grupos es la siguiente:

1. Investigación y propuesta
2. Análisis de amenazas
3. Servicios y mecanismos de seguridad
4. Introducción
5. Decisiones en fase de diseño
6. Resumen de resultados
7. Conclusiones y recomendaciones
8. Repaso y cierre de memoria

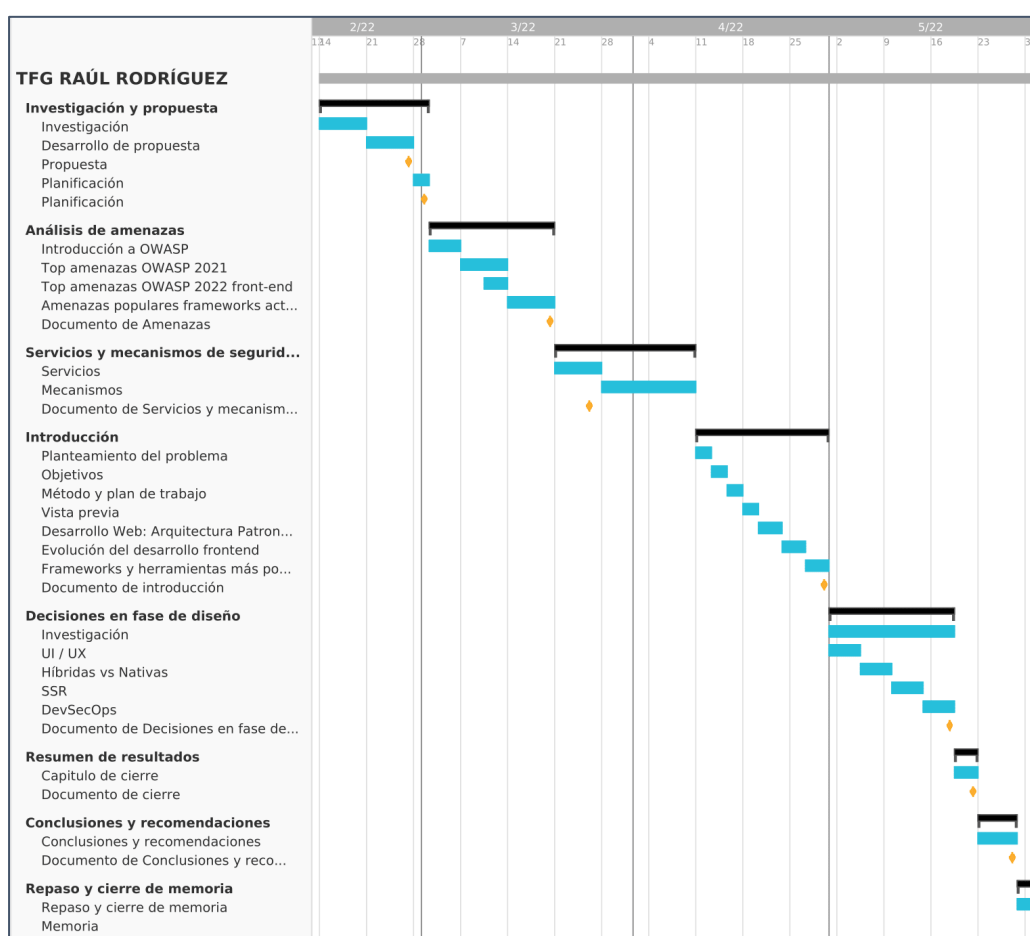


ILUSTRACIÓN 1 PLANIFICACIÓN DE TRABAJO



## 1.4 Descripción del documento

Descripción de los diferentes capítulos del libro a nivel general.

- **Introducción:** el primero de ellos es el capítulo introductorio en el que se pondrá en contexto al lector y donde se enseñan aquellos conocimientos que son básicos para el buen entendimiento del resto de capítulos, también es en este capítulo donde se presenta la problemática que servirá de guía para el resto del libro.
- **Análisis de amenazas:** estudio y análisis de las amenazas actuales que rodean al mundo de las aplicaciones web, partiendo de las más generales a las ya más específicas que afectan directamente al desarrollo front-end.
- **Servicios y mecanismos de seguridad:** una vez se han introducido las amenazas, pasamos a comentar aquellos servicios de seguridad que son esenciales en las aplicaciones web, y cuáles son los mecanismos de seguridad que nos van a permitir ofrecer dichos servicios.
- **Decisiones en fase de diseño:** en este capítulo veremos cuáles son las decisiones más importantes, hablando en términos de seguridad, a la hora de diseñar una aplicación front-end.
- **Resumen y discusión:** expuesta la problemática y sus posibles soluciones hacemos un resumen sobre todo lo visto hasta el momento discutiendo aquellos aspectos más importantes.
- **Conclusiones y recomendaciones:** finalmente, a modo de cierre de este libro, se exponen las conclusiones a las que hemos podido llegar tras realizar un estudio y análisis exhaustivo de la seguridad en aplicaciones front-end, ofreciendo un conjunto de recomendaciones finales.

## 2 DESARROLLO WEB

El desarrollo web hace referencia a la construcción y mantenimiento de aplicaciones web, entendiendo por aplicaciones web, aquellos programas software que son ejecutados por un navegador web.

El desarrollo de estas aplicaciones está compuesto por múltiples tareas, pero a nivel general podemos hacer una primera separación en dos tipos de tareas:

- Parte *client side* (**front-end**): está formada por aquellas tareas que se encargan de la parte de la aplicación que va a ejecutarse en el propio navegador del cliente, y por lo tanto, visualizada por los usuarios y que va a comunicarse con los servicios back-end necesarios para funcionar correctamente.
- Parte *server side* (**back-end**): que comprende aquellas tareas de construcción del servicio o los servicios web que van a implementar la lógica de la aplicación necesaria dando soporte al cliente ante las acciones tomadas por los usuarios.

Estas dos partes son igual de importantes y hoy en día en la mayoría de las aplicaciones web el front-end no puede vivir sin backend y viceversa. Pero, para los objetivos y el alcance de este libro la parte que más nos interesa a nosotros es la parte **front-end**.

El desarrollo front-end por lo tanto, se encarga de construir la capa visual que va a servir a los usuarios para interactuar con la aplicación de una manera lo más amigable posible.

A continuación, vamos a presentar las principales arquitecturas del desarrollo front-end y después presentaremos los principales patrones de arquitectura para el desarrollo front-end.

### 2.1 Patrones de Arquitectura

Antes de presentar las diferentes arquitecturas vamos a dar una definición de arquitectura software:

*“The software architecture of a system represents the design decisions related to overall system structure and behavior. Architecture helps stakeholders understand and analyze how the system will achieve essential qualities such as modifiability, availability, and security.”*

Fuente: <https://www.sei.cmu.edu/our-work/software-architecture/>

Esta definición tiene dos ideas principales, la primera de ellas que la arquitectura de un sistema software representa las decisiones de diseño del sistema que van a marcar su estructura y su comportamiento. Por otro lado, indica que la arquitectura va a servir como ayuda para alcanzar los requerimientos establecidos para el sistema software. Dicho esto, pasamos a ver las arquitecturas front-end más conocidas hoy en día.

### 2.2 MVC

Fue uno de los primeros patrones en el diseño software que se aplicó a las aplicaciones front-end, el principal objetivo de este era el de realizar una separación lógica de la aplicación entre

la lógica de negocio y su visualización, más concretamente en las siguientes 3 partes, *Model*, *View*, *Controller*.

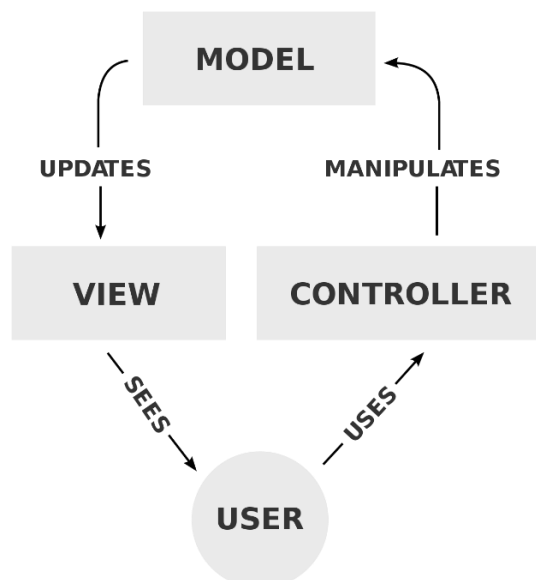


ILUSTRACIÓN 2 MVC SOURCE: [HTTPS://ES.WIKIPEDIA.ORG/WIKI/MODELO%E2%80%93VISTA%E2%80%93CONTROLADOR](https://es.wikipedia.org/wiki/Modelo-Vista-Controlador)

Estas tres partes se pueden definir de la siguiente manera:

- **Modelo:** encargado de manejar los datos y de la lógica de negocio, es el modelo el encargado de comunicarse con la vista para notificar que un estado en concreto ha cambiado y de esta manera la vista pueda enterarse de cuando hay que hacer cambios en la pantalla.
- **Vista:** es el diseño y la presentación de la aplicación.
- **Controlador:** encargado de dirigir los diferentes comandos a los modelos y las vistas, por lo tanto, el encargado de comunicarse con el modelo para actualizar los datos y con la vista para reaccionar ante entradas proporcionadas por los usuarios de la aplicación.

Uno de los primeros frameworks que implementaron este patrón fueron [AngularJS](#), [Ember](#), [Backbone](#), [React](#) + [Redux](#), etc.

## 2.3 MVVM

*Model View View-Model (MVVM)* es una de las alternativas a MVC en la que se fragmenta la vista y el modelo, estableciendo una asociación entre las diferentes partes de la vista y las partes del modelo, de manera que, a un conjunto de elementos de la vista se le asocia únicamente aquellas partes del modelo que les es de interés, a esto se le denomina la capa *View-Model*.

Este patrón también mantiene la separación entre la capa de presentación y la capa de lógica de negocio, y además incorpora lo que se denomina *two-way data binding* entre la vista y cada vista-modelo, esto, lo que permite es, automatizar la propagación de las modificaciones realizadas por cada vista-modelo a la vista principal de la aplicación, agilizando y simplificando la manipulación de los datos del modelo.

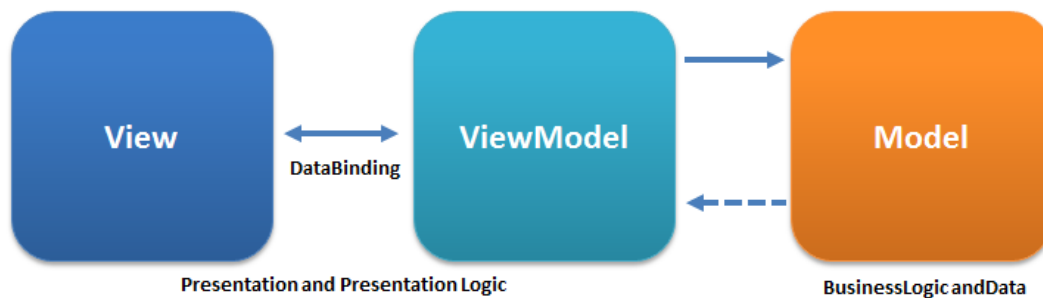


ILUSTRACIÓN 3 MVVM SOURCE: <https://en.wikipedia.org/wiki/Model%E2%80%93View%E2%80%93ViewModel>

Para entenderlo mejor, a continuación, se muestra una tabla extraída de la página de [guru99](https://www.guru99.com/mvc-vs-mvvm.html) en la que se muestran las principales diferencias entre ambos patrones:

TABLA 1 MVC VS MVVM SOURCE: <https://www.guru99.com/mvc-vs-mvvm.html#:~:text=In%20MVC%2C%20controller%20is%20the,AND%20code%20is%20event%2Ddriven>

MVC	MVVM
El controlador es el punto de entrada a la aplicación.	La vista es el punto de entrada a la aplicación.
Relaciones <i>One to Many</i> entre el controlador y la vista.	Relaciones <i>One to Many</i> entre la vista y la vista-modelo.
La vista no tiene referencia al controlador.	La vista tiene referencias a cada vista-modelo.
MVC es un modelo antiguo.	MVVM es un modelo relativamente nuevo.
MVC puede ser probado de forma separada al usuario	Fácil separación entre los test unitarios y el código que mantiene las relaciones entre el modelo y cada vista-modelo.

Este patrón es el que utilizan la mayoría de los nuevos frameworks de desarrollo de aplicaciones JavaScript en las denominadas [Single Page Application](#) (SPA).

## 2.3.1 Otros patrones

A partir de las dos arquitecturas vistas previamente MVC y MVVM, empezaron a salir diferentes patrones de arquitectura que las complementaban permitiendo, según el framework, que la arquitectura de la aplicación fuera mucho más legible y eficiente. Las más importantes hoy en día entre los frameworks más utilizados son:

### 2.3.1.1 Modular Architecture

Consiste en estructurar la aplicación en diferentes módulos basados en los diferentes dominios que pueda tener. El principal framework que utiliza este patrón es Angular, que lo veremos más adelante.

### 2.3.1.2 Component Architecture

Este patrón es utilizado por la mayoría de frameworks hoy en día y consiste en dividir las diferentes vistas de la aplicación en componentes, la idea es que esta división se realice en base a un criterio en el que todos los elementos que forman un componente tengan sentido juntos y no

separados, que dentro del contexto de la vista tenga sentido agruparlos. Por ejemplo, un listado de elementos puede ser un componente, y ese componente está formado por tantos componentes como elementos tenga ese listado. De esta forma, cada componente tendrá su parte HTML, JavaScript y CSS separada del resto de componentes.

#### 2.3.1.3 Dumb-Smart Components

A partir del anterior patrón, surge una separación más entre componentes tontos y componentes inteligentes, entendiendo por componentes tontos aquellos componentes que no tienen ninguna lógica de negocio, que únicamente son elementos visuales que no van a alterar el estado de la aplicación, una cabecera, por ejemplo. Por otro lado, los componentes inteligentes son aquellos que sí que contienen lógica de negocio y, por lo tanto, sí que pueden modificar el estado de la aplicación.

#### 2.3.1.4 State Management

Este patrón ha surgido a medida que las aplicaciones front-end han ido creciendo y adquiriendo mayor complejidad, lo que va a hacer es ayudarnos a no depender constantemente de un back-end a la hora de realizar operaciones intermedias que requieran de un estado temporal, como, por ejemplo, un formulario, en algunos formularios con cierta complejidad, se necesitan realizar operaciones que requieren mantener un estado temporal hasta que finalmente tras rellenar todo el formulario se manda al back-end, esto se consigue gracias a unas librerías denominadas como State Management como pueden ser Redux para React, NgRx para Angular, o Vuex para Vue, que permiten mediante un conjunto de elementos gestionar estado en la parte cliente de la aplicación, evitando así saturar el backend cada vez que se requiera mantener un estado temporal. Se recomienda la lectura de la introducción de estas tres librerías para entender más en profundidad este patrón.

#### 2.3.1.5 Unidirectional Architecture

Partiendo del anterior patrón de arquitectura, este lo que dice es que el flujo de datos solo debería viajar en una sola dirección, en concreto, hacia la vista, la cual nunca debería alterar directamente el estado de la aplicación frontend, lo que sí que va a hacer es con la ayuda de las librerías de State Management vistas anteriormente, lanzar señales para que se modifique el estado, pero siempre desde estos módulos de gestión del estado.

#### 2.3.1.6 Micro Frontends

Este patrón se asemeja al patrón de arquitectura [Micro Servicios](#) aplicado en los back-end, y es que en algunas aplicaciones, en las que la parte frontend empieza a crecer y a desarrollar nuevas funcionalidades que de por sí ya son bastantes grandes, la gestión del front-end se vuelve inmanejable, este es uno de los principales motivos por los que se realiza una separación de cada funcionalidad / caso de uso en una aplicación front-end independiente, permitiendo que todos estos *micro-frontends* se comuniquen y compartan recursos entre ellos.

Este modelo de arquitectura cada vez se está volviendo más popular y en aplicaciones que se prevé que van a crecer bastante se plantea desde un comienzo como *micro-frontends*. Para profundizar más en este patrón se recomienda leer el libro de [Building Micro-Frontends by Luca Mezzalana](#).

### 2.3.2 Patrones de diseño

Entendemos por *patrones de diseño* aquellas soluciones generales y reusables que dan solución a ciertos tipos de problemas en un contexto en concreto y que aparecen repetidas veces

en la fase de diseño y en la fase de desarrollo de una aplicación. Estos patrones pueden ser categorizados en las siguientes tres categorías:

- **Patrones estructurales:** establecen unas pautas para las conexiones entre los diferentes componentes del sistema haciéndolas más eficientes y aportando flexibilidad.
- **Patrones de creación:** implementan mecanismos de creación de objetos para que el código sea reutilizable y sea más flexible.
- **Patrones de comportamiento:** identifica y simplifica los procesos comunes de interacción entre los objetos del sistema.

Muchos de los patrones que vamos a ver a continuación se utilizaban únicamente en el desarrollo de servicios back-end que seguían el paradigma de *Programación Orientada a Objetos*, pero el desarrollo front-end cada vez está adoptando más este paradigma, esto es debido a que [TypeScript](#) está cogiendo mucha popularidad hoy en día, TypeScript es un lenguaje de programación escrito sobre JavaScript que permite adoptar este paradigma mediante el uso de tipos, clases, interfaces, etc. Es por ello por lo que en el desarrollo front-end se esté volviendo de utilidad algunos de estos patrones que anteriormente únicamente se utilizaba en servicios back-end.

A continuación, vamos a presentar brevemente algunos de los patrones de diseño más comunes en las aplicaciones front-end, pero por lo general, la mayoría de los patrones de diseño son aplicables en el desarrollo de aplicaciones front-end.

Para conocer todos estos patrones o para profundizar más en algunos de ellos se recomienda la web de [Refactoring Guru](#).

### 2.3.2.1 Patrones estructurales

#### 2.3.2.1.1 Patrón de fachada

Cuando un sistema está compuesto por múltiples subsistemas complejos, se puede utilizar este patrón de diseño para crear una interfaz que internamente interactúe con cada uno de los subsistemas encapsulándolos en esta interfaz abstrayendo su complejidad, de esta manera simplificamos la interacción y comunicación con cada uno de los subsistemas.

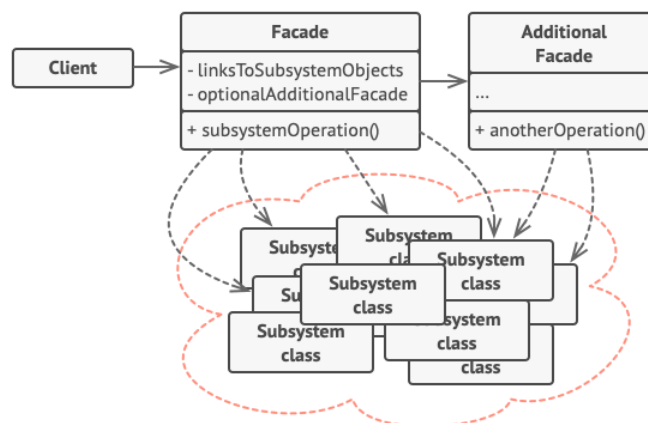


ILUSTRACIÓN 4 FACADE DESIGN PATTERN SOURCE: [HTTPS://REFACTORING.GURU/ES/DESIGN-PATTERNS/FACADE](https://refactoring.guru/es/design-patterns/facade)

### 2.3.2.2 Patrones de creación

#### 2.3.2.2.1 Patrón de fábrica (*Factory Method*)

Este patrón nos ayuda a crear objetos sin tener que conocer el detalle de sus clases abstrayendo su lógica y simplificando su creación, con esto se consigue que el código sea más flexible ante cambios en la creación de dichos objetos. Para ver un ejemplo de cómo implementarlo con TypeScript se recomienda el siguiente [artículo](#).

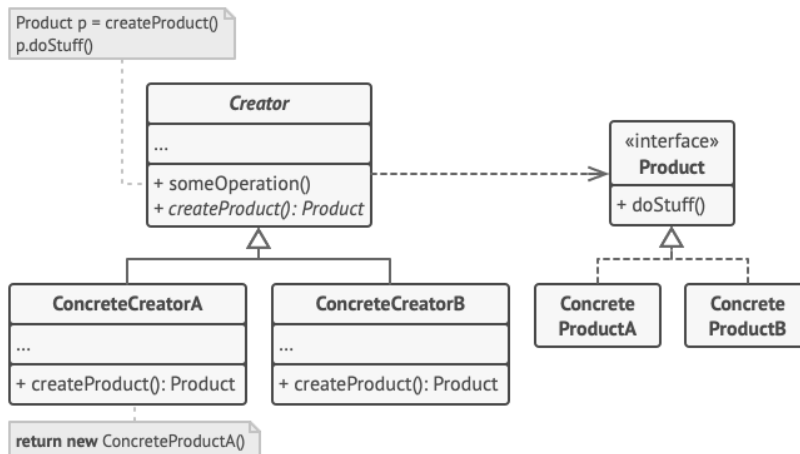


ILUSTRACIÓN 5 FACTORY METHOD DESIGN PATTERN SOURCE: [HTTPS://REFACTORING.GURU/ES/DESIGN-PATTERNS/FACTORY-METHOD](https://refactoring.guru/es/design-patterns/factory-method)

#### 2.3.2.2.2 Patrón Singleton

Aplicando el siguiente patrón de creación conseguimos que las clases que se van creando tengan una única instancia, proporcionando un punto de acceso global a la misma.

Algunos frameworks como Angular ya proporcionan este patrón en sus servicios a través de la inyección de dependencias.

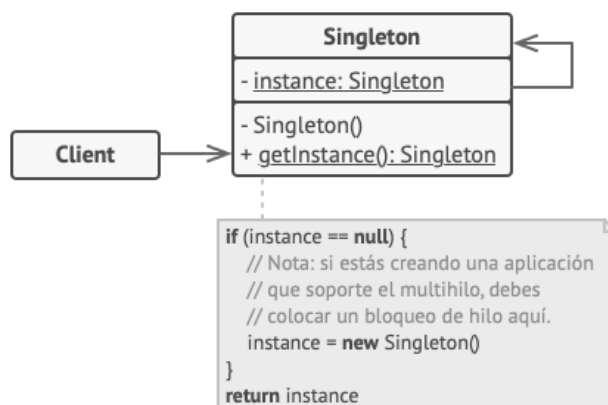


ILUSTRACIÓN 6 SINGLETON DESIGN PATTERN SOURCE: [HTTPS://REFACTORING.GURU/ES/DESIGN-PATTERNS/SINGLETON](https://refactoring.guru/es/design-patterns/singleton)

### 2.3.2.3 Patrones de comportamiento

#### 2.3.2.3.1 Patrón de estrategia (*Strategy*)

En la mayoría de las aplicaciones se tienen objetos de clase que en función del contexto del escenario en el que se encuentren tienden a comportarse de una manera u de otra. En estos casos el patrón *Strategy* recomienda extraer la lógica específica de cada uno de estos escenarios a clases

separadas, de esta manera tenemos la flexibilidad de añadir nuevos escenarios sin tener que alterar la clase principal.

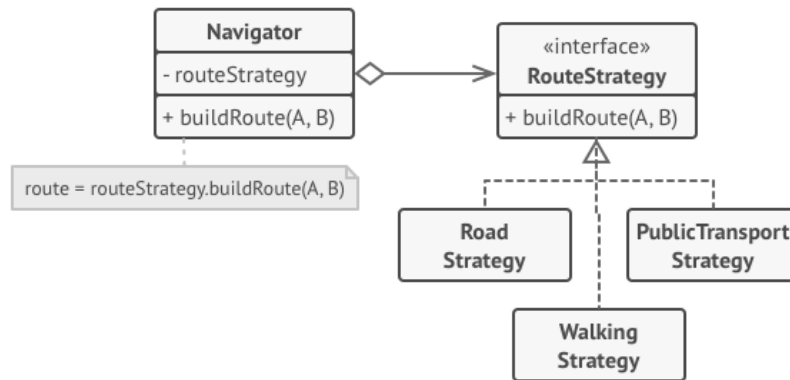


ILUSTRACIÓN 7 STRATEGY DESIGN PATTERN SOURCE: [HTTPS://REFACTORING.GURU/ES/DESIGN-PATTERNS/STRATEGY](https://refactoring.guru/es/design-patterns/strategy)

### 2.3.2.3.2 Patrón de observador

Este patrón establece un mecanismo que permite notificar a un grupo de interesados sobre los posibles cambios de estado sufridos por un objeto, estos interesados se subscriben a determinados objetos para enterarse de los posibles cambios en el momento en el que se producen.

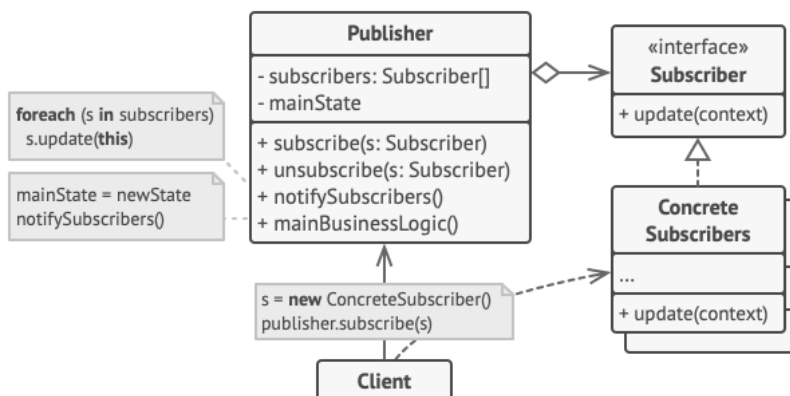


ILUSTRACIÓN 8 OBSERVER DESIGN PATTEN SOURCE: [HTTPS://REFACTORING.GURU/ES/DESIGN-PATTERNS/OBSERVER](https://refactoring.guru/es/design-patterns/observer)

Este patrón es muy común en las librerías y frameworks modernos de desarrollo front-end, ya que con este mecanismo de subscripciones se puede reaccionar ante los cambios de estado producidos con el propósito de que solo ciertas partes de la vista se actualicen.

## 2.4 Evolución del desarrollo front-end

En un primer momento las aplicaciones web no tenían la separación actual entre front-end y back-end, **el código era compartido** y se enviaba la página a mostrar directamente desde el servicio al navegador. La siguiente evolución ya supuso la separación que conocemos actualmente y lo único que **las páginas eran estáticas** ya que únicamente estaban compuestas por [HyperText Markup Language](#) (HTML) y [Cascading Style Sheets](#) (CSS), lo que quiere decir que para que el contenido visualizado en el navegador cambiara era necesario realizar una recarga de la página en el navegador.



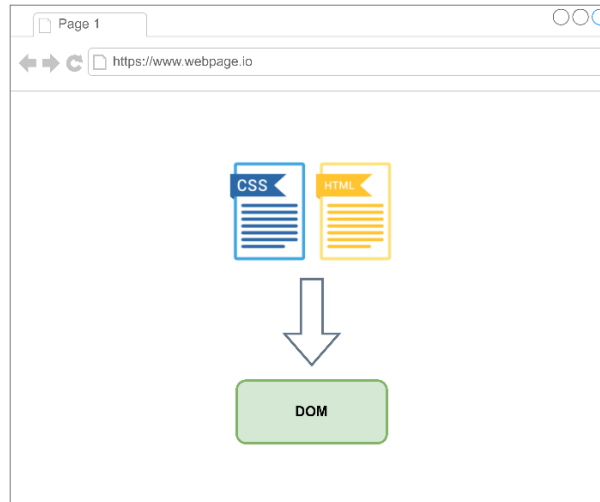


ILUSTRACIÓN 9 DESARROLLO FRONTEND I

El siguiente paso que se dio con la necesidad de eliminar la restricción existente del contenido estático y es que en el año 1995 [Brendan Eich](#) creó el **lenguaje de programación JavaScript** con el principal objetivo de crear un **estándar** entre los navegadores para tener un lenguaje con el que poder realizar operaciones sobre el [Document Object Model](#) (DOM) del navegador.

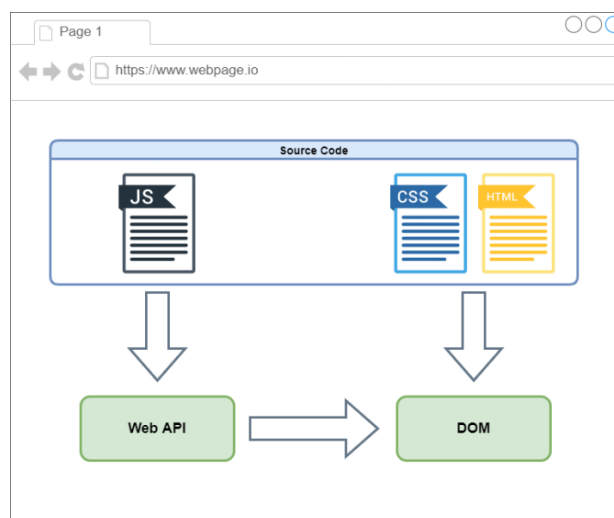


ILUSTRACIÓN 10 DESARROLLO FRONTEND II

El siguiente surgimiento fue el de [Asynchronous JavaScript And XML](#) (**AJAX**) supuso un gran avance ya que AJAX permitía por primera vez que las páginas no solo mostraran contenido si no que podían **administrar los datos de la aplicación de manera independiente** debido a que el script de AJAX permite lanzar peticiones de solicitud de datos al back-end, además este script también permite **interaccionar con los usuarios** ya que permite realizar actualizaciones del contenido mostrado en base a los datos recuperados del servidor. En este punto la complejidad de la parte front-end de las aplicaciones web ya era considerable ya que no solo se encargaba de mostrar las páginas necesarias, si no que ahora ya podía reaccionar ante las interacciones del usuario, pero lo hacía de una manera un poco caótica, hasta que finalmente los desarrolladores se dieron cuenta de que era necesario añadir una capa para abstraer todas esas operaciones realizadas sobre las [Web APIs](#) como Ajax y las operaciones de manipulación del DOM, finalmente

apareció [jQuery](#) para solucionar todos estos problemas **aumentando la legibilidad del código, su mantenibilidad, y facilitando la compatibilidad entre navegadores.**

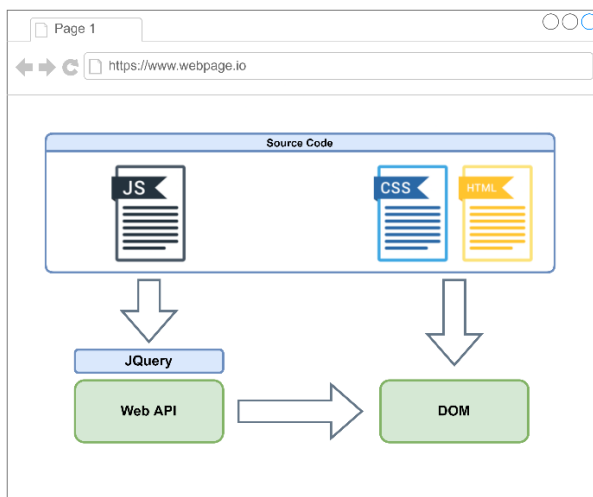


ILUSTRACIÓN 11 DESARROLLO FRONTEND III

Si que es cierto que JQuery mejoró mucho la forma de trabajar en desarrollo frontend, pero había algunos problemas que aún persistían en las aplicaciones web de gran tamaño, el principal problema es el mecanismo que ofrece JQuery para la manipulación del DOM, a través de consultas basadas en selectores de clase o identificadores de elementos, y esto cuando se realizan muchas operaciones sobre el DOM se vuelve un poco problemático.

