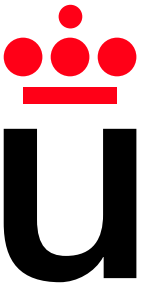


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

**Grado en Ingeniería Informática**

**Curso 2022-2023**

**Trabajo Fin de Grado**

# COWORDLE, UN JUEGO COMPETITIVO ONLINE

**Autor: Jorge Moreno Fernández**
**Tutor: Michel Maes Bermejo**
**Cotutor: Micael Gallego Carrillo**

# Agradecimientos

Lo primero quiero agradecer a Michel, mi tutor, por guiarme y enseñarme como se prepara un proyecto de principio a fin. Estoy muy contento con el trabajo que hemos realizado juntos.

También quiero agradecer a mis amigos, Lei, Miguel, Jorge y Christian, que me han ayudado a comprobar que todo funcionaba, que esperaban a que subiera un fix 'rápido' para poder seguir jugando y que me han ayudado a que la interfaz de la aplicación sea más intuitiva.

Y por supuesto agradecer a mi madre, que me ha apoyado durante todos estos años, me ha guiado cuando me encontraba perdido y me ha dado fuerzas para poder terminar la carrera y el TFG.

# Resumen

En esta memoría se va a documentar todo el trabajo realizado para el desarrollo de la aplicación CoWordle, un juego web accesible desde cualquier dispositivo, realizado utilizando las últimas tecnologías web.

CoWordle ha sido la oportunidad perfecta para aprender tecnologías web como WebSockets o Deno, y para realizar un juego multijugador accesible desde cualquier dispositivo que combina el famoso juego Wordle con la competición en tiempo real.

La aplicación cuenta con dos proyectos, por un lado la *webapp*, que permite la visualización de la aplicación utilizando navegadores web. El otro proyecto es el servidor de WebSockets, que realiza la comunicación en tiempo real para todos los jugadores de una partida.

A lo largo de este documento se explican las tecnologías usadas, como SvelteKit o Deno, también se explican aquellos sistemas que se hayan ido creando durante la realización de la aplicación, como los sistemas de tipados para la comunicación Websocket.

Describiremos los objetivos y el avance de la aplicación a lo largo del tiempo, así como la arquitectura escogida, que cuenta con dos proyectos distintos, una webapp y un servidor de WebSockets. Se explicará el diseño visual inspirado por la aplicación de Wordle original. Finalmente se describirán cuales son las posibilidades para desplegar la aplicación utilizando GitHub y utilizando Docker.

**Palabras clave**:

- Javascript
- Typescript
- Web
- Deno
- Node
- Socket.IO
- SvelteKit
- Wordle

# Índice de contenidos

# Índice de figuras

# Índice de códigos

# 1 Introducción

## 1.1. Contexto y alcance

Este trabajo representa la finalización del Grado de Ingeniería Informática, en el cual, se demostrarán algunas de las habilidades obtenidas a lo largo de los cursos, así como la capacidad de aprendizaje que se obtiene tras adquirir una amplía base de conocimientos.

La realización de CoWordle se centra en el aprendizaje por parte del autor de algunas tecnologías web, como la utilización de WebSockets como método de comunicación entre cliente y servidor. Además, permite aprender también tecnologías como Docker, que permiten desplegar la aplicación de manera sencilla, independientemente del lugar de despliegue.

La tecnología de WebSockets es una tecnología que se estudia brevemente en el grado, pero no se llega a utilizar de manera práctica.

El objetivo de este trabajo es aprender a utilizar WebSockets, una tecnología que permite comunicar el cliente y el servidor de manera bidireccional, y que se utiliza en aplicaciones de tiempo real, como juegos online, aplicaciones de mensajería web o como base para otras tecnologías como WebRTC.

**CoWordle** es el juego web multijugador derivado de Wordle desarrollado para este trabajo. CoWordle añade a la versión original la capacidad de desafiar a otros jugadores y competir en tiempo real jugando una partida de Wordle simultánea.

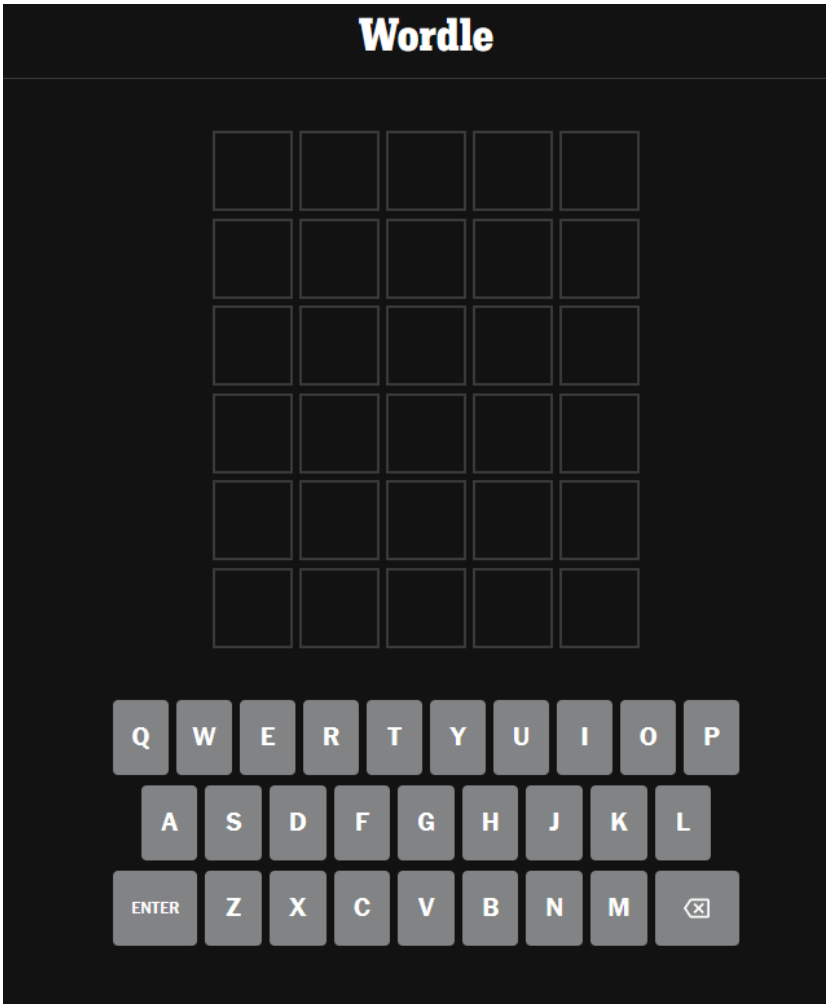



Figura 1.1: Aplicación oficial de Wordle.

## 1.2. Wordle

**Wordle** (Figura 1.1) es un juego de navegador creado por el ingeniero Josh Wardle durante la pandemia de COVID-19. Desde el lanzamiento Wordle ha conseguido que millones de jugadores se conecten cada día para encontrar una palabra objetivo.

### 1.2.1. La historia de Wordle

En sus comienzos Wordle solo era jugado por los amigos y familiares de Josh Wardle (Junio 2021), más tarde, Wardle publicó su juego en internet alcanzando menos de 100 jugadores durante las primeras semanas. Sin embargo, en enero de 2022 ya contaba con más de 300.000 jugadores cada día, y para finales de ese mismo mes ya jugaban millones de personas.

A finales de Enero de 2022, con la espectacular recepción del juego, The New York Times compró el juego por una cifra que no se ha publicado, pero que según el propio Wardle, ronda las 7 cifras.

En el informe de ganancias trimestrales de Marzo de 2022, The New York Times anunció que la compra de Wordle había traído a millones de jugadores a su página, y además muchos de estos nuevos jugadores, habían probado otros juegos ofrecidos por el noticiario.

### 1.2.2. ¿Comó se juega?

Wordle es un juego en el que cada día, todos los jugadores del mundo tienen que descubrir la misma palabra. Cada jugador tiene seis oportunidades para encontrar la palabra secreta, y cada oportunidad ofrece pistas sobre lo bien que has colocado cada letra del intento.

El juego utiliza colores como pistas para cada letra del intento:

- El color gris indica que la letra no se encuentra en la palabra objetivo.
- El color naranja indica que la letra se encuentra en la palabra, pero no está en su posición correcta.
- El color verde indica que la letra se encuentra tanto en la palabra, como en la posición de la palabra correcta.

Cuando un jugador ha descubierto todas las letras de la palabra objetivo antes de quedarse sin intentos, el juego se termina y se muestra al jugador el resultado de la partida, mostrando entre otras estadísticas, el número de intentos requeridos.

Esta estadística ha sido uno de los grandes éxitos del juego, millones de jugadores intentan cada día conseguir la palabra en el menor número de intentos posibles, y comparan entre ellos los resultados con las integración de Facebook y Twitter de las que dispone la aplicación oficial.

### 1.2.3. Otras aplicaciones derivados de Wordle

Tras el éxito de Wordle, multitud de aplicaciones comenzaron a aparecer que clonaban las mismas mecánicas que Wordle, entre ellas, existen versiones en diferentes lenguajes o versiones donde tienes que resolver múltiples palabras simultáneamente como Octordle (Figura 1.2) o Dordle (Figura 1.3).

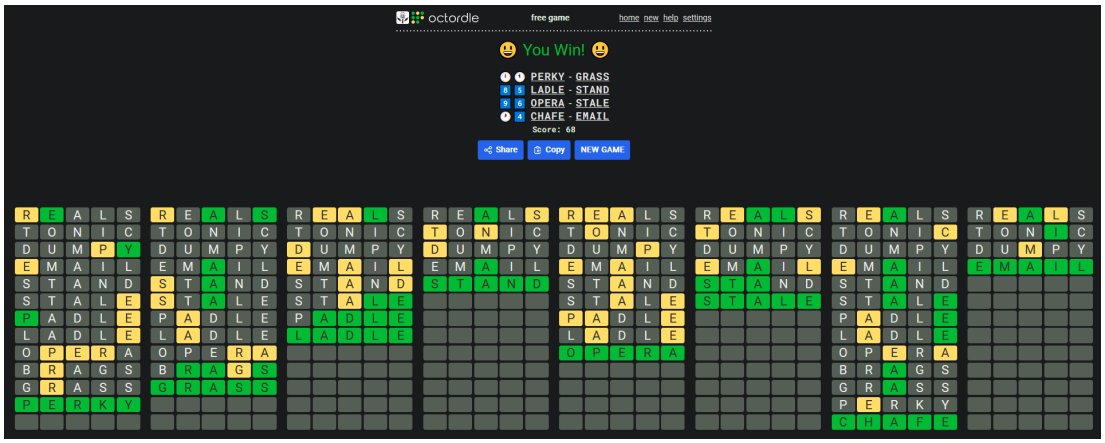



Figura 1.2: Aplicación Octordle.


