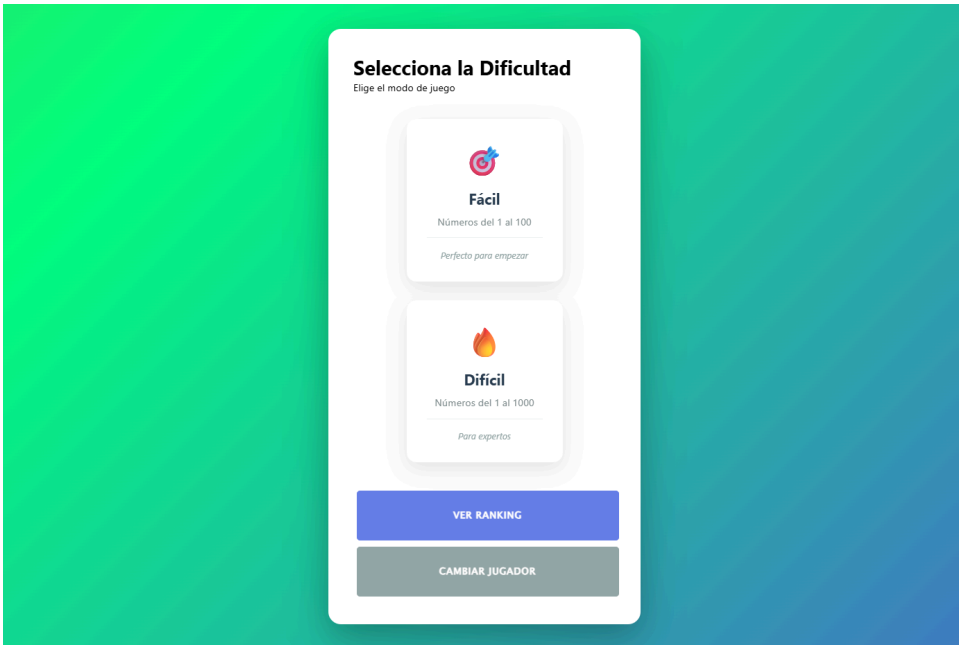

GuessIt

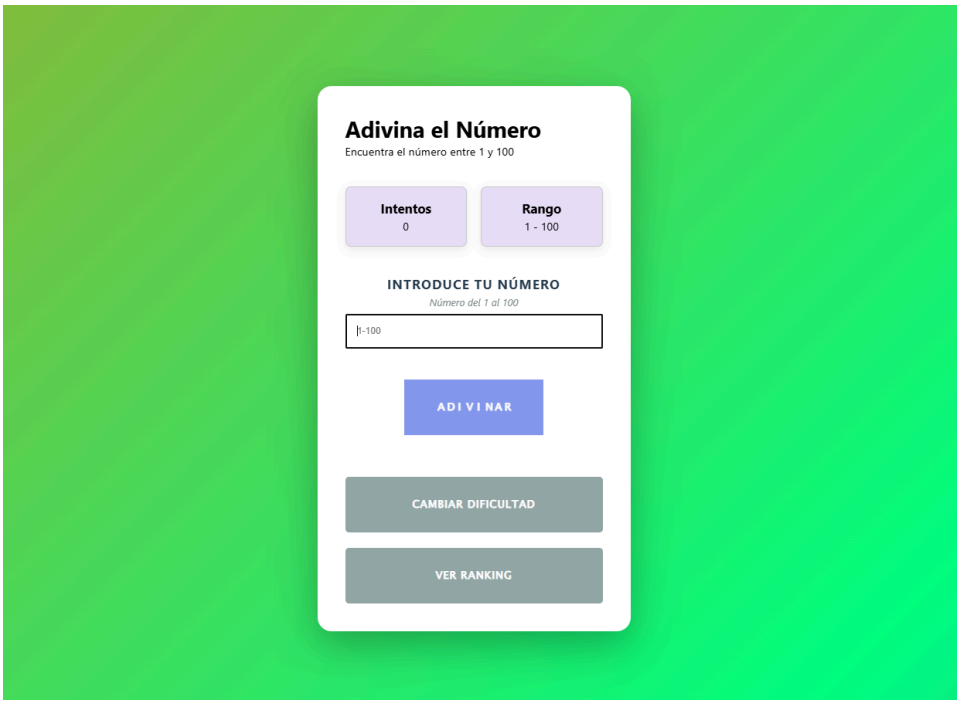
Índice

Explicación detallada

Resumen funcional

La aplicación es un juego **"adivina el número"** con dos niveles de dificultad (1–100 y 1–1000). Pide el nombre del jugador, genera un número aleatorio, registra los intentos, da pistas (MAYOR / MENOR), guarda las partidas por jugador en `localStorage` y muestra un ranking por promedio de intentos. Incluye pequeños toques UX: confeti al ganar, historial de últimos intentos, botones de menú, cambio de jugador y navegación entre pantallas.





Variables y tipos de datos

- Variables de estado (mutables, `let`):
  - `numRand`, `count`, `gameOver`, `attempts`, `currentDifficulty`, `maxNumber`, `confettiAnimationId`.
- Variables relacionadas con jugadores:
  - `players` (array de objetos), `currentPlayer` (string).
- Elementos del DOM (inmutables, `const`): referencias devueltas por `document.getElementById` o `querySelectorAll` (por ejemplo `difficultyScreen`, `gameScreen`, `form`, `guessInput`, etc).

Tipos usados: `Number`, `String`, `Boolean`, `Array`, `Object`.

Estructura con funciones

- `selectDifficulty(difficulty)` — gestiona selección de dificultad, pide nombre si hace falta, configura rango y arranca el juego.
- `startGame()` — inicializa variables del juego, genera el número aleatorio con `random`, resetea la UI y detiene confeti si procede.
- `showGameScreen()` / `showDifficultyScreen()` / `showScreen(screenName)` — cambian pantallas usando clases CSS (`classList.add`/`remove`).
- `random(min, max)` — genera número aleatorio entre `min` y `max` usando `Math.random()` y `Math.floor()`.
- `evaluateGuess(num)` — lógica principal que compara el intento con `numRand`, actualiza `count`, da retroalimentación (MAYOR / MENOR / acierto), guarda la partida, dispara confeti.