Fueron este tipo de acciones tan repetitivas y comunes en las aplicaciones web las que originaron la necesidad de una **librería** o **framework** que las incorporaran y que además incorporaran aquella funcionalidad que pudiera ser de utilidad en cualquier aplicación web, es aquí donde nace esta última capa de abstracción que es finalmente la que utiliza el desarrollador para llevar a cabo el desarrollo de la parte front-end de una aplicación.

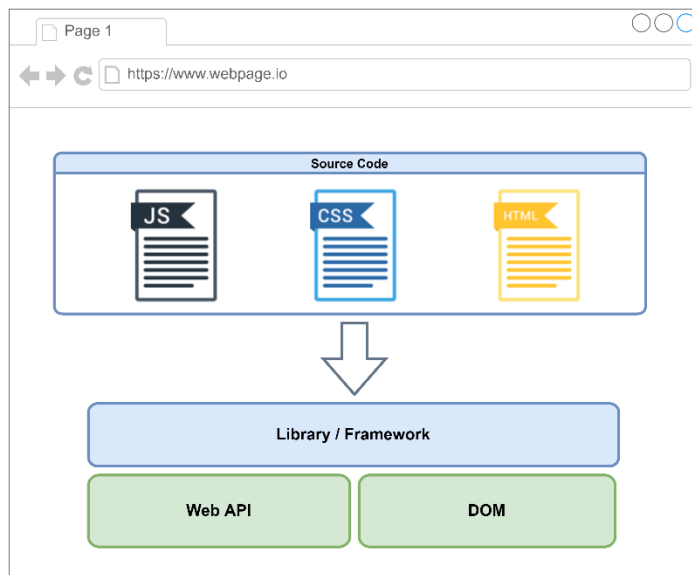


ILUSTRACIÓN 12 DESARROLLO FRONTEND IV

Con este último esquema no solo se solucionan los problemas asociados a las tareas repetitivas y comunes, si no que se solucionan muchos problemas asociados al diseño de la

arquitectura, haciendo que las aplicaciones web desarrolladas siguiendo este último esquema sean mucho más flexibles y escalables.

Una vez alcanzado este punto cercano a la actualidad es cuando hay un incremento considerable de apariciones de nuevos frameworks y librerías de todo tipo para el desarrollo de aplicaciones frontend, en el siguiente diagrama se puede ver un resumen a lo largo del tiempo de la aparición de cada una de las nuevas tecnologías.

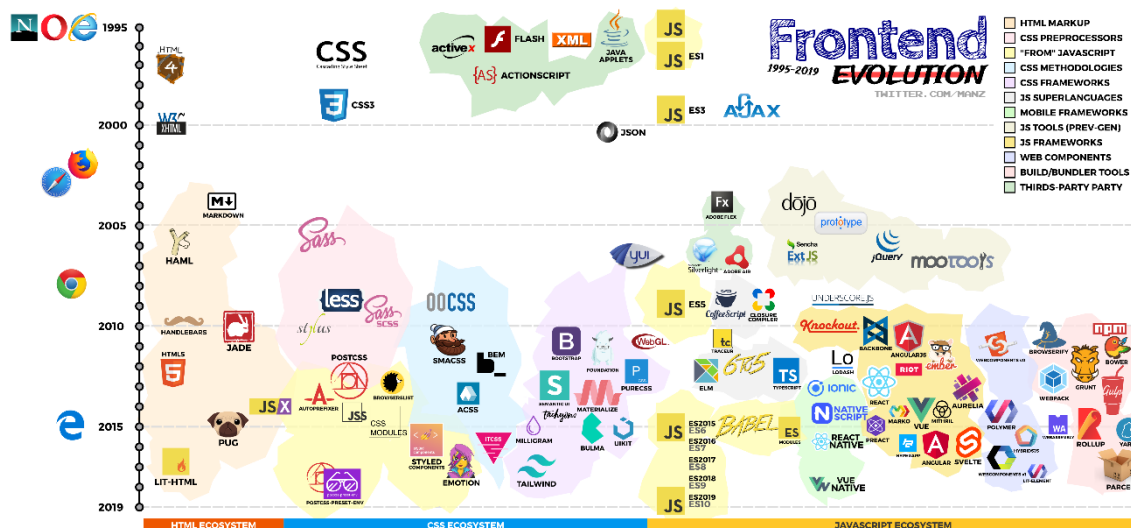


ILUSTRACIÓN 13 FRONTEND EVOLUTION SOURCE: <https://github.com/ManzDev/FRONTEND-EVOLUTION>

Actualmente este mercado está dominado por 3 o 4 librerías / frameworks que son los más utilizados hoy en día. En el siguiente diagrama obtenido de la página web de [stateofjs](https://stateofjs.com) a partir de los datos recogidos en las encuestas realizadas por diferentes desarrolladores de todo el mundo, podemos ver la evolución a lo largo del tiempo de cada uno de estos frameworks siendo en este último año 2021 **React**, el framework más **utilizado**, seguido de **Angular**, **Vue.js**, y **Svelte**. En la siguiente sección entraremos un poco más en detalle en cada uno de ellos.

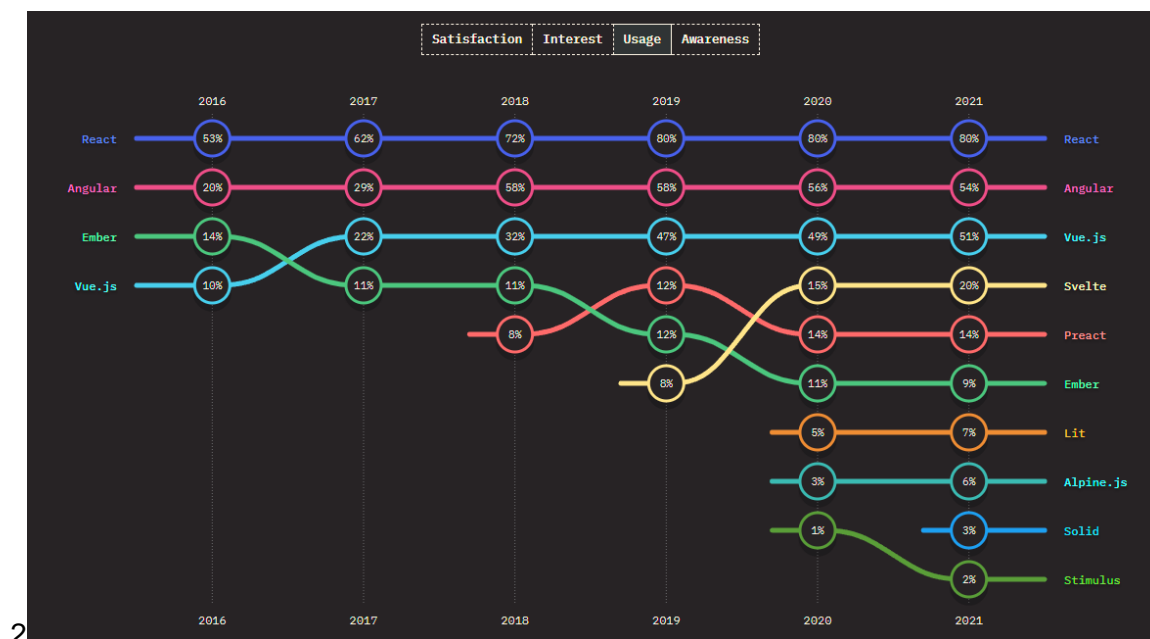


ILUSTRACIÓN 14 FRONTEND FRAMEWORKS 2021 USAGE SOURCE: <https://2021.stateofjs.com/en-US/libraries/front-end-frameworks>

En cambio, si los analizamos por el grado de satisfacción vemos que por ejemplo Angular queda de los últimos y en cambio Svelte que es de los nuevos por encima.

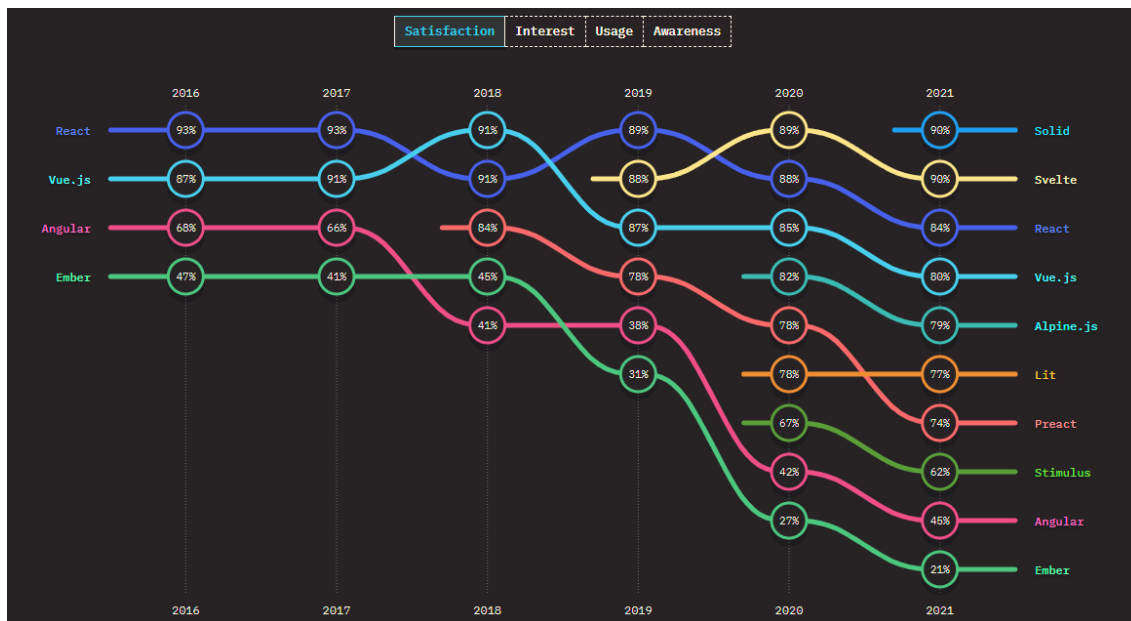


ILUSTRACIÓN 15 FRONTEND FRAMEWORKS INTEREST 2021 SOURCE: <https://2021.STATEOFS.COM/EN-US/LIBRARIES/FRONT-END-FRAMEWORKS>

Por otro lado, dentro del desarrollo front-end hay diversas aplicaciones multiplataforma que son desarrolladas utilizando tecnologías web como lo son HTML, JavaScript, y CSS. Estas herramientas cada vez son más populares dentro de las tareas de los desarrolladores front-end, y por ello, también vamos a analizar dos de las más importantes en el mercado actual de aplicaciones multiplataforma.

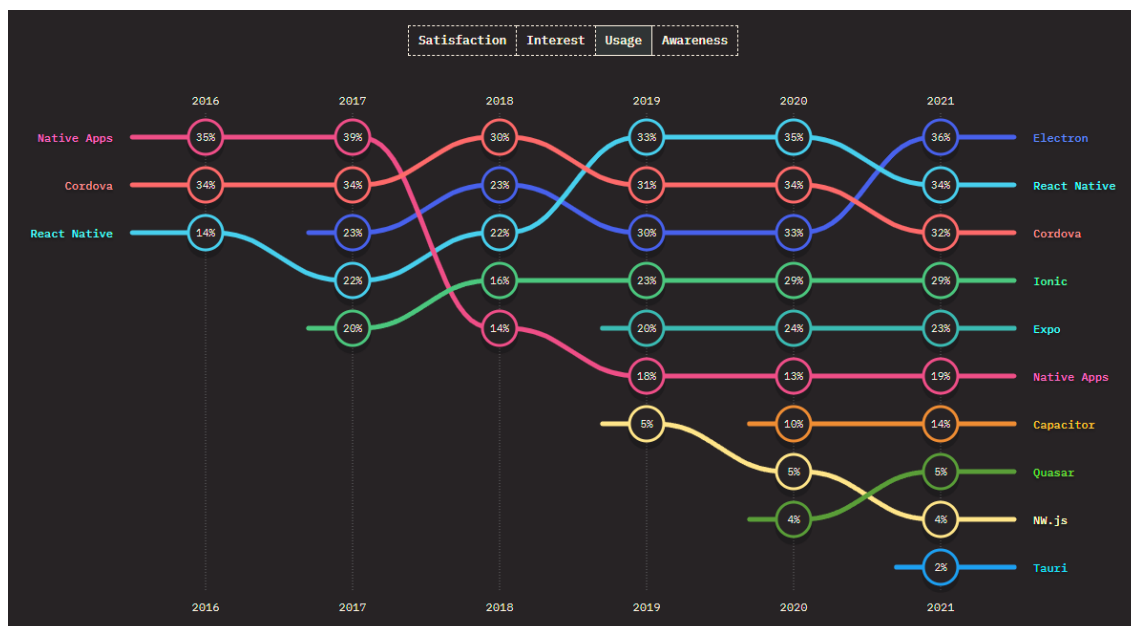


ILUSTRACIÓN 16 MOBILE-DESKTOP LIBRARIES SOURCE: <https://2021.STATEOFS.COM/EN-US/LIBRARIES/MOBILE-DESKTOP>

Como podemos apreciar, las dos librerías más utilizadas actualmente para dicho propósito son **Electron**, y **React Native**, en tercera posición tenemos **Cordova** que es una alternativa a React Native que cada vez se está utilizando menos.

## 2.5 Frameworks y herramientas

### 2.5.1 Frameworks y librerías de desarrollo

#### 2.5.1.1 React

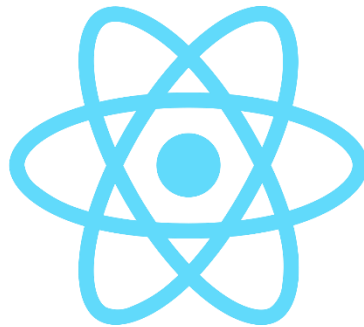


ILUSTRACIÓN 17 LOGOTIPO LIBRERÍA REACT

[React](#) es una librería que fue lanzada por Jordan Walke (ingeniero software de **Facebook**) en el año 2013, y desde entonces siempre se ha mantenido como una de las alternativas más utilizadas debido a su **rendimiento**, su **flexibilidad**, y su **ecosistema**. El hecho de que fuera creada y mantenida por [Facebook](#) (actualmente conocida como Meta) y por toda la comunidad de desarrolladores libres ha ayudado bastante a que siempre se mantuviera en la cúspide.

En un comienzo su principal competencia era Angular pese a que eran dos opciones bastante diferentes, ya que React está pensada como una librería para el desarrollo de componentes UI, mientras que Angular es considerado un framework para el desarrollo de aplicaciones web completas. Los principales puntos fuertes de React son frente al resto de opciones son:

- **Flexibilidad:** toda la funcionalidad de la aplicación va a través de librerías de terceros, pudiendo elegir entre diferentes alternativas a la hora de implementar por ejemplo el enrutado de la aplicación o la gestión del estado.
- **Tamaño:** debido a su sencillez es de las alternativas con menor peso final.
- **[Virtual DOM](#):** mejora el rendimiento de la aplicación optimizando los cambios realizados sobre el DOM.
- **[JSX](#):** facilita el desarrollo de plantillas HTML haciendo uso de JavaScript.
- **Rendimiento:** al ser una librería bastante ligera los tiempos de arranque y de procesamiento en el cliente son bastante reducidos.
- **Curva de aprendizaje:** como no trae un *toolkit* propio para el desarrollo de aplicaciones web su curva de aprendizaje es reducida.

Otra de las ventajas que ofrece React es [React Native](#) una librería para el desarrollo de aplicaciones híbridas en la que simplemente con saber React puedes llegar a hacer aplicaciones nativas para Android e iOS partiendo el mismo código.



ILUSTRACIÓN 18 LOGOTIPO FRAMEWORK ANGULAR

[Angular](#) desarrollado por **Google** y por la comunidad de código libre es uno de los **frameworks** de desarrollo de aplicaciones web más utilizados hoy en día, no confundir con su primero lanzamiento de no [AngularJs](#) que fue uno de los primeros frameworks en surgir, pero no ganó popularidad hasta su versión 2.0 (Angular) en la que mejoró bastante su rendimiento y su *Developer Experience* (DX), este cambio no fue un simple incremento de versión con mejoras, si no que fue el desarrollo de un nuevo framework desde 0 que ofrecía estas mejoras gracias a la [Inyección de dependencias](#), mejoras en la manipulación del DOM, etc.

Hay varios puntos por los que Angular siempre se ha mantenido como una de las opciones más utilizadas, aunque hoy en día de las menos populares hablando en términos de comenzar un nuevo proyecto. Una de las principales ventajas es que está **mantenido por Google** y esto desde el principio ha hecho que la comunidad de desarrolladores apueste por dicha tecnología, ya que ante cualquier problema hay una comunidad que lo abala y que en la mayoría de los casos te vas encontrar fácil solución a los problemas que puedan surgir. Otra de las ventajas que ofrece Angular es que por defecto viene con un **toolkit** tan **completo** que apenas es necesario integrar librerías de terceros para el desarrollo de aplicación web complejas. Esto introduce uno de los defectos a ojos de muchos desarrolladores, y es que **la curva de aprendizaje de Angular es de las más elevadas** entre las diferentes alternativas.

Finalmente, cabe destacar que otra de las características que ofrecía Angular frente al resto es la posibilidad de utilizar [TypeScript](#), ofreciendo la posibilidad de utilizar el [modelo de programación orientada a objetos](#), esto antes marcaba una gran diferencia debido a que está demostrado que las aplicaciones desarrolladas utilizando TypeScript son más seguras y estables, y se prevé que aumente mucho el uso de esta tecnología. Esto era un punto a favor hasta hace poco, debido a que la mayoría de las alternativas hoy en día permiten desarrollar haciendo uso de las ventajas de TypeScript.

### 2.5.1.3 Vue



ILUSTRACIÓN 19 LOGOTIPO FRAMEWOR VUE.JS

[Vue](#) fue lanzada al poco tiempo de la salida de React, en el 2014 [Evan You](#) creador de Vue.js, lanzó una primera versión que quizá no fue tan popular como React o Angular en esos años, pero con el tiempo se volvió una de las alternativas favoritas para el desarrollo de aplicaciones front-end.

Vue al igual que React es una librería, aunque en temas de peso final se acerca más a Angular que a React, esto es debido a que Vue ofrece por defecto funcionalidad que React deja de la mano de librerías externas. Vue comparte con React otras características como:

- Utiliza un [DOM virtual](#).
- Componentes **reactivos** y **componibles**.
- Ambas están pensadas como **librerías centrales** en las que temas como el enrutamiento y la gestión global del estado son manejadas por librerías asociadas.
- Permite **JSX**, aunque también da la posibilidad de trabajar con plantillas HTML.

Pero sin lugar a duda la característica que destaca de Vue frente al resto de alternativas es su **curva de aprendizaje**, esta librería cuenta con **muy buena documentación** y una forma sencilla de empezar a trabajar con ella que hacen que suela ser la opción elegida cuando se quiere empezar aprendiendo alguno de los frameworks. Esto ha hecho que el mercado se incline hacia esta alternativa ya que la curva de aprendizaje de los nuevos integrantes de un equipo abarata bastante el coste de los proyectos. Otra de las características por las que los desarrolladores eligen Vue es su flexibilidad, ya que permite llevar a cabo rápidamente una idea inicial permitiendo su crecimiento escalando su arquitectura sin ningún problema. Muchas veces cuando se piensa en proyectos grandes se descarta unívocamente la idea de utilizar esta alternativa, ya que está demostrado que está a la altura del resto de opciones cuando se habla de proyectos de gran escala.

#### 2.5.1.4 Svelte



ILUSTRACIÓN 20 LOGOTIPO FRAMEWORK SVELTE

[Svelte](#) es uno de los últimos frameworks en aparecer, fue desarrollado por *Rich Harris* en 2016 y su última versión con la que alcanzo bastante popularidad fue desarrollada con [TypeScript](#) en 2019. Este framework ha ido ganando bastante popularidad debido a que es considerado uno **de los más rápidos** reduciendo el tiempo de arranque y mejorando el rendimiento del tiempo de ejecución en el cliente. Esto es debido a que Svelte a diferencia del resto trabaja más **como un compilador**, en vez de realizar todo el trabajo en el propio navegador, lo realiza en su mayoría al construir la aplicación.

Otra de las ventajas que ofrece es que no trabaja con el [virtual DOM](#), si no que **opera directamente sobre el DOM** del navegador, y esto en algunos casos puede mejorar el rendimiento de la aplicación, en el siguiente [post](#) escrito por los desarrolladores de Svelte viene explicado más profundamente.

Finalmente, otro punto que destaca de Svelte frente al resto de alternativas, es **su tamaño** ya que **ronda los 1.5kB**, frente a los 62.2kB de Angular y los 23kB de Vue, en cambio React se acerca bastante con 2.6kB.

Svelte es una muy buena opción cuando se quiere desarrollar aplicaciones de calidad en un tiempo récord.



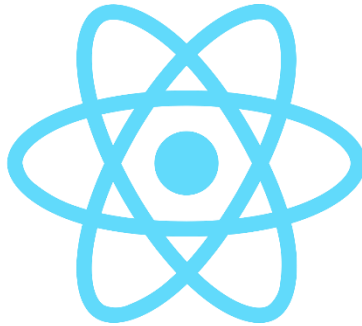


ILUSTRACIÓN 21 LOGOTIPO FRAMEWORK REACT NATIVE

[React Native](#) a diferencia de React es un **framework** que sirve para el **desarrollo de aplicaciones móviles**, este framework está basado puramente en React incluyendo un conjunto de herramientas útiles para el desarrollo de aplicaciones móviles.

Las aplicaciones desarrolladas con este React Native tienen la capacidad de ejecutarse nativamente tanto en dispositivos Android, como IOs, y pesar de que estas aplicaciones estén desarrolladas con tecnologías web, finalmente permiten generar las aplicaciones nativas para los diferentes sistemas operativos, esto es una gran ventaja ya que permite tener una única base del código para ambos sistemas.

Todos los puntos fuertes vistos anteriormente para React son igualmente aplicables a este tipo de aplicaciones, pero introducen algún problema cuando se trata de dispositivos móviles, más adelante veremos qué ventajas y desventajas tienen sobre una opción puramente nativa, pero en generales este tipo de aplicaciones la principal ventaja que nos ofrece es la de agilizar el proceso de desarrollo de aplicaciones móviles abaratando su coste, y permitiendo a desarrolladores web a desarrollar aplicaciones móviles sin tener que invertir demasiado tiempo en formación.

### 2.5.1.6 Electron



ILUSTRACIÓN 22 LOGOTIPO FRAMEWORK ELECTRON

[Electron](#) es un framework que fue creado por un desarrollador llamado Cheng Zhao pero que actualmente es mantenido por [GitHub](#), este framework nos permite desarrollar aplicaciones de escritorio multiplataforma compatibles con los sistemas operativos de Linux, Windows, y Mac, además este framework nos sirve para desarrollar tanto la parte del cliente (renderizada por el navegador [Chromium](#)), como la parte del servidor (utilizando [Node.js](#)), todo ello haciendo uso de HTML, CSS, y JavaScript.

Aplicaciones de escritorio de gran renombre como lo son [Visual Studio Code](#), [Whatsapp](#), [Twitch](#), [Teams](#), han sido desarrolladas utilizando este framework, esta es una de las principales razones que llevan a las organizaciones a utilizar esta tecnología en vez de las opciones puramente nativas a la hora de desarrollar aplicaciones de escritorio.

Al igual que el framework de React Native visto anteriormente, Electron, nos da la posibilidad de disponer de una única base del código que nos sirve para generar las aplicaciones para los distintos sistemas operativos, es por ello por lo que una de las principales ventajas que nos ofrece es agilizar el desarrollo de aplicaciones de escritorio abaratando su coste final, así como la posibilidad de que sea desarrollada por desarrolladores web.

A diferencia de lo que se pueda pensar, esta tecnología al no ser una opción puramente híbrida (de las que hablaremos más adelante) permite hacer uso de las características que son propias de cada plataforma, por ejemplo, utilizar el sistema de notificaciones del sistema operativo.

Más adelante, junto a React Native analizaremos cuales son las ventajas y desventajas de utilizar esta opción frente a una puramente nativa, y hablaremos de los aspectos de seguridad que deberemos tener en cuenta a la hora de tomar esta decisión.

## 2.5.2 Herramientas

### 2.5.2.1 npm



ILUSTRACIÓN 23 LOGOTIPO NPM

*Node Package Manager* [npm](#) es uno de los registros de paquetes software más utilizados en todo el mundo por los desarrolladores de JavaScript, existen otros como puede ser [yarn](#), pero en aquellos sitios donde tengamos que poner un ejemplo sobre algo que involucre a un gestor de paquetes vamos a utilizar npm. En el desarrollo de aplicaciones web esta pieza es imprescindible porque nos permite administrar sus dependencias y por lo general la mayoría de aplicación web requieren de muchas dependencias que son descargadas a través de este tipo de herramientas.

Esta herramienta no solo sirve para la descarga de dependencias, sino que también nos ayuda a instalarlas o desinstalarlas de nuestras aplicaciones, mantener una buena gestión de versiones, etc. Otro de los usos que se le dan, es el de una plataforma en línea que nos permite publicar y compartir módulos o librerías con el resto de los desarrolladores. Para más información se recomienda ir a la documentación oficial de npm.



ILUSTRACIÓN 24 LOGOTIPO JAVASCRIPT

[JavaScript](#) (JS) es el lenguaje de programación o de scripting más utilizado hoy en día por los desarrolladores de aplicaciones web, más en concreto, el más utilizado en el desarrollo de las aplicaciones front-end. Todos los frameworks modernos vistos anteriormente están escritos con JS y por lo tanto el desarrollo de las aplicaciones utilizando alguno de estos frameworks es con JS. Pero JS no solo es utilizado para dichos propósitos sino que también es utilizado fuera del desarrollo de aplicaciones web.

Posee un estándar oficial que es el [ECMAScript](#) mantenido por la organización de estándares [ECMA Internacional](#). Este lenguaje se creó con el objetivo de hacer que las páginas web dejaran de ser estáticas y que el usuario pudiera interaccionar con ellas mejorando la experiencia de usuario, esto se consigue gracias a que los scripts desarrollados con JavaScript permiten la manipulación del *Document Object Model* (DOM) y el CSS de las páginas, otro de los propósitos de JS es que pudiera ser ejecutado por cualquiera de los navegadores existentes y de esta manera que las páginas web funcionaran independientemente del navegador.

La gran comunidad que rodea a este lenguaje y su cada vez más extendida omnipresencia en diferentes aplicaciones ejecutadas por diferentes sistemas hacen que JavaScript tenga un futuro asegurado entre los lenguajes de programación.



ILUSTRACIÓN 25 LOGOTIPO TYPESCRIPT

[TypeScript](#) es un lenguaje de programación escrito sobre JavaScript que facilita el desarrollo de aplicaciones permitiendo la definición de tipos en el código JavaScript. No solo mejora la experiencia del desarrollador facilitando su trabajo, sino que también hace que el código JavaScript resultante sea más robusto y estable gracias a la comprobación estática de tipos. Sus principales funcionalidades son:

- **Tipos de datos primitivos:** a la hora de declarar las variables se les puede asignar un tipo primitivo de los que ofrece TypeScript.
- **Arrays:** a la hora de indicar el tipo de las variables permite indicar que lo que va a contener es un Array del tipo se le indique.
- **Any:** declaración de variables de cualquier tipo, las variables declaradas con *any* pueden contener cualquier tipo.
- **Functions:** al igual que las variables declaradas TypeScript permite declarar los tipos de los parámetros de entrada a las funciones, así como el tipo de los valores devueltos.
- **Object Types:** la estructura de datos *Object* de JavaScript ahora puede ser tipada, pudiendo declarar los tipos de cada uno de los atributos que lo forman.
- **Clases:** definición de clases JavaScript que hacen uso de tipos.
- **Interfaces:** las interfaces sirven para definir la forma de ciertos objetos pudiendo dar la posibilidad de que estas interfaces sean extendidas por otras.
- **Types:** permite la creación de nuevos tipos para ser usados en la declaración de variables a modo de alias.

El uso de TypeScript en las nuevas aplicaciones web está en constante crecimiento, actualmente todos los frameworks y librerías de desarrollo modernos permiten incorporar TypeScript al proyecto, y su uso es cada vez más recomendado por la comunidad de desarrolladores front-end. A lo largo de este libro se expondrán varios ejemplos y por lo general todos ellos estarán escritos en TypeScript.

## 3 ANÁLISIS DE AMENAZAS

Con el propósito de analizar aquellos servicios y mecanismos de seguridad que son más importantes incluir en el desarrollo de interfaces web independientemente del cliente que lo ejecute, ya sea un ordenador, un teléfono móvil, o una tablet, lo primero que vamos a hacer es tomar como principales amenazas las listadas por la organización OWASP en el Top 10 de 2021.



ILUSTRACIÓN 26 LOGOTIPO OWASP TOP 10

### 3.1 Amenazas Aplicaciones web

#### 3.1.1 OWASP

La "Open web Application Security Project" (OWASP) es una comunidad sin ánimo de lucro que tiene como objetivo promover el desarrollo de aplicaciones web seguras y [fiables](#). Su proyecto es totalmente abierto y que está en constante actualización por la comunidad. Cada 4 años sacan un documento con el Top 10 de vulnerabilidades de seguridad más comunes y sus riesgos en aplicaciones web, aplicaciones de dispositivos móviles.

Este documento tiene como objetivo concienciar, y aunque muchas organizaciones lo usen de estándar de seguridad, OWASP cuenta con un estándar oficial [OWASP Application Security Verification Standard](#) (ASVS). Otro de los usos que se le dan a este Top es a modo de métrica para evaluar la seguridad que ofrece una aplicación [web](#). Las recomendaciones de uso de este Top 10 de OWASP son las siguientes:

Use Case	OWASP Top 10 2021	OWASP Application Security Verification Standard
Awareness	Yes	
Training	Entry level	Comprehensive
Design and architecture	Occasionally	Yes
Coding standard	Bare minimum	Yes
Secure Code review	Bare minimum	Yes
Peer review checklist	Bare minimum	Yes
Unit testing	Occasionally	Yes
Integration testing	Occasionally	Yes
Penetration testing	Bare minimum	Yes
Tool support	Bare minimum	Yes
Secure Supply Chain	Occasionally	Yes

FIGURA 1. RECOMENDACIONES OWASP TOP 10. FUENTE:  
[HTTPS://OWASP.ORG/Top10/A00\\_2021\\_How\\_to\\_Use\\_the\\_OWASP\\_Top\\_10\\_as\\_a\\_Standard/](https://owasp.org/Top10/A00_2021_How_to_Use_the_OWASP_Top_10_as_a_Standard/)

Por lo tanto, este Top 10 es un buen punto de partida para dotar nuestras aplicaciones de seguridad, pero no es suficiente por sí solo.