Figura 1.3: Aplicación Dordle.


Figura 1.4: Aplicación Worldle.

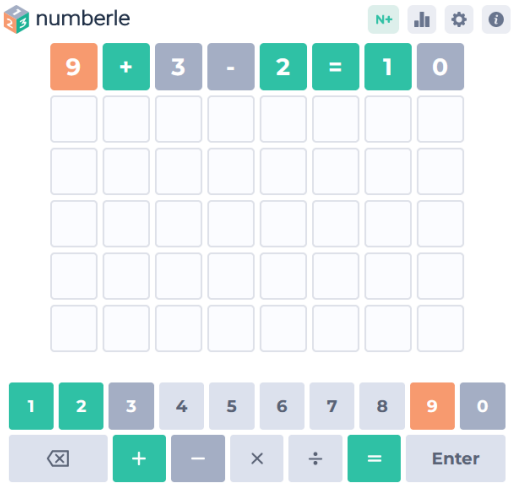



Figura 1.5: Aplicación Numberle.

A parte de inspirar multitud de juegos sobre palabras, existen variaciones como Globle, Worldle (Figura 1.4), en las que el objetivo es descubrir países utilizando las formas de estos, o Numberle (Figura 1.5), donde el objetivo el completar ecuaciones matemáticas.

## 1.3. CoWordle

CoWordle (Figura 1.6) añade otro *twist* a Wordle, añadiendo un valor competitivo que aparece cuando múltiples jugadores tienen que descubrir la misma palabra en tiempo real. Los jugadores se encuentran con dos problemas simultáneos, el primero es conseguir averiguar la palabra antes de que se agoten todos los

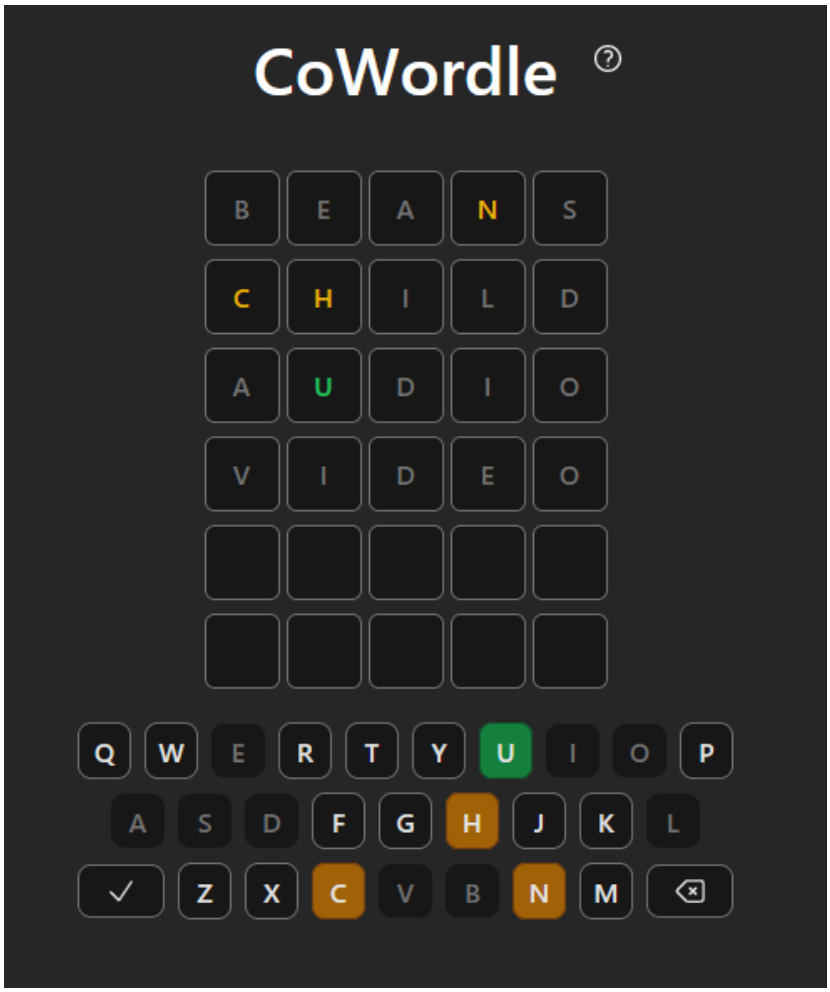



Figura 1.6: Vista de juego CoWordle.

intentos, y el segundo es hacerlo antes que lo consigan los demás jugadores.

Para poder hacer que los jugadores compitan simultáneamente es necesario alguna tecnología que los permita comunicarse en tiempo real, por ello, CoWordle presenta una muy buena oportunidad para aprender sobre cómo funcionan y cómo se usan las tecnologías de tiempo real, aunque para la realización de este proyecto se ha utilizado únicamente WebSockets.

# 2 Objetivos

## 2.1. Objetivos principales

Los objetivos principales son:

- El aprendizaje de la tecnología de WebSockets en su implementación de cliente y servidor.
- La creación de un juego multijugador dentro de una página web, manejo de estados de todos los jugadores de la partida y comunicación en tiempo real.
- Desarrollar una aplicación web que se pueda usar desde el navegador y desde el móvil, compartiendo el mismo código.
- Desplegar la aplicación utilizando herramientas como Docker y Docker Compose.

## 2.2. Objetivos secundarios

Los objetivos secundarios son:

- Realizar un sistema de comunicación entre front-end y back-end fuertemente tipado, utilizando Typescript.

- Utilizar un desarrollo por sustracción, eliminando partes del juego que no añaden valor, y haciendo que las partes que permanecen se sientan fluidas usando animaciones.

# 3

# Tecnologías, Herramientas y Metodologías

## 3.1. Tecnologías

### 3.1.1. Typescript

Typescript [1] es un lenguaje desarrollado por Microsoft que se compila a Javascript. Las principales funcionalidades que añade Typescript son: un robusto sistema de tipados y un completo sistema de genéricos.

Typescript reduce los tiempos de desarrollo permitiendo detectar errores de tipados mientras programas, además, ayuda al desarrollador utilizando autocompletado sobre miembros de clases, variables y funciones.

#### ¿Cómo se ha utilizado Typescript?

Typescript ha sido el único lenguaje de programación utilizado en toda la práctica. Gracias a los avances de Node/Deno, Typescript se ha podido utilizar tanto en el front-end, como en el back-end.

Además, en el trabajo se han creado sistemas específicos de ayuda sobre los tipados, así como para la auto compleción de, por ejemplo, las llamadas de web-sockets.

### 3.1.2. Node.js

Node.js [2] es probablemente una de las creaciones más influyentes en el mundo web, creada por Ryan Dahl en 2009, permite ejecutar código Javascript en el lado del servidor, sin necesidad de usar un navegador.

Node.js utiliza internamente el motor de Javascript (y WebAssembly) V8 desarrollado para Chromium por Google.

Una de las principales diferencias entre Node.js y otros sistemas como el motor de PHP, es que no es necesario levantar un proceso para cada llamada, en su lugar, Node.js es capaz de ir recibiendo llamadas e irlas respondiendo en un único hilo de ejecución pero con un sistema de eventos que permite responder varias peticiones asíncronamente, esto le permite responder mucho más rápido y eficientemente que otros sistemas.

En 2022 y según la encuesta de StackOverflow [3] en la que participaron más de cuarenta y cinco mil personas, Node.js es la tecnología web más usada entre desarrolladores profesionales siendo usada por un 46.31 % de estos.

#### ¿Cómo se ha utilizado Node.js?

Aunque Node.js no se ha utilizado directamente en la práctica, sí que ha permitido indirectamente que el resto de herramientas hayan podido ser utilizadas, especialmente SvelteKit.

### 3.1.3. Deno

Deno [4] es una tecnología open-source alternativa a Node, creada por el mismo desarrollador que creó Node.js, Ryan Dahl, que presentó Deno en una charla llamada “10 things I Regret About Node.js” en la conferencia de Javascript que se estaba celebrando en Europa.

Los principales cambios respecto a Node.js son la incorporación de Typescript sin herramientas adicionales, y la incorporación de un sistema de permisos muy estricto que impide la ejecución de código malicioso al usar librerías externas.

Además, Deno incluye mejoras como el manejo automático de dependencias, una instalación muy rápida, un corredor de tests interno así como otro de benchmarks, o la simplicidad de uso al no necesitar herramientas externas como linters o formatters, que ya vienen incluidos con la propia instalación de Deno.

Deno es capaz de ejecutar código Typescript directamente utilizando un compilador internamente, sin necesidad de generar archivos Javascript por parte del

desarrollador, y sin generar archivos en el proyecto resultado de la compilación.

#### ¿Cómo se ha utilizado Deno?

Deno no estaba planteado como herramienta cuando se comenzo el trabajo, pero despues de necesitar hacer tests para la parte de servidor de WebSockets, y buscar posibles herramientas de testeo para Node.js, surgió la idea de cambiar esta parte de Node.js por Deno.

Como ya se ha comentado, Deno cuenta con su propio ejecutor de tests, y además, es casi completamente compatible con Node.js, migrar la aplicación fue un trabajo de un par de horas, en las que simplemente hubo que cambiar la forma de inicializar la aplicación para ajustarse con el sistema de Deno.

#### Curiosidad, el nombre de Deno

Como el creador de Deno es también el creador de Node.js, Ryan Dahl de manera cómica eligió el nombre de Deno invirtiendo las sílabas de Node (No-de -¿De-no).

### 3.1.4. SvelteKit

SvelteKit [5] es una tecnología para el desarrollo front-end y back-end desarrollada por Rich Harris.

SvelteKit es la unión de dos tecnologías, por una lado Svelte, que es un framework de desarrollo front-end, alternativo a React, Angular o Vue. Y por otro lado, la parte del nombre *Kit* viene de la incorporación a Svelte de un sistema de routing, y la posibilidad de ejecutar código en el lado de servidor.

SvelteKit, también permite hacer SSR (Server Side Rendering), al igual que Next permite hacer SSR para React, o Nuxt permite hacer SSR para Vue.

#### Svelte

Svelte [6] es un compilador de componentes web que permite al desarrollador programar la vista, ejecución y estilado de un componente en un solo archivo, y tras ser compilado, devuelve código que manipula el DOM y es reactivo a cambios en la información presentada.