- `resetGame()` — reinicia partida llamando a `startGame()`.
- `lanzarConfeti()` — controla la animación de confeti con `requestAnimationFrame` y `cancelAnimationFrame`.
- `saveGame(attempts)` — guarda la partida en el objeto `players` del jugador actual.
- `savePlayers()` — persiste `players` en `localStorage`.
- `showRanking()` / `updateRanking()` — calculan y muestran el ranking.

- `switchPlayer()` — borra el jugador actual y vuelve a la pantalla de selección.

Operadores y control de flujo

- **Condicionales** `if`, `else if`, `else` en muchas partes:
  - Validación de entrada (`isNaN`, rango), comparación de intentos (`num < numRand`, `num > numRand`, `num === numRand`), control de jugador, existencia de players, etc.
- **Operador ternario** para construir el `innerHTML` del ranking (`cond ? valor1 : valor2`).
- **Operadores lógicos** `&&`, `||` implícitos en condicionales (por ejemplo `if (!currentPlayer)`).
- **Comparaciones** estrictas (`===`) y no estrictas donde procede (en código actual usas `===` al comparar `num` con `numRand`).
- **Aritmética** básica: incremento de `count` (`count++`), cálculo de promedio (suma / longitud).
- **Template literals** (backticks `` `...${var}...` ``) para construir texto HTML dinámico y mensajes.

Arrays y métodos de array

- `players` es un array de objetos `{ name, games }`.
- `attempts` guarda los números que ha probado el jugador.
- Métodos usados:
  - `push()` — añadir intentos y partidas.
  - `find()` — buscar si ya existe un jugador por `name`.
  - `map()` — transformar `players` en estructura para ranking.
  - `reduce()` — sumar intentos de todas las partidas de un jugador (`player.games.reduce((sum, game) => sum + game.attempts, 0)`).
  - `filter()` — quitar jugadores sin partidas.
  - `sort()` — ordenar por promedio de intentos.
  - `slice(