### 3.1.2 OWASP Top 10

Este listado de vulnerabilidades se ha obtenido a partir de la recolección de datos, categorizados y estructurados en base a la lista de *Common weakness Enumeration (CWE)* esta lista contiene los tipos de debilidades software y hardware más comunes, entendiendo por debilidad, un fallo, bug, o cualquier otro error en la implementación, en el código, en el diseño, o en la arquitectura del software o hardware, que si no se abordan pueden comprometer el sistema haciéndolo vulnerable, entendiendo por vulnerable a una aplicación en la que se ha detectado un agujero de seguridad, o una debilidad, debido a un fallo en el diseño o a un bug, que permite a un atacante causar algún tipo de daño a las partes interesadas de la aplicación, estas partes interesadas incluyen al propietario de la aplicación, a los usuarios u otras partes relacionadas con la aplicación. Para la realización de este Top 10 han contado con datos de más de 500.000 aplicaciones proporcionados por:

- AppSec Labs.
- Cobalt.io.
- Contrast Security.
- GitLab.
- HackerOne.
- HCL Technologies.
- Micro Focus.
- PenTest-Tools.

- Probely.
- Sgreen.
- Veracode.
- WhiteHat (NTT).

Otros años en la recolección de datos únicamente se han solicitado datos de incidencias reportadas de las 30 primeras CWEs, pero este año se ha eliminado ese filtro con el objetivo de ver que CWEs desaparecen o cuales aguantan a lo largo de los años. Finalmente se analizan los datos en base a un conjunto de CWEs que pueden ser de dos tipos diferentes, **root cause o symptom**, el primero de ellos hace referencia a aquellos CWEs que tienen que ver con problemas que surgen por ejemplo de una mala configuración o de un fallo criptográfico, y la otra categoría se corresponde al resto de CWEs como por ejemplo la exposición de datos sensibles o la denegación de servicios. Pero por lo general el que más importa para la realización de este Top 10 es el de *root cause* debido a que para este grupo es más factible proporcionar un método de identificación y de corrección. Finalmente, los datos son categorizados en función de la tasa de incidencias y los *Common Vulnerability Scoring System* (CVSS).

Para cada una de las amenazas del Top 10 se recopilan los siguientes factores asociados a los datos de los que se ha extraído cada una de ellas:

- **CWEs Mapped:** Número de CWEs mapeados a la categoría del Top por el equipo que desarrolla el Top 10.
- **Incidence Rate:** porcentaje de aplicaciones vulnerables al CWE de la población probada por esa organización para ese año.
- **Weighted Exploit:** Puntuación del 0 al 10 correspondiente a la subpuntuación de explotación de las puntuaciones obtenidas CVSSv2 y CVSSv3 asignadas a los CVEs y mapeadas a CWEs.
- **Weighted Impact:** Puntuación del 0 al 10 correspondiente a la subpuntuación de impacto de las puntuaciones CVSSv2 y CVSSv3 asignadas a CVEs y mapeadas a CWEs.
- **(Testing) Coverage:** Porcentaje de aplicaciones testeadas por todas las organizaciones para un CWE determinado.
- **Total Occurrences:** Número total de aplicaciones encontradas que contienen CWEs mapeados a una categoría.
- **Total CVEs:** Número total de CVEs en NVD DB (*National Vulnerability Database*) que se han mapeado a los CWs mapeados a una categoría.



### 3.1.3 OWASP Top 10 - 2021

Este Top 10 surgió hace 20 años con el propósito de analizar las amenazas más importantes para el desarrollo de aplicaciones web seguras. La última versión de este Top 10 se hizo en el año 2017 sin ninguna actualización, pero desde el pasado año 2021 se ha empezado a trabajar en una nueva versión de este. Los cambios realizados respecto a la anterior versión se pueden ver en la siguiente imagen:

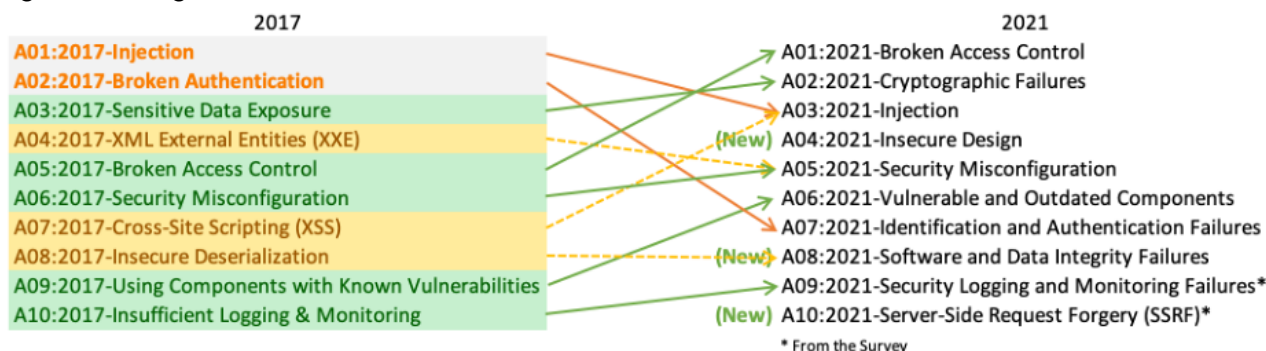


FIGURA 2. OWASP TOP 10 2021. FUENTE: [HTTPS://OWASP.ORG/Top10/](https://owasp.org/Top10/)

En este Top 10 hemos podido ver que han aparecido nuevas vulnerabilidades y algunas otras han sido renombradas y tanto las nuevas como las renombradas han adquirido un nombre que tiene más en cuenta la causa raíz de la vulnerabilidad y no la consecuencia de la misma. Principalmente los cambios más destacables de este Top son los siguientes:

- **Broken access control:** pasa a estar en primer lugar, escalando 4 puestos, hoy en día es la vulnerabilidad con más ocurrencias para los CWEs mapeados.
- **Using Components with Known Vulnerabilities:** se renombra a **Vulnerable and Outdated Components**, y escala del puesto 9 al 6. El renombre permite a la vulnerabilidad a no solo contemplar las vulnerabilidades asociadas a componentes con vulnerabilidades conocidas, sino que también contempla aquellas vulnerabilidades producidas por componentes que han sido obsoletos.
- **Injection:** deja de estar en primer lugar y desciende al tercer puesto, más adelante hablaremos del motivo por el cual esta vulnerabilidad ya no es tan recurrente como las otras.
- **Sensitive Data Exposure:** se renombra a **Cryptographic Failures** y pasa del puesto número 3 al número 2. El nuevo nombre deja más claro cuál es la vulnerabilidad y no cual es el riesgo.
- **Broken Authentication:** se renombra a **Identification and Authentication Failures** y desciende del puesto número 2 al número 7.
- **(NEW) Insecure Design:** aparece en el puesto número 4, y representa aquellos riesgos que provienen de un mal diseño de alguna de las fases del desarrollo de la aplicación web.
- **(NEW) Software and Data Integrity Failures:** aparece en el número 8 y nace de la vulnerabilidad de **Insecure Deserialization** para darle más importancia a las vulnerabilidades producidas no solo por una deserialización insegura sino por la falta de verificación de integridad.
- **(NEW) SSRF (Server-Side Request Forgery):** en el último puesto del Top aparece esta nueva categoría que parece que cada vez es más habitual que un atacante introduzca peticiones HTTP hacia un dominio arbitrario.

A continuación, vamos a repasar cada una de las vulnerabilidades del Top 10 analizando cuales de ellas nos afectan de manera más directa en el desarrollo front-end de las aplicaciones web.

### 3.1.3.1 A01:2021-Broken Access Control

La primera de las vulnerabilidades tiene que ver con el control de acceso implantado en las aplicaciones web, se ha convertido en la más importante de todas debido a que estos últimos años se registraron alrededor de 318k incidencias causadas por dicha amenaza, con una tasa de incidencia promedio de 3,48% para el 94% de las aplicaciones en las que fue probado.

Otro de los factores que la hacen tan crítica es que es el origen de muchas otras ya que una vez se ha violado la política de control de acceso de una aplicación se pueden realizar diferentes ataques como exponer información sensible, alterar la información del usuario contenida en la aplicación, etc. Por lo que dicha vulnerabilidad afecta a la mayoría de los aspectos más importantes de la seguridad, como es la confidencialidad, integridad, autenticidad, etc. Esta categoría recoge las siguientes amenazas listadas por la OWASP:

- Violación del principio de menor privilegio, o denegación por defecto. Este principio dice que el acceso de un usuario ha de ser mediante grupos, roles, o capacidades, donde cualquier operación sobre cualquier recurso está prohibida por defecto.
- Eludir las comprobaciones implantadas por el control de acceso mediante la alteración de la URL, la modificación del estado interno de la aplicación, o la interceptación de peticiones a un API.
- Visualización, modificación no permitida de la información contenida en la cuenta del usuario.
- Uso no autorizado de las *APIs* de la aplicación suplantando la identidad.
- Elevación de privilegio, ya que sin haber pasado los controles de acceso puede haber adquirido privilegios de otro usuario.
- Manipulación de los metadatos, JWT (*JSON web Token*), cookies, campos ocultos, etc.
- Configuración (*Cross-Origin Resource Sharing*) CORS incorrecta permitiendo el acceso a las *APIs* desde orígenes no autorizados.
- Acceso no autorizado a partes de la aplicación que requieren previa autenticación

Las medidas que propone OWASP para prevenir los posibles ataques producidos por dicha vulnerabilidad son:

- **Denegar por defecto**, excepto para recursos públicos. Más adelante veremos la importancia de este punto a la hora de establecer restricciones de acceso en el enrutado de una aplicación front-end.
- Implementar mecanismos de control de acceso y su reutilización a lo largo del flujo de la aplicación. Validando los permisos del usuario en cada parte de la aplicación, estableciendo permisos atómicos.
- Hacer buen uso del CORS esta recomendación se establece directamente en los APIs que consume la aplicación.
- Deshabilitar la lista de directorios del servidor web y asegurarse de que los metadatos y copias de seguridad no estén presentes.
- Monitorizar errores de control de acceso estableciendo alertas a administradores en caso de repetidos fallos.
- Establecer límites en los *APIs* para minimizar daños en caso de ataques automáticos.

- Invalidar las sesiones de usuario tan pronto como se cierre sesión, y **establecer tokens JWT de corta duración** para minimizar la franja de tiempo en la que un atacante puede actuar. **Seguir lo estándares de OAuth** para la gestión de los tokens. Estos últimos puntos son muy importantes a la hora de establecer nuestro mecanismo de sesión de usuario.

Uno de los ataques más comunes realizados sobre dicha vulnerabilidad es el acceso no autorizado a URLs que únicamente deberían estar expuestas a usuarios administrador.  
<https://example.com/app/getappInfo> [https://example.com/app/admin\\_getappInfo](https://example.com/app/admin_getappInfo)

### 3.1.3.2 A02:2021-Cryptographic Failures

"*Cryptographic Failures*" más conocido anteriormente como "*Sensitive Data Exposure*", hace referencia no tanto al síntoma en sí de la vulnerabilidad, si no a la causa que la provoca, los fallos relacionados con la criptografía o con su ausencia finalmente desembocan en la exposición de datos sensibles. Dentro de los 29 CWEs mapeados para esta vulnerabilidad encontramos que los 3 más recurrentes son *CWE-259: Use of Hard-coded Password*, *CWE-327: Broken or Risky Crypto Algorithm*, y *CWE-331 Insufficient Entropy*.

El mundo de las aplicaciones web cada día es más extenso y el tipo de datos que manejan cada vez es más variado, contraseñas, números de tarjetas de crédito, datos personales, etc. Es por ello que una parte del diseño de una aplicación debería determinar cuáles de los datos que maneja han de ser protegidos y de qué manera. OWASP nos propone los siguientes puntos para identificar fallos criptográficos:

- Se utilizan protocolos como HTTP, SMTP, FTP, para datos que viajan en claro.
- Se están utilizando algoritmos o protocolos de cifrado antiguos o débiles.
- Se utilizan claves de cifrado débiles o con mala rotación.
- No se está aplicando cifrado donde se debería.
- Mala aleatoriedad de los sistemas de cifrado.
- Se hace uso de funciones [hash](#) obsoletas como MD5 O SHA1.
- Se hace uso de métodos de relleno de cifrado obsoletos.
- Los mensajes de error de cifrado son vulnerables.

Por otro lado, recomienda los siguientes puntos como mínimo para prevenir ataques originados a causa de fallos criptográficos:

- Clasificar los datos procesados, almacenados, o transmitidos por la aplicación identificando que datos son sensibles acorde a las [leyes de privacidad](#), requerimientos regulatorios, o necesidades de negocio.
- No almacenar datos sensibles innecesarios. La información que no existe no puede ser robada.
- Asegurarse de que los datos sensibles que no están en movimiento están cifrados, como por ejemplo las contraseñas.
- Asegurar que los algoritmos, protocolos y claves estándar son seguros y actualizados, además de utilizar una gestión de claves adecuada.
- Cifrar todos los datos que vaya a ser enviados con protocolos como *Transport Layer Security* (TLS).
- Deshabilitar la caché para respuestas que contienen datos sensibles.

- Implementar los controles de seguridad necesarios según la clasificación de los datos.
- No hacer uso de protocolos heredados como *File Transfer Protocol* (FTP) y *Simple Mail Transfer Protocol* (SMTP) para transportar datos confidenciales.
- Almacenar contraseñas utilizando funciones de hash *salted*, dichas funciones utilizan un dato variable denominado *salt* para dotar al hash generado de la suficiente entropía posible.
- Utilizar mecanismos de autenticación cifrados.
- Utilizar claves cifradas generadas al azar y almacenarlas en memoria en modo binario.
- Para los métodos de cifrado aleatorio asegurarse que las semillas utilizadas son seguras.
- No hacer uso de funciones de cifrado obsoletas como MD5, SHA1, PKCS, etc
- Verificar de forma independiente la eficiencia de la configuración establecida y sus ajustes.
- Desactivar el autocompletado en los formularios que recogen datos sensibles.

Uno de los ejemplos más comunes de ataque a esta vulnerabilidad es cuando no se hace uso del protocolo TLS o se utiliza un mecanismo de cifrado débil. Un atacante podría capturar por ejemplo las credenciales utilizadas en el envío de peticiones y utilizarlas con fines maliciosos.

### 3.1.3.3 A03:2021-Injection

La inyección de código baja de la primera a la tercera posición, el 94% de las aplicaciones han sido testeadas contra inyecciones de formularios obteniendo una tasa de incidencia máxima del 19%, y una tasa media del 3%, con 274k de ocurrencias. Dentro de las 33 CWEs mapeadas destacan la *CWE-79: Cross-site Scripting*, *CWE-89: SQL Injection*, y *CWE-73: External Control of File Name or Path*.

Esta vulnerabilidad se produce en el envío de datos no controlados a un intérprete. Los principales motivos propuestos por OWASP de que algunas aplicaciones sufran ataques debidos a dicha vulnerabilidad son:

- Los datos introducidos en la aplicación por el usuario no son validados, filtrados, o borrados cuando no son necesarios.
- Consultas dinámicas o llamadas no parametrizadas sin escapar que son utilizadas directamente por los intérpretes.
- Uso de datos de procedencia desconocida en los parámetros de búsqueda del *Object Relational Mapping* (ORM)
- Uso directo o concatenación de datos de procedencia desconocida.

Por otro lado, para evitar este tipo de ataques recomiendan la revisión del código fuente, y la automatización de pruebas que verifiquen las peticiones que lanza la aplicación. Otro punto importante para prevenir este tipo de ataques es mantener los datos separados de los comandos y las consultas a bases de datos, por ello también recomiendan seguir los siguientes puntos:

- Evitar el uso de *APIs* inseguras que utilizan intérpretes, utilizar *APIs* que proporcionen una interfaz parametrizada o que hacen uso de algún *Object Relational Mapping* (ORM).
- Validar la entrada del lado del servidor.
- Escapar los caracteres especiales usando la sintaxis específica de cada interprete para consultas dinámicas.
- Hacer uso de LIMIT en las consultas a la base de datos.

A día de hoy en el mundo de desarrollo front-end esta vulnerabilidad suele ser muy habitual, y aunque que las herramientas utilizadas ya están protegidas por defecto contra este tipo de ataques igualmente hay que tener cuidado en aquellas zonas en las que almacenamos código en una variable que posteriormente va a ser ejecutado, esto es debido a que si permitimos almacenar código que posteriormente va a ser interpretado damos la posibilidad a un atacante de que inyecte código en dicha variable.

#### 3.1.3.4 A04:2021-Insecure Design

Esta categoría es nueva en el Top 10 de 2021 y engloba aquellas amenazas provenientes de un fallo en el diseño o en la arquitectura de una aplicación. Para recopilar datos acerca de esta nueva categoría se han mapeado 40 CWEs de las que destacan: *CWE-209: Generation of Error Message Containing Sensitive Information*, *CWE-256: Unprotected Storage of Credentials*, *CWE-501: Trust Boundary Violation*, y *CWE-522: Insufficiently Protected Credentials*.

Esta categoría habla únicamente de diseños seguros y no de implementaciones seguras, uno de los principales factores que hacen que un diseño sea inseguro es la falta de perfiles de riesgo de negocio, que determinan el nivel de diseño de seguridad que cierta aplicación requiere.

En la parte de seguridad del diseño de una aplicación es muy importante determinar que requerimientos son necesarios y que gestión de recursos debe tener. Dentro de los requisitos de protección de datos encontramos la confidencialidad, integridad, disponibilidad, y autenticidad de los activos de la aplicación y su lógica de negocio. Por ello a la hora de diseñar una aplicación se recomienda recopilar todo este conjunto de requisitos funcionales y no funcionales que dotan a nuestra aplicación de determinados servicios de seguridad. Estos requisitos están contenidos en diferentes fases del desarrollo de la aplicación, como, por ejemplo, el desarrollo, las pruebas, etc.

OWASP define el diseño seguro como una cultura y metodología que evalúa constantemente las amenazas y que asegura que el código desarrollado es robusto y que es probado contra métodos de ataque bien conocidos. Para garantizar que ambos objetivos se cumplen es necesario que exista un ciclo de vida de desarrollo seguro, siguiendo unos patrones de diseño seguros, haciendo uso de metodologías, librerías de componentes seguros, herramientas seguras, etc.

Para prevenir que nuestra aplicación tenga un diseño inseguro, la OWASP recomienda cumplir con los siguientes puntos:

- Establecer y usar un ciclo de vida de desarrollo seguro con expertos en Ciberseguridad que evalúen y diseñen los controles relacionados con la seguridad y privacidad.
- **Establecer y utilizar una biblioteca de patrones de diseño seguros, o componentes seguros ya implementados.**
- Utilizar el modelado de amenazas para la autenticación crítica, el control de acceso, la lógica de negocio y los flujos de claves para la aplicación.
- Integrar el lenguaje y controles de seguridad en las historias de usuario del desarrollo de la aplicación.
- Integrar verificaciones de plausibilidad en cada nivel de la aplicación (desde el **front-end** hasta el backend).
- Disponer de tests unitarios y de integración.

- Separar en capas de sistema y de red en función de las necesidades de exposición y protección.
- Limitar el consumo de recursos por usuario o servicio.

#### 3.1.3.5 A05:2021-Security Misconfiguration

Esta vulnerabilidad se debe principalmente a la mala configuración de alguna de las partes de una aplicación web, como, por ejemplo, no cambiar las contraseñas de por defecto de las cuentas utilizadas en las diferentes herramientas. De las 20 CWEs mapeadas para esta categoría podemos destacar las siguientes: *CWE-16 Configuration* y *CWE-611 Improper Restriction of XML External Entity Reference*.

OWASP propone los siguientes puntos para detectar que hay algún componente mal configurado:

- Falta de seguridad en alguna de las herramientas utilizadas por la aplicación, o permisos mal configurados en servicios en la nube.
- Funciones innecesarias están habilitadas o instaladas (puertos, **servicios, páginas, cuentas o privilegios innecesarios**).
- Errores controlados que exponen demasiada información a los usuarios.
- Funciones de seguridad deshabilitadas o no configuradas de forma segura en sistemas actualizados.
- Disponer de frameworks, librerías, bases de datos, con una configuración insegura.
- El servidor no envía cabeceras o directivas de seguridad, o no están configuradas en valores seguros.
- Software desactualizado o vulnerable (véase *A06:2021-Vulnerable and Outdated Components*)

Además, para asegurarnos de que una aplicación dispone de todas las configuraciones de seguridad bien establecidas recomiendan que en los procesos de instalación se incluyan los siguientes puntos:

- Los entornos de desarrollo, QA, y producción, han de configurarse de manera idéntica, utilizando diferentes credenciales para cada entorno. Automatizar dicho proceso para que añadir un nuevo entorno no suponga un esfuerzo.
- Eliminar aquellos frameworks, librerías, servicios, etc, que no se estén utilizando.
- Hacer revisiones de la configuración de seguridad de las diferentes herramientas de las aplicaciones.
- Realizar segmentaciones de la arquitectura de la aplicación.
- Envío de directivas de seguridad a los clientes.
- Automatizar los procesos de verificación de los ajustes y configuraciones en todos los entornos.

#### 3.1.3.6 A06:2021-Vulnerable and Outdated Components

La siguiente vulnerabilidad se presenta cuando integramos en alguna aplicación algún componente que posee algún tipo de vulnerabilidad no parcheada o que se esté utilizando una versión atrasada del componente en la que alguna vulnerabilidad aún no ha sido parcheada. Dentro de las CWEs mapeadas para esta vulnerabilidad encontramos *CWE-1104: Use of Unmaintained*



*Third-Party Components* y las otras dos CWEs se han obtenido del Top 10 2013 y 2017. Los motivos por los cuales OWASP determina que una aplicación puede contener dicha vulnerabilidad son los siguientes:

- Desconocimiento de las versiones utilizadas en los diferentes componentes utilizados.
- Si se está haciendo uso de cualquier software que sea vulnerable, no soportado por la aplicación o obsoleto.
- Si no se suele realizar escaneos de seguridad regularmente o si no estás suscrito a boletines de noticias relacionadas con los componentes que utilizas.
- Si no se parchea o se sube de versión cuando es necesario de las dependencias utilizadas.
- Si los desarrolladores no testean la compatibilidad, de las librerías actualizadas o parcheadas.
- Si no se dispone de una configuración segura de los componentes utilizados (véase *A05:2021-Security Misconfiguration*).

Para prevenir esta vulnerabilidad OWASP recomienda disponer de un sistema de parcheo en el que se realicen las siguientes acciones:

- Eliminar dependencias en desuso, elementos no necesarios, ya sean *features*, componentes, ficheros o documentación.
- Realizar un inventario continuo de las versiones utilizadas en los componentes tanto de la parte cliente como la parte del servidor, también de las dependencias utilizadas.
- Utilizar componentes provenientes de fuentes seguras. (Ver *A08:2021-Software and Data Integrity Failures*)
- Monitorizar aquellas librerías y componentes que no tienen un mantenimiento vigente, o que simplemente no se ha creado un parche de seguridad para una versión antigua.

Más adelante veremos cómo abordar esta vulnerabilidad a la hora de seleccionar las librerías que necesita una aplicación cualquiera, y de cómo podemos hacer controles para monitorizar que componentes y librerías requieren de algún parche o actualización.

#### 3.1.3.7 A07:2021-Identification and Authentication Failures

Fallos en la autenticación y en la identificación, esta vulnerabilidad se suele producir debido a un mecanismo de autenticación débil o una mala configuración de seguridad del sistema de autenticación. Las CWEs asociadas a fallos en la identificación más destacables son: *CWE-297: Improper Validation of Certificate with Host Mismatch*, *CWE-287: Improper Authentication*, y *CWE-384: Session Fixation*.

La autenticación de una aplicación, así como la gestión de la sesión de un usuario son de los puntos más críticos de una aplicación por ello cada vez se integran más mecanismos de seguridad y más potentes. El siguiente listado ofrecido por OWASP contiene los indicativos de que una aplicación pueda ser vulnerable a este tipo de fallos:

- Permitir la automatización de ataques como introducir usuarios y contraseñas de manera automática.
- Permitir ataques automáticos O ataques por fuerza bruta.
- Permitir el uso de contraseñas bien conocidas, débiles, o contraseñas por defecto.

- Utilizar mecanismos de recuperación de contraseñas débiles, por ejemplo, preguntas y respuestas.
- Utilizar almacenamientos de datos de contraseñas que se encuentran en texto plano, cifradas o débilmente cifradas. (véase *A02:2021-Cryptographic Failures*)
- Carecen de mecanismos de autenticación multi factor o son débiles.
- Exposición del identificador de la sesión en una URL.
- Desechar el identificador de la sesión tras un login correcto.
- No se invalida correctamente los identificadores de sesión tras realizar un cierre de sesión o si se ha superado el periodo de inactividad.

Por otro lado, las recomendaciones que ofrecen para evitar esta vulnerabilidad son las siguientes:

- Implementar mecanismos de multi factor para prevenir de ataques automáticos o por fuerza bruta, y robo de credenciales reusadas.
- No realizar despliegues con contraseñas por defecto.
- Implementar un mecanismo de detección de contraseñas débiles o bien conocidas.
- Establecer políticas de seguridad del cambio de contraseñas de manera que se asegure una correcta rotación de estas.
- Asegurarse de que las rutas de registro de cuentas, recuperación de credenciales y uso de *APIs* estén reforzadas contra ataques de enumeración de cuentas.
- Limitar el número de reintentos de login, o aumentar el tiempo establecido entre cada reintento. Además, se recomienda configurar avisos a los administradores ante casos de muchos reintentos.
- Utilizar un administrador de sesión incorporado, seguro y server-side que genere un nuevo identificador de sesión aleatorio con alta entropía después de cada inicio de sesión. El identificador de sesión no debe estar presente en la URL, debe almacenarse de forma segura y debe invalidarse después de los tiempos de espera de cierre de sesión e inactividad.

Esta es una de las vulnerabilidades que mayor carga tiene en la parte front-end de cualquier aplicación, debido a que hay que implementar métodos de verificación e implementar los mecanismos de autenticación oportunos para asegurar que no se produzcan este tipo de ataques.

#### 3.1.3.8 A08:2021-Software and Data Integrity Failures

Esta categoría vista por primera vez en este Top de 2021 está asociada a aquellas amenazas producidas por dar por supuesto que cierto software o herramienta es seguro sin previamente haber comprobado su integridad. Esta categoría es de las que mayor peso tiene su impacto en los datos de (CVE / CVSS). Entre las 10 CWEs mapeadas destacan: *CWE-829: Inclusion of Functionality from Untrusted Control Sphere*, *CWE-494: Download of Code Without Integrity Check*, y *CWE-502: Deserialization of Untrusted Data*.

Esta vulnerabilidad la contiene aquellas aplicaciones que utilizan cualquier tipo de software y no se protegen frente a la violación de integridad. El ejemplo que expone OWASP en el Top 10 es que, si no se hace una verificación de los plugins, librerías o módulos utilizados provenientes de fuentes no conocidas, pueden terminar en accesos no autorizados, código malicioso, o cualquier acción que comprometa al sistema. Otro punto muy importante se presenta a la hora de desarrollar



el ciclo de vida de las aplicaciones ya que en este punto se automatiza la descarga de software como dependencias, librerías, etc. Es por ello por lo que en el CI de una aplicación se automatiza la verificación de integridad de todos los componentes que provienen del exterior.

Para prevenir los ataques producidos contra dicha vulnerabilidad OWASP recomienda cumplir con los siguientes requisitos:

- Utilizar firmas digitales o un mecanismo similar para realizar comprobaciones sobre los datos y el software utilizado.
- Asegurarse de que las librerías y dependencias provenientes de gestores como **npm** o **maven**, provienen de repositorios de confianza.
- Utilizar un software que compruebe la integridad de los datos y el software utilizado.
- Incluir revisiones de los cambios introducidos en las fases de desarrollo.
- Asegurarse de que los pipelines CI/CD desarrollados tengan la segregación, configuración, y control de acceso adecuados.
- Asegurarse de que los datos serializados sin firmar o sin cifrar no se envíen a clientes que no sean de confianza sin algún tipo de verificación de integridad o firma digital para detectar la manipulación o la reproducción de los datos serializados.

Más adelante veremos cómo podemos asegurarnos de que se están realizando las comprobaciones de integridad convenientes a la hora de descargar las librerías y dependencias desde el gestor utilizado, ya sea **npm**, **yarn**, etc

#### 3.1.3.9 A09:2021-Security Logging and Monitoring Failures

Para la siguiente categoría no se cuenta con muchos datos CVE/CVSS debido a que el registro de logs y la monitorización de una aplicación es complejo de testear, pero tiene un gran impacto sobre la contabilidad, la visibilidad, las alertas de ataques, y los análisis forenses de una aplicación. Para esta categoría se han mapeado 4 CWEs, *CWE-778 Insufficient Logging to include CWE-117 Improper Output Neutralization for Logs*, *CWE-223 Omission of Security-relevant Information*, y *CWE-532 Insertion of Sensitive Information into Log File*.

Sin el registro de logs y monitorización de una aplicación es muy difícil detectar posibles brechas de seguridad, los indicativos que hacen referencia a dicha vulnerabilidad según la OWASP son los siguientes:

- No se está registrando ningún tipo de evento auditable, como inicios de sesión, inicios de sesión fallidos, y transacciones importantes.
- No se registran advertencias o errores, o el registro que generan es inadecuado o escaso.
- No se monitorizan los registros de las aplicaciones y o APIs en busca de actividades sospechosas.
- Los registros generados son almacenados únicamente de manera local.
- Umbrales de alerta y procesos de escalada de respuesta inexistentes o poco eficientes.
- Los *pentesting* realizados y los análisis de pruebas de seguridad de aplicaciones dinámicas no activan alertas.
- La aplicación no detecta, escala, o alerta sobre ataques activos en tiempo real o casi real.

Para prevenir este tipo de fallos recomiendan implementar un conjunto de controles en función del riesgo de la aplicación:

- Asegurar que los registros generados disponen de suficiente contexto del usuario como para identificar cuentas sospechosas o maliciosas, y que estos registros puedan ser retenidos durante un tiempo suficiente como para realizar un análisis forense.
- Asegurar que los registros generados puedan ser tratados fácilmente.
- Asegurar la buena codificación de los registros para evitar inyecciones o ataques sobre los sistemas de registro y monitoreo.
- Asegurar la auditoría de transacciones importantes estableciendo controles de integridad.
- Asegurar que los equipos de *DevSecOps* hayan establecido monitorización y un sistema de alertas efectivos.
- Establecer o adoptar un plan de recuperación y respuesta ante incidentes.

Cada vez es más popular implementar mecanismos para el registro de eventos y monitorización en las aplicaciones front-end, en los últimos años únicamente se establecían dichos mecanismos en los servidores. Más adelante veremos ¿Cómo? ¿Cuándo? Y ¿Qué? Es importante registrar y monitorizar.

### 3.1.3.10 A10:2021-Server-Side Request Forgery

Esta nueva categoría al ser nueva únicamente cuenta con una CWE mapeada. SSRF se produce cuando se consume un recurso externo a través de una URL no validada proporcionada por un usuario, esto permite a un atacante cualquiera a forzar a una aplicación a realizar el envío de peticiones manipuladas hacia un destino inesperado, independientemente de si la aplicación está protegida por un firewall, VPN un ACL.