Svelte, a diferencia de React o Vue, no necesita de un framework en runtime para funcionar, el compilador devuelve el código Javascript necesario para mostrar

la información necesaria y manejar la reactividad, sin necesidad de un shadow-DOM, esto disminuye la cantidad de código que hay que enviar al cliente en la petición inicial, lo que mejora el rendimiento y disminuye el "time to interactive".

Además, la sintaxis de Svelte es muy simple y muy cómoda de usar. Svelte incluye también una librería de animaciones de componentes, y un sistema para manejar la reactividad que simplifica el desarrollo utilizando una sintaxis sencilla para hacer que cualquier función o variable sea reactiva.

#### SvelteKit

SvelteKit añade a Svelte la posibilidad de realizar operaciones en la parte de servidor, permitiendo hacer SSR, que permite mejorar reducir el time to interactive enviando al cliente solo el HTML resultante de realizar el renderizado en el servidor, y otras operaciones.

SvelteKit es considerada como la manera más sencilla y rápida de desarrollar una aplicación de Svelte, y por esa razón se ha utilizado en este trabajo.

#### ¿Cómo se ha utilizado SvelteKit?

SvelteKit ha fundamentado la base que ha permitido añadir reactividad a la aplicación de manera sencilla, su cómoda sintaxis facilita el trabajar con los datos, especialmente aquellos que requieren de reactividad.

#### Tailwind

Tailwind [7] es un framework de CSS que permite utilizar la mayor parte del poder de CSS directamente desde la plantilla HTML, además, es compatible con el sistema de componentes de Svelte.

Tailwind funciona utilizando clases HTML, que tienen una representación en el framework de CSS, esto consigue que con muy poquito esfuerzo tengamos una página lista.

Tailwind no es una librería de componentes ya hechos, como lo puede ser Bootstrap, Tailwind solo incluye código CSS que es el que será interpretado por el navegador.

#### ¿Cómo se ha usado Tailwind?

Tailwind se ha usado para la mayoría de la maquetación de la aplicación, aunque ha habido pequeñas partes que si han necesitado de su propio CSS para crear estilos más complejos.

### 3.1.5. Git y GitHub

Git es un sistema de control de versiones distribuido, que permite a desarrolladores sincronizar su código entre versiones.

GitHub es el servicio web que ha almacenado el código, es el servicio de Git más grande e importante.

### 3.1.6. Docker y DockerHub

Docker es una plataforma para desarrollar y desplegar aplicaciones de todo tipo, permite separar las aplicaciones de la infraestructura reduciendo los tiempos de desarrollo.

Cada aplicación de Docker se puede subir a DockerHub, la plataforma para compartir aplicaciones Docker oficial, a través de esta plataforma es posible desplegar tu aplicación en cualquier computadora con tan solo saber el nombre del repositorio y las credenciales (en caso de ser necesarias).

### 3.1.7. Playwraight

Playwraight [8] es el framework para testear componentes recomendado por SvelteKit, es un software muy capaz que permite con muy poco trabajo y con una sintaxis muy sencilla, hacer testeo determinista sobre los componentes web. En numerosos frameworks de testeo de páginas web hay problemas para saber cuando la página web ha cargado los componentes, Playwraight provee de sistemas asíncronos (basados en Promesas) para esperar a que los componentes hayan cargado sin complicar el código.

#### ¿Cómo se ha usado Playwraight?

Playwraight se ha usado para hacer tests para componentes, y parcialmente para flujos. Hacer tests de flujos completos ha sido más complejo, ya que es difícil interceptar las comunicaciones de Websockets.

## 3.2. Metodología

Este trabajo ha seguido una metodología iterativa e incremental en espiral, en la cual el tutor y el alumno se reúnen de manera periódica y hacen revisión de objetivos y progreso.

En cada reunión se hacen propuestas de mejora, y estas se convierten en objetivos para la siguiente revisión. Poco a poco el desarrollo del trabajo se fue materializando, empezando con los pasos más básicos como tener una plantilla sobre la que poder ir probando desarrollos, pasando por el aprendizaje de la tecnología de WebSockets, como el arreglo de bugs que han ido surgiendo a lo largo del tiempo.

La metodología escogida ha funcionado muy bien para el tipo de proyecto, la progresión iterativa y en fragmentos ha permitido que el TFG se haya desarrollado lentamente, y con la atención puesta en cada fragmento de desarrollo, sin llegar a encontrarse con un gran panel de tareas que pueda resultar abrumador, además, el planteamiento a futuro ha permitido que la aplicación haya crecido adecuadamente, sin necesidad de realizar grandes cambios.

# 4

# Descripción informática

## 4.1. Requisitos

### 4.1.1. Requisitos funcionales

Los primeros requisitos funcionales del TFG se centraron en la creación de los sistemas y las mecánicas básicas que definen a CoWordle. Después, se fueron detallando funcionalidades más avanzadas.

- RF1. Como usuario puedo crear una partida a la que otros jugadores pueden unirse.
- RF2. Como usuario puedo unirme a partidas creadas por otros jugadores si poseo el link o el código de la partida.
- RF3. Como usuario puedo cambiarme el nombre público en una partida.
- RF4. Como usuario puedo ver a todos los jugadores de la partida.
- RF5. Como usuario creador de una partida (*host*), tengo el control sobre cuándo quiero que comience ésta.
- RF6. Como usuario *host* puedo expulsar a otros jugadores de la partida.
- RF7. Como usuario puedo introducir palabras una vez haya comenzado la partida.

- RF8. Como usuario puedo ganar si introduzco la palabra objetivo.
- RF9. Como usuario puedo conocer cómo de cerca están otros jugadores de la palabra objetivo.
- RF10. Como usuario puedo abandonar la partida en cualquier momento y unirme a otras.
- RF11. El sistema es capaz de detectar cuándo un jugador se ha unido a la partida, y mostrar al resto de jugadores su nombre.
- RF12. El sistema es capaz de detectar cuándo un jugador ha abandonado la partida, y de eliminar su información de ésta. Además, en caso de ser el último jugador, el sistema eliminará la memoria reservada para la partida.
- R13. El sistema es capaz de detectar cuándo todos los jugadores han acabado sus intentos, y mostrar una pantalla con la solución y un mensaje indicando que todos los jugadores han perdido.
- RF14. Como usuario puedo acceder a la aplicación desde dispositivos móviles, tablets, portátiles y de ordenadores de sobremesa.

### 4.1.2. Requisitos no funcionales

- RNF1. La página web debe ser fácil de usar.
- RNF2. Los colores usados en la página web deben ser característicos y reconocibles.
- RNF3. El código único de cada partida debe contener pocos caracteres y ser reconocible, para que en caso de que sea transmitido de manera oral, este sea fácil de comunicar.
- RNF4. Una vez haya terminado la partida, debe indicarse al los jugadores, quién ha ganado y cuál era la palabra objetivo.
- RNF5. Desde la consola de despliegue, se puede atender al avance de la partida, siguiendo los eventos principales que ocurren.

## 4.2. Arquitectura general

El proyecto está dividido en dos repositorios, por un lado está la aplicación web (webapp), que se encarga de realizar la lógica de presentación, así como el *game loop*, y por otro lado tenemos el servidor de backend, donde va toda la lógica de websockets en la parte de servidor.

La comunicación entre aplicaciones se realiza mediante peticiones HTTP y websockets, para ello se utiliza la librería estándar de Javascript proporcionada por el navegador, la librería estándar de Node.js, la librería estándar de Deno y la librería Socket.io.

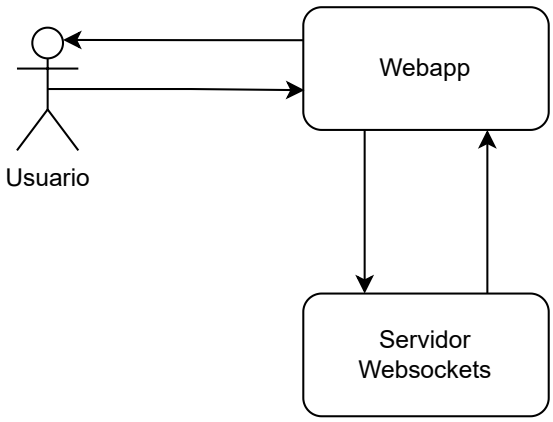



Figura 4.1: Diagrama de comunicación entre aplicaciones.

El usuario siempre interactúa directamente con la aplicación web, y es esta la que realizará llamadas al servidor.

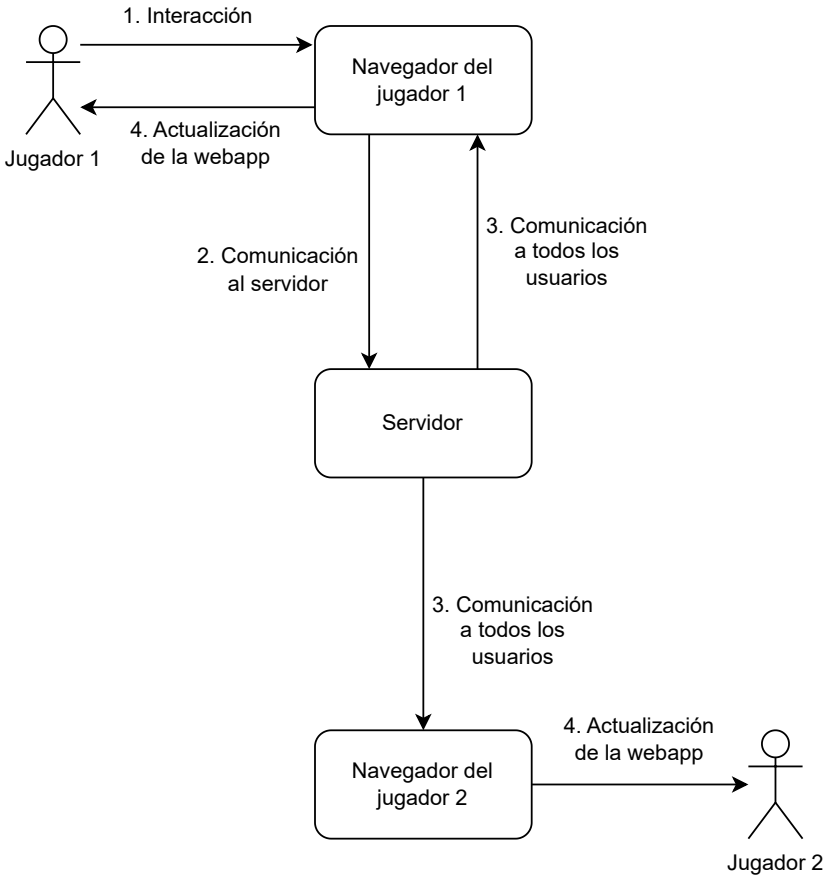



Figura 4.2: Diagrama de comunicación a todos los usuarios.