Este tipo de incidencias se encuentran en crecimiento debido a que cada vez es más común que las aplicaciones brinden a los usuarios la posibilidad de realizar dichas funciones. OWASP recomienda implementar todos o alguno de los mecanismos de seguridad que se presentan a continuación:

- **Desde la capa de red**
  - Segmentar la funcionalidad de acceso a recursos remotos en redes separadas.
  - Aplicar políticas de firewall de "*deny by default*" o reglas de control de acceso a la red, con el objetivo de bloquear todo el tráfico de internet no esencial.
- **Desde la capa de aplicación**
  - Verificar y validar todos los datos de entrada proporcionados por el usuario.
  - Asegurar que se cumple el esquema de URL, implementando una lista positiva de puestos y destinos permitidos.
  - No enviar respuestas sin procesar a los clientes.
  - Deshabilitar las redirecciones HTTP.
  - Asegurar la consistencia de las URLs.
  - Evitar reducir el SSRF mediante el uso de listas de denegación o expresiones regulares.
- **Medidas adicionales**
  - **Evitar el uso de mecanismos de seguridad front-end como por ejemplo OpenID.**
  - Para front-ends con grupos de usuarios dedicados y manejables, usar cifrado de red (por ejemplo, VPN) en sistemas independientes para considerar necesidades de protección muy altas.

Por lo general los ataques SSRF más comunes son el escaneo de puertos para obtener información de estos, exponer información sensible a través de URLs locales, acceso a metadatos alojados en una URL de un servicio CLOUD, etc.

#### 3.1.3.11 A11:2021 – Next Steps

OWASP Top 10 se limita a las 10 vulnerabilidades más relevantes, pero como es lógico hay muchas otras vulnerabilidades que quizá no son tan recurrentes, pero si igual de importantes. A continuación, se listan las **3 vulnerabilidades** que se recomienda tener en cuenta con el objetivo de identificarlas y remediarlas en caso de que la vulnerabilidad sea real.

##### 3.1.3.11.1 Errores de código

Esta categoría incluye aquellos defectos o patrones de seguridad conocidos, a continuación, se listan los más importantes:

- Reutilización de variables.
- Exponer información sensible en los registros debug.
- Errores *off-by-one*.
- Condiciones de carrera.
- Errores de conversión.
- ...

Por lo general los lenguajes de programación modernos evitan por defecto muchos de estos errores, lo que se suele hacer es establecer condiciones de compilación estrictas para evitar que dado un código no pueda compilar si detecta uno de estos errores. Para ello se suelen implementar análisis de código estático en el IDE utilizado y en el propio lenguaje, esta es una parte muy importante a la hora de montar proyectos front-end y que está ganando mucha popularidad entre los desarrolladores, en la siguiente sección veremos que herramientas nos van a ayudar a cumplir estos criterios de calidad y cómo configurarlas.

##### 3.1.3.11.2 Denegación de servicios

Esta vulnerabilidad se puede presentar siempre si se utilizan los recursos necesarios. Para poder estar previstos ante estos ataques lo más común es realizar pruebas de rendimiento de CPU, E/S de disco, y uso de memoria, con el objetivo de rediseñar, optimizar, o almacenar en caché operaciones costosas. Lo más normal es aplicar estas medidas de seguridad sobre los servidores y no tanto sobre las interfaces.

##### 3.1.3.11.3 Errores de gestión de memoria

Las aplicaciones web suelen estar desarrolladas utilizando lenguajes de memoria administrada, como, Java, .NET, JavaScript, y por lo general estos lenguajes tienen problemas de gestión de memoria, como buffer o *heap overflows*. Actualmente el desarrollo de las *APIs* se está orientado hacia lenguajes sin este tipo de problemas como Rust o Go. Por lo general en las aplicaciones front-end no se tiene muchas alternativas salvo el estar previstos frente a la primera de estas últimas categorías, asegurar que la calidad del código es eficiente mejora en algunos casos este tipo de problemas.

### 3.1.4 Otras vulnerabilidades

Fuera ya del Top10 de OWASP surgen nuevas vulnerabilidades constantemente, y no por no figurar en dicha referencia son menos importantes. En la siguiente imagen publicada por el blog de seguridad de *synk* podemos ver reflejadas aquellas vulnerabilidades no contempladas por el top de OWASP y que son de relevancia hoy en día.

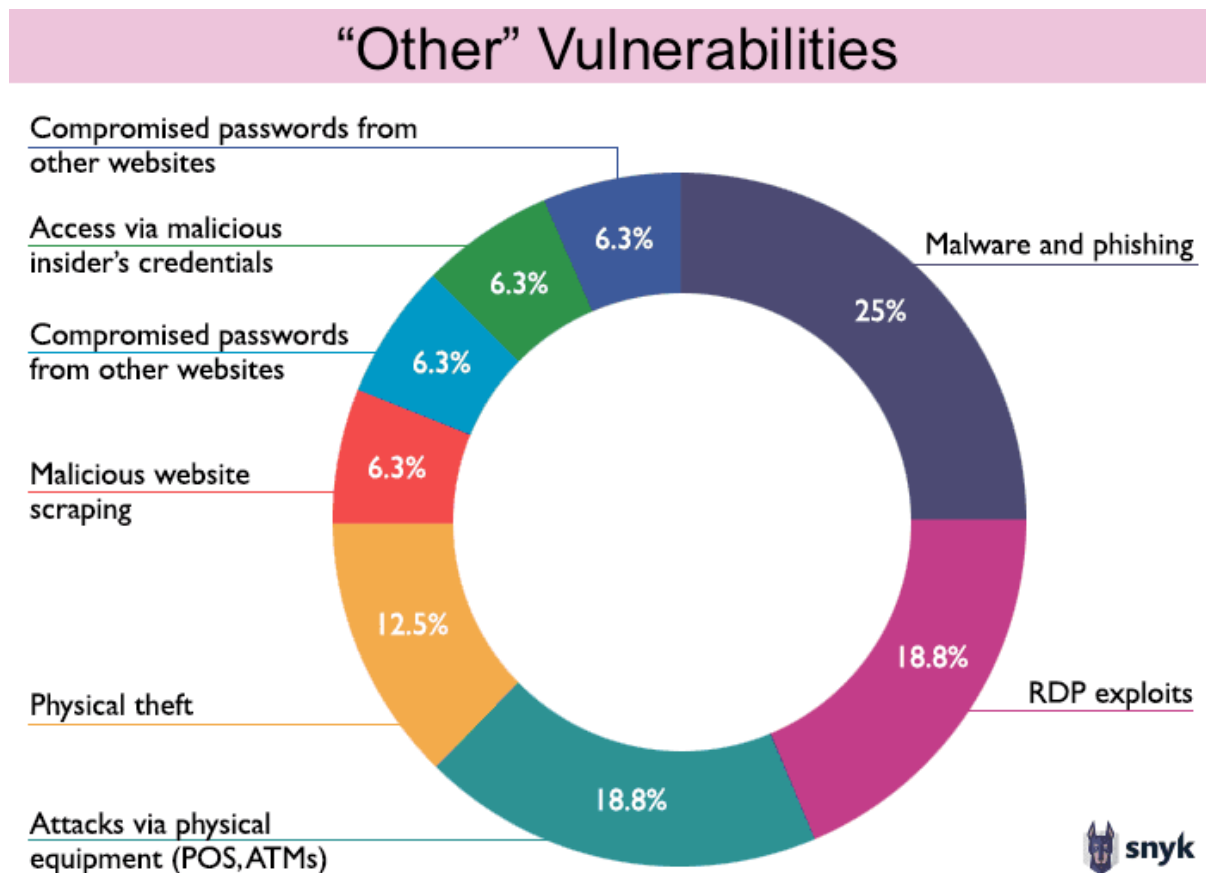


FIGURA 3. "OTHER" VULNERABILITIES. FUENTE: [HTTPS://SNYK.IO/LEARN/OWASP-TOP-10-VULNERABILITIES/](https://synk.io/learn/owasp-top-10-vulnerabilities/)

A continuación, analizaremos aquellas que nos afectan directamente en el desarrollo de aplicaciones front-end, yendo desde vulnerabilidades a bajo nivel como las que afecta a JavaScript Vanilla, hasta vulnerabilidades más comunes hoy en día en los frameworks modernos más utilizados.

## 3.2 Amenazas JavaScript

El creciente uso de JavaScript para el desarrollo de la parte cliente de las aplicaciones web es indudable, ya sea una aplicación desarrollada en JavaScript Vanilla, TypeScript, o utilizando alguno de los frameworks modernos, lo que finalmente se ejecuta en el navegador es JavaScript, es por ello que se dedica una sección exclusiva para el análisis de las amenazas y vulnerabilidades más comunes de este lenguaje, cierto es que el uso de JavaScript para el desarrollo backend también es bastante popular, pero únicamente nos vamos a enfocar en aquellas que afectan al desarrollo front-end.

Las vulnerabilidades que vamos a ver a continuación, aunque se produzca en el lado del cliente finalmente se pueden convertir en amenazas del lado del servidor ya que una vez explotadas permiten lanzar peticiones contra APIS en las que en un principio no se tiene autorización. En esta

sección únicamente vamos a listar las más comunes y ver cuáles son las mejores medidas de prevención, pero más adelante veremos como poder aplicar dichas medidas.

### 3.2.1 Riesgos y problemas

#### 3.2.1.1 Ejecución de código malicioso

A través del *Document Object Model* (DOM) de una página web un atacante puede inyectar un script para que sea interpretado desde el HTML y que en consecuencia todos los clientes que estén cargando esa página web ejecutaran ese script en sus navegadores.

#### 3.2.1.2 Robo de datos del usuario

Un atacante se puede aprovechar de los scripts de la aplicación para acceder a los datos compartidos entre el navegador y la aplicación web. Normalmente la mayoría de las aplicaciones web tienen que hacer uso de cookies u otros mecanismos como *LocalStorage* o *SessionStorage*, y por lo general el uso que se les da a estos almacenes de datos es para guardar datos de la sesión del usuario, por lo tanto, dentro de la información que puede sacar un atacante encontramos: identificadores de sesión con el que hacer accesos no autorizados, información sensible del usuario, etc.

#### 3.2.1.3 Actividad del usuario no intencionada

Utilizando técnicas para manipular el navegador un atacante puede realizar actividades maliciosas en los sitios donde un usuario haya iniciado sesión. Otra forma de realizar estas acciones no intencionadas es alterando el contenido visualizado por el usuario para hacerle que el propio usuario realice una acción diferente.

#### 3.2.1.4 Explotación de vulnerabilidades en el código fuente

Normalmente toda aplicación web hace uso de paquetes y librerías externas en las que confían pero que pueden traer consigo vulnerabilidades ocultas, finalmente este código es ejecutado junto al de la propia aplicación web exponiendo dichas vulnerabilidades, ampliando las posibilidades de un atacante

#### 3.2.1.5 Vulnerabilidades más comunes

Se pueden agrupar en los siguientes tipos de vulnerabilidad: *Cross-Site Scripting* (XSS), código malicioso, *Cross-Site Request Forgery* (CSRF o XSRF), ataques *man-in-the-middle* y explotación de vulnerabilidades del código fuente de la aplicación web.

##### 3.2.1.5.1 Cross-Site Scripting (XSS)

XSS como hemos visto en la sección anterior se encuentra en el número 3 del top 10 es de las vulnerabilidades más peligrosas hoy en día en las aplicaciones web. Los ataques XSS se centran en la inyección de scripts maliciosos, la manera en la que se produce este ataque es debida a como el *Document Object Model* (DOM) del navegador interacciona con la aplicación web, ya que este permite que un script sea anidado en el DOM y finalmente ejecutado por el navegador del usuario. Normalmente la inyección se produce a través de los campos de un formulario o directamente sobre la URL del navegador.

Existen dos tipos de ataque XSS los que se ejecutan directamente a través de un link malicioso denominadas *Reflective XSS*, y los que almacenan en la base de datos un script malicioso

escondido en una imagen o en un comentario que posteriormente ejecutarán todos los navegadores que visualicen el contenido que anida dicho script, estos últimos son denominados *Stored XSS*. El alcance de estos ataques es indiscutible pudiendo llegar a comprometer la aplicación web completamente. Las acciones recomendadas para prevenir este tipo de ataques son las siguientes:

- **Escapar / Codificar la entrada de los usuarios**, muchos de los ataques XSS vienen acompañados de caracteres especiales que son usados en el código HTML, JS, CSS. Lo que se suele hacer es sustituir estos caracteres por sus respectivos en código ASCII, de manera que a la hora de interpretarlo el navegador no lo tome como alguna instrucción a ejecutar. Existe una [Cheat Sheet](#) creada por la OWASP que es muy recomendable leer que muestra un conjunto de estrategias de validación, expresiones regulares, etc.
- **Filtrar la entrada de los usuarios**, otra de solucionar el problema de la entrada proporcionada por el usuario es directamente filtrándola para descartar aquellos caracteres especiales que puedan ser peligrosos a la hora de ser interpretados por el navegador.
- **Validar la entrada de los usuarios**, para evitar tener que escapar o filtrar los caracteres especiales que son peligrosos lo que se puede hacer es validar la entrada del usuario mediante expresiones regulares en función de la información que esperamos recibir por cada campo. Por ejemplo, si esperamos un número avisemos al usuario de que no puede introducir otra cosa diferente.
- **No confiar solo en las validaciones front-end**, una buena práctica a la hora de prevenir ataques XSS establecer siempre validaciones en ambos lados de las aplicaciones web, por ello independientemente de si existen o no validaciones en la parte del servidor, es conveniente validar siempre la parte del cliente.
- **Content Security Policy (CSP)**, CSP establece una capa de seguridad entre el cliente y el servidor de una aplicación web a través de unas reglas establecidas en las cabeceras de las peticiones HTTP o mediante la etiqueta HTML *meta*. Por lo general estas reglas nos van a ayudar a definir qué tipo de contenido esperamos recibir de un determinado dominio, y todo el contenido que no cumpla con dicha política será descartado. Para una lectura más extendida sobre CSP se recomienda la [cheatsheet](#) proporcionada por OWASP.

Para más detalle de cómo protegernos frente a este tipo de ataques son recomendables las siguientes lecturas "[Cross Site Scripting Prevention](#)", "[DOM based XSS Prevention](#)" de OWASP.

#### 3.2.1.5.2 Cross-Site Request Forgery (CSRF)

Esta vulnerabilidad aparece cuando un atacante consigue lanzar peticiones o realizar alguna acción no esperada por el usuario en una aplicación web. Normalmente esto se consigue mediante ingeniería social, en forma de webs falsas, correos engañosos, o links falsos incrustados en una aplicación web, que inintencionadamente inducen al usuario a realizar alguna acción inesperada, que por lo general aprovecha la sesión establecida por el usuario en una aplicación o las cookies almacenadas. Normalmente los ataques CSRF son usados con diferentes propósitos:

- Secuestro de datos, obtenidas a partir de las cookies de sesión o mediante peticiones al servidor
- Manipulación de datos
- Propagación de virus de tipo gusano por redes sociales
- Manipulación de datos en línea como por ejemplo encuestas online
- Instalación de malware en dispositivos móviles



- Acciones maliciosas

Este tipo de ataques se pueden evitar de diversas maneras, la primera de ellas es mediante **tokens** de comunicación utilizados entre el cliente y el servidor de la aplicación web, estos tokens se suelen generar en el momento de iniciar sesión en una aplicación y se envían junto a cada petición realizada con el servidor, en casos extremos en los que es crítico evitar este tipo de ataques se suelen generar tokens para cada petición. Más adelante veremos que herramientas actualmente nos ayudan a implementar dicho mecanismo. Independientemente del nivel de seguridad que establezcamos en la política de tokens, esto no nos cubre en caso de que una aplicación cuente con alguna vulnerabilidad XSS.

Otra acción que ayuda a prevenir este tipo de ataques es enseñando a los usuarios a reconocer este tipo de ataques para que sepan identificar dado un correo, enlace, ... si es falso o no. Para más detalle de cómo protegernos frente a este tipo de ataques es recomendable la lectura del "[Cross-Site Request Forgery Prevention](#)" de OWASP.

#### 3.2.1.5.3 Vulnerabilidades del código fuente

El código fuente de las aplicaciones front-end puede contener agujeros de seguridad ocultos, ya sea en el código de la propia aplicación, o en alguna de las dependencias que utiliza, estos agujeros pueden ser aprovechados por un atacante y por lo tanto aumentan la superficie de riesgo de la aplicación. Una de las cosas a tener en cuenta a la hora de analizar estas vulnerabilidades es que JavaScript no es un lenguaje compilado, sino que es interpretado, es por ello que la ofuscación de código no previene este tipo de ataques ya que con el tiempo suficiente cualquiera puede hacer el código legible, igualmente la ofuscación del código es una buena práctica que puede ayudar a reducir este tipo de ataques.

Por lo tanto, una de las principales fuentes de este tipo de vulnerabilidades son los registros de paquetes como NPM, ya que su uso en las aplicaciones web es altamente extendido y cada vez se utiliza mayor número de librerías externas en las aplicaciones, con el fin de reducir los tiempos de desarrollo. Por un lado, puede ser interesante el uso de estas librerías, pero por otro lado hay que tener mucho cuidado y es por ello por lo que una de las primeras recomendaciones para esta clase de vulnerabilidades es utilizar alguna herramienta para auditar y escanear las librerías que integra una aplicación web en busca de vulnerabilidades, en la siguiente sección veremos que alternativas existen y como utilizarlas. Si se va a utilizar npm en una aplicación web es muy recomendable conocer la [Cheat Sheet](#) de buenas prácticas ofrecida por la OWASP, estas buenas prácticas han sido sacadas de un artículo publicado por el [blog de Synk](#), en general este listado es exclusivo para npm si no que es aplicable a otros gestores como bit, yarn, npm, ...

## 1. Avoid publishing secrets to the npm registry

- 1 Run `npm publish --dry-run` to review the package before publishing
- 2 Put sensitive files in `.gitignore`
- 3 Use the `files` property in `package.json` to whitelist files and directories

## 2. Enforce lockfile

Freeze lockfile and ensure the npm CLI installs per lockfile only, without changing it. In CI and build environments favor:

- 1 `$ npm ci`
- 2 `$ yarn install --frozen-lockfile`

## 3. Minimize attack surface—ignore run-scripts

Malicious packages take advantage of key lifecycle events when an npm install runs arbitrary commands.

To minimize this attack surface:

- 1 Assess a project's health status and credibility before installing a package
- 2 Disable run-scripts during install such as:

```
$ npm install <package> --ignore-scripts
```

## 4. Assess npm project health

Review a project for outdated dependencies, and assess environment health with CLI commands:

```
$ npm doctor
$ npm outdated
```

## 5. Scan and monitor for vulnerabilities source dependencies

Don't let vulnerabilities in your project dependencies compromise the security of your application. Make sure to:

- 1 Connect Snyk to GitHub or other SCMs for CI integration with your projects
- 2 Run `snyk test` to scan a new project from CI
- 3 Run `snyk monitor` to track and open PRs to fix security vulnerabilities in open source dependencies

## 6. Use a local npm proxy

A local private registry such as [Verdaccio](#) will give you more control of security, enabling you:

- 1 Full control of lightweight private packages
- 2 To cache packages and avoid being affected by network issues and external incidents

Easily spin up verdaccio using docker:

```
$ docker run verdaccio/verdaccio
```

## 7. Responsible disclosure

Publicly disclosed security vulnerabilities without proper coordination pose a potentially serious threat to the security of the npm community.

We are happy to collaborate on responsible security disclosures. Contact us at [security@snyk.io](mailto:security@snyk.io).

- 1 Report a security issue via the [vulnerability](#) page
- 2 Email us at [security@snyk.io](mailto:security@snyk.io)

FIGURA 4. CHEAT SHEET NPM. SOURCE: <https://snyk.io/blog/ten-npm-security-best-practices/>



## 8. Enable 2FA

Enable two-factor authentication on npm with:

```
$ npm profile enable-2fa auth-and-writes
```

## 9. Use npm author tokens

Make use of restricted tokens for querying npm packages and functionalities from CI by creating a read-only and IPv4 address range restricted token:

```
$ npm token create --read-only --cidr=192.0.2.0/24
```

## 10. Understand typosquatting risks

Typos in package installation can be deadly.

- 1 Be mindful when copy-pasting package install instructions to the terminal and verify authenticity.
- 2 Opt to have a logged-out npm user in your developer environment
- 3 Favor npm install with `--ignore-scripts`

Authors:

- [@iliran\\_tal](#)  
Node.js Security WG & Developer Advocate at Snyk
- [@jcradeveloper](#)  
Core maintainer at Verdaccio



Otra recomendación para prevenir la generación de agujeros de seguridad es el desarrollo de test unitarios, no con el único fin de verificar comportamientos, si no para verificar que el código desarrollado se ejecuta de manera segura. De igual manera que las herramientas para auditar paquetes, en la siguiente sección veremos cuales son las herramientas más populares para implementar estos test unitarios.

### 3.2.1.6 Buenas prácticas

Las medidas preventivas vistas anteriormente no suelen ser suficientes ya que o bien no cubren todas las vulnerabilidades o bien aparecen nuevas con el tiempo, es por ello por lo que a la hora de desarrollar una aplicación front-end con JavaScript se recomienda cumplir con el siguiente listado de buenas prácticas:

- **Evitar el uso de la función `eval()` con entradas del usuario**, el uso de esta función permite a un usuario cualquier a ejecutar código dañino en una aplicación.
- **Habilitar cifrado mediante TLS/SSL**, el cifrado de los datos transmitidos entre un cliente y un servidor ayuda a prevenir muchos de los ataques en las aplicaciones web, como, por ejemplo, los ataques CSRF.
- **Tokens de seguridad**, es muy importante que a la hora de establecer conexión con un API configuremos debidamente un token de seguridad para cada usuario, de esta manera aseguramos un acceso seguro a la misma.
- **Content Security Policy**, independientemente de si detectamos o no vulnerabilidades XSS, es altamente recomendado el uso de directivas CSP.
- **Utilizar herramientas de escaneo**, otra de las recomendaciones que en muchos casos nos van a ahorrar mucho tiempo, es el uso de herramientas de escaneo que utilicen una base de datos de vulnerabilidades bien actualizada, además se recomienda integrar este tipo de herramientas con el ciclo de vida de la aplicación para que este escaneo sea dinámico en cada nueva entrega de la aplicación.
- **Establecer cookies seguras**, de esta manera nos aseguramos de que las cookies solo pueden ser utilizadas por una única página web.
- **Utilizar métodos de manipulación del DOM seguros**, métodos como `innerHTML` son muy peligrosos ya que no hacen este escapado de caracteres especiales que hemos comentado previamente, por ello se recomienda utilizar métodos como `innerText` que si que realiza este escapado de la entrada.

## 3.3 Amenazas librerías / frameworks modernos

Para el siguiente análisis vamos a entrar en detalle de cuáles son las vulnerabilidades más comunes que hoy en día afectan a los frameworks modernos. Para ello vamos a tomar como referencia la página de [2021.stateofjs](https://2021.stateofjs.com/) en la que figuran los frameworks front-end más **utilizados** en el pasado **2021** elegidos por la comunidad. Todo lo que hemos visto anteriormente en la sección de “[Amenazas JavaScript](#)” es aplicable a cualquiera de las herramientas que vamos a ver a continuación, ya que todas ellas utilizan JavaScript como lenguaje. Antes de pasar a comentar más en concreto las vulnerabilidades de cada alternativa y sus posibles medidas preventivas, vamos a hacer un repaso de aquellas vulnerabilidades más comunes que por lo general afectan al desarrollo front-end utilizando tecnologías modernas.

### 3.3.1 Cross-Site Scripting (XSS)

Esta vulnerabilidad vista anteriormente es una de las más comunes en la parte front-end de las aplicaciones web. Ninguna de las alternativas puede implementar un mecanismo de seguridad contra este tipo de ataques debido a que su principal inconveniente es el origen de los datos, es inviable que el propio framework o librería determine cuando es fiable o no determinado contenido que se desea mostrar a través de un *innerHTML* o algo similar. Es por ello por lo que la regla principal que se establece desde las referencias oficiales de las diferentes alternativas es que se asegure que el HTML, CSS, o JS a ejecutar en este tipo de funciones se encuentre desinfectado y que por lo tanto está libre de peligros.

### 3.3.2 Broken Access Control

Los problemas derivados de una implementación pobre o ineficiente de los mecanismos de autenticación son muy comunes entre las diferentes alternativas. Por lo tanto, independientemente de con que esté desarrollada una aplicación web es posible que se presenten amenazas como las vistas anteriormente en la [A01:2021-Broken Access Control](#) del Top 10 de OWASP.

Las recomendaciones a la hora de proteger una aplicación frente a este tipo de vulnerabilidades son las siguientes:

- Una de las primeras recomendaciones es asegurarse de que la conexión entre el cliente y el servidor es segura, y utiliza algún protocolo de cifrado para la comunicación.
- Establecer **validaciones** en los formularios de registro con el fin de que se cumplan las **políticas de seguridad de contraseñas seguras**.
- **Reducir lo máximo posible la exposición de identificadores de sesión.**
- Gestionar correctamente las sesiones de los usuarios, **asegurando que son invalidadas cuando ya no se estén utilizando**. Establecer timeouts de sesión.
- Utilizar **cifrado en el envío de credenciales e identificadores de sesión**.
- **Establecer mecanismos de autenticación multi factor.**



FIGURA 5. MULTI-FACTOR. FUENTE: [HTTPS://WWW.CYBER.GOV.AU](https://www.cyber.gov.au)

- **Utilizar un gestor de sesiones alojado en el lado del servidor**, no se debe proporcionar identificadores de sesión directamente desde la parte front-end ya que esto puede comprometer su seguridad.

### 3.3.3 SQL Injection

La inyección SQL también está presente en aplicaciones desarrolladas con alguna de estas tecnologías modernas, los indicios y las posibles medidas de seguridad contra esta vulnerabilidad

son las vistas anteriormente en la [A03:2021-Injection](#) del Top 10 de OWASP. Las mejores prácticas que se recomiendan ya enfocadas en esta sección serían las siguientes:

- Aplicar el **principio de menor privilegio**. Esto se consigue estableciendo una gestión de permisos y roles segura y eficiente, estableciendo las denominadas *White list* y asegurando que por defecto no se tiene permiso para nada, más adelante veremos cómo podemos implementar esta gestión en cada una de las alternativas.
- Al igual que para la vulnerabilidad de XSS una buena práctica contra la inyección SQL es el **filtrado y desinfección de la entrada del usuario** por los diferentes puntos de entrada de una aplicación web, como por ejemplo los formularios.
- Otra buena práctica que ayuda no tanto a prevenir, pero si a detectar este tipo de ataques es **monitorizar** la aplicación web tanto del lado del cliente como del lado servidor, haciendo especial foco en esta sección en las integraciones con terceros.

### 3.3.4 XML External Entity Attack (XXE)

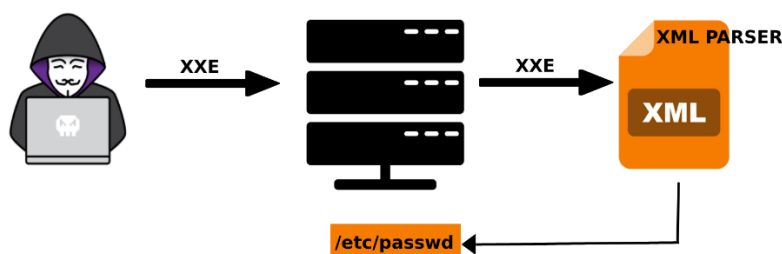


FIGURA 6. XXE. FUENTE: [HTTPS://SECURITYSOULS.COM/WHAT-ARE-XXEXML-EXTERNAL-ENTITY-ATTACKS/](https://securitysouls.com/what-are-xxexml-external-entity-attacks/)

Los ataques XXE son un tipo de inyección de código malicioso en los que el objetivo son los parsers XML utilizados por la aplicación, a través de estos ataques se extrae información sensible o desencadenan otros ataques como pueden ser *CSRF* o ataques *DDoS*. Para prevenir este tipo de ataques lo mejor es limitar las capacidades de estos parsers, teniéndolos siempre actualizados.

### 3.3.5 Vulnerabilidades del código fuente

Debido a que independientemente del framework / librería que utilicemos para desarrollar la parte front-end de una aplicación web finalmente el código resultante no es más que una aplicación JavaScript existe la posibilidad de encontrar vulnerabilidades en el código fuente ya sea desarrollado por nosotros o proveniente de una librería externa.

Todas las alternativas que vamos a ver utilizan el gestor de paquetes **npm** para la descarga de dependencias, este gestor por defecto realiza escaneos de seguridad proporcionando un reporte de vulnerabilidades cada vez que se realiza alguna operación, o podemos lanzar dicho proceso manualmente utilizando *npm audit*, por lo general las vulnerabilidades encontradas pueden ser corregidas por npm, pero no siempre es posible.

### 3.3.6 Buenas prácticas

- Realizar **escaneos de seguridad de las dependencias** esto se puede realizar con una herramienta externa o bien con *npm* más adelante veremos cómo realizar estos escaneos y como interpretarlos.

- Implementar siempre que sea posible **interceptores de las peticiones para la validación de URLs**, de esta manera con una *White list* podemos tener contemplado el conjunto de URLs que están permitidas en la aplicación.
- **Establecer cabeceras CSP**, actualmente todas las alternativas nombradas cuentan con mecanismos suficientes para implementar este tipo de cabeceras.
- **Leer la documentación oficial** del framework o librería a utilizar, en la mayoría de los casos esta cuenta con un apartado de seguridad que cubre aquellas amenazas que son específicas de cada herramienta.

### 3.3.7 Amenazas React

React es una librería desarrollada por Facebook y es de las más populares hoy en día y es de las que más tiempo se ha mantenido siendo de la más utilizadas, su comunidad es bastante extensa y es por ello por lo que nos puede llevar a pensar que es la más segura y que está libre de las vulnerabilidades más comunes, pero no es así.


Otro de los puntos que da a pensar que React pueda ser más segura que alguna otra librería o framework es su tamaño, ya que cuanto menos código contenga menor será la superficie de ataque, todo esto es cierto pero un detalle que no se suele tener en cuenta es el motivo por el cual la librería de React es tan pequeña, y es que React basa su flexibilidad en la integración de otros componentes de código abierto, por lo que la cantidad de código proveniente de otras fuentes es mayor que la de cualquier otra librería o framework, esto por lo tanto aumenta la superficie de ataque ya que los componentes que integre pueden contener agujeros de seguridad escondidos. Por lo que una de las primeras recomendaciones a la hora de utilizar React en el desarrollo de nuestros proyectos es **ser más selectivos en la elección de componentes** a integrar dentro de la aplicación.

A continuación, vamos a ver cuáles de las vulnerabilidades vistas anteriormente afectan en mayor proporción a la librería de React y que acciones se recomienda tomar para evitarlas.

#### 3.3.7.1 Cross-Site Scripting (XSS)

Dentro de la [referencia](#) de React encontramos unos cuantos consejos para prevenir este tipo de ataques. Buenas prácticas para prevenir ataques XSS en aplicaciones React:

- **Deshabilitar las etiquetas HTML que permitan la ejecución de código**, como lo son `<object>`, `<script>`, `<link>` y `<embed>`.
- Utilizar por defecto `{}` cuando se quiera pintar el contenido de una variable en el HTML, ya sea un string o un bloque HTML, ya que en React gracias a la tecnología de *JavaScript XML* (JSX) podemos escribir HTML en un componente React. El principal motivo de **utilizar `{}` para el bindeo de datos** es que React automáticamente va a escapar los valores introducidos.



```

1  const SecureComponent = () => {
2    return (
3      <div className="secure-component">
4        <h1>Secure component</h1>
5        <p> { inputData } </p>
6      </div>
7    );
8  }
9
10 export default SecureComponent;

```

- Utilizar la propiedad *dangerouslySetInnerHTML* con conocimiento, por defecto el nombre es disuasorio ya que nos avisa de que es peligrosa, es por ello por lo que si la vamos a utilizar nos tenemos que asegurar que el contenido que le pasemos sea fiable y estemos seguros de que se ha aplicado alguna de las técnicas de desinfección a dicho contenido. Esta propiedad viene a realizar la tarea que desempeña *innerHTML*. El uso de librerías como [DOMPurify](#) facilita dicha tarea realizando escaneos de las entradas de los usuarios eliminando aquel contenido que pueda ser malicioso.
- Utilizar *React.createElement()* ya que con esta función podemos crear elementos HTML de una forma segura.
- Utilizar una **librería de desinfección** que se encargue de aplicar el escapado de los datos a mostrar en el DOM.

### 3.3.7.2 Zip Slip

Esta vulnerabilidad es específica de las aplicaciones **React**, Zip Slip es una funcionalidad de la librería que permite a un usuario subir archivos comprimidos con el propósito de reducir el tamaño de los ficheros que se desean subir a la aplicación. Este módulo de la librería lo que hace es utilizar un directorio dedicado a estos archivos que finalmente puede ser explotado por un atacante permitiendo la manipulación de información sensible o incluso llegar a afectar a un servidor que conecte con el cliente.

Si se desea hacer uso de esta funcionalidad se recomienda tener especial cuidado con el nombre que se establece a los archivos que se desean subir. Para asegurar que el nombre no es un problema se recomienda utilizar una política de nombrado de los ficheros que se suben. Realizando un escapado de caracteres especiales, utilizando expresiones regulares, etc. O bien renombrar todos los ficheros que se desean subir con el propósito de establecer siempre nombres seguros.

### 3.3.7.3 Server Side Rendering (SSR)

Por lo general cuando implementamos SSR en una aplicación debemos tener cuidado en muchos aspectos, y en las aplicaciones de React debemos tener especialmente cuidado si estamos utilizando la librería de Redux. En caso de que estemos utilizando Redux se recomienda que los estados que queramos almacenar lo hagamos directamente en la parte del servidor ya que si

establecemos el estado en el cliente estamos expuestos a que nos inyecten código malicioso en el estado.

Otra recomendación cuando se utiliza SSR es el uso de librerías que se encarguen de la desinfección del contenido a renderizar por el navegador, de esta manera nos aseguramos de que en caso de haber código malicioso no va a ser ejecutado.

### 3.3.8 Amenazas Angular

Angular ha tenido mucha popularidad a lo largo de su existencia, este framework desarrollado por Google se volvió muy popular en su versión 2.0 ya que presentaba una forma de trabajar no vista hasta el momento y que pese a su compleja curva de aprendizaje hoy en día Angular es uno de los frameworks **más utilizados**. Aunque no sea la opción más escogida hoy en día por la comunidad, no podemos negar que es de las que más se sigue utilizando tanto para nuevos proyectos como proyectos ya existentes, es por ello por lo que es importante conocer que vulnerabilidades son la más comunes y que podemos hacer para prevenirlas utilizando el propio framework. A diferencia de React, Angular **posee un conjunto de herramientas bastante extenso** que te permite hacer casi cualquier cosa que requiera una aplicación web cualquiera, de primeras es un **framework muy potente** pero que **carece de flexibilidad** en muchos aspectos ya q. En temas de seguridad el hecho de que Angular te proporcione tantas herramientas de seguridad es en cierta manera bueno porque puedes **asegurarte de que las herramientas que utilizas provienen de una fuente conocida y abalada por una gran comunidad y por Google**.

La propia página de Angular nos proporciona una sección dedicada a la seguridad en la que ofrecen una pequeña [guía](#) de todo lo que tienes que saber para realizar un desarrollo seguro utilizando este framework, se recomienda su lectura ya que no solo se centra en Angular y siempre es de ayuda consultar información de diferentes proveedores.

#### 3.3.8.1 Cross-Site Scripting (XSS)

- De igual manera que para React el principal mecanismo de seguridad de Angular frente a este tipo de ataques es utilizar las **{{}}** de **interpolación de datos** ya que por defecto escapa la entrada introducida previniendo de posibles scripts maliciosos.



- Otra forma de prevenir estos ataques es utilizando alguna **librería de componentes**, de manera que los componentes ya estén desarrollados de manera segura y todas las medidas preventivas sean delegadas a la librería.

- **Utilizar con precaución la propiedad [innerHTML]** en este caso React renombró dicha propiedad para avisar de que era peligrosa, pero Angular no, simplemente es nombrada como lo que finalmente hace. Por lo tanto, es recomendable seguir aquellos consejos vistos anteriormente para *innerHTML* realizando una desinfección del contenido pasado a dicha propiedad.
- Evitar la concatenación de datos de entrada del usuario directamente sobre el *template* de los componentes. A la hora de definir el template de cada componente existen dos maneras de hacerlo, una es sacando el template a un fichero HTML, y la otra es poniendo el contenido HTML directamente sobre el atributo *template* del componente, es esta última forma la que puede causarnos problemas si realizamos una concatenación directa de datos de entrada no validados previamente.



- Cuando se desea realizar **Server Side Rendering (SSR)** evitar completamente el uso de **motores de plantillas** como *Handlebars*, *Pug*, etc. Esto se debe a que los datos inyectados provienen de un agente externo y por lo tanto está fuera del alcance del API de Angular, por lo que no tenemos la certeza de que Angular los haya tratado para que sean seguros.
- **Utilizar el compilador de plantillas AOT**, este compilador es el que trae por defecto Angular y nos ayuda a prevenir muchas vulnerabilidades, entre ellas XSS.

### 3.3.9 Amenazas Vue

Vue es un framework creado por *Evan You* en 2014 que desde entonces no ha parado de crecer, su presencia entre las diferentes alternativas a la hora de desarrollar una aplicación front-end está más que justificada ya que es un framework y no una librería, y que por lo tanto te ofrece casi todo lo necesario para desarrollar una aplicación, finalmente otra característica que lo destaca es que su curva de aprendizaje es bastante menor que la de Angular.



En temas de seguridad nos encontramos con el mismo conjunto de vulnerabilidades comunes que puede tener una aplicación de Angular o React, se recomienda la lectura de la [referencia](#) de la librería, en ella encontramos un conjunto de buenas prácticas que nos ayuda a empezar a desarrollar de manera segura utilizando Vue.

### 3.3.9.1 Cross-Site Scripting (XSS)

De igual manera, encontramos que para Vue la vulnerabilidad más común es la de XSS, y las recomendaciones para prevenir este tipo de ataques son similares que al resto de alternativas:

- No realizar concatenaciones en el template de los componentes.
- Utilizar el data binding proporcionado por Vue que escapa directamente los datos a renderizar `<h1>{{ userInput }}</h1>`
- Utilizar el atributo `v-bind` que también escapa directamente el contenido suministrado.
- Si se utilizan los diferentes mecanismos de inyección HTML asegurarse que el contenido suministrado se encuentra escapado y libre de caracteres especiales. Los diferentes mecanismos que por debajo utilizan *innerHTML* son:
  - **v-html** en los templates.
  - **domProps: { innerHtml: ... }**, en las funciones de render.
  - **domPropsInnerHTML**, en las funciones de render utilizando JSX.
- Realizar un filtrado o escapado de las URLs suministradas a través de `v-bind:href`.

### 3.3.9.2 Server Side Rendering (SSR)

Vue al igual que React permite hacer SSR y por lo tanto corre los mismos peligros de inyección vistos anteriormente, Vue, establece un conjunto de buenas prácticas a la hora de implementar SSR en una aplicación de Vue, su recomienda su lectura en la [referencia oficial](#).

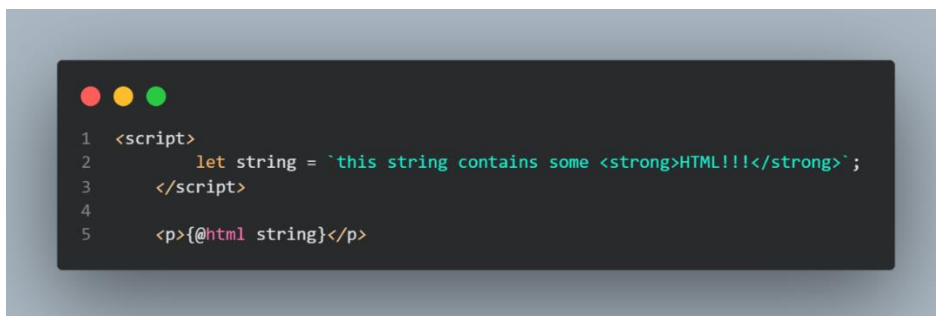
## 3.3.10 Amenazas Svelte

Este nuevo framework que ha aparecido recientemente se diferencia del resto en que el código de la aplicación no es interpretado en tiempo de ejecución por el navegador, sino que es convertido a JavaScript en tiempo de compilación, esto permite que el famoso virtual DOM no sea necesario, y que el navegador no tenga que ejecutar ni descargar nada específico de la librería de Svelte, a diferencia de React por ejemplo que requiere descargar su biblioteca para poder trabajar con el virtual DOM. Todo esto hace que las aplicaciones desarrolladas con Svelte sean más pequeñas y rápidas, esto nos hace pensar si esta nueva forma de funcionar puede aumentar la seguridad de una aplicación, y en cierta manera si, ya que el simple hecho de reducir la superficie expuesta hace que se reduzcan las posibilidades de un posible atacante. Svelte también tiene la posibilidad de realizar SSR por lo que también se recomienda tener especial cuidado en este aspecto. En la guía oficial de Svelte no encontramos una sección dedicada a la seguridad, pero sí que va indicando aquellos lugares donde debemos tener cuidado.

### 3.3.10.1 Cross-Site Scripting (XSS)

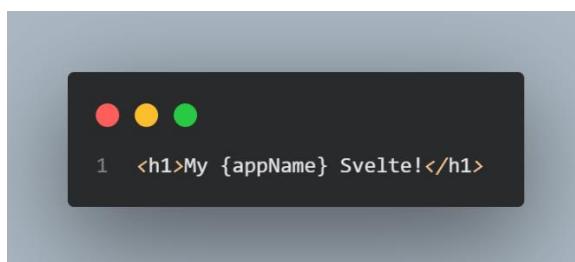
Svelte de igual manera que el resto de las alternativas cuenta con un mecanismo para inyectar HTML en el template de un componente, desde mi punto de vista es el que lo hace de la manera menos segura ya que la etiqueta que utiliza no es del todo descriptiva de lo que hace por debajo, la etiqueta a utilizar es **@html**, por lo tanto, si no utilizamos con cuidado esta etiqueta podemos sufrir ataques XSS si no desinfectamos el contenido suministrado.





```
1 <script>
2   let string = `this string contains some <strong>HTML!!!</strong>`;
3 </script>
4
5 <p>{@html string}</p>
```

Por otro lado, Svelte también dispone de un mecanismo para el *data binding*, que nos asegura que el contenido suministrado es escapado y que está fuera de peligro.



```
1 <h1>My {appName} Svelte!</h1>
```

### 3.3.11 Amenazas React Native

Antes de comentar las amenazas más importantes que afectan a React Native, es importante saber que todas las amenazas vistas anteriormente asociadas a React también afectan directamente a las aplicaciones desarrolladas con React Native, por lo que uno de los principales inconvenientes que introduce este tipo de tecnologías es que **por defecto** no va a ser la opción más segura a la hora de desarrollar aplicaciones móviles, ya que aparte de los posibles problemas de seguridad que se tengan en las propias apps móviles, React Native, introduce las posibles amenazas del lenguaje, y del propio framework, pero todo esto no evita que se pueda llegar a desarrollar aplicaciones móviles seguras utilizando React Native.

Por lo general, las aplicaciones móviles contienen información bastante importante de los usuarios, debido a esto, la seguridad en este tipo de aplicaciones sea un aspecto de alta importancia, uno de los principales fallos que se cometen a la hora de desarrollar este tipo de aplicaciones por desarrolladores web, es la falta de información acerca de la tecnología que rodea a los dispositivos móviles, y es que no es lo mismo dotar de seguridad una aplicación web, que una aplicación móvil, es por ello por lo que la primera recomendación que se suele hacer es, que el desarrollador adquiera los conocimientos necesarios de estas tecnologías para llegar a entender cuáles son las nuevas amenazas a las que se enfrentan, y tenerlas presentes a la hora de desarrollar.

A continuación, vamos a presentar aquellas amenazas que afectan más a la parte del framework y no tanto a las amenazas más comunes de las aplicaciones Android o IOS, para obtener mayor información acerca de los diferentes problemas de seguridad se recomienda leer la [referencia de Seguridad de React Native](#), la [referencia de seguridad de Android](#), y la [referencia de seguridad de IOS](#).

### 3.3.11.1 Broken Access Control

En la mayoría de las aplicaciones móviles necesitan almacenar información sensible como son los *secrets*, o parámetros de autenticación, que son necesarios a la hora de integrar librerías de terceros como son Firebase, Google, o Facebook, para llevar a cabo parte de la funcionalidad de la aplicación, o simplemente necesitamos un almacenar un *secret* para poder utilizar un API. Asegurar que esta información no sea comprometida es una de las primeras medidas de seguridad que hay que llevar a cabo en este tipo de aplicaciones, ya que si un atacante se hace con ellas podría realizar operaciones sobre las que no tiene autoridad pudiendo desencadenar múltiples problemas de seguridad.

Para protegernos ante esta amenaza se recomienda el uso de la siguiente librería [react-native-dotenv](#), esta librería nos permite declarar variables de entorno para ser utilizadas a lo largo de la aplicación, de esta manera no quedarán expuestas en texto plano en el compilado de la aplicación. Aunque esta medida ya nos da cierta seguridad, la manera ideal de proteger este tipo de información es utilizando funciones *serverless* como lo son [AWS Lambda](#) o [Google Cloud Functions](#).

### 3.3.11.2 Persistencia de datos

En ocasiones es necesario poder almacenar datos en el propio dispositivo en el que se está ejecutándose una aplicación con el objetivo de persistir información utilidad, por ejemplo, la sesión del usuario para no tener que autenticar al usuario cada vez que inicia la aplicación. Para dichos propósitos React Native ofrece una herramienta llamada *Async Storage* que dispone un almacén asíncrono al estilo clave-valor para poder persistir datos en el dispositivo móvil, el principal problema de esta opción es que los datos se almacenan sin encriptar por lo que no es una buena opción cuando deseamos almacenar datos sensibles como lo son tokens de sesión de usuario.

Una posible solución a este problema sería utilizar los servicios nativos de [Keychain Services](#) de iOS y [Android Keystore](#) de Android que ofrecen una manera de persistir datos en el dispositivo móvil de forma segura utilizando algoritmos criptográficos, para poder utilizar estos servicios podemos utilizar alguna de las siguientes librería que lo implementan directamente ([react-native-encrypted-storage](#), [react-native-keychain](#), [redux-persist-sensitive-storage](#)) o desarrollar un [bridge](#) para poder utilizar dichos servicios.

### 3.3.11.3 Deep Linking

[Deep Linking](#) es el mecanismo que tienen las aplicaciones móviles de recibir datos provenientes de fuentes externas mediante el uso de links, estos son detectados por la aplicación y tienen la siguiente forma, *app://* donde *app* hace referencia al identificador de la aplicación, y lo que precede a *//* es el enlace interno que la aplicación detectará para operar en función del enlace recibido.

Este mecanismo se suele utilizar para acceder externamente a las diferentes partes de una aplicación, para lanzar alguna operación desde fuera de la misma, o para acceder directamente al detalle de un objeto de la aplicación, por ejemplo, *app://cars/1*, por lo general este dato que aparece en el enlace no es peligroso de exponer, pero, por ejemplo, si en vez de un identificador queremos recibir un *token*, ya tenemos un problema, ya que una aplicación maliciosa podría secuestrar los datos presentes en los enlaces a través de la declaración del mismo esquema de *deeplinks*, esto

se debe a que no hay un sistema centralizado para el registro de *deeplinks*, de manera que cada *deeplink* solo pueda pertenecer a una única aplicación.

Este problema de seguridad no es específico de React Native sino que también lo tienen las apps desarrolladas nativamente, en el caso de Android para añadir una barrera de seguridad lo que hace es mostrar un dialogo indicando que aplicación es de la que proviene el *deeplink*, por otro lado, IOs en su versión 9 introduce lo que se denominan [universal links](#) para solventarlo.

### 3.3.12 Amenazas Electron

Electron, al utilizar HTML, CSS, y JavaScript para el desarrollo de aplicaciones *desktop*, hereda todo el conjunto de vulnerabilidades vistas previamente para dichas tecnologías, pero hay que tener una cosa clara, y es que este framework no sirve simplemente para hacer una especie de aplicación web ejecutada por un sistema operativo, sino que aparte de las capacidades que tiene una aplicación web, este framework permite explotar las capacidades de una aplicación de *desktop*, y es por ello que la superficie de riesgo es mayor ante posibles vulnerabilidades.

Las vulnerabilidades más comunes en las aplicaciones *desktop* desarrolladas utilizando el framework de Electron son, *Cross-Site Scripting (XSS)* y las vulnerabilidades en el código fuente, igualmente, estas aplicaciones poseen diferentes amenazas asociadas a la propia tecnología y al mundo de las aplicaciones *desktop*, es por ello que en la referencia oficial del framework se listan un conjunto de [buenas prácticas](#) que se recomienda tener en cuenta a la hora de empezar a desarrollar utilizando Electron.

#### 3.3.12.1 Cross-Site Scripting (XSS)

Una de las vulnerabilidades más importantes en este tipo de aplicaciones son los ataques XSS, ya que pueden llegar a tener un alto impacto en la seguridad tanto sobre la propia aplicación, como sobre sistema operativo en el que se está ejecutando, esto se debe principalmente a que al tener la posibilidad de utilizar parte de la funcionalidad del sistema operativo, también lo estamos exponiendo al peligro, por lo que se recomienda que una de las primeras acciones a tomar sea llevar a cabo todas las recomendaciones vistas anteriormente para poder prevenir este tipo de ataques.

#### 3.3.12.2 Vulnerabilidades del código fuente

En estas aplicaciones que te permiten desarrollar apps de escritorio utilizando otro tipo de tecnologías como en este caso, tecnologías web, es de suma importancia tener constantemente actualizadas las versiones de las dependencias, pero sobre todo la versión del propio framework, ya que este será el principal encargado de ir parcheando aquellos agujeros de seguridad relacionados con la tecnología final que abstrae el propio framework.

# 4 SERVICIOS Y MECANISMOS

## 4.1 Servicios

Todos los servicios de los que vamos a hablar a continuación se refieren a servicios en términos de seguridad Informática, más concretamente a los servicios que se aplica en la parte front-end de las aplicaciones web, estos servicios se pueden categorizar en:

- Autenticación
- Autorización
- Integridad de datos
- Confidencialidad
- Disponibilidad

### 4.1.1 Autenticación

El servicio de **autenticación** nos ofrece una manera de identificar **usuarios, entidades, o dominios** dentro de una aplicación web y **garantizar** que son quienes dicen ser, por lo tanto, este servicio nos garantiza por ejemplo que a la hora de que un usuario inicie sesión en una aplicación podamos estar **seguros** de que ese usuario es quien dice ser y no un posible atacante.

Hoy en día casi cualquier aplicación web cuenta con mecanismos de autenticación, es algo bastante común que en el desarrollo de una aplicación se tenga que desarrollar una pantalla de registro y de login, con los respectivos mecanismos de autenticación que serán los que nos ayuden a ofrecer este servicio.

Existen diversos tipos de autenticación, pero en nuestro caso el que más nos interesa es el de **autenticación de usuarios**, y por lo general este proceso de autenticación de un usuario se va a centrar en **algo que sabe, algo que tiene, o algo que es**. En la mayoría de los casos esta autenticación se realiza utilizando un nombre de usuario o un identificador y algún otro elemento que solo pueda provenir de dicho usuario.

Otro concepto importante dentro de las aplicaciones web es el de **sesión de usuario** estas sesiones son mantenidas en la parte del servidor y disponen de un identificador que las identifica inequívocamente y nos va a facilitar la autenticación frente al servidor en cada interacción realizada con él. En los mecanismos veremos que, sí que hay que interactuar con las sesiones de usuario en la parte front-end, pero como tal la complejidad se encuentra en el servidor. La OWASP dispone un documento de [buenas prácticas](#) para la gestión de sesiones que es muy recomendable leer.

### 4.1.2 Autorización

La **autorización** es el servicio por el cual se le **autoriza** a un usuario dentro de una aplicación web a realizar alguna **acción**, entendiendo por acción el acceder a una sección de la aplicación, acceder a un recurso concreto, realizar una operación, etc. Este término a veces se asocia incorrectamente al proceso de realizar login dentro de una aplicación web pero dicho proceso pertenece exclusivamente al servicio de autenticación, suele ser justo después de ese proceso en

el que se empieza a producir el servicio de autorización, pero cierto es que no siempre es así, ya que a un usuario no autenticado se le puede autorizar a realizar algún tipo de acción, por ejemplo, registrarse en la aplicación.

Otro punto importante es que a partir de la violación de este servicio es cuando se determina que una aplicación es vulnerable a la vulnerabilidad anteriormente vista en el Top 10 *"Broken Access Control"*, y los ataques producidos contra esta vulnerabilidad ponen en peligro el cumplimiento del resto de servicios (confidencialidad, integridad, disponibilidad) al permitir escalar privilegios pudiendo realizar operaciones que no se le permitían en un principio, es por este motivo por el que es tan crítico el servicio de autorización dentro de una aplicación web.

Cada vez es más frecuente establecer mecanismos de autorización en las aplicaciones, y por lo general las aplicaciones de mayor tamaño tienden a necesitarlos más con el objetivo de separar la funcionalidad en roles, de manera que cada usuario disponga de uno o más roles y que a través de estos roles se determine que puede hacer dentro de la aplicación. Más adelante veremos un conjunto de recomendaciones establecidas por la OWASP y cuáles son los mecanismos de autenticación podemos implementar en una aplicación web para cumplir con el servicio de autorización.

### 4.1.3 Integridad

La **integridad** en una aplicación web es el servicio que nos **garantiza** que la **información** que viaja por la aplicación **no sufre ninguna alteración**, por lo tanto, este servicio nos protege ante cualquier modificación, inserción destrucción, retransmisión, reordenamiento, y duplicación de la información.

Como tal en el desarrollo front-end de una aplicación no se suelen implementar mecanismos que aseguren la integridad de la información que fluye, pero sí que hay un punto en el desarrollo front-end en el que sí que es importante que se asegure la integridad, y es, **a la hora de descargar dependencias** del proyecto, es muy importante que estemos seguros que las dependencias descargadas **no han sido manipuladas**, para ello se suelen utilizar gestores de dependencias como [\*npm\*](#), que se encargan automáticamente por debajo de realizar estas comprobaciones de dependencia.

### 4.1.4 Confidencialidad

El servicio de **confidencialidad** nos **garantiza** que no han revelado datos sin previa autorización. En el entorno de las aplicaciones web es muy importante prevenir que los datos sensibles que están en movimiento o guardados sean revelados, el principal mecanismo que se establece para asegurar la confidencialidad es el cifrado de los datos sensibles, de esta manera nos aseguramos de que si un atacante los intercepta no pueda interpretarlos.

En la parte front-end de las aplicaciones web una pieza muy importante a implementar para asegurar la confidencialidad de los datos es el sistema de roles y permisos, estableciendo un conjunto de acciones y accesos restringidos a través de una *White list* de roles – permisos.

### 4.1.5 Disponibilidad

El servicio de **disponibilidad** nos **garantiza** que, dado un sistema, este siempre está **disponible cuando se lo requiere**, es un poco confuso pero dicho de otra manera este servicio lo

que hace es protegernos frente a posibles ataques que puedan inhabilitar una aplicación web dejando esta de estar disponible para su acceso y por lo tanto para su uso.

Una de las maneras de cumplir al máximo con este servicio es estableciendo un mecanismo de **monitorización** con el objetivo de llevar un seguimiento del correcto funcionamiento de una aplicación. Hasta hace bien poco esta práctica únicamente se realizaba en la parte servidor de una aplicación, pero actualmente la monitorización de aplicaciones front-end está ganando bastante importancia y cada vez se implementa en más proyectos. En la siguiente sección veremos que alternativas tenemos para implementar dicho mecanismo en el desarrollo front-end de una aplicación.

Otro de los factores que influyen bastante en la disponibilidad de una aplicación es el **desarrollo de tests**, más concretamente los test e2e o de integración, que a través de ellos podemos asegurar que todo funciona correctamente y que no hay ningún caso de uso que hace que una aplicación deje de estar disponible.

Finalmente, otra pieza importante que suele **prevenir de errores humanos** que afectan contra la disponibilidad, es el **desarrollo CI/CD** de una aplicación web que automatice todo el proceso de integración y despliegue, así como asegurar en cada nueva versión de la aplicación que los tests se ejecutan correctamente y que no se detecta ninguna anomalía respecto a anteriores versiones. Según vayamos avanzando iremos viendo la relación entre las aplicaciones web, la seguridad, las operaciones, y la infraestructura que terminaremos por nombrar “**DevSecOps**” a la que se dedicará una sección para hablar de este enfoque en el diseño de las aplicaciones web.

## 4.2 Mecanismos

Los mecanismos y consejos que vamos a ver a continuación están enfocados en la parte front-end de las aplicaciones web. La OWASP tiene un [documento](#) dedicado a la autenticación que se recomienda leer, en el encontraremos los mecanismos que vamos a ir viendo y un conjunto de buenas prácticas bastante interesantes, pero además no solo se centra en la parte front-end, sino también en la de back-end.

### 4.2.1 Autenticación

#### 4.2.1.1 Usuario y contraseña

El primer y más antiguo de los mecanismos es la autenticación a través de usuario y contraseña. Este mecanismo lo forman dos partes, el registro y el login. En el registro el usuario establece unas credenciales normalmente formadas por un nombre de usuario y una contraseña, estas se registran en el servidor que lo autenticará de ahora en adelante, y la segunda parte del mecanismo, el login, en el que el usuario va a proporcionar dichas credenciales para ser autenticado. El nombre de usuario suele ser un identificador único como por ejemplo el email o el DNI, y la contraseña por lo general tiene que cumplir un conjunto de requisitos que aseguran que sea segura.

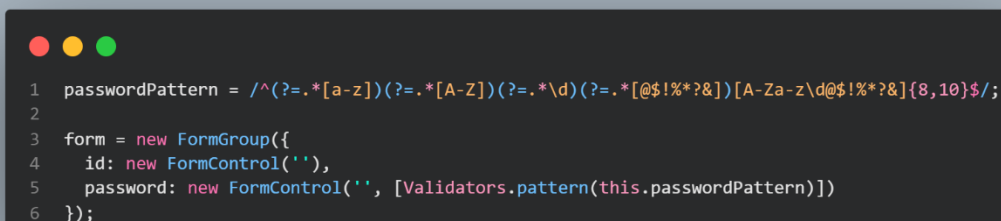
##### 4.2.1.1.1 Validaciones de contraseñas

Es importante que la página de registro desarrollada en una aplicación realice validaciones sobre las contraseñas introducidas, para evitar que un usuario establezca una contraseña demasiado débil y que un atacante pueda obtenerla sin mucho esfuerzo, normalmente las

validaciones establecidas son obligatorias, de manera que nadie pueda saltárselas y crear una cuenta sin pasar dichas validaciones. Por lo tanto, para asegurarnos de que una contraseña sea segura tenemos que validar que cumpla las siguientes características:

- Longitud de al menos 8 caracteres para que no sea muy memorizable.
- Longitud de 64 caracteres como máximo, por lo general se establece este límite con el objetivo de prevenir ataques de denegación de servicios debido a la transformación de las contraseñas utilizando algoritmos *hash*.
- Contiene al menos una letra minúscula y otra mayúscula.
- Contiene al menos un número.
- Contiene caracteres especiales, es muy común que se requiera que la contraseña contenga al menos 1 carácter especial, por ejemplo, ! @ # ? ].

Lo más fácil para implementar este tipo de validaciones en una aplicación es utilizar expresiones regulares, de manera que verifiquemos que la entrada del usuario cumple el patrón establecido para las contraseñas. Ejemplo en Angular:



```
1 passwordPattern = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,10}$/;
2
3 form = new FormGroup({
4   id: new FormControl(''),
5   password: new FormControl('', [Validators.pattern(this.passwordPattern)])
6 });
```

Además de validar que una contraseña cuente con ciertas características, se recomienda el uso de librerías que consulten bases de datos de contraseñas vulnerables, para que las contraseñas establecidas no sean las más comunes del momento en el que se crean.

#### 4.2.1.1.2 Recuperación de contraseñas

Otro factor bastante importante en la seguridad de las contraseñas es asegurar que si una aplicación cuenta con un mecanismo de recuperación de contraseñas este sea seguro y que no permita que un atacante extraiga contraseñas de usuarios a través de este. Para ello siguiendo la [Cheat Sheet de OWASP](#) vemos que es muy importante que este procedimiento siga las siguientes buenas prácticas:

- Retorno de mensajes claros y concisos, tanto si la cuenta existe como si no.
- Hay que asegurar que los tiempos de respuesta de los usuarios son uniformes.
- Utilizar un canal complementario para comunicar el método de reseteo.
- Utilizar [URL tokens](#) para acceder al procedimiento, esto es, que para poder acceder al servicio de recuperación de la contraseña se envíe por otro canal una URL que sirva de token para acceder.
- Hay que asegurar que los *tokens* utilizados cumplen con los estándares de seguridad.



#### 4.2.1.1.3 Política de cambio de contraseñas

Otra de las cosas importantes a la hora de proteger las contraseñas de los usuarios es establecer una política de cambio de contraseñas, de esta manera el periodo de exposición de las contraseñas es menor y por lo tanto nos aseguramos de que las contraseñas no puedan ser descifradas cuando el tiempo que supone descifrarlas es mayor que el de la política de cambio de contraseñas. En este punto desde la parte frontend se tendrá que desarrollar una pantalla que permita al usuario realizar dicha acción cada vez que se le requiera cambiar la contraseña.

#### 4.2.1.1.4 Protección contra ataques automáticos

Uno de los ataques más comunes para extraer contraseñas son los ataques automáticos y pueden ser de tres tipos diferentes:

- **Fuerza bruta:** se trata de probar contra una misma cuenta un gran conjunto de contraseñas almacenadas en un diccionario o en alguna estructura de datos.
- **Relleno de credenciales ([Credential Stuffing](#)):** se trata de probar múltiples combinaciones usuario/contraseña que han sido expuestas en otro sistema.
- **Pulverización de contraseñas ([Password Spraying](#)):** se trata de probar contraseñas débiles contra un gran conjunto de cuentas conocidas.