Al ser una aplicación en tiempo real, es necesario que todos los usuarios estén al tanto de toda la información relevante para la partida. Como se puede ver en la Figura 4.2, cada vez que surge una interacción por parte de un usuario, como la inserción de una nueva palabra en el tablero, el servidor es capaz de procesar esta información y enviar a todos los usuarios el resultado relevante para ellos de la acción del otro jugador.

### 4.2.1. Aplicación web

La aplicación web es la herramienta con la cual el usuario interactúa, permite a este realizar todas las acciones necesarias utilizando un navegador web.

Está desarrollada utilizando SvelteKit para la realización de la lógica, y Tailwind para dar color y estilo.

Como se puede ver en la Figura 4.3, el flujo principal que sigue el usuario para poder jugar una partida comienza abriendo la página web de la aplicación web,

después, creará o se unirá a una partida ya creada, cuando todos los jugadores estén listos, el creador de la partida le dará al botón de comenzar, y después de una cuenta atrás, a todos los jugadores se les presentará un tablero de Wordle donde pondrán ir escribiendo sus intentos. Si alguno de los jugadores descubre la palabra, todos los jugadores verán una nueva pantalla con el nombre de la persona ganadora, en esta nueva pantalla, los jugadores pueden elegir entre salir de la partida o volver a jugar. En caso de que ningún jugador consiga ganar, se les presentará a todos una pantalla en el que se mostrará la palabra objetivo, y las mismas opciones que en el caso de haber habido un ganador.

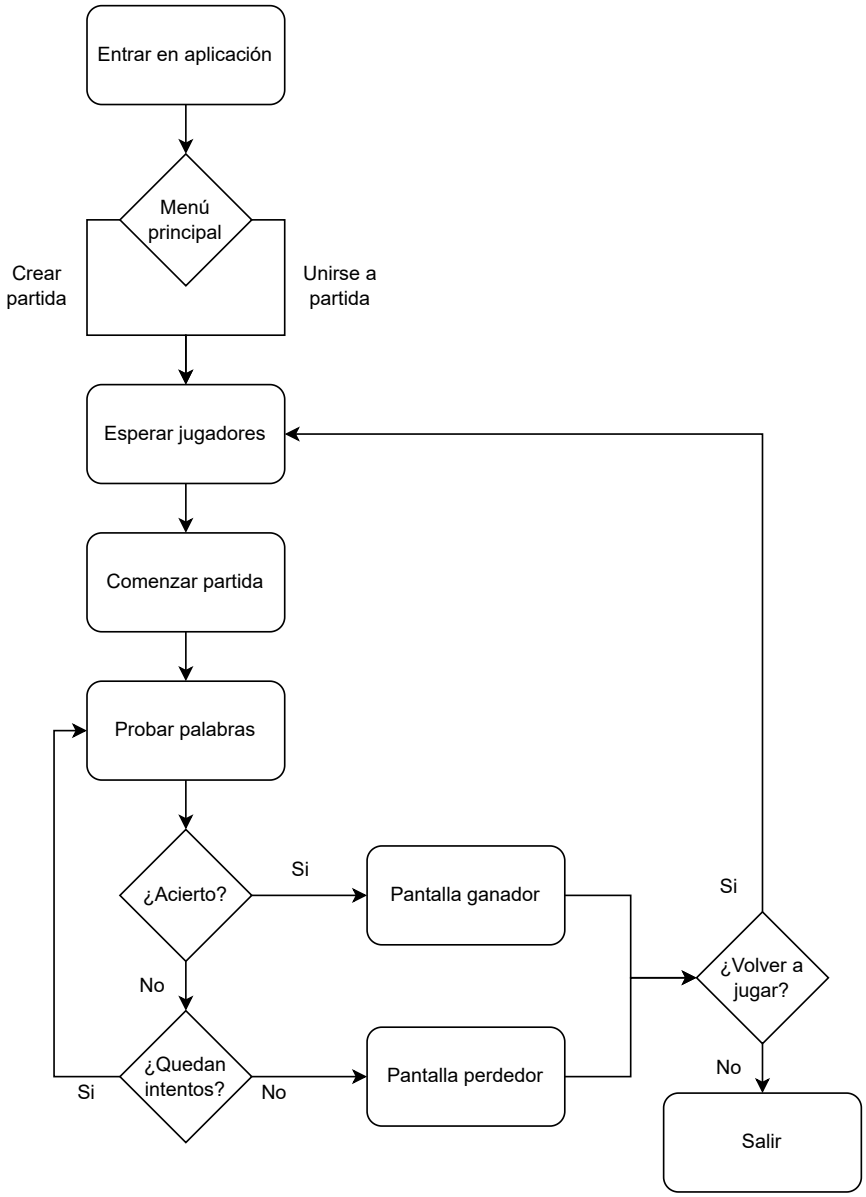



Figura 4.3: Diagrama de flujo de la aplicación web.

### 4.2.2. Diseño visual

El diseño visual de la aplicación está inspirado por el diseño original de Wordle tal como se puede ver en la aplicación oficial de Wordle.

En CoWordle se están utilizando diferentes tonalidades para representar los estados de las palabras. Para ver si los colores eran claros y entendibles, se escogió a varios voluntarios que afirman no encontrar mucha diferencia con los significados de los colores oficiales.

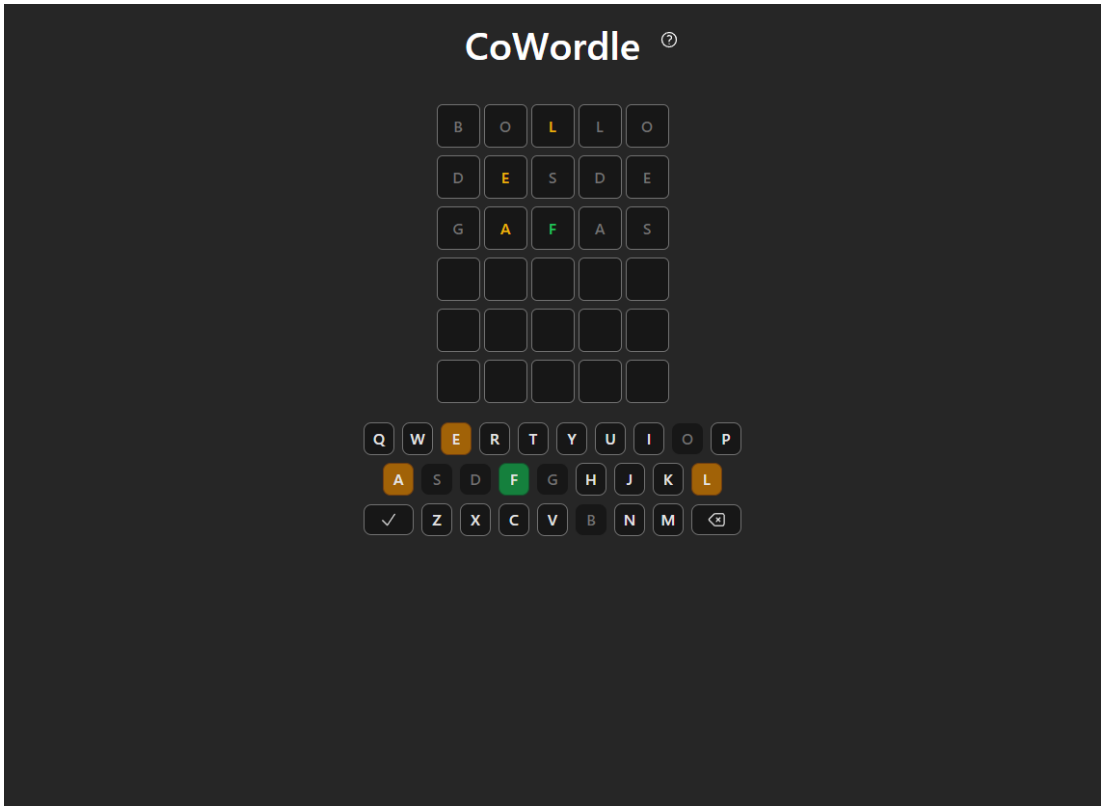



Figura 4.4: Aplicación web en vista escritorio.

Los colores elegidos utilizan tonalidades con colores intensos para poder diferenciarlos con el fondo oscuro. A lo largo de toda la aplicación se reutilizan los mismos colores, creando una sensación de marca.

Los bordes redondeados hacen más amigable la aplicación, y sigue con tendencias modernas que utilizan grandes empresas como Microsoft, por ejemplo, con los bordes de ventana redondeados de Windows 11.

El tablero dispone de un teclado para indicar las letras usadas, oscureciendo aquellas que no se encuentran en la palabra, marcando verde aquellas que se encuentran en la posición correcta y naranjas aquellas que no se encuentran en la posición correcta. El teclado es además completamente funcional.

Toda la aplicación es *responsive*, lo cual significa que es adaptable para cualquier pantalla de cualquier dispositivo, por ejemplo, asi es que como se ve la aplicación en un telefono movil de la compañia Apple.

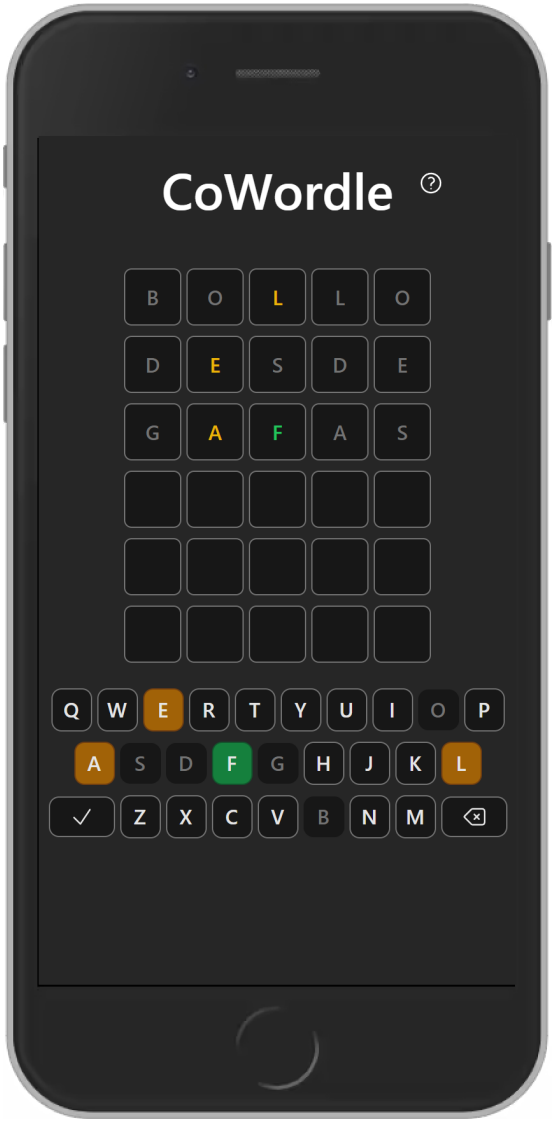



Figura 4.5: Aplicación web en un dispositivo móvil.