Los mecanismos de seguridad que se recomiendan implementar para prevenir este tipo de ataques son los siguientes:

- **Autenticación multi-factor:** mecanismo que veremos a continuación
- **Bloqueo de cuenta:** es el mecanismo más común de prevención, y consiste en determinar un número máximo de reintentos en un periodo de tiempo establecido, de manera que, si se realizan, por ejemplo, 5 intentos fallidos en menos de 5 minutos, se notifica al propietario de la cuenta lo sucedido y se procede a bloquear su cuenta durante un tiempo establecido o hasta que un administrador desbloquee la cuenta.
- **[CAPTCHA](#):** este mecanismo se suele implementar de manera que se requiera resolver un CAPTCHA tras a ver realizado un número de intentos fallidos. Más que prevenirnos de estos ataques lo que hace es reducir su probabilidad ya que al implementar CAPTCHA los ataques por fuerza bruta, por ejemplo, consumen muchos más recursos.
- **Preguntas de seguridad y palabras memorizables:** este mecanismo no sirve para autenticar a un usuario pero sí que reduce la probabilidad de este tipo de ataques, aunque si no se implementa debidamente estas preguntas o palabras pueden ser fácilmente explotables, la OWASP cuenta con una [Cheat Sheet](#) de buenas prácticas a la hora de implementar este mecanismo.

#### 4.2.1.2 Autenticación multi-factor

Los mecanismos de autenticación multi-factor consisten en requerir más de un tipo de evidencia a la hora de autenticar a los usuarios de una aplicación, estas evidencias pueden ser:

- **Algo que sabe:** contraseña, PINs, preguntas de seguridad.
- **Algo que tiene:** token, *Time-based One-Time Password* (TOTP), certificado, email, SMS, o llamada telefónica.
- **Algo que es:** huella dactilar, reconocimiento facial, escáner de iris.
- **Localización:** Geolocalización, o rango de IP.



Se ha demostrado que la incorporación de este mecanismo mejora considerablemente la protección contra los ataques que se realizan sobre las contraseñas, por ejemplo, ataques de fuerza bruta, [credential stuffing](#), o [password spraying](#). En algunas aplicaciones ya es obligatorio, como en las aplicaciones del sector bancario, y la tendencia es que las nuevas aplicaciones lo incorporen. En la siguiente [Cheat Sheet de la OWASP](#) encontramos más información importante acerca de los métodos de autenticación multi-factor. Por lo general cuando se quiere incorporar este mecanismo a una aplicación se suele integrar librerías externas en el lado del servidor que ya lo traen, por ejemplo, *Google Authenticator*,

#### 4.2.1.3 Otros mecanismos de autenticación

Hoy en día es muy popular que los mecanismos de autenticación sean piezas externas a la propia aplicación, de manera que toda la responsabilidad de ofrecer este servicio de autenticidad recaiga sobre un agente externo que hace de proveedor de identidades, por lo tanto, la seguridad de estos mecanismos depende completamente de la confianza puesta en estos proveedores de identidades.

En la mayoría de los casos la implementación de un mecanismo multi-factor es la opción más segura, pero hay en otros casos que no, o simplemente se tienen unos requerimientos diferentes, y es para estos casos que existen soluciones de terceros como las que vemos a ver a continuación que cada vez se están volviendo más populares y que por ejemplo para grandes infraestructuras con múltiples componentes suele ser la mejor de las soluciones.

Por lo general estos mecanismos se suelen implantar en la parte cliente de una aplicación, pero eso no quita que también puedan ser utilizadas en servidores para semejantes propósitos.

##### 4.2.1.3.1 OAuth

[Open Authorization](#) (OAuth) es un protocolo que permite delegar este servicio de autenticación a un agente externo que se denomina proveedor de identidades. Este agente externo se encarga de generar un *token* que va a servir para autorizar un conjunto de operaciones permitidas por dicha entidad. Actualmente se utiliza la versión [OAuth 2.0](#) que soluciona a través de HTTPS unas vulnerabilidades detectadas en la primera versión

Uno de los ejemplos más populares es la autenticación a través de un *application programming interfaces* (API), [API de Google](#), este API hace uso del protocolo OAuth para poder autenticar a una aplicación web a realizar operaciones sobre algún servicio de los que ofrece Google. A través de una librería JavaScript de Google se proporciona las herramientas suficientes como para poder generar un *token* que va a servir para ser autenticado en cada operación que se quiera realizar contra este proveedor de servicios.

Como este ejemplo hay muchos otros en los que una aplicación externa ofrece servicios que para ser utilizados hay que autenticarse siguiendo el estándar OAuth.

#### 4.2.1.3.2 OIDC

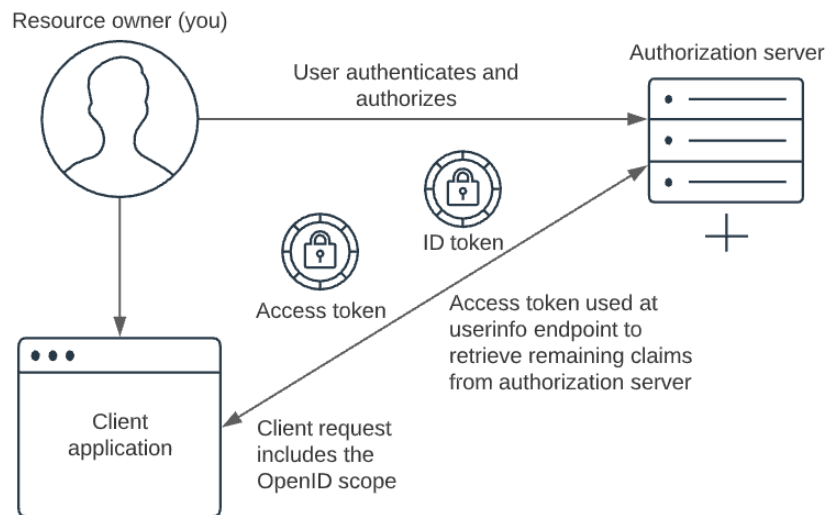
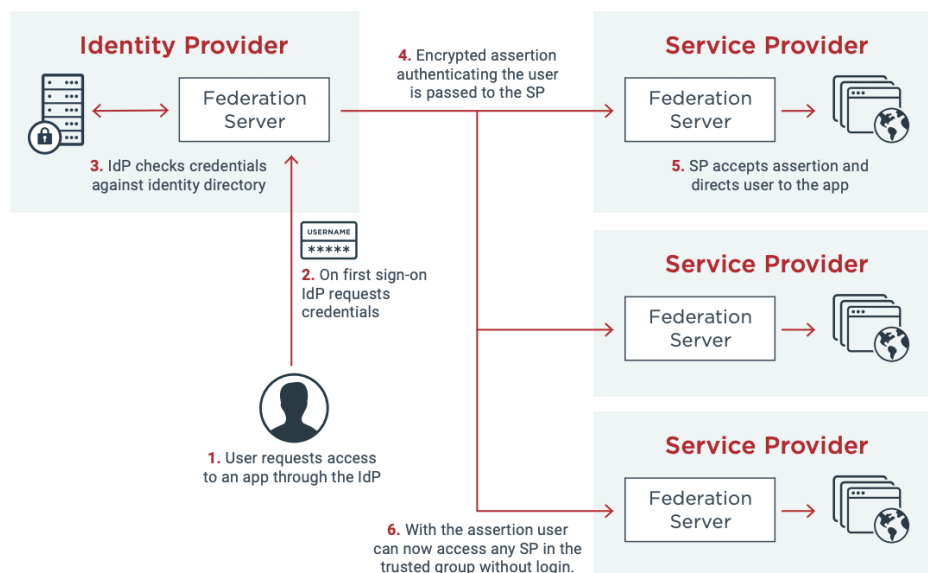


ILUSTRACIÓN 27 OIDC SOURCE: [HTTPS://WWW.PINGIDENTITY.COM/EN/RESOURCES/CONTENT-LIBRARY/ARTICLES/OPENID-CONNECT.HTML](https://www.pingidentity.com/en/resources/content-library/articles/openid-connect.html)

OpenID connect (OIDC) es protocolo de autenticación que implementa OAuth 2.0 y que a través de algún proveedor de identidades en el que se confía permite garantizar que un usuario es quien dice ser. Este protocolo permite realizar lo que se denomina como *Single Sign On* (SSO), SSO es un procedimiento que habilita a un usuario a ser autenticado en múltiples sistemas realizando una única autenticación, de manera que en la misma sesión del navegador tú puedes estar autenticado en múltiples aplicaciones web que confíen en el mismo proveedor de identidades habiendo realizado una sola autenticación.

### IdP-initiated Federated SSO



The six-step sequence illustrates a typical federated SSO use case.

ILUSTRACIÓN 28 SSO SOURCE: [HTTPS://WWW.PINGIDENTITY.COM/EN/RESOURCES/BLOG/POSTS/2021/WHAT-IS-SINGLE-SIGN-ON-SSO.HTML](https://www.pingidentity.com/en/resources/blog/posts/2021/what-is-single-sign-on-ssso.html)

Si cogemos como ejemplo otra vez la API de Google, vemos que también ofrece un servicio de autenticación a través de OpenId, de manera que, teniendo la librería JavaScript integrada en una aplicación podemos incorporar un botón que lo que va a hacer es redirigirnos a la plataforma

de autenticación de Google, una vez Google ha realizado esta autenticación el navegador redirige al usuario al sitio origen del que provenía asegurando a la aplicación origen y a todas las que confían en Google como proveedor de identidades que dicho usuario es quien dice ser.

#### 4.2.1.3.3 FIDO

*The Fast Identity Online* (FIDO) es una alianza que ha creado un conjunto de estándares de autenticación con el propósito de ofrecer un mecanismo de autenticación sin contraseñas o multi-factor, buscando mejorar la experiencia de un usuario a la hora de autenticarse transmitiendo simplicidad, rapidez, y seguridad al usuario.

Para implementar este mecanismo en las aplicaciones web lo que recomienda la asociación de FIDO es integrar la librería de *Web Authentication Api* ([WebAuthn](#)), en el caso de las aplicaciones front-end la librería de JavaScript que ofrecen está desarrollada por la propia asociación de FIDO, y a través de esta librería se realizan los procesos de registro y autenticación de usuarios. Hay un tutorial ofrecido por FIDO muy recomendado si se quiere implementar este mecanismo [aquí](#), Google también cuenta con un tutorial bastante recomendado [aquí](#).

### 4.2.2 Autorización y Confidencialidad

La parte front-end de las aplicaciones web es muy importante a la hora de establecer un mecanismo que cubra los servicios de autorización y confidencialidad, esto es debido a que en la gran mayoría de las aplicaciones la interfaz se va a tener que comportar de diferente manera en función del usuario que esté interaccionando en ese momento, de forma que un usuario solo pueda acceder, visualizar, y operar únicamente sobre lo mínimo necesario para ese usuario. Independientemente de la herramienta que utilicemos para desarrollar una aplicación web finalmente los activos a proteger son los mismos:

- **Accesos:** las diferentes opciones de navegación dentro de la aplicación (rutas).
- **Contenido:** dentro de cada pantalla el contenido que se le muestra al usuario (elementos).
- **Operaciones:** del contenido que puede visualizar el usuario qué puede hacer con él (acciones).

El cómo vamos a proteger estos activos también es independiente de la tecnología, el mecanismo es el mismo pero diferentes formas de implementarlo. La opción más recomendada es establecer una estructura de datos con **roles**, **permisos**, y **scopes**, estos último son las diferentes acciones que se pueden realizar sobre un activo (crear, editar, ...).

#### 4.2.2.1 Autorización basada en Roles y Permisos

##### 4.2.2.1.1 Scopes

Como hemos comentado previamente los scopes son las posibles acciones que se van a poder realizar sobre las diferentes entidades de la aplicación, primeramente, siempre vamos a contar con los siguientes scopes: *create*, *update*, *delete*, *read*. Estas son las operaciones más básicas que se pueden realizar, pero en algunos casos es interesante contar con otras como *hide* para ocultar elementos, *show* para mostrarlos, *move* para moverlos, etc.

Por lo general en una aplicación no vamos a tener una única entidad, asique lo normal será definir este conjunto de *scopes* para cada una de ellas. Hay una posible alternativa para mejorar la gestión de los *scopes* resultantes cuando son muchos, y es hacer una categorización por entidades

o *features* (unidad funcional dentro de la aplicación, ej. *Feature* libros) pudiendo determinar para cada una de ellas que *scopes* aplican.

#### 4.2.2.1.2 Roles

Cada rol va a significar una agrupación de usuarios en función de los permisos que se le vayan a asignar. Un ejemplo muy claro es el caso de rol *USER* y rol *ADMIN*, los usuarios con rol *ADMIN* por lo general van a disponer de más permisos que los usuarios con rol *USER*, pues según van creciendo una aplicación web esta jerarquía de roles se va volviendo cada vez más necesaria, por ello una buena práctica a la hora de montar una aplicación web es implementar este mecanismo al inicio del proyecto, aunque en un principio no sea necesario, pero si se decidiera implementarlo más adelante posiblemente supondrá un problema.

#### 4.2.2.1.3 Permisos

Finalmente teniendo los *scopes* y los roles bien definidos lo que vamos a crear es la estructura de datos a la que podamos consultar, por ejemplo, dado un role saber los *scopes* que tiene permitidos, o que dado un *scope* que roles lo tienen permitido. Para el caso en el que hacemos una separación lógica de las entidades / *features* la estructura de datos en vez de contener un listado de *scopes* por cada rol contendrá un listado de pares categoría – *scope* por ejemplo.

#### 4.2.2.2 Feature Flags

Las *Feature Flags* son un mecanismo que nos permite la integración de funcionalidades que incompletas o no, nos interesa tenerlas en un entorno productivo sin que estas sean accesibles. Esto se consigue estableciendo unos valores booleanos que van a determinar si cierta funcionalidad está habilitada o no.

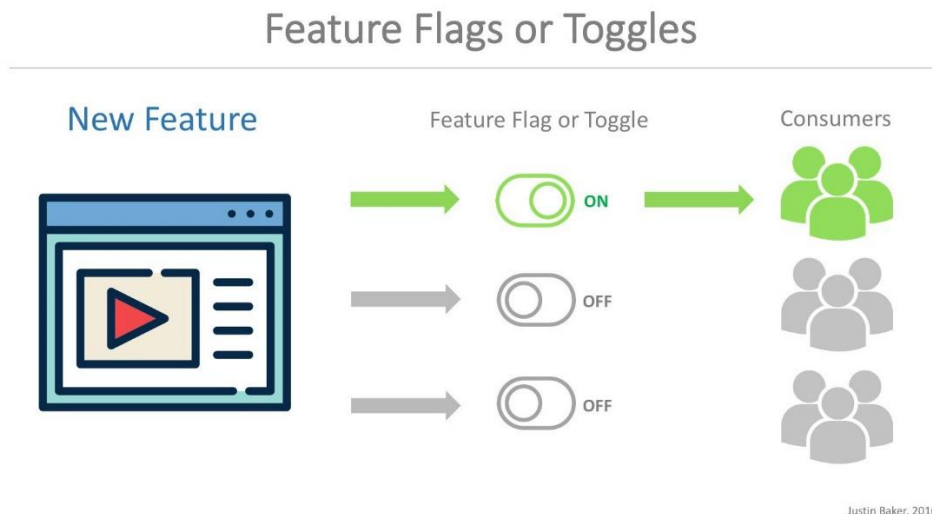


ILUSTRACIÓN 29 FEATURE FLAGS SOURCE: [HTTPS://MEDIUM.COM/CROWDBOTICS/INSTANTLY-RELEASING-FEATURES-IN-A-WORLD-THAT-NEVER-SLEEPS-USING-FEATURE-FLAGS-E0E9E6C56D26](https://medium.com/crowdbotics/instantly-releasing-features-in-a-world-that-never-sleeps-using-feature-flags-e0e9e6c56d26)

Otro uso de los *Feature Flags* son los test A/B habilitando cierta funcionalidad solo a una parte de los usuarios, esto es muy popular últimamente ya que algunos usuarios categorizados como usuarios *beta* dan su consentimiento para que se les habiliten estas funcionalidades que aún no están completas y de esta manera sean probadas por usuarios reales en un entorno productivo.

A la hora de implementar estos valores booleanos lo ideal es que puedan ser cambiados desde variables de entorno, de manera que sin tener que volver a hacer una nueva puesta en producción podamos controlar dichas variables.

#### 4.2.2.3 Buenas prácticas

- **Scopes atómicos:** A la hora de establecer los *scopes* de los permisos es buena práctica que estos sean lo más atómicos posibles, de manera que un *scope* no te permita realizar múltiples operaciones, si no que cada *scope* haga referencia a una única acción.
- **Separación de los permisos:** los mapas de permisos establecidos tienen que estar almacenados en una base de datos de un servidor y no en el cliente, este punto es **muy importante** debido a que si almacenamos esta información en el front-end de una aplicación corremos el peligro de que si en un momento determinado de la aplicación surge una brecha de seguridad, un atacante puede alterar estos permisos con el objetivo de escalar privilegios y realizar acciones que antes no estaban permitidas para su rol. Otra ventaja de tener esta información en una base de datos es que podemos actualizar los permisos concedidos a un grupo de usuarios sin tener que realizar un nuevo despliegue de la aplicación.
- **Nombrado de los roles:** para que este mecanismo sea lo más mantenible posible es conveniente que el nombre dado a los roles tenga el mayor sentido posible en relación con las acciones que puede realizar cada rol.
- **Construcción constante de scopes:** otro punto relevante a la hora de que el código sea más mantenible es ir creando los *scopes* al mismo tiempo que se va integrando nueva funcionalidad, de manera que si se añaden nuevas entidades a la aplicación se cree su estructura de *scopes* aunque en ese momento no sea necesario, de esta manera no es un problema si en un futuro se desea aplicar permisos sobre una entidad que no disponía de ellos previamente.
- Establecer **Feature Flags** para que el código se vaya integrando en entornos productivos con el propósito de detectar errores en una fase previa a su verdadera puesta en producción.

#### 4.2.3 Integridad

Como se ha contado previamente, en la parte front-end de las aplicaciones web hay una parte importante a la hora de asegurar la integridad de los datos y los componentes de la propia aplicación, cuando realizamos la descarga las dependencias del proyecto a través de un gestor de dependencias como puede ser *npm* hay que realizar las comprobaciones pertinentes para asegurar la integridad de las dependencias, esta herramienta por ejemplo, integra un mecanismo que hace estas verificaciones de integridad de la siguiente manera, a la hora de realizarla publicación de algún paquete genera un [checksum](#) que posteriormente utiliza para realizar [comprobaciones de integridad](#) cuando se realizan descargas de dicho paquete.

Además, estas herramientas como *npm* ofrecen algún comando para lanzar un conjunto de comprobaciones, una de ellas suele ser la comprobación de integridad de las dependencias descargadas, en el caso de *npm* este comando es el [npm doctor](#).

Otra de las recomendaciones para asegurar la integridad de las dependencias descargadas es, el uso de registros locales privados como [nexus](#), o [verdaccio](#), de esta manera cada vez que se realice una descarga de las dependencias establecemos una nueva capa de seguridad evitando pasar por registros públicos.

## 4.2.4 Disponibilidad

### 4.2.4.1 Monitorización y Trazabilidad

Cada vez la complejidad de las aplicaciones front-end es más y más compleja, y que la aplicación deje de estar disponible o alguna de sus partes no esté accesible o no funcione correctamente, puede generar grandes pérdidas a una compañía, es por ello la importancia de implementar estos mecanismos de monitorización y trazabilidad de la aplicación que sean capaces de **detectar errores** en su etapa inicial con el fin de corregirlos en el menor tiempo posible, otros factores importantes de la detección de estos errores es determinar la frecuencia y de donde provienen, ya que pueden provenir del **propio JavaScript de la aplicación**, **fallos de conexión**, de una **librería de terceros**, o incluso del **framework utilizado** para el desarrollo de la aplicación web.

Otra de las funciones bastante importantes que ofrecen estas herramientas es la detección de **problemas de rendimiento**, estos pueden suponer una mala usabilidad “User Experience” (UX) y en algunas ocasiones esto suponga un problema de seguridad, como errores humanos, ya que una mala UX puede llevar a un usuario a realizar acciones no deseadas, por ejemplo, si hay algún elemento que está haciendo de cuello de botella en el rendimiento de la aplicación front-end y a la hora de realizar un clic sobre un botón que realiza una acción, un usuario si nota que no ha funcionado puede volver a realizar múltiples clics sobre dicho botón llevando a un estado no deseado en el que se han realizado dicha acción múltiples veces.

Finalmente, para asegurar la disponibilidad de una aplicación web es importante **monitorizar los usuarios** de la propia aplicación, que acciones toman dentro de la misma, y más importante aún, el flujo de usuarios activos dentro de la aplicación con el objetivo de analizar se es necesario realizar algún escalado de los servicios utilizados por la aplicación o realizar alguna mejora en la lógica de la aplicación. En resumen, los mecanismos que hemos de implementar en una aplicación web para asegurar la disponibilidad y el buen funcionamiento son los siguientes:

- Monitorización de errores
- Monitorización del rendimiento
- Monitorización de usuarios

A continuación, vamos a pasar a describir algunas de las herramientas que nos facilitan la implementación de dichos mecanismos.

#### 4.2.4.1.1 Pingdom



ILUSTRACIÓN 30 LOGOTIPO PINGDOM

[Pingdom](#) (desarrollada por [SolarWinds](#)) esta herramienta ofrece un conjunto amplio de servicios como la monitorización SSL, monitorización de usuarios en tiempo real, monitorización del tiempo de actividad, etc. Las ventajas de esta herramienta respecto al resto son:

- **Uptime Monitoring:** esta funcionalidad nos ofrece la posibilidad de ver el tiempo que tarda una página o un recurso en ser totalmente cargado, y si ocurre algún error en el proceso de carga genera unos reportes detallados que te facilitan detectar el origen del problema, además se pueden obtener las trazas de los errores ocurridos.
- **PageSpeed:** con esta herramienta podremos obtener información detallada de la propia aplicación, tiempo de carga, número de peticiones, puntuación de los rendimientos, tamaños de las páginas cargadas, diagramas en cascada que muestran el timeline de los *assets* (recursos de la aplicación) que se van cargando, etc.
- **Alerting:** el **sistema de alertas** que ofrece Pingdom, es **muy personalizable** pudiendo determinar grupos de personas notificadas, el medio de notificación (**sms**, email, ...), etc.

#### 4.2.4.1.2 RayGun



ILUSTRACIÓN 31 LOGOTIPO RAYGUN

[RayGun](#) es una aplicación que ofrece **servicios en tiempo real** para monitorizar la calidad y el rendimiento de aplicaciones web y móviles, con el objetivo de detectar y diagnosticar posibles problemas de rendimiento, como lo son los cuellos de botella, y así poder solucionarlos rápidamente mejorando la experiencia de usuario de la aplicación.

Esta herramienta cuenta con una **interfaz muy intuitiva** en la que podemos visualizar rápidamente datos de la velocidad de una página web, tiempos de carga, el número de sesiones activas, el número de usuarios activos, etc.

La compañía ofrece tres productos por separado:

- **Crash Reporting:** para detectar y diagnosticar errores.
- **Real User Monitoring:** para monitorizar el rendimiento front-end.
- **Application Performance Monitoring:** para monitorizar el rendimiento de la parte back-end.

Esta herramienta es una muy buena alternativa para monitorizar los usuarios y el rendimiento de una aplicación, ofreciendo los resultados a través de un interfaz gráfico bastante competitivo.



#### 4.2.4.1.3 Sentry



ILUSTRACIÓN 32 LOGOTIPO SENTRY

La herramienta de [Sentry](#) es un SDK que permite monitorizar el rendimiento de una aplicación web, pero sobre todo **monitorizar en búsqueda de errores en tiempo real**, desde la parte front-end hasta la parte back-end.

*Sentry* destaca en la representación de la información extraída de las excepciones capturadas ya que permite ver detalles como el número de usuarios afectados, la propia traza del error, que navegadores han sido afectados, y **qué commit introdujo los cambios que causaron el error**. Además, ofrece un sistema de alertas a través de email que te notifica cuando se ha desplegado una versión que está generando algún tipo de error. Esta herramienta es muy flexible permitiendo integrarla en casi cualquier lenguaje / *framework* actual, ofreciendo una instalación sencilla en cualquiera de ellos.

Finalmente, otro punto a destacar de *Sentry*, es que dispone de una versión gratuita orientada a desarrolladores.

#### 4.2.4.1.4 LogRocket



ILUSTRACIÓN 33 LOGOTIPO LOGROCKET

[LogRocket](#) es otra herramienta desarrollada para la monitorización de sesiones de usuario, suele ser de las alternativas más elegidas en este ámbito, ya que **permite reproducir las sesiones de los usuarios**, facilitando en gran medida el rastreo de errores o posibles problemas de rendimiento, ya que te permite ver exactamente lo que el usuario estaba viendo. Aparte de poder reproducir las sesiones de los usuarios esta herramienta monitoriza la aplicación para la detección de errores JavaScript capturando las trazas de error e información acerca de los errores producidos y de cómo solucionarlo.

Otra de las ventajas de esta herramienta es que se integra fácilmente con otras herramientas, como, por ejemplo, con *Sentry*, y con los frameworks JavaScript más utilizados como, Vue, React, Angular, etc.

Finalmente, *LogRocket* ofrece un sistema de alertas a través de email o la aplicación de [Slack](#).





ILUSTRACIÓN 34 LOGOTIPO APPSIGNAL

[AppSignal](#) es una herramienta **todo en uno** que ofrece diferentes servicios sin tener que realizar integraciones complejas, si no que, se caracteriza por su **rápida y fácil instalación**.

Los servicios que ofrece son:

- Monitorización de errores.
- Monitorización de rendimiento front-end.
- Monitorización back-end.
- Visualización de métricas en tiempo real.
- Detección de anomalías y sistema de alertas.

Los inconvenientes de *AppSignal* son que únicamente admite *Elixir*, *Node.js*, *Ruby*, y *JavaScript*, y que la información registrada del rendimiento y de los errores, es un poco pobre respecto al de otras alternativas.

#### 4.2.4.1.6 Firebase



ILUSTRACIÓN 35 LOGOTIPO FIREBASE

[Firebase](#) es una plataforma de servicios desarrollada por Google que ofrece múltiples herramientas, y en concreto una de ellas es [Firebase Crashlytics](#), esta herramienta no ofrece soporte para aplicaciones web, pero sí que es la herramienta por excelencia en la **monitorización de errores en tiempo real para aplicaciones, tanto híbridas como nativas**, es por ello que merece la pena nombrarla, por ser una alternativa muy interesante en la monitorización y trazabilidad de las aplicaciones híbridas.

En los errores detectados por esta herramienta encontramos información que nos permite identificar el problema de raíz y además cuenta con un sistema de alertas en tiempo real de los nuevos errores que nos permite llevar una trazabilidad de estos.

#### 4.2.4.1.7 Otras herramientas

Hoy en día, existen múltiples alternativas a estas herramientas que perfectamente son igual de válidas, entre ellas encontramos: [Sematext](#) que es un todo en uno muy completo, [Site24x7](#) otro todo en uno, [SpeedCurve](#) que ofrece monitorización de usuarios en tiempo real desde una interfaz muy usable, [Rollbar](#) que ofrece un sistema para la clasificación de errores, [Airbrake](#) herramienta para la trazabilidad de errores fácil de utilizar, [Azure Application Insights](#), [Datadog](#) que es otra herramienta que ofrece todos los servicios necesarios, y muchas otras herramientas.

Como se puede apreciar, el mundo de la monitorización y trazabilidad de aplicaciones front-end está bastante avanzado, dispone de diferentes opciones para su implementación entre las que hay que elegir las que más se adapten a los requisitos de cada aplicación.

#### 4.2.4.2 Escaneo de vulnerabilidades

Una de las principales vulnerabilidades en aplicaciones front-end que comentábamos en el Top 10 de OWASP, son los componentes desactualizados o que poseen algún tipo de vulnerabilidad. Hoy en día las aplicaciones front-end utilizan muchas librerías de terceros que ayudan a ahorrar tiempo a la hora de implementar cierta funcionalidad, pero en algunas ocasiones puede suponer un problema de seguridad debido a que una dependencia tenga una versión desactualizada o que contenga una vulnerabilidad aún no detectada.

Por lo general, las herramientas de descarga de dependencias como *npm* traen mecanismos de seguridad que permiten realizar un escaneo de vulnerabilidades sobre las dependencias descargadas en el proyecto, en el caso de *npm*, [npm audit](#), este comando consulta registros actualizado de vulnerabilidades bien conocidas para detectar que dependencias son vulnerables. También da la posibilidad de auto corregir estas dependencias vulnerables con el comando *npm audit fix*. A continuación, se muestra una imagen de la información devuelta por este comando:

<b>Moderate</b>	<b>Prototype pollution</b>
Package	hoek
Patched in	> 4.2.0 < 5.0.0    >= 5.0.3
Dependency of	numbat-emitter
Path	numbat-emitter > request > hawk > boom > hoek
More info	<a href="https://nodesecurity.io/advisories/566">https://nodesecurity.io/advisories/566</a>

ILUSTRACIÓN 36 NPM AUDIT SOURCE: [HTTPS://DOCS.NPMJS.COM/ABOUT-AUDIT-REPORTS](https://docs.npmjs.com/about-audit-reports)

Entre la información devuelta hay que destacar el primer dato que es la severidad de la vulnerabilidad, que nos va a servir como indicativo de la gravedad de la vulnerabilidad detectada, ese dato podrá tomar los siguientes valores:

TABLA 2 NPM AUDIT SEVERITY

Severity	Recommended action
Critical	Address immediately
High	Address as quickly as possible
Moderate	Address as time allows
Low	Address at your discretion

Otro de los comandos que ofrece npm que puede servir de utilidad es [npm outdated](#) que se utiliza para comprobar si alguno de los paquetes instalados está deprecado en cuyo caso lo más recomendable es sustituir dicho paquete por otro que esté a día y que sea mantenido por el autor. Más adelante veremos que estas comprobaciones pueden ser automatizadas en el ciclo de vida del desarrollo de aplicaciones con el objetivo de que siempre se mantengan bien actualizadas.

#### 4.2.4.3 Testing

Otro de los mecanismos que ayuda a reducir los problemas que afectan a la disponibilidad de una aplicación es el desarrollo de test. Años atrás no era una práctica muy común en las aplicaciones front-end, pero cada vez se está volviendo más necesario debido a que cada vez suponen mayor complejidad y es por ello por lo que es buena práctica el desarrollo de test para mejorar la calidad del producto asegurando el correcto funcionamiento de las funcionalidades que ofrece, y pudiendo detectar errores en una fase lo más temprana posible.

Los test que podemos tener en una aplicación front-end por lo general se van a centrar en validar lo que los usuarios pueden ver a través del navegador, y que las funcionalidades ofrecidas por la aplicación funcionan como es esperado. Un ejemplo sería validar que al rellenar un formulario de creación finalmente el elemento que hemos creado se visualiza por pantalla, independientemente de si de verdad se ha creado en una base de datos.

Otro factor importante de los test front-end es que a diferencia de los test en back-end no se centran tanto en la cobertura del código si no que lo más importante es asegurarse que la aplicación se comporta como es esperado teniendo en cuenta todos los casos extremos.

Existen varios tipos de test, y por lo general las herramientas que se utilizan para su desarrollo se integran con cualquiera de los nuevos frameworks de desarrollo de aplicaciones. En la siguiente tabla podemos ver cada tipo, su nivel de abstracción, scope (que partes cubre), y algunas de las herramientas que nos facilitan su implementación.