Como se puede ver, en lugar del teclado único de cada teléfono móvil, se utiliza el teclado interno de CoWordle, lo cual permite mantener la información que el jugador recibe independientemente del dispositivo.

Además, todas las vistas de la aplicación cuentan con un botón de información que explica brevemente cómo se utiliza la aplicación.

#### Comunicación websocket

La comunicación websocket ha sido el punto de aprendizaje principal, según se fueron desarrollando las diferentes partes para el trabajo, se fue definiendo lo que acabó siendo una serie de reglas sobre la comunicación diferenciando estas en tres tipos de comunicación diferentes.

Para entender estos tres tipos de comunicación hay que entender primero el sistema de comunicación HTTP. En este tipo de comunicación es el cliente el que inicia la llamada y es el servidor el que responde a esta. Sin embargo, en websockets, el servidor puede enviar información al cliente sin que sea el cliente el iniciador de la llamada. Por ello Websockets funciona por sistemas de eventos, donde el socket puede enviar y esperar comunicación (comunicación bidireccional).

Una vez entendida la comunicación HTTP y la comunicación websocket, y las diferencias entre ambas, va a quedar claro porque en CoWordle se diferencia entre tres tipos distintos de comunicación.

En primer lugar tenemos la comunicación cliente - servidor, donde el cliente envía eventos al servidor, y el servidor puede actuar ante estos eventos, pero no responde a través de ellos. Estos eventos se clasifican como eventos OUT, ya que el cliente envía información al exterior. Por ejemplo, la llamada cuando un jugador ha sido expulsado de la partida (*remove_player*), se envía desde el host de la partida hacia el servidor, cuando el servidor ha procesado este evento, envía a todos los jugadores de la partida el evento *player_disconnected*, y serán los clientes los que eliminen de sus listados locales al jugador que se ha eliminado.

En segundo lugar tenemos la comunicación servidor - cliente, un ejemplo de esto es el recién descrito *player_disconnected*, en el que el servidor envía un evento a los jugadores de la sala. Este tipo de eventos se clasifica en CoWordle como IN, porque el cliente recibe información del servidor.

En último lugar el tercer tipo de comunicación se clasifica como DIALOGUE, y se corresponde con comunicación cliente - servidor - cliente, es decir, es el cliente el que comienza la comunicación, el servidor procesa la respuesta y se la envía al cliente con el mismo nombre de evento con el que la llamada había comenzado, hasta que la llamada no ha sido respondida, el cliente se quedará esperando una respuesta de manera asíncrona, es decir, sin bloquear al resto de la aplicación.

### 4.2.3. Servidor Websockets

El servidor de websockets está implementado utilizando Deno y una librería que ayuda al manejo de eventos para los websockets llamada Socket.io. Deno se utiliza para levantar el servidor y para recibir peticiones HTTP, existe un

endpoint necesario para que Socket.io cree la conexión de Websockets, y uno más para la creación de la partida.

### Sistema de eventos

El servidor responde a peticiones HTTP y Websockets creando eventos que son respondidos de manera secuencial por funciones lambda, cada función sólo responde a un evento, devolviendo una respuesta. Javascript es una tecnología uni-hilo, pero es capaz de responder peticiones asíncronamente utilizando primitivas llamadas promesas. Cuando una función marcada como asíncrona se ejecuta, el motor de Javascript la guarda como evento, este evento será manejado cuando el motor de Javascript lo considere oportuno. Por ejemplo, cuando la CPU está esperando una lectura en memoría, Javascript puede cambiar el contexto y procesar otras funciones encoladas. De está manera, Javascript es capaz de maximizar la utilización de su hilo de ejecución.

### Eventos disponibles

El servidor permite la comunicación WebSocket utilizando los siguientes eventos:

- *setup* es un evento de tipo diálogo que une a un jugador a la partida, es necesario enviar como parámetros el código de la sala generado por /create-room y el nombre del usuario que se va a unir a la partida, este evento avisa a todos los demás jugadores de la sala que se ha unido el jugador emitiendo el evento *player_connection*.
- *update_player_name* es un evento de tipo diálogo que permite a los jugadores actualizar su nombre, especialmente útil porque el primer nombre que recibe cada jugador al conectarse a la partida es aleatorio entre una combinación de posibilidades.
- *remove_player* es un evento de tipo out que solo pueden utilizar el creador de la partida, y permite expulsar a jugadores de la partida. Cuando ha expulsado a un jugador emite un evento *player_disconnected* que notifica al resto de usuarios de la desconexión.
- *validate_word* es un evento de tipo diálogo que comprueba un intento de un jugador con la palabra objetivo. El resultado de la operación de comprobación es un array de una enumeración (*WordlePoints*) que indican el resultado de cada letra en la palabra (si no se encuentra en la palabra, si se encuentra en la palabra, si está en la posición correcta). Este resultado se envía directamente al jugador que ha ejecutado el evento, pero también se

envía a los demás jugadores para indicar en el scoreboard el conocimiento de cada jugador. Además, se comprueba si la partida ha terminado por que el intento coincide con la palabra objetivo, y si todos los jugadores han perdido porque se han quedado sin intentos.

- *start_game* es un evento de tipo out que permite al host de la partida empezar. En el momento que se lanza este evento todos los jugadores reciben el evento *start_prematch* que les indica cuándo comenzará la partida.

- *disconnect* es un evento nativo de Socket.io, que permite al servidor conocer cuando un jugador se ha desconectado de la partida, existen varias razones para que se ejecute este evento, por ejemplo, que el jugador haya perdido la conexión con internet, o que el jugador haya decidido abandonar la partida cerrando el navegador. En este evento se comunica al resto de jugadores que este ha abandonado la partida, en caso de ser el jugador que ha creado la partida el que la abandona, se cierra también para el resto de jugadores.

### Endpoints HTTP

Como el objetivo del trabajo era el aprendizaje de la comunicación WebSocket, la mayor parte de la comunicación se realiza a traves de estos, aún asi existen dos rutas disponibles para la realización de comunicación HTTP:

- */create-route*, genera un nuevo identificador de partida. Este identificador es fundamental para la comunicación entre el cliente y el servidor ya que permite la identificación de la partida para realizar las operaciones necesarias.

- Existe también una ruta reservada por la librería de Socket.IO que se utiliza internamente para establecer la conexión websocket. Esta ruta está completamente manejada por Socket.IO.

### Identificador de partida

Cada partida tiene un identificador único de seis cifras generado pseudoaleatoriamente llamado roomCode. El roomCode lo generará el servidor tras la petición HTTP. Esta ruta únicamente reserva el código generado para la partida, utilizando ese código los jugadores son capaces de unirse a la partida inicialmente, el código también se utiliza para realizar la comunicación cliente-servidor.

## 4.3. Diseño e Implementación

### 4.3.1. La reutilización de componentes

Svelte, la tecnología que he usado para hacer los componentes de front, se basa en la creación de componentes que puedan ser reutilizados en diferentes situaciones, la reutilización de código se considera buena práctica en todo el ámbito de la ingeniería informática. Por ello parece importante destacar que una buena elección de la tecnologías es solo la base, y es necesario hacer un buen uso de estas para poder llegar a un buen resultado, por ello en este apartado quiero destacar un buen uso de componentes de Svelte.

Existe un componente que se reutiliza varias veces a lo largo de la aplicación, el componente llamado *<Word>*, que se encuentra en el archivo *Word.svelte* de la aplicación cowordle-webapp.

Este componente se utiliza como botones estilizados en el menú principal, en este primer uso, el componente muestra la palabra elegida por el desarrollador, indicando al usuario que es un botón con el que puede interaccionar.

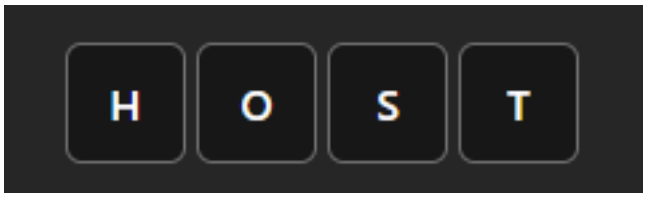



Figura 4.6: Imagen del componente como botón.

Después, se vuelve utilizar como huecos para las palabras mientras se juega, en este caso la palabra tiene distinto número de huecos disponibles, además está completamente vacía de texto.

Figura 4.7: Imagen del componente con texto vacio.

Cuando el jugador interacciona con el teclado mientras juega, las palabras se van llenando de letras.


Figura 4.8: Imagen del componente con texto.

Además, las letras muestran el progreso de la palabra objetivo, utilizando los colores característicos.

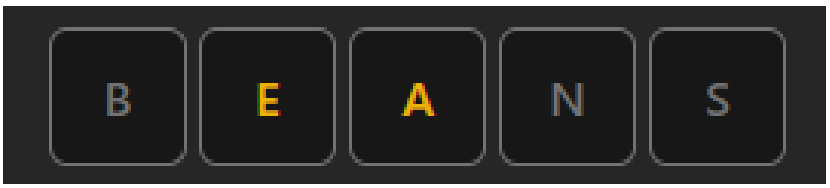



Figura 4.9: Imagen del componente con texto y colores.

Todas las imágenes mostradas representan el mismo componente de Svelte, y permiten mostrar las ventajas de utilizar tecnologías web reactivas. Comenzando por el principio, necesitamos un componente que sea capaz de mostrar texto.

```
<Word word="Host" length={4} />
```

Figura 4.10: Imagen código para utilizar el componente.

Así es como se utiliza el componente $<Word>$, para mostrar el texto de la primera imagen simplemente es necesario definir la palabra a mostrar y su longitud.

También queremos que el componente reciba texto del usuario dinámicamente, esto sería posible reprogramando al componente o creando uno nuevo, pero realmente ya tenemos esta funcionalidad sin necesidad de programar nada más. En lugar de utilizar como argumento para el parámetro word un string fijo, podemos utilizar una variable, e indicarle a Svelte que esa variable es reactiva utilizando, la directiva bind, que recalcula el componente cada vez que cambia el contenido de la variable pasada.

```
<Word bind:word={code} length={6} />
```

Figura 4.11: Imagen código para utilizar el componente con texto dinámico.