Tipo	Nivel de abstracción	Scope	Herramientas
<b>Análisis estático</b>	Low	Código fuente. Errores de sintaxis, malas prácticas, formato, etc.	<ul style="list-style-type: none"><li>• <a href="#">ESLint</a></li><li>• <a href="#">Prettier</a></li><li>• <a href="#">SonarLint</a></li><li>• <a href="#">Husky</a></li><li>• <a href="#">JSHint</a></li><li>• <a href="#">JSLint</a></li></ul>
<b>Unitarios</b>	Low	Funciones y métodos de la aplicación. Comportamiento de cada componente de la aplicación	<ul style="list-style-type: none"><li>• <a href="#">AVA</a></li><li>• <a href="#">Jasmine</a></li><li>• <a href="#">Jest</a></li><li>• <a href="#">Karma</a></li><li>• <a href="#">Mocha</a></li><li>• <a href="#">Testing Library</a></li></ul>

Tipo	Nivel de abstracción	Scope	Herramientas
Integración	Medium	Interacción entre los diferentes componentes que componen la aplicación.	<ul style="list-style-type: none"> <li>• <a href="#">Jest</a></li> <li>• <a href="#">Cypress</a></li> <li>• <a href="#">Testing Library</a></li> <li>• <a href="#">AVA</a></li> </ul>
End-to-end	High	Interacciones de un usuario con la aplicación a través de un navegador. Convirtiendo cada posible acción ejecutada por un usuario en instrucciones a ejecutar para finalmente validar el resultado esperado.	<ul style="list-style-type: none"> <li>• <a href="#">Cypress</a></li> <li>• <a href="#">Testing Library</a></li> <li>• <a href="#">Puppeteer</a></li> <li>• <a href="#">Selenium</a></li> </ul>
Accesibilidad	High	Cumplimiento de los estándares de <a href="#">accesibilidad</a> .	<ul style="list-style-type: none"> <li>• <a href="#">AccessLint</a></li> <li>• <a href="#">axe-core</a></li> <li>• <a href="#">Lighthouse</a></li> <li>• <a href="#">pa11y</a></li> </ul>
Regresión visual	High	Estructura visual, cambios visuales producidos por cambios en el código.	<ul style="list-style-type: none"> <li>• <a href="#">Applitools</a></li> <li>• <a href="#">Cypress</a></li> <li>• <a href="#">Percy</a></li> <li>• <a href="#">Needle</a></li> </ul>
Performance	High	Estabilidad y rendimiento.	<ul style="list-style-type: none"> <li>• <a href="#">Lighthouse</a></li> <li>• <a href="#">PageSpeed Insights</a></li> <li>• <a href="#">WebPageTest</a></li> <li>• <a href="#">YSlow</a></li> </ul>

TABLA 3 TESTING FRONT-END FUENTE: [HTTPS://CSS-TRICKS.COM/FRONT-END-TESTING-IS-FOR-EVERYONE/#H-UNIT-TESTING](https://css-tricks.com/front-end-testing-is-for-everyone/#h-unit-testing)

Hoy en día, las herramientas más **utilizadas** según las estadísticas del [stateofjs](https://stateofjs.com) son:

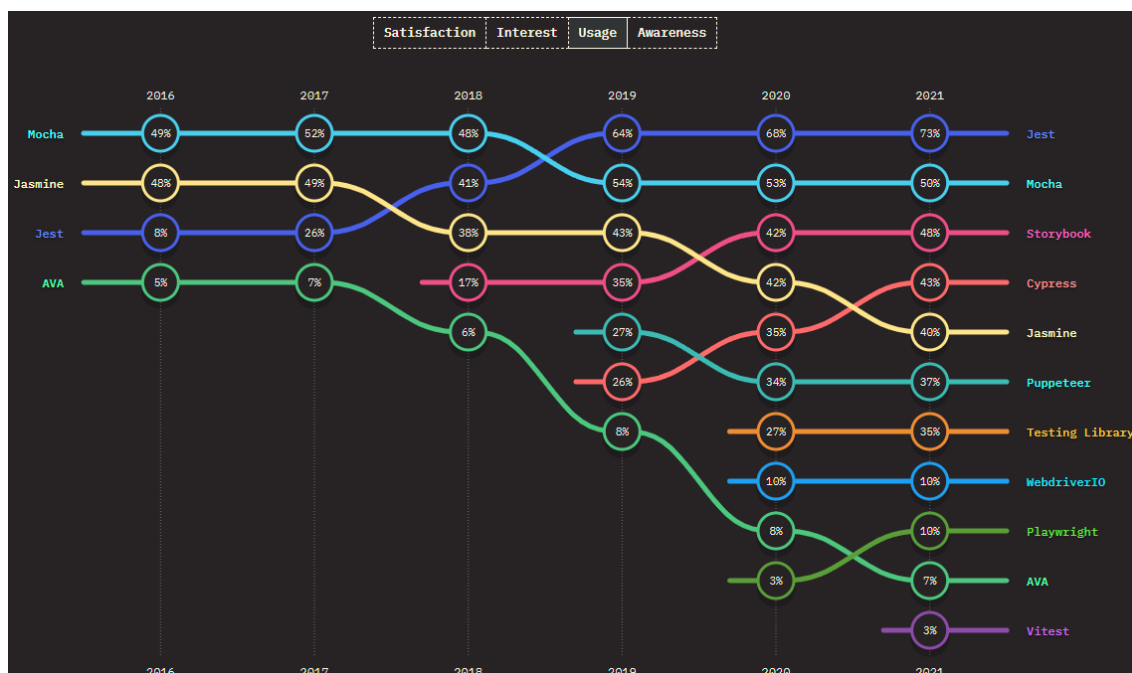


ILUSTRACIÓN 37 TESTING SOURCE: [HTTPS://2021.STATEOFSJS.COM/EN-US/LIBRARIES/TESTING](https://2021.STATEOFSJS.COM/EN-US/LIBRARIES/TESTING)

Existe un gran debate entre cuantos test hay que hacer de cada tipo debido a que el coste de alguno de ellos es bastante elevado, pero por lo general se tiende a desarrollar unos pocos test unitarios, bastantes de integración y algún test E2E.

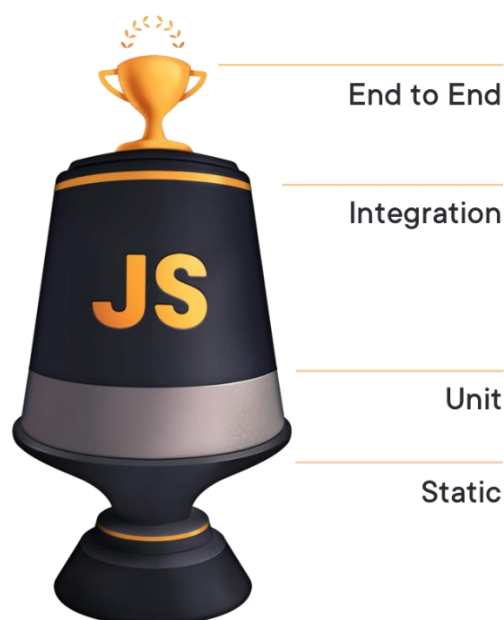


FIGURA 7 TEST FRONTEND FUENTE: [HTTPS://TESTINGJAVASCRIPT.COM/](https://testingjavascript.com/)

En el siguiente punto veremos otro mecanismo que mejora la disponibilidad de las aplicaciones juntando todos los mecanismos vistos hasta el momento convirtiéndose en una pieza fundamental en cualquier aplicación.

#### 4.2.4.4 CI/CD

Continuous Integration (CI) y Continuous Deployment (CD) es una estrategia de desarrollo software que permite acelerar la puesta en producción de las aplicaciones asegurando que los criterios de calidad establecidos se cumplen en cada integración de código realizada, todo ello se consigue a través de los *pipelines*, estos se encargan de la puesta en producción de las aplicaciones automatizando todo el proceso que ello conlleva, no solo la preparación del artefacto que finalmente va a ser desplegado, si no también todas las comprobaciones pertinentes para asegurar que la versión que va a ser desplegada es estable y ha superado las barreras de calidad establecidas.

La parte de **CI** se centra en la automatización de la construcción y validación del código, pudiendo incorporar al proceso automático la ejecución de los test vistos anteriormente, mientras que la parte **CD** se encarga de automatizar el despliegue de cada versión de la aplicación que ha pasado dichas validaciones, este proceso de despliegue por lo general se realiza haciendo uso de lo que se denominan contenedores, más adelante veremos que sobre estas piezas que componen los *pipelines* también es necesario realizar comprobaciones de seguridad.

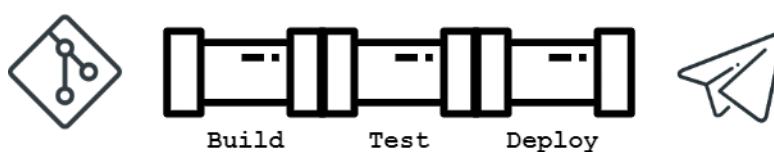


ILUSTRACIÓN 38 CD PIPELINE

Este mecanismo agiliza la puesta en producción de las aplicaciones y, además, al estar todo el proceso automatizado eliminamos el factor humano de la cadena que suele ser de los mayores riesgos que se suelen tener hablando en términos de seguridad, por lo tanto, evitamos que puedan ocurrir errores humanos en algunos de los pasos a realizar, o por ejemplo de que se despliegue una versión que no cumple con los criterios de calidad. Todo ello hace que implementar un mecanismo de CI/CD en una aplicación web sea esencial a la hora de pensar en **estabilidad**, **seguridad** y en **disponibilidad**, ya que nos da la posibilidad de automatizar ciertas comprobaciones de seguridad en cada despliegue realizado de la aplicación.

Hoy en día, existen múltiples herramientas para la creación de pipelines entre ellas encontramos algunas como: [Jenkins](#), [CircleCI](#), [Bitbucket](#), [TravisCI](#), etc. Dicho todo esto, gran parte de estas tareas siempre han sido desempeñadas por un perfil [DevOps](#), pero cada vez los desarrolladores son más autónomos en este tipo de tareas, y por lo general un perfil DevOps está mucho más especializado en el ámbito de la Ciberseguridad que un desarrollador, es por ellos que hay muchos aspectos que se tienen que tener en cuenta a la hora de construir estos *pipelines* de CI/CD, en los últimos años a partir de esta necesidad nació el termino *DevSecOps* que es aquella persona encargada de dotar de seguridad al ciclo de vida de las aplicaciones, implementando mecanismos de seguridad sobre los *pipelines* CI/CD. Por lo general, esta tarea desempeñada por los *DevSecOps* suele tener en cuenta:

- **Monitorización** y **audición** los pipelines.
- Implementación de **comprobaciones de seguridad en cada capa de la aplicación**, asegurando que las vulnerabilidades de una capa en concreto no puedan afectar a capas

adyacentes. Para entender mejor la lógica que hay detrás de esto se recomienda leer que es el modelo [Swiss cheese model](#).

- Hacer uso de **permisos de menor privilegio temporales**, uno de los fallos que se comete cuando el proceso realizado por los *pipelines* se realizaba manualmente era que los encargados de desempeñar dicha tarea disponían de permisos sobre todos los recursos necesarios por un tiempo indefinido, esto puede suponer una amenaza por el robo de credenciales, por ello se recomienda que en los *pipelines* se utilicen permisos de menos privilegio que tengan un tiempo limitado de manera que todo quede bien cerrado tras completarse el *pipeline*.
- Adicionalmente a las fases de test se recomienda implementar mecanismos que realicen **escaneos de seguridad entre cada una de las fases del *pipeline*** en busca de vulnerabilidades en el código fuente o alguna de sus dependencias (visto anteriormente con *npm audit*), [en alguno de los contenedores utilizados](#) para el despliegue, o en la propia infraestructura.
- Otra de las consideraciones de seguridad que se tienen en cuenta es, la **separación de los *pipelines* a un repositorio aparte**, de esta manera implementamos una capa de seguridad entre los *pipelines* y las aplicaciones, con esto podemos aplicar diferentes permisos sobre el repositorio que contiene la aplicación y el repositorio que contiene los *pipelines*, esta práctica es muy útil cuando la persona que controla los *pipelines* es diferente a la persona que desarrolla la aplicación, evitando que puedan producirse errores humanos como por ejemplo, que se modifique un *pipeline* por error.

En el capítulo siguiente profundizamos más en este perfil de *DevSecOps* y en qué casos será necesario disponer de una persona especializada en este ámbito.

# 5 DECISIONES EN FASE DE DISEÑO

En este capítulo se presenta el conjunto de decisiones esenciales que deben tomarse en la fase de diseño de una aplicación web. Entre ellas, las que tienen que ver con la seguridad y constituyen una parte esencial a tener en cuenta, si bien deben siempre balancearse adecuadamente con otras de no menos importancia como son las relacionadas con la experiencia de usuario (UX) o con la selección de aplicaciones nativas o híbridas, que se discuten en las dos secciones iniciales.

## 5.1 UX

*User Experience (UX) design* se define por la [Interaction Design Foundation](#) como:

“Is the process design teams use to create products that provide meaningful and relevant experiences to users. This involves the design of the entire process of acquiring and integrating the product, including aspects of branding, design, usability and function.”

Por lo tanto, entendemos por UX aquel proceso de creación de productos por el cual dotamos a una aplicación de experiencias significativas y relevantes a los usuarios, a partir de aspectos como el *branding*, el diseño, la usabilidad y la funcionalidad de la aplicación, estos son los que finalmente determinan el diseño UX de las aplicaciones.

A la hora de diseñar la propia interfaz de las aplicaciones web es muy importante tener en cuenta la experiencia de usuario, ya que si una interfaz no es muy usable posiblemente no sea del todo segura, y es que un alto porcentaje de los problemas de seguridad de las aplicaciones se pueden deber a errores humanos y estos a su vez pueden haber sido originados por un mal diseño UX, estos errores se pueden deber a alguno de los siguientes motivos:

- El usuario no ve algún elemento de la pantalla.
- El usuario no ha leído o entendido completamente un mensaje o aviso que la aplicación tiene que transmitirle.
- El usuario ha cometido un error de escritura.
- El usuario ha cometido un error al realizar un clic.
- El usuario se ha saltado algún procedimiento.
- El usuario no hace un uso correcto de alguna de las funcionalidades.

Por lo general las aplicaciones web con una interfaz fácil de utilizar suelen inspirar mayor confianza y seguridad a los usuarios, esto es muy importante a la hora de requerir a los usuarios que realicen algún proceso de seguridad, ya que es más probable que este lo lleve a cabo si confía en la propia aplicación y en su seguridad.

Es importante que las medidas de seguridad que se deseen implantar en una aplicación web se apliquen desde las primeras fases del desarrollo del producto y que estén presentes durante el ciclo de vida de desarrollo, es por ello por lo que es muy importante a la hora de tener que tomar decisiones sobre el diseño UX de una aplicación que se tengan ciertos aspectos en cuenta:



- **Privacidad del usuario**, es importante que a la hora de diseñar los textos y mensajes de la aplicación se tenga en cuenta cual es la información sensible para evitar mostrarla por la pantalla sustituyéndola por ejemplo por mensajes genéricos.
- **Transparencia con el usuario**, el usuario debe tener claro en todo momento cuales son nuestras intenciones respecto a cómo van a ser utilizados sus datos por la aplicación, y a ser posible dar la flexibilidad al usuario de elegir que se puede y que no se puede hacer con los datos que quiera suministrar.
- **Sorprender al usuario** con nuevos mecanismos, por ejemplo, a la hora de solicitar a un usuario la confirmación de borrar algún elemento, obligar al usuario a escribir la palabra BORRAR para habilitar el botón que finalmente borra el elemento, de esta manera evitamos que los usuarios vayan pulsando los botones de confirmar automáticamente.
- Pedir **confirmación al realizar acciones importantes**, sin ser intrusivo para no aburrir al usuario.
- **Advertir al usuario** cuando esté haciendo algo incorrecto o que pudiera acabar en un estado de error, en vez de esperar a tener que mostrarle un mensaje de error, es mejor pensar en cómo evitar que llegue a ese estado de error. Un ejemplo de esto sería el caso de Gmail con los archivos adjuntos, si Gmail detecta que en el cuerpo de un correo escribes la palabra “*adjunto*” y se te olvida adjuntar el archivo que ibas a adjuntar, a la hora de enviar el correo Gmail te saca un mensaje de confirmación avisándote que has escrito esa palabra y te pregunta que, si estás seguro de que deseas enviar el correo sin adjuntar ningún archivo, de esta manera estamos ayudando a los usuarios a no cometer errores. En el caso de otros gestores de correo, finalmente se enviaría el correo sin previo aviso.
- **Evitar los procesos aburridos**, ya que es en estos procesos en los que los usuarios realizan dichos procesos de manera automática sin prestar casi atención, o sencillamente dejan de realizar estos procesos, esto suele pasar con muchos mecanismos de seguridad, por ejemplo, si para cumplir la política de contraseñas el usuario tiene que dedicar demasiado tiempo en encontrar una contraseña segura, un porcentaje de los usuarios no la cambiara, en cambio sí facilitamos al usuario este proceso mediante consejos y ejemplos, posiblemente consigamos que gran parte de los usuarios no terminen por desistir.
- **Mensajes de error explícitos**, ejemplo, a la hora de notificar a un usuario que ha introducido mal sus credenciales es muy recomendable decirle el motivo y aportarle cualquier tipo de ayuda que le sirva para llegar a introducir correctamente las credenciales, un buen ejemplo de esto es *Facebook* que te avisa de la última vez que la contraseña ha sido cambiada si te equivocas al introducirla.
- Prevenir errores debidos a acciones inconscientes por parte del usuario ofreciendo **sugerencias y avisos**, para que el usuario no se despiste a la hora de utilizar la aplicación.
- Utilizar **patrones de diseño que ayuden a comunicar** a los usuarios a cómo deberían trabajar con la aplicación, pidiendo doble confirmación en las acciones importantes, avisando de los errores cometidos, etc.
- Ofrecer a los usuarios **mecanismos para deshacer acciones**, si es posible en operaciones como borrar o archivar, operaciones en las que un elemento desaparece perdiendo el control sobre él, es recomendable dar la opción de “*deshacer*” para dar la posibilidad al usuario de que en caso de error tenga una manera de solucionarlo y volver a un estado anterior.
- Utilizar **mapas de calor** para detectar en un futuro aquellas vistas de la aplicación en las que los usuarios no están interaccionando correctamente pasando por alto alguna parte importante de la pantalla.

- **Valores por defecto**, a la hora de ofrecer al usuario una vista en la que tenga que rellenar valores de un formulario ya sea de creación, de configuración o de lo que sea, si le ofrecemos unos **buenos** valores por defecto en base a algún tipo de predicción, podemos prevenir que el usuario no cometa algún error al introducir los valores incorrectamente.
- Realizar **pruebas con usuarios reales** simulando un entorno lo más cercano posible a producción, de esta manera muchos errores de usabilidad serán detectados en fases previas a su verdadera puesta en producción.
- **Mantener informado al usuario** acerca de los procesos de seguridad y el motivo por el que se llevan a cabo.
- Evaluar la aplicación para **determinar el grado de seguridad que es necesario**. En algunas aplicaciones en las que apenas se recolectan datos sensibles y no se realizan operaciones peligrosas, establecer mecanismos como lo son la autenticación de doble factor lo único que van a hacer es empeorar la experiencia del usuario.
- **No utilizar los emails como nombres de usuario**, esta información es sensible y no puede verse expuesta.

Únicamente nos hemos centrado en aquellos aspectos que son importantes a la hora de proteger una aplicación frente a posibles fallos de seguridad, pero hay unas guías UX establecidas por [Nielsen Norman Group](#) que se recomienda leer para profundizar más en algunas de las técnicas aplicadas en el diseño UX de las aplicaciones.

En resumen, un buen diseño UX en las aplicaciones web va a contribuir en gran medida en la prevención de errores, por ello, es muy importante que a la hora de diseñar la interfaz de una aplicación se tengan en cuenta los puntos vistos anteriormente para no descuidar estos detalles a la hora de tomar decisiones sobre el diseño UI/UX de las aplicaciones web.

## 5.2 Tipos de renderizado

Otra de las decisiones que tendremos que tomar a la hora de empezar el diseño de una aplicación web es, determinar la **técnica de renderizado** de la interfaz en base a los requerimientos de la aplicación a diseñar. Existen diferentes tipos de renderizado que analizaremos a continuación, y veremos en qué casos es mejor cada uno de los tipos. Finalmente veremos los aspectos de seguridad que deberemos tener en cuenta a la hora de tomar la decisión.

### 5.2.1 Server Side Rendering (SSR)

Este tipo de renderizado no es algo novedoso ya que antes de la aparición de JavaScript y los frameworks modernos, era la manera normal de funcionar. En páginas web desarrolladas con [JSP](#) o [PHP](#) por ejemplo, cada vez que se solicitaba mostrar una página se enviaba una petición al servidor de la aplicación y este generaba el HTML con todos los datos ya procesados al frontend, de manera que el navegador únicamente tenía que mostrar el HTML devuelto, sin tener que hacer ningún tipo de procesamiento.

Actualmente este tipo de renderizado puede ser implementado por alguno de los nuevos frameworks JavaScript, y por lo general suele utilizarse para páginas web en las que el contenido mostrado es puramente estático, ya que si tuviera que realizar algún cambio tendría que volver a renderizarlo en el servidor para posteriormente volver a enviarlo al cliente. Por lo tanto, al no depender tanto de la parte JavaScript en la parte del navegador la carga de la página es mucho más rápida.

Una de las principales ventajas de este método es el posicionamiento *Search Engine Optimization* (SEO), ya que al recibir la página totalmente renderizada los rastreadores de Google pueden examinarla e indexarla más rápidamente mejorando considerablemente el posicionamiento SEO.

En temas de seguridad realizar SSR consigue eliminar casi cualquier vulnerabilidad asociada a la parte frontend y a la ejecución de código en el navegador, no solo eso si no que de esta manera aseguramos la privacidad de los datos, ya que al cliente le llegan únicamente los datos que son necesarios mostrar por pantalla, y no necesita traer todos los datos del servidor al cliente para coger necesarios. Otra de las ventajas que ofrece SSR es que no requiere de tener un servidor arrancado ya el procesamiento de los datos se realiza directamente en el servidor, consiguiendo así reducir la superficie de la infraestructura expuesta al cliente.

### 5.2.2 Client Side Rendering (CSR)

Este tipo de renderizado es el más común en las aplicaciones frontend *Single Page Application* (SPA), y el que por lo general utilizan los frameworks y librerías vistas anteriormente. Consiste en **renderizar cada página en el propio navegador** del usuario que la va a visualizar, este proceso de renderizado suele ser tarea del propio framework o librería que se esté utilizando, al solicitar una página el servidor envía un HTML vacío que pasa por todo el código JavaScript que se encarga de ir generando el contenido que finalmente será mostrado por el navegador, este proceso hace que estas aplicaciones tengan buenas métricas *First Paint* (FP), *First Contentful Paint* (FCP), pero debido a que el navegador necesita renderizar nuevamente algunas partes del HTML en cada cambio el *Time To Interactive* (TTI) se ve afectado por el tiempo invertido por el código JavaScript. El problema de esta alternativa es que el archivo HTML final, es generado en el navegador y en ocasiones este proceso toma bastante tiempo, afectando como hemos comentado al TTI, y desmejorando el posicionamiento *Search Engine Optimization* (SEO).

En temas de seguridad, puede que esta opción sea la menos segura, ya que todo el procesamiento del código JavaScript se realiza en la parte del cliente, y es en la construcción de los ficheros HTML donde se producen la mayoría de los ataques, como las inyecciones de código o cualquiera de las vulnerabilidades vistas anteriormente.

### 5.2.3 Static Side Generation (SSG)

Con *Static Side Generation* en vez de renderizar las páginas en cada una de las peticiones, las páginas **son pre renderizadas en tiempo de construcción** generando los archivos HTML estáticos de cada página, estos archivos pueden ser alojados en algún *Content Delivery Network* (CDN) con el objetivo de que todo el contenido de la web sea estático, consiguiendo una gran mejora en la **velocidad**, en los **costes** de alojamiento (esto se debe a que ahora la aplicación puede residir completamente en un CDN), y una **mejor experiencia de usuario** y a su vez una **mejora considerable en el posicionamiento SEO** debido a que se mejoran las métricas de *First Paint* (FP), *First Contentful Paint* (FCP), and *Time To Interactive* (TTI), en el siguiente [artículo](#) se explica más en profundidad este último punto.

Al igual que en SSR, con esta alternativa desaparecen **la mayoría de las preocupaciones de seguridad en la parte frontend de las aplicaciones**, no se necesitan conexiones a bases de datos ni servidores corriendo, la información nunca va al frontend ya que nunca es solicitada, no

hay posibilidad de inyección de código, ni nada que sea causado por vulnerabilidades del código, se aumenta la resiliencia contra ataques DDoS. Es por todo ello por lo que esta opción mejora considerablemente la confianza en la aplicación, mejorando la seguridad y la privacidad de esta. El único problema de seguridad que plantea esta tecnología surge de la manera de realizar el renderizado en la parte del servidor, y es que, si la aplicación utiliza el método de stringify de JSON para la conversión de datos a objetos, existe la posibilidad de que la aplicación sufra ataques en la parte del servidor produciendo fugas de datos. La desventaja de esta opción es la escalabilidad de la aplicación, ya que, si la aplicación tiene que mostrar páginas muy extensas con muchos elementos puede tomar mucho tiempo realizar este pre renderizado, por ello, esta opción se suele recomendar para aplicaciones que por lo general no tienen mucha funcionalidad, ni contiene contenido dinámico, un ejemplo claro sería una *landing page*.

## 5.2.4 Conclusión

Por lo general, la decisión a tomar va a depender bastante del tipo de aplicación que se quiera desarrollar y no tanto de la seguridad que ofrece cada alternativa. Las aplicaciones en las que es buena decisión realizar un *pre-rendering* (**SSR**, **SSG**) son aquellas en las que el principal objetivo sea mostrar información de manera rápida, con una buena UX y con un buen posicionamiento SEO, en cambio, en aquellas aplicaciones en las que necesitamos que el contenido mostrado se vaya actualizando sin necesidad de un servidor, es mejor opción **CSF**.

Hoy en día existen múltiples frameworks como [Next.js](#) o [Nuxt](#), que permiten hacer un modelo híbrido en el que puedes determinar que contenido es SSR o SSG, y cual no, esta opción cada vez es más popular debido a los beneficios y ventajas que presenta, y en cuestiones de seguridad es posible que este modelo híbrido también sea buena elección de cara al futuro de las aplicaciones web.

## 5.3 Híbridas vs Nativas

Uno de los campos cada vez más extendidos dentro del desarrollo frontend es el de las aplicaciones híbridas, ya sea una aplicación de escritorio o una aplicación móvil, hay situaciones en las que optar por una aplicación híbrida es la mejor opción, pero en otras ocasiones no lo es. A continuación, vamos a analizar cuáles son las ventajas y desventajas de cada una de las alternativas, y vamos a analizar los aspectos de seguridad de estas aplicaciones para poder saber qué decisión tomar a la hora de determinar la tecnología con la que desarrollar alguna de estas aplicaciones, pero antes de nada vamos a definir que es una aplicación híbrida. Una aplicación híbrida no es más que una aplicación web que funciona sobre el navegador web nativo del dispositivo sobre el que se está ejecutando.

### 5.3.1 Ventajas y desventajas

En rasgos generales las **ventajas** que ofrecen las apps híbridas frente a las nativas son:

- **Time to market:** el tiempo que se tarda en lanzar una aplicación para ser utilizada por los usuarios. Al disponer de una única base del código y poder generar las diferentes versiones de los distintos sistemas operativos, nos ahorramos el tiempo de desarrollar cada una de ellas.
- **Coste:** al igual que nos ahorramos el tiempo de desarrollo de cada una de las versiones, también nos ahorramos el coste de desarrollarlas, y lo mismo con el desarrollo de las posibles actualizaciones y bugs que surjan.

- **Escalabilidad:** fácil de escalar a múltiples plataformas, no se requiere desarrollar de nuevo la aplicación para cada una de las nuevas plataformas que se desee incluir.
- **Mantenibilidad:** ante nuevos evolutivos o bugs no es necesario desarrollarlos para cada una de las plataformas a las que se da soporte.

Por otro lado, las **desventajas** que nos encontramos son:

- **Rendimiento:** tanto la carga como el funcionamiento en general de las aplicaciones híbridas es más lento que el de las nativas.
- **User Experience (UX):** por lo general, la experiencia de usuario en las aplicaciones híbridas se ve afectado debido a las limitaciones de la tecnología que se utiliza para su desarrollo.
- **Funcionalidad:** las posibilidades que se tienen a la hora de elegir una opción híbrida en muchas ocasiones se ven limitadas, hay ciertas características propias de cada plataforma a las que no se tiene acceso.
- **Navegador:** las apps híbridas dependen del navegador web de la plataforma (esto únicamente afecta a las apps que son puramente híbridas).

Existen otras alternativas, como [React Native](#), o [Electron](#), que tratan de juntar lo mejor de los dos mundos, por un lado, se desarrollan utilizando tecnologías web como JavaScript, ofreciendo la posibilidad de obtener a través de una única base del código las diferentes versiones para las distintas plataformas, y por otro lado, en el caso de alternativas como React Native tenemos las mismas ventajas de las apps nativas ya que el código desarrollado finalmente es transformado a código nativo permitiendo que actúe como una aplicación nativa cualquiera, pudiendo acceder a aquellas características propias de cada plataforma.

### 5.3.2 Conclusiones

En términos de seguridad por lo general las aplicaciones híbridas suelen ser menos seguras que las nativas ya que la superficie expuesta es mayor, esto se debe a que aparte de las vulnerabilidades que pueda tener la propia aplicación, las híbridas, traen consigo las vulnerabilidades del lenguaje en el que están desarrolladas, y las vulnerabilidades que afectan directamente a los navegadores web, ya que las apps híbridas se ejecutan sobre el navegador del dispositivo. En el caso de las herramientas como React Native reducimos bastante el conjunto de vulnerabilidades limitándolo únicamente a las vulnerabilidades asociadas al lenguaje y a la librería, sin tener que preocuparnos de la seguridad del navegador.

Todo esto no quiere decir que una aplicación híbrida tenga que ser menos segura que una nativa, pero sí que implica que tengamos que aplicar los mismos mecanismos de seguridad que aplicaríamos a una aplicación web. Otro punto bastante importante a la hora de proteger estas aplicaciones es tener constantemente actualizada la versión de la plataforma híbrida ya que está será la principal responsable de encargarse de aquellos aspectos más relacionados con la seguridad del dispositivo que lanza la aplicación.

Si nos encontramos en la posición de tener que tomar una decisión de con que tecnología va a ser desarrollada una aplicación, si con tecnología web o con tecnología nativa, deberemos tener en cuenta los siguientes puntos para poder tomar la decisión:

- El **tiempo** del que se dispone, ya que no es lo mismo que se tengan apenas unos pocos meses para desarrollar la aplicación y subirla a las tiendas, que tener tiempo más que

suficiente para llevarlo a cabo. Cuando nos encontramos en el caso de que el tiempo es una variable importante y que **no disponemos de demasiado tiempo**, la mejor opción es desarrollar una app híbrida ya que el tiempo de puesta en producción es mucho menor, pero en cambio si disponemos de **tiempo suficiente** como para desarrollar la aplicación como es debido, la mejor opción es hacer una **app nativa**.