En este caso estamos utilizando la directiva *bind* para hacer que el componente se redibuje cada vez que cambia la variable code. Esto significa que nuestro componente es ajeno al texto que presentamos, y es responsabilidad de otros componentes seleccionar el texto a mostrar.

Parece un pequeño detalle, pero esta flexibilidad de desarrollo simplifica el código haciéndolo muy reutilizable y componible.

La primera implementación del componente *<Word>* se hizo cuando se comenzó el proyecto, y sin mucho cambio, ha llegado a utilizarse en otras partes de la aplicación en las que no se había llegado a pensar cuando se creó el componente, y eso demuestra que se realizó un buen diseño inicialmente.

### 4.3.2. Tipados en la comunicación websocket

Como se ha explicado en el apartado 4.2.2, la comunicación websocket con el servidor de CoWordle se clasifica en tres grupos, según sea cliente - servidor, servidor - cliente y cliente - servidor - cliente. En este apartado se va a explicar cómo de fácil es realizar esta comunicación independientemente del tipo de esta.

Toda esta capa de tipado se construye como un wrapper sobre el socket expuesto por Socket.IO. Esto permite utilizar la potencia de Socket.IO, mientras se provee la seguridad de tipos y la comodidad del auto-completado del IDE, además permite la utilización de distintos métodos del wrapper según el tipo de comunicación que vayamos a realizar, siendo estos: *emit* para los eventos cliente-servidor, *on* para los eventos servidor-cliente, y *dialogue* para los eventos cliente-servidor-cliente.

#### Comunicación IN

Para la comunicación IN, es el servidor quien envía el evento al cliente, por ello el cliente debe quedarse esperando a que el servidor envíe la petición.

El wrapper expone un método *on*, que permite añadir un callback cuando el evento sea emitido, es decir, cuando el servidor lo emite y el cliente lo haya recibido.

En el framento del Código 4.1 se puede observar cómo se está definiendo que cuando el evento *player_connected* ocurra, la función lambda debe ejecutarse. El evento *player_connected* está declarado utilizando tipos de la siguiente manera (fragmento de Código 4.2).

En esta declaración tan solo necesitamos indicar los parámetros que nos van a llegar del servidor, esta declaración de tipos permite dos mejoras con respecto a los eventos directos de Socket.IO. Por un lado, el IDE ofrece ayuda sobre los

Código 4.1: Uso de eventos websocket IN.

```
this.socket.on("player_connected", ({ newPlayer }) => {
    this.addPlayer(newPlayer);

    this.notifies.playersUpdated.broadcast(this.players);
});
```

Código 4.2: Ejemplo de declaración de la comunicación IN.

```
export type WebsocketInEvent = {
    ...
    player_connected: {
        newPlayer: InitialPlayerInfoDto;
    };
    ...
};
```

nombres de los eventos disponibles sobre los que hacer la espera, además, nos ofrecerá ayuda sobre los parámetros que vamos a recibir según el nombre que indicamos en el primer argumento.

Todo esto es posible gracias al poderoso sistema de genéricos proporcionado por Typescript.

Código 4.3: Declaración del método *on*.

```
on<TEvent extends EventName<WebsocketInEvent>, TArgs
   extends WebsocketInEvent[TEvent]>(
event: TEvent,
callback: (args: TArgs) => void,
): void {
    this.socket.on(event as string, callback);
}
```

Como se puede ver en la definición del método *on* (fragmento de Código 4.3), el primer parámetro es el nombre del evento, que está tipado como el genérico *TEvent* que está a su vez definido como un nombre de evento de la lista de eventos IN. El segundo parámetro es el callback a que se llamará cuando el evento se ejecute, este parámetro está tipado como una función que recibe un argumento del tipo *TArgs*, que es de la lista de eventos IN, aquel que coincide con la el genérico *TEvent*. Todo ello ofrece suficiente información al IDE para ser capaz de autocompletar todo lo necesario para que el programador esté seguro de lo que está programando es correcto, y además, mejora la mantenibilidad a largo plazo, porque en caso de cambiar el tipado de algún evento IN de la lista

de eventos, el cambio sería reflejado automáticamente en toda la aplicación y en caso de que no coincidieran los nuevos tipos con los antiguos, Typescript indicaría al programador donde ha ocurrido el fallo de tipado.

### Comunicación OUT

La comunicación OUT es aquella que ocurre en dirección cliente - servidor, el cliente solo pretende comunicar información al servidor, sin esperar nada de respuesta.

Código 4.4: Ejemplo de uso del método *emit*.

```
1     startGame(wordListId: string): void {
2         this.socket.emit("start_game", {
3             wordListId,
4         });
5     }
```

En el ejemplo 4.4 se puede ver cómo se emite un evento al servidor, para emitir un evento necesitamos el nombre del evento a emitir, y necesitamos también la información que queremos enviar por el socket.

Al igual que la comunicación IN, el sistema dispone de un lugar donde definir todos los eventos OUT, para el ejemplo mostrado en la figura anterior, el evento está definido como se ve en el fragmento de Código 4.5.

Código 4.5: Ejemplo de declaración de la comunicación OUT.

```
1     export type WebsocketOutEvent = {
2         ...
3         start_game: {
4             wordListId: string;
5         };
6     };
```

Esta definición es la versión inversa de la mostrada en los eventos IN, lo que declaramos dentro del evento representan los argumentos que vamos a enviar al servidor, mientras que la comunicación IN se definen los argumentos que íbamos a recibir. A pesar de la diferencia de uso, la forma de declarar ambos es la misma.

Si nos adentramos en la declaración del método *Emit* (fragmento de Código 4.6), que como ya hemos visto es el que se utiliza para la comunicación OUT, volvemos a ver la aparición de un tipo genérico *TEvent*, con la diferencia de que este *TEvent* es un evento de la lista de eventos OUT, la otra diferencia con el

Código 4.6: Declaración del método *emit*.

```
1     emit<TEvent extends EventName<WebsocketOutEvent>,TArgs
         extends WebsocketOutEvent[TEvent]>(
2     event: TEvent,
3     args: TArgs,
4     ): void {
5         this.socket.emit(event, args);
6     }
```

Código 4.7: Ejemplo de uso del método *dialogue*.

```
1     const { localPlayer, players, hostPlayer } = await this.
         socket.dialogue("setup", {
2         playerName: this.localPlayer.name,
3         roomCode: this.roomCode,
4     });
```

método *on*, es que en este utilizamos el argumento *args* como el argumento del método *emit* de Socket.IO, que es el que realizara la petición siguiendo el estándar websocket.

El método *emit* ofrece al programador la misma seguridad y mantenibilidad que el método *on*, además, su similitud de uso, ayuda a que sea más fácil usarlo por programadores de todos los niveles.

### Comunicación Dialogue

En la comunicación de tipo Dialogue, el cliente envía un evento al servidor, esperando una respuesta de vuelta. Este tipo de comunicación es más compleja de implementar que las otras, pero su uso en CoWordle sigue siendo muy simple.

Como se puede ver en el ejemplo 4.7, para usar un evento de tipo Dialogue tan solo tenemos que llamar al método *dialogue* del wrapper. Aunque el tipo Dialogue es muy similar a lo que es una petición HTTP al uso, usando el evento permitimos que toda la comunicación del mismo tipo (comunicación en tiempo real sobre el juego) se realice utilizando un único canal de comunicación, además, nos permite utilizar la información guardada en el socket desde el servidor.

El primer argumento define el evento que vamos a emitir y sobre el que vamos a esperar la respuesta. El segundo argumento describe los datos que vamos a enviar. Como respuesta obtendremos una *promesa* de la respuesta que recibiremos del servidor cuando este responda.

Código 4.8: Ejemplo de declaración del tipo de evento Dialogue.

```
export type WebSocketDialogueEvent = {
    ...
    setup: (args: { roomCode: string; playerName: string }) => {
        players: InitialPlayerInfoDto[];
        hostPlayer: InitialPlayerInfoDto;
        localPlayer: InitialPlayerInfoDto;
        roomState: RoomState;
    };
    ...
};
```

Para declarar un evento del tipo Dialogue, se utiliza una sintaxis muy cómoda, que permite diferenciar en un solo vistazo los tipos que vamos a enviar y los tipos que vamos a recibir (fragmento de Código 4.8).

El evento se declara como una función donde los argumentos son los datos a enviar a servidor, y cuya respuesta son los tipos que vamos a recibir del servidor.

Como explicaba anteriormente, la implementación es un poco más compleja que en los tipos de comunicación anteriores, como se ve en el fragmento de Código 4.9. El primer parámetro coincide con los de los otros tipos de comunicación, volviendo a usar el *TEvent*. El segundo argumento extrae los parámetros de la función declarada como evento. El valor de retorno del método se define como una *promesa* del valor de retorno definido en función del evento.

Con esta implementación estamos manteniendo la sencillez de uso que teníamos con los otros tipos de comunicación, pero estamos creando una comunicación más clásica, donde el cliente y el servidor se comunican como si fuera una petición HTTP normal.

## 4.4. Pruebas

Como el proyecto está dividido en dos repositorios distintos con tecnologias distintas, es necesario explicar como se han realizado los tests en ambos repositorios.

Código 4.9: Implementación del método *dialogue*.

```
1     async dialogue<TEvent extends EventName<
          WebSocketDialogueEvent>, TEventMethod extends
          WebSocketDialogueEvent[TEvent]>(
2     event: TEvent,
3     params: Parameters<TEventMethod>[0],
4     eventResponse?: string
5     ): Promise<ReturnType<WebSocketDialogueEvent[TEvent]>> {
6         return new Promise((resolve, reject) => {
7             this.socket.emit(event, params);
8
9             this.socket.on<string>(eventResponse || event, (
                  result) => {
10                this.socket.off(event);
11                resolve(result);
12            });
13
14            setTimeout(() => reject("timeout"), 10000);
15        });
16    }
```

### 4.4.1. Webapp

Para la webapp se han realizado pruebas unitarias de componentes, para ello se ha utilizado Playwright. Como se ha explicado en la sección de tecnologías, Playwright es la librería recomendada para el testeo de componentes.

Playwright utiliza un navegador *headless* internamente para comprobar que los componentes están funcionando. Además, permite seleccionar qué tecnología de navegador se quiere probar. Para CoWordle se realizan tests en todas las tecnologías principales, Firefox, Chromium y Webkit.