- Otro de los aspectos a tener en cuenta es el **presupuesto**, cosa que suele ir ligada al tiempo y, por lo tanto, si disponemos de un **presupuesto ajustado** la mejor opción será una **aplicación híbrida**, en cambio si disponemos de **presupuesto suficiente**, sin duda es mejor opción una **app nativa**.
- La **frecuencia de los cambios** también influye a la hora de tomar la decisión y es que, si se espera que la aplicación esté requiriendo actualizarse constantemente, será mejor desarrollar una app híbrida ya que el coste de estas actualizaciones es menor, pero como hemos comentado en el anterior punto, si el presupuesto no es ajustado y disponemos de suficiente tiempo para realizar dichas actualizaciones será mejor opción una app nativa.
- Finalmente, uno de los puntos más importantes es determinar si la aplicación va a disponer de **conexión a internet**, porque en el caso de las apps híbridas la conexión a internet es un requisito principal, ya que estas requieren constantemente el estar conectadas a internet, es por ello que si la aplicación va a funcionar sobre un dispositivo sin conexión a internet la **única** opción es desarrollar una **app nativa**.

A la hora de empezar a desarrollar una aplicación siempre va a haber debate en que tecnología utilizar ya que en función de cuales sean los intereses de cada persona será mejor una opción u otra, pero en términos de seguridad hay que tener claro que **por defecto** una aplicación híbrida es menos segura que una nativa, pero eso no excluye que se pueda desarrollar una aplicación híbrida segura aplicando los mecanismos de seguridad pertinentes. Lo que sí que es aconsejable es que si lo que se desea es una opción híbrida debido al tiempo o alguna de las razones vistas previamente, si tenemos la posibilidad de utilizar una tecnología como Electron o React Native, en términos de seguridad siempre va a ser más eficiente que una opción totalmente híbrida.

## 5.4 DevSecOps

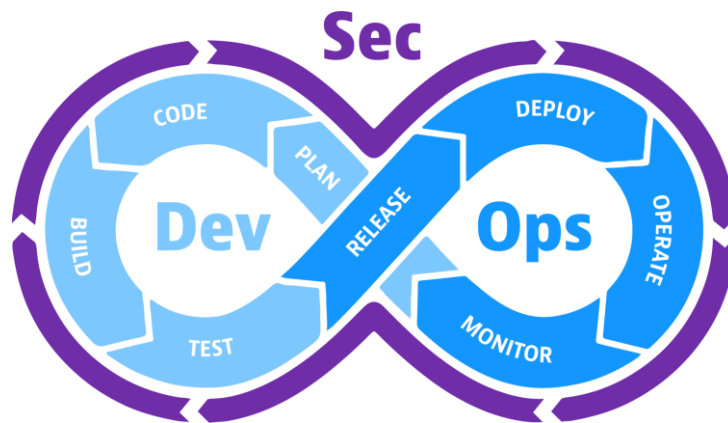


ILUSTRACIÓN 39 DevSecOps SOURCE: [HTTPS://WWW.DYNATRACE.COM/NEWS/BLOG/WHAT-IS-DEVSECOPS/](https://www.dynatrace.com/news/blog/what-is-devsecops/)

*"El propósito y la intención de DevSecOps es crear sobre la mentalidad de que todo el mundo es responsable por la seguridad, con el objetivo de distribuir de manera segura las **decisiones de seguridad** rápidamente y escalar para aquellos que mantienen el nivel más alto de contexto, sin sacrificar la seguridad necesaria"*

(Shannon Lietz, coautora del "Manifiesto DevSecOps")

Es muy importante que a la hora de diseñar el ciclo de vida de una aplicación web se tengan en cuenta ciertas consideraciones de seguridad, y es tanta la importancia de estas consideraciones que con el paso del tiempo ha ido emergiendo un nuevo perfil entre los equipos de desarrollo, con el principal objetivo de preocuparse acerca de las consideraciones de seguridad de las aplicaciones que hay que tener en cuenta en los ciclos de vida.

Normalmente estos criterios de seguridad eran revisados por un equipo de QA en cada publicación de una nueva versión de las aplicaciones, pero, el problema viene cuando se adquiere una metodología Agile y DevOps, ya que, en esta metodología la integración del código y del despliegue de versiones se hace de manera incremental, y cuando estas tareas de integración y despliegue son muy recurrentes el tema de la seguridad y la calidad supone un cuello de botella para los proyectos.

Por lo tanto, *Develop Security Operations (DevSecOps)* nació de la necesidad de disponer de un experto que tuviera conocimientos de estos 3 aspectos (desarrollo, seguridad, operaciones) del desarrollo de aplicaciones, con el principal objetivo de automatizar la integración de dichos aspectos de seguridad en las diferentes fases del ciclo de vida de las aplicaciones web.

Lo que conseguimos dotando de seguridad a cada una de las fases del ciclo de vida es que aquellos problemas de seguridad que puedan surgir a lo largo del desarrollo de una aplicación sean detectados en fases tempranas, donde son menos costosos de arreglar, consiguiendo así, agilizar el proceso y reducir costes de desarrollo. *DevSecOps* se ha convertido en uno de los principales pilares en las metodologías Agile que permite, no solo aumentar la velocidad de las entregas, sino también su calidad.



## 5.4.1 Buenas prácticas

### 5.4.1.1 Shift left & shift right

Uno de los conceptos más importantes según los *DevSecOps* es aplicar lo que denominan “**Desplazamiento a la izquierda**” (*Shift left*), esto consiste en **concienciar** a los desarrolladores de una aplicación de que la seguridad no se implanta al final de los proyectos sino al principio, por ello es necesario mover la seguridad de derecha (fin) a izquierda (inicio) del desarrollo de las aplicaciones, de tal manera que la seguridad se tenga en cuenta en fases de diseño donde aún no se han tomado decisiones, con esto, prevenimos ciertos problemas de seguridad identificándolos en fases tempranas, donde son más fácil de solucionar. Sin embargo, no hay que dejar de lado aquellas fases que se encuentran más a la derecha, y que tienen que ver con los despliegues de las aplicaciones en los diferentes entornos que tenga, ya que, es en estos entornos de producción donde más ataques se realizan y donde tendremos que poner el foco a la hora de monitorizar una aplicación, es por ello por lo que también es importante que los *DevSecOps* apliquen lo que se denomina “**Desplazamiento a la derecha**” (*Shift right*).

### 5.4.1.2 Capacitar al equipo

Para que todo esto funcione, todos los integrantes del equipo de desarrollo deben de disponer de unos mínimos **conocimientos** base relacionados con la seguridad en aplicaciones web, ya que, a la hora de integrar los diferentes mecanismos de seguridad por los *DevSecOps*, el resto del equipo debe entender lo esencial asociado a su parte de trabajo y así dar la posibilidad de mejorar la implantación de dichos mecanismos aportando ideas o consejos.

### 5.4.1.3 Organizar al equipo

Debido a que la figura del *DevSecOps* no es tan conocida y que en ocasiones es difícil de identificar, una buena práctica es **organizar al equipo** comunicando las **responsabilidades** de seguridad asociadas a los procesos de desarrollo de las aplicaciones, de manera que, cada integrante tenga claro cuál es su tarea dentro de su trabajo y así poder identificar aquellos puntos en los que es necesario aplicar los conocimientos de seguridad adquiridos.

### 5.4.1.4 Analizar los resultados

Como todo nuevo proceso en el desarrollo de un producto, es de interés saber si finalmente ha dado buenos resultados, para ello lo mejor es acudir a números para determinar si el nuevo proceso cumple los objetivos establecidos, esto es posible mediante la implantación de los siguientes mecanismos que permiten realizar dichas comprobaciones.

- **Trazabilidad:** implementar un mecanismo de trazabilidad que nos permita recopilar datos a lo largo del CI, como el registro de errores, por ejemplo.
- **Auditabilidad:** establecer controles de seguridad para garantizar el cumplimiento de los criterios de calidad establecidos al inicio de la implantación de seguridad.
- **Visibilidad:** dar visibilidad a los controles establecidos, por ejemplo, estableciendo alertas de seguridad.

### 5.4.1.5 Actualizarse

Como hemos visto a lo largo de este libro, la seguridad es algo vivo que crece a la misma velocidad que lo hace la tecnología IT, es por ello por lo que a medida que pasa el tiempo hay que seguir actualizándose para poder estar al corriente de las nuevas amenazas que puedan afectar a



las aplicaciones web, consultando la OWASP, los CVEs actuales, leyendo posts de seguridad actuales, etc.

Por ejemplo, para este año 2022 [dynatrace](#) ha compartido el siguiente [blog](#) con un top de las ocho tendencias actuales para los *DevSecOps*, y como este blog hay muchos otros que salen cada poco tiempo y que ayudan a ir actualizándose a medida que pasa el tiempo.

## 6 CONCLUSIONES

Como hemos podido observar, la seguridad, sin lugar a duda, es una parte muy importante en las aplicaciones web, ya que según han ido evolucionando, hemos podido ver que cada vez son más complejas y que los tipos de datos que manejan cada vez son más sensibles, y que las amenazas del OWASP Top 10 de 2021 parecen ser cada vez más peligrosas debido a la importancia de los datos. Por otro lado, hemos visto que los nuevos frameworks de desarrollo JavaScript utilizados para el desarrollo de aplicaciones front-end, también poseen múltiples vulnerabilidades de las que preocuparse y, que debido al constante crecimiento de la parte front-end de las aplicaciones, es necesario empezar a preocuparse aún más por la seguridad a la hora de diseñarlas. También, hemos visto cuales son los servicios más esenciales que ha de disponer una aplicación web y cuales son aquellas herramientas y buenas prácticas que nos facilitan su implantación. Finalmente, se han presentado cuales son aquellas decisiones que debemos tener en cuenta a la hora de diseñar una aplicación front-end.

Dentro de las decisiones a tomar en el diseño de las aplicaciones front-end, cabe destacar la importancia de tener la figura del *DevSecOps* en un equipo de desarrollo que posea una metodología de trabajo *Agile*, y la aparición de estas nuevas tecnologías de pre renderizado, las aplicaciones denominadas como “*Static Site Generation*”, suponen una gran mejora en temas de performance y de SEO, pero no solo eso, sino que con esta tecnología conseguimos reducir el área expuesta en la parte del cliente consiguiendo así reducir considerablemente las posibles amenazas.

Otra de las conclusiones más importantes a la que hemos podido llegar es, que la implantación de mecanismos de seguridad en las aplicaciones front-end han de pensarse en fases tempranas, donde no supone un sobrecoste implantarlas, y donde aún estamos a tiempo de enfrentar aquellas amenazas que en un futuro pueden ser mucho más grandes y posiblemente intratables, por ello, a la hora de diseñar las aplicaciones es muy importante que tengamos presentes todos los conceptos de seguridad vistos y estemos provistos de todos los posibles mecanismos que nos aseguren que se cumplen los servicios de seguridad necesarios para las aplicaciones.

# 7 REFERENCIAS

- [1]. A Brief History of Frontend Frameworks. (2018, 10 septiembre). YouTube. Recuperado 25 de mayo de 2022, de <https://www.youtube.com/watch?v=Kzeog8yTFaE>
- [2]. The evolution of front-end development. (2020, 24 marzo). CloudReports. Recuperado 25 de mayo de 2022, de <https://cloudreports.net/the-evolution-of-front-end-development/>
- [3]. Mezzalana, L. (s. f.). Building Micro-Frontends. O'Reilly Online Learning. Recuperado 25 de mayo de 2022, de <https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/ch04.html>
- [4]. Pineda, E. (2022, 19 febrero). The evolution of Frontend Development. Eduardo Pineda. Recuperado 25 de mayo de 2022, de <https://www.epineda.net/the-evolution-of-front-end-development/>
- [5]. GeeksforGeeks. (s. f.). Web Development. Recuperado 25 de mayo de 2022, de <https://www.geeksforgeeks.org/web-development/>
- [6]. Why svelte is revolutionary. (2020, 1 octubre). DEV Community. Recuperado 25 de mayo de 2022, de <https://dev.to/hanna/why-svelte-is-revolutionary-415e>
- [7]. Gawkowski, E. (2022, 13 mayo). React vs Svelte – Which Is Better For Your Business in 2022? Pagepro. Recuperado 25 de mayo de 2022, de <https://pagepro.co/blog/react-vs-svelte/>
- [8]. Hartman, J. (2022, 16 abril). React vs Angular: 10 Most Important Differences You Must Know! Guru99. Recuperado 25 de mayo de 2022, de <https://www.guru99.com/react-vs-angular-key-difference.html>
- [9]. Nowak, M. (s. f.). Vue vs React in 2022 - Which Framework to Choose and When? Monterail. Recuperado 25 de mayo de 2022, de <https://www.monterail.com/blog/vue-vs-react#projects>
- [10]. MVC - Glosario | MDN. (2020, 8 diciembre). Developer.Mozilla. Recuperado 25 de mayo de 2022, de <https://developer.mozilla.org/es/docs/Glossary/MVC>
- [11]. MVC - Glosario | MDN. (2020b, diciembre 8). MDN. Recuperado 26 de mayo de 2022, de <https://developer.mozilla.org/es/docs/Glossary/MVC>
- [12]. Modelo–vista–controlador. (2021, 27 septiembre). Wikipedia, la enciclopedia libre. Recuperado 26 de mayo de 2022, de <https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador>
- [13]. Martin, M. (2022, 17 marzo). MVC vs MVVM: Key Differences with Examples. Guru99. Recuperado 26 de mayo de 2022, de <https://www.guru99.com/mvc-vs-mvvm.html>
- [14]. Patil, H. (2021, 11 diciembre). Contemporary Front-end Architectures - webf. Medium. Recuperado 26 de mayo de 2022, de <https://blog.webf.zone/contemporary-front-end-architectures-fb5b500b0231>
- [15]. Zeynalli, A. (2022, 17 enero). Software Architecture Patterns for Front-End Development. Medium. Recuperado 26 de mayo de 2022, de <https://azeynalli1990.medium.com/software-architecture-patterns-for-front-end-development-9e43e43cdfb3>
- [16]. 9 patrones de diseño que el front-end necesita conocer - programador clic. (s. f.). programmerclick. Recuperado 26 de mayo de 2022, de <https://programmerclick.com/article/79831663709/>
- [17]. Patrones de diseño / Design patterns. (s. f.). Refactoring Guru. Recuperado 26 de mayo de 2022, de <https://refactoring.guru/es/design-patterns/>

- [18]. Ferry, C. (2019, 27 noviembre). Singleton - Design Patterns meet the Frontend. DEV Community. Recuperado 26 de mayo de 2022, de <https://dev.to/coly010/singleton-design-patterns-meet-the-frontend-12m1>
- [19]. Bala, S. (2021, 22 junio). 3 Design Patterns in TypeScript for Frontend Developers. OpenReplay Blog. Recuperado 26 de mayo de 2022, de <https://blog.openreplay.com/3-design-patterns-in-typescript-for-frontend-developers/>
- [20]. JavaScript | MDN. (2022, 2 febrero). Developer Mozilla. Recuperado 26 de mayo de 2022, de <https://developer.mozilla.org/es/docs/Web/JavaScript>
- [21]. The starting point for learning TypeScript. (s. f.). Typescriptlang. Recuperado 26 de mayo de 2022, de <https://www.typescriptlang.org/docs/>
- [22]. npm Docs. (s. f.). Npm. Recuperado 26 de mayo de 2022, de <https://docs.npmjs.com/>
- [23]. Learning React Native. (s. f.). O'Reilly Online Learning. <https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html>
- [24]. React Native · Learn once, write anywhere. (s. f.). React Native. Recuperado 26 de mayo de 2022, de <https://reactnative.dev/>
- [25]. Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS. (s. f.). Electron. Recuperado 26 de mayo de 2022, de <https://www.electronjs.org/>
- [26]. OWASP Top 10:2021. (s. f.). Owasp. Recuperado 26 de mayo de 2022, de <https://owasp.org/Top10/>
- [27]. Jiménez, F. J. (2022, 13 abril). Novedades de OWASP Top 10 2021 (I). Security Art Work. Recuperado 26 de mayo de 2022, de <https://www.securityartwork.es/2021/12/22/novedades-de-owasp-top-10-2021-i/>
- [28]. Top 10 vulnerabilidades web de 2021. (2022, 18 enero). INCIBE. Recuperado 26 de mayo de 2022, de <https://www.incibe.es/protege-tu-empresa/blog/top-10-vulnerabilidades-web-2021>
- [29]. OWASP publica el Top 10 - 2021 de riesgos de seguridad en. (2021, 30 septiembre). INCIBE-CERT. Recuperado 26 de mayo de 2022, de <https://www.incibe-cert.es/alerta-temprana/bitacora-ciberseguridad/owasp-publica-el-top-10-2021-riesgos-seguridad-aplicaciones>
- [30]. OWASP – Seguridad en la web. (2020, 24 enero). Ciberseguridad. Recuperado 26 de mayo de 2022, de <https://Ciberseguridad.com/guias/desarrollo-seguro/owasp/#%C2%BFQue es OWASP>
- [31]. colaboradores de Wikipedia. (2021, 1 febrero). Common Weakness Enumeration. Wikipedia, la enciclopedia libre. Recuperado 26 de mayo de 2022, de [https://es.wikipedia.org/wiki/Common\\_Weakness\\_Enumeration](https://es.wikipedia.org/wiki/Common_Weakness_Enumeration)
- [32]. CWE - About - CWE Overview. (s. f.). CWE. Recuperado 26 de mayo de 2022, de <https://cwe.mitre.org/about/index.html>
- [33]. Tal, L. (2022, 19 mayo). OWASP Top 10 | OWASP Top 10 Vulnerabilities 2021. Snyk. Recuperado 26 de mayo de 2022, de <https://snyk.io/learn/owasp-top-10-vulnerabilities/>
- [34]. OWASP Application Security Verification Standard | OWASP Foundation. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de <https://owasp.org/www-project-application-security-verification-standard/>
- [35]. Usage Statistics and Market Share of JavaScript for Websites, May 2022. (s. f.). W3techs. Recuperado 26 de mayo de 2022, de <https://w3techs.com/technologies/details/pl-js>
- [36]. Crashtest Security GmbH. (2022, 24 mayo). **【Secure your JavaScript Vulnerabilities】** Web Security Guide. Crashtest Security. Recuperado 26 de mayo de 2022, de <https://crashtest-security.com/javascript-vulnerabilities/>

- [37]. Tal, L. (2021, 14 noviembre). 10 npm Security Best Practices. Snyk. Recuperado 26 de mayo de 2022, de <https://snyk.io/blog/ten-npm-security-best-practices/>
- [38]. Miller, A. (2022, 19 mayo). JavaScript Security | JavaScript Vulnerabilities. Snyk. Recuperado 26 de mayo de 2022, de <https://snyk.io/learn/javascript-security/>
- [39]. NPM Security - OWASP Cheat Sheet Series. (s. f.). Owasp. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/NPM\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html)
- [40]. Miller, A. (2021, 18 noviembre). Cross Site Request Forgery (CSRF). Snyk. Recuperado 26 de mayo de 2022, de <https://snyk.io/learn/csrf-cross-site-request-forgery/>
- [41]. Content Security Policy (CSP) - HTTP | MDN. (2022, 30 abril). Developer Mozilla. Recuperado 26 de mayo de 2022, de <https://developer.mozilla.org/en-US/docs/web/HTTP/CSP>
- [42]. Swadia, S. (2021, 2 noviembre). How to Secure Your React.js Application. freeCodeCamp.Org. Recuperado 26 de mayo de 2022, de <https://www.freecodecamp.org/news/best-practices-for-security-of-your-react-js-application/>
- [43]. Bilgili, D. (2021, 19 agosto). Using dangerouslySetInnerHTML in a React application. LogRocket Blog. Recuperado 26 de mayo de 2022, de <https://blog.logrocket.com/using-dangerouslysetinnerhtml-in-a-react-application/>
- [44]. What are XXE(XML External Entity) Attacks? - Security Souls | 2022. (2021, 3 febrero). Security Souls. Recuperado 26 de mayo de 2022, de <https://securitysouls.com/what-are-xxe-xml-external-entity-attacks/>
- [45]. What is XXE (XML external entity) injection? Tutorial & Examples | Web Security Academy. (s. f.). portswigger. Recuperado 26 de mayo de 2022, de <https://portswigger.net/web-security/xxe>
- [46]. Li, V. (2022, 5 marzo). Angular + React: Vulnerability Cheatsheet - ShiftLeft Blog. Medium. Recuperado 26 de mayo de 2022, de <https://blog.shiftleft.io/angular-react-vulnerability-cheatsheet-a3b36f22a0fd>
- [47]. Tal, L. (2021a, noviembre 14). 6 Angular Security Best Practices | Cheat Sheet. Snyk. Recuperado 26 de mayo de 2022, de <https://snyk.io/blog/angular-security-best-practices/>
- [48]. Security Vue.js. (s. f.). Vuejs. Recuperado 26 de mayo de 2022, de <https://v2.vuejs.org/v2/guide/security.html>
- [49]. Server-Side Rendering (SSR) | Vue.js. (s. f.). Vue.Js. Recuperado 26 de mayo de 2022, de <https://vuejs.org/guide/scaling-up/ssr.html#cross-request-state-pollution>
- [50]. Svelte. (s. f.). Svelte. Recuperado 26 de mayo de 2022, de <https://svelte.dev/>
- [51]. Svelte from a React developer's perspective: The Basics. (2020, 21 noviembre). Delvalle. Recuperado 26 de mayo de 2022, de <https://delvalle.dev/posts/svelte-from-react-perspective-basics/svelte-from-react-perspective-part-1/>
- [52]. Typosquatting: cuidado con lo que escribes. (2021, 3 junio). INCIBE. Recuperado 26 de mayo de 2022, de <https://www.incibe.es/aprendeciberseguridad/typosquatting>
- [53]. Input Validation - OWASP Cheat Sheet Series. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Input\\_Validation\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html)
- [54]. Application Security Training For Developers. (s. f.). Kontra. Recuperado 26 de mayo de 2022, de <https://application.security/free/kontra-front-end-top-10>
- [55]. Vachhani, H. (2022, 6 enero). Security Aspects to consider for a React Native Application. Medium. Recuperado 26 de mayo de 2022, de <https://medium.com/simform-engineering/security-aspects-to-consider-for-a-react-native-application-95556f0e4244>
- [56]. Seguridad | Electron. (s. f.). Electron. Recuperado 26 de mayo de 2022, de <https://www.electronjs.org/es/docs/latest/tutorial/security>

- [57]. Security · React Native. (2022, 4 abril). React Native Dev. Recuperado 26 de mayo de 2022, de <https://reactnative.dev/docs/security>
- [58]. <https://www.redeszzone.net/tutoriales/seguridad/diferencias-autenticacion-autorizacion/>
- [59]. Authentication - OWASP Cheat Sheet Series. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Authentication\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html)
- [60]. Authorization - OWASP Cheat Sheet Series. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Authorization\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html)
- [61]. Authentication - OWASP Cheat Sheet Series. (s. f.-b). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Authentication\\_Cheat\\_Sheet.html#authentication-general-guidelines](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#authentication-general-guidelines)
- [62]. Forgot Password - OWASP Cheat Sheet Series. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Forgot\\_Password\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html)
- [63]. Multifactor Authentication - OWASP Cheat Sheet Series. (s. f.). OWASP. Recuperado 26 de mayo de 2022, de [https://cheatsheetseries.owasp.org/cheatsheets/Multifactor\\_Authentication\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html)
- [64]. OpenID Connect | Google Identity |. (s. f.). Google Developers. Recuperado 26 de mayo de 2022, de <https://developers.google.com/identity/protocols/oauth2/openid-connect>
- [65]. Hunter, A. (2021, 23 septiembre). OAuth vs SAML vs OpenID: Learn the Differences between Them. Parallels Remote Application Server Blog - Application Virtualization, Mobility and VDI. Recuperado 26 de mayo de 2022, de <https://www.parallels.com/blogs/ras/oauth-vs-saml-vs-openid/>
- [66]. FIDO authentication with passkeys | Google Identity |. (s. f.). Google Developers. Recuperado 26 de mayo de 2022, de <https://developers.google.com/identity/fido>
- [67]. FIDO Alliance. (2022, 19 mayo). FIDO2. Recuperado 26 de mayo de 2022, de <https://fidoalliance.org/fido2/>
- [68]. A demonstration of the WebAuthn specification. (s. f.). WebAuthn.io. Recuperado 26 de mayo de 2022, de <https://webauthn.io/>
- [69]. WORKSHOP: Authenticating your web like a boss. (2018, 26 enero). Slides. Recuperado 26 de mayo de 2022, de <https://slides.com/fidoalliance/jan-2018-fido-seminar-webauthn-tutorial>
- [70]. Remdt, A. (2021, 17 marzo). Handling User Permissions in JavaScript. CSS-Tricks. Recuperado 26 de mayo de 2022, de <https://css-tricks.com/handling-user-permissions-in-javascript/>
- [71]. Klimenchenko, E. (2021, 4 junio). Front-End Testing is For Everyone. CSS-Tricks. Recuperado 26 de mayo de 2022, de <https://css-tricks.com/front-end-testing-is-for-everyone/>
- [72]. Testing JavaScript with Kent C. Dodds. (s. f.). Testingjavascript. Recuperado 26 de mayo de 2022, de <https://testingjavascript.com/>
- [73]. Testing Pipeline 101 For Frontend Testing. (2022, 21 febrero). Smashing Magazine. Recuperado 26 de mayo de 2022, de <https://www.smashingmagazine.com/2022/02/testing-pipeline-101-frontend-testing/>
- [74]. Continuous integration - CI. (s. f.). CircleCI. Recuperado 26 de mayo de 2022, de <https://circleci.com/continuous-integration/#how-to-get-started-with-cicd>



- [75]. CR M. Á. (2021, 19 julio). ¿Qué son los feature flags? DEV Community. Recuperado 26 de mayo de 2022, de <https://dev.to/marianocodes/que-son-los-feature-flags-kc7>
- [76]. Sanchez, C. (2021, 7 diciembre). Instantly releasing features in a world that never sleeps, using Feature Flags. Medium. Recuperado 26 de mayo de 2022, de <https://medium.com/crowdbotics/instantly-releasing-features-in-a-world-that-never-sleeps-using-feature-flags-e0e9e6c56d26>
- [77]. Hodgson, P. (s. f.). Feature Toggles (aka Feature Flags). martinfowler.com. Recuperado 26 de mayo de 2022, de <https://martinfowler.com/articles/feature-toggles.html>
- [78]. Gorej, V. (2020, 6 julio). How to use npm audit with Continuous Integration in 3 simple steps. Linkedin. Recuperado 26 de mayo de 2022, de <https://www.linkedin.com/pulse/how-use-npm-audit-continuous-integration-3-simple-steps-gorej>
- [79]. npm-audit | npm Docs. (s. f.). Npmjs. Recuperado 26 de mayo de 2022, de <https://docs.npmjs.com/cli/v8/commands/npm-audit/>
- [80]. Best practices for scanning images. (2022, 25 mayo). Docker Documentation. Recuperado 26 de mayo de 2022, de <https://docs.docker.com/develop/scan-images/>
- [81]. Powell, R. (2022, 18 febrero). How to secure your CI pipeline. CircleCI. Recuperado 26 de mayo de 2022, de <https://circleci.com/blog/secure-ci-pipeline/>
- [82]. D'Onofrio, F. (2021, 11 diciembre). Everything you need to know: human error in UX design. Medium. Recuperado 26 de mayo de 2022, de <https://uxdesign.cc/human-error-bf3a79cc5d37>
- [83]. Ermigiotti, L. (2022, 19 marzo). 7 Ways to Use UX Design to Enhance User Data Security. Codemotion Magazine. Recuperado 26 de mayo de 2022, de <https://www.codemotion.com/magazine/frontend/design-ux/ux-design-enhance-data-security/>
- [84]. Bittner, M. (2021, 16 diciembre). The ultimate UX Research cheat sheet - UX Collective. Medium. Recuperado 26 de mayo de 2022, de <https://uxdesign.cc/the-ultimate-ux-research-cheat-sheet-b70862c086a6>
- [85]. UX Research Cheat Sheet. (s. f.). Nielsen Norman Group. Recuperado 26 de mayo de 2022, de <https://www.nngroup.com/articles/ux-research-cheat-sheet/>
- [86]. Yildirim, I. (2019, 24 abril). Error prevention: One of the Usability Heuristics. DEV Community. Recuperado 26 de mayo de 2022, de <https://dev.to/izzet/error-prevention-one-of-the-usability-heuristics-2l4b>
- [87]. Hall, R. (2022, 4 mayo). A Look At Security: Hybrid Apps vs. Native Apps. MindSea: App Design & Dev Agency, Focusing on HealthTech. Recuperado 26 de mayo de 2022, de <https://mindsea.com/security-hybrid-apps-vs-native-apps/>
- [88]. Dhaduk, H. (2022, 16 mayo). Native vs Hybrid App: What's the Best Approach? Insights on Latest Technologies - Simform Blog. Recuperado 26 de mayo de 2022, de <https://www.simform.com/blog/native-vs-hybrid-app/#Section34>
- [89]. Dragomir, B. (2022, 6 enero). React Native: Under the Hood - Better Programming. Medium. Recuperado 26 de mayo de 2022, de <https://betterprogramming.pub/react-native-under-the-hood-281df5f548f>
- [90]. P., A. (2021, 11 mayo). What is the difference between React Native and Hybrid Apps? Linkedin. Recuperado 26 de mayo de 2022, de <https://www.linkedin.com/pulse/what-difference-between-react-native-hybrid-apps-abhishek-pareek>
- [91]. Marshall, T. (2021, 21 junio). Pre-rendered, server-rendered, or hybrid: Which should I use? Kontent by Kentico. Recuperado 26 de mayo de 2022, de <https://kontent.ai/blog/pre-rendered-server-rendered-or-hybrid-which-should-i-use/>

- [92]. What Is Server Side Rendering | SSR Pros & Cons. (2022, 11 mayo). Prerender. Recuperado 26 de mayo de 2022, de <https://prerender.io/what-is-srr-and-why-do-you-need-to-know/>
- [93]. Ibsen, M. (2022, 6 mayo). 3 Ways of Rendering on the Web | by Marius Ibsen | Compendium. Medium. Recuperado 26 de mayo de 2022, de <https://medium.com/compendium/3-ways-of-rendering-on-the-web-4363864c859e>
- [94]. Ibsen, M. (2022a, mayo 3). Benefits of Static Site Generation | by Marius Ibsen | Dfind Consulting. Medium. Recuperado 26 de mayo de 2022, de <https://medium.com/dfind-consulting/benefits-of-static-websites-5fb187d1ffe6>
- [95]. Troutman, J. (2022, 30 abril). The 3 Types of Rendering in Next.js | JavaScript in Plain English. Medium. Recuperado 26 de mayo de 2022, de <https://javascript.plainenglish.io/the-three-types-of-rendering-in-next-js-bd6f780270ac>
- [96]. DevSecOps. (2021, 9 septiembre). ibm. Recuperado 26 de mayo de 2022, de <https://www.ibm.com/co-es/cloud/learn/devsecops>
- [97]. Marsal, J. (2022, 10 marzo). What is DevSecOps? And what you need to do it well. Dynatrace News. Recuperado 26 de mayo de 2022, de <https://www.dynatrace.com/news/blog/what-is-devsecops/>