Los tests realizados incluyen comprobaciones para ver si los componentes son montados correctamente en la pantalla, si los componentes muestran los textos correctamente y si los colores, especialmente en los colores de juego, se muestran correctamente.

Un ejemplo de test donde se ha comprobado si el componente se monta correctamente, se puede observar en el archivo *tests/components/word.test.ts*. En este test se puede comprobar que primero se monta el componente con los props necesarios, y después se realiza la comprobación de que el componente sea visible.

Un ejemplo de test donde se comprueba el contenido textual de un componente se puede ver en el mismo archivo que el test anterior. En este test (fragmento de Código 4.11) se puede comprobar cómo se monta el componente y se prueba que

Código 4.10: Ejemplo test visibilidad componente.

```
test("component is displayed correctly", async ({ mount })
   => {
    const comp = await mount(Word, {
        props: {
            word: "Hello",
            length: "Hello".length,
        },
    });

    await expect(comp).toBeVisible();
});
```

el texto interno del componente coincide con lo esperado.

Código 4.11: Ejemplo test con texto.

```
test("contents are displayed correctly", async ({ mount })
   => {
    const comp = await mount(Word, {
        props: {
            word: "Hello",
            length: "Hello".length,
        },
    });


    await expect(comp).toContainText("Hello");
});
```

Un ejemplo de test donde se comprueba que los colores de las letras se puede volver a ver en el archivo ya visto (fragmento de Código 4.12).

En este ejemplo se monta el componente, con las letras a mostrar y los resultados que deberían de mostrar los colores de las letras CoWordle. Una vez montado el componente, se extraen todas las letras de este, y se comprueba que cada letra tenga la clase HTML correcta. Se comprueban las clases HTML porque internamente a cada clase HTML le corresponde un color CSS. Si en algún momento se decidiera cambiar los colores, no habría que modificar los tests, ya que en ningún momento se comprueban colores concretos.

Para poder ejecutar los tests en local, es necesario utilizar el comando 4.13.

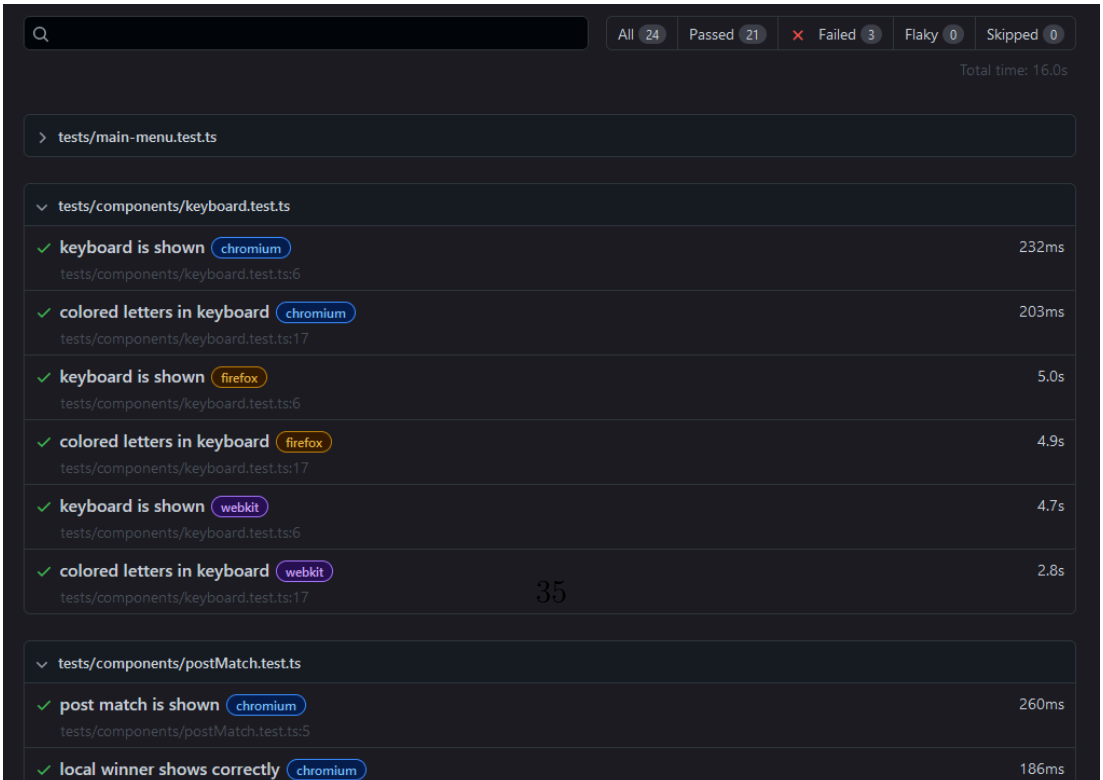
Una vez ejecutado el comando, Playwright comenzará a realizar los tests, una vez haya terminado nos abrirá una página web donde mostrarán los resultados.

Código 4.12: Ejemplo test colores.

```
test("colors are shown correctly", async ({ mount }) => {
    const word = "SUN";


    const comp = await mount(Word, {
        props: {
            word,
            length: word.length,
            results: [WordlePoints.Exact, WordlePoints.
               InWord, WordlePoints.Missing]
        },
    });


    const innerLetters = await comp.getByRole("paragraph").
       all();


    await expect(innerLetters[0]).toHaveClass(/letter-exact
       /);
    await expect(innerLetters[1]).toHaveClass(/letter-in-
       word/);
    await expect(innerLetters[2]).toHaveClass(/letter-
       missing/);
});
```

Código 4.13: Comando para ejecutar los tests de Playwright

```
$ npm run test-ct
```



En la página mostrada en la Figura 4.12 podemos inspeccionar cada test realizado, pudiendo ver las comprobaciones que se han realizado en cada uno, como se puede ver en la Figura 4.13, y en caso de haber fallado, la razón del error.

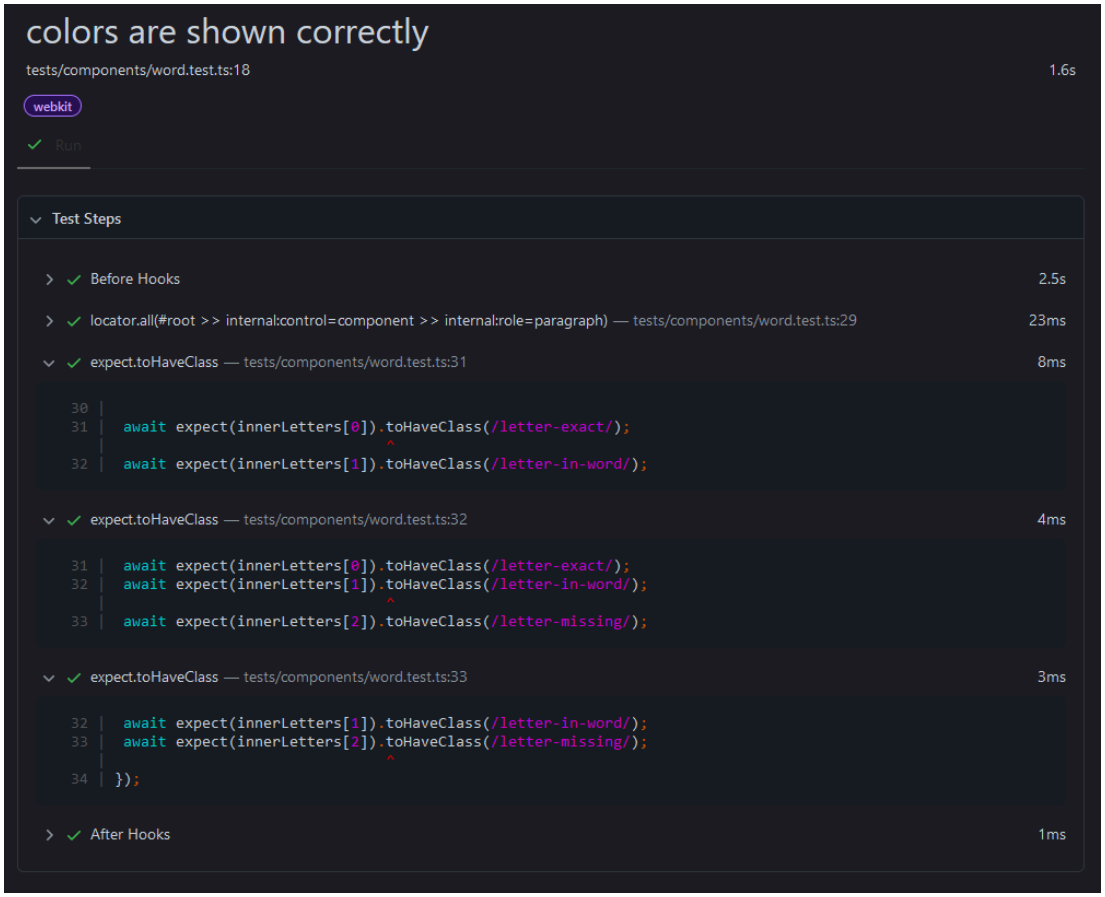



Figura 4.13: Inspección de un test en Playwright.

### Problemas encontrados

Playwright puede llegar a ser complejo de instalar según sea el sistema del usuario a ejecutar los tests. Para la realización de este TFG se han dedicado varias horas únicamente a conseguir que Playwright consiga ejecutar los tests en el entorno en el cual se ha realizado esta (WSL 2 - Ubuntu). Al final el problema se soluciono instalando las librerías necesarias manualmente, en lugar de utilizar el instalador automático. Si durante la instalación de las librerías ocurre el sistema operativo no encuentra alguna, sera necesario limpiar la cache de librerías, en mi caso APT tal y como se indica en [9].

### 4.4.2. Servidor de Websockets

Los tests para el servidor de Websockets se han centrado en el testeo unitario de la comprobación de palabras para la generación de resultados Wordle, además se ha comprobado con test unitarios el comportamiento de alguna función de dominio.

Con la flexibilidad de testeo que ofrece Deno, se ha programado un sistema que es capaz de generar tests unitarios para las palabras que se requiera (fragmento de Código 4.14). Para ello, se genera un test con varios steps (pasos), cada step es una nueva palabra para probar, todo ello es posible generando una estructura de datos que almacene el intento del jugador, y el resultado que se debería de obtener tras el intento. Después, tan solo es necesario recorrer la estructura de datos generando los steps (fragmento de Código 4.15 ).

Tras ejecutar este test obtenemos los siguientes resultados:

```
lorwen@DESKTOP-719CVOE:~/dev/TFG/cowordle-ws$ deno test tests/wordle-word.test.ts
running 1 test from ./tests/wordle-word.test.ts
Validate word ...
  TESTS -> TESTS ... ok (13ms)
  TESTS -> TESTT ... ok (6ms)
  TESTS -> SPLIT ... ok (6ms)
  TESTS -> SPLTT ... ok (5ms)
  TESTS -> STLAT ... ok (6ms)
  TESTS -> ATTAT ... ok (6ms)
  TESTS -> AAAAA ... ok (5ms)
Validate word ... ok (61ms)

ok | 1 passed (7 steps) | 0 failed (119ms)
```

Figura 4.14: Resultado del test de validación de palabras.

Como se puede comprobar, siguiendo la estructura de datos, cada step ejecutado ha comprobado si el resultado tras ejecutar la función *validateWord* ha retornado el resultado esperado o no. Además, nos muestra por pantalla todos los intentos que se han realizado.

#### Problemas encontrados

Deno es un entorno para ejecutar Typescript muy reciente y todavía no dispone de todas las mejoras para realizar tests que pueden llegar a ofrecer sistemas más consolidados, como sistemas para hacer mocks, este problema es aliviado por la flexibilidad ofrecida por Javascript, que permite realizar pseudo-mocks de manera manual.

Aun así, la experiencia de testeo con Deno es satisfactoria.

Código 4.14: Implementación del sistema de creación de test automáticos.

```
    const validations: WordTest[] = [
        { solution: "TESTS", words: [
            {
                word: "TESTS",
                expected: [2, 2, 2, 2, 2],
            },
            {
                word: "TESTT",
                expected: [2, 2, 2, 2, 0],
            },
            {
                word: "SPLIT",
                expected: [1, 0, 0, 0, 1],
            },
            {
                word: "SPLTT",
                expected: [1, 0, 0, 2, 1],
            },
            {
                word: "STLAT",
                expected: [1, 1, 0, 0, 1],
            },
            {
                word: "ATTAT",
                expected: [0, 1, 1, 0, 0],
            },
            {
                word: "AAAAA",
                expected: [0, 0, 0, 0, 0],
            },
            ]
        }
    ];
```

## 4.5. Distribución y despliegue

Existen varias maneras de ejecutar la aplicación, la opción más sencilla es utilizar Docker, aunque también está disponible de levantar la aplicación desde el código directamente.

Código 4.15: Implementación del sistema de creación de test automáticos.

```
    for await (const validation of validations) {
        for await (const word of validation.words) {
            await test.step({
                name: "${validation.solution} -> ${word.word}",
                fn: () => {
                    const result = validateWord(word.word,
                       validation.solution);
                    assertEquals(result, word.expected);
                },
            });
        }
    }
```

### 4.5.1. Utilizando Docker

Utilizando Docker, podemos levantar la aplicación al completo, tanto la webapp, como el servidor de websockets. Esta es la opción más sencilla y la manera recomendada de levantar la aplicación para su despliegue.

Para levantar la aplicación utilizando Docker necesitamos tener Docker y Docker Compose instalado. Una vez instalado podemos utilizar el siguiente archivo *docker-compose.yaml* (fragmento de Código 4.16). Este archivo indica a Docker que aplicaciones (llamadas servicios) queremos que se ejecuten, aqui podemos agregar aplicaciones propias, o de terceros.

Para desplegar CoWordle, todos las aplicaciones que necesitamos que se ejecuten son propias. La primera es la webapp, y la segunda es el servidor de websockets. Además, en el archivo estamos creando una red entre ambas aplicaciones que permite la comunicación interna de ambas, este paso no es necesario para realizar el despliegue en Okteto, pero si puede llegar a ser necesario en caso de realizar el despliegue en otros sistemas, como en local. Las variables de entorno las utilizan las aplicaciones para comunicarse entre si.

Para la aplicación webapp es necesario tener en cuenta que la comunicación WebSocket se realiza entre el cliente y el servidor de WebSockets, utilizando la *PUBLIC_WEBSOCKET_URL*. El resto de la comunicación se realiza entre el servidor de la webapp y el servidor de WebSockets, para ello es necesario utilizar la red interna entre ambas aplicaciones.

Una vez dispongamos de un archivo *docker-compose.yaml*, podemos utilizar el siguiente comando para levantar la aplicación.

Si queremos levantar la aplicación en segundo plano, podemos utilizar el siguiente comando en su lugar.

Código 4.16: *docker-compose.yaml* para despliegue en local.

```
version: "3.9"
services:
webapp:
    image: "lorwen/cowordle-webapp:latest"
    ports:
    - 4173:4173
    networks:
    - inner-network
    environment:
    - PUBLIC_WEBSOCKET_URL=http://websockets:9000
    - PUBLIC_WEBSOCKET_EXTERNAL_URL=http://localhost:9000
    depends_on:
    - websockets
websockets:
    image: "lorwen/cowordle-ws:latest"
    ports:
    - 9000:9000
    networks:
    - inner-network
    environment:
    - WEBAPP_EXTERNAL_URL=http://localhost
networks:
    inner-network:
```

Código 4.17: Comando para levantar todas las aplicaciones utilizando Docker.

```
$ docker compose up
```

Código 4.18: Comando para levantar todas las aplicaciones utilizando Docker en segundo plano.

```
$ docker compose up -d
```

En la documentación oficial de Docker Compose explican más opciones para utilizar Docker Compose [10], pero con los comandos mostrados anteriormente es suficiente para levantar CoWordle.

Una vez esté levantada la aplicación, podemos utilizar la URL para entrar en la aplicación desde cualquier navegador.

### 4.5.2. Utilizando Git o GitHub

También está disponible la opción de levantar la aplicación desde el repositorio de GitHub, para ello será necesario descargar el código de ambas aplicaciones.

Es posible descargar el código utilizando git con los comandos mostrados (fragmento de Código 4.19):

Código 4.19: Comandos git para clonar los repositorios.

```
1    $ git clone https://github.com/Dokest/cowordle-ws.git
2    $ git clone https://github.com/Dokest/cowordle-webapp.git
```

Una vez descargado el código, tenemos que levantar ambas aplicaciones: para levantar la webapp, será necesario utilizar el comando 4.20.

Código 4.20: Comando para levantar la webapp.

```
1    $ npm run dev
```

Para levantar el servidor de websockets, será necesario utilizar el comando 4.21.

Código 4.21: Comando para levantar el servidor de Websockets.

```
1    $ deno task dev
```

Una vez levantadas ambas aplicaciones, podemos acceder a CoWordle utilizando la URL http://localhost:4173/.

# 5

# Conclusiones y trabajos futuros

Tal y cómo se planteó para el desarrollo de la aplicación, se han llegado a alcanzar la mayoría de los objetivos, incluso aquellos que parecían más complejos.

## 5.1. Resolución de objetivos

### 5.1.1. Aprendizaje websocket

Este objetivo ha sido cumplido. No solo se han llegado a entender las diferencias entre la comunicación HTTP y websockets, si no que se han construido herramientas y sistemas sobre la comunicación websocket que han permitido realizar el trabajo de manera más sencilla y mantenible.

### 5.1.2. Creación de un juego en una aplicación web

Este objetivo ha sido cumplido, el juego se considera completo pero mejorable. El *game loop* está cerrado y es completamente jugable.

### 5.1.3. Independencia de dispositivos

Se ha conseguido crear una aplicación a la que se puede acceder desde ordenadores de sobremesa, portátiles, tablets y móviles. Además, la aplicación se ha diseñado de manera que en todos los dispositivos se dispone de la misma información.

### 5.1.4. Despliegue

Se han conseguido *Dockerizar* ambas aplicaciones y se provee de un archivo docker-compose que permite levantar ambas de manera sencilla. Se ha puesto especial cuidado en que la aplicación se pueda lanzar en cualquier entorno con independencia de las IPs de los equipos.

## 5.2. Futuras ampliaciones

A futuro la aplicación tiene varias posibles mejoras:

- La creación de un sistema de leaderboard global, para poder comparar tus resultados con los de jugadores de todo el mundo. Este desarrollo se llegó a plantear en una de las revisiones con el tutor.
- Añadir más listas de palabras para jugar, de esta forma podría haber partidas en más lenguajes, o incluso partidas temáticas. El sistema de creación de listas es adaptable, y está preparado para introducir más listas en el futuro sin mucho trabajo.
- Añadir un tutorial para personas que no conozcan el juego de Wordle.
- Estudiar la accesibilidad de la página y mejorarla donde sea necesario.

## 5.3. Conclusiones personales

El trabajo me ha supuesto un gran esfuerzo, un curso entero pensando, programando y trabajando en un único proyecto parece un gran cometido para cualquier estudiante de grado.

Estoy muy contento con el resultado, aunque una vez terminado y analizado, existen muchos sitios donde es mejorable tanto el código, como la arquitectura. Dicho esto, estoy satisfecho con lo bien que han salido para ser la primera vez

que intento hacer un juego en el navegador. También estoy muy contento con el aspecto visual, creo que ha quedado vistoso, y sorprendentemente mejor de lo que me esperaba.

También estoy muy contento, no solo con todo lo aprendido en el área de la programación, si no también con lo aprendido escribiendo este documento. Es la primera vez que tengo que analizar a fondo un trabajo, y he aprendido mucho más de lo que me esperaba haciéndolo.

Con todo lo aprendido, cada vez que quiera emprender un nuevo proyecto lo prepararé de manera diferente, utilizando todo lo aprendido en este.

# Bibliografía


[1] T. Team, *Página principal Typescript*. [Online]. Available: https://www.typescriptlang.org/

[2] N. Team, *Página principal Node.js*. [Online]. Available: https://nodejs.org/es

[3] StackOverflow, *Encuesta, StackOverflow, 2022*. [Online]. Available: https://survey.stackoverflow.co/2022/

[4] D. Team, *Página principal Deno*. [Online]. Available: https://deno.com/runtime

[5] S. Team, *Página principal SvelteKit*. [Online]. Available: https://kit.svelte.dev/

[6] ——, *Página principal Svelte*. [Online]. Available: https://svelte.dev/

[7] T. Team, *Página principal Tailwind*. [Online]. Available: https://tailwindcss.com/

[8] P. Team, *Página principal Playwright*. [Online]. Available: https://playwright.dev/

[9] B. y Joshua Pinter, *Solución al problema de librerías en Ubuntu*. [Online]. Available: https://askubuntu.com/a/310

[10] D. C. Team, *Documentación oficial de Docker Compose up*. [Online]. Available: https://docs.docker.com/engine/reference/commandline/compose_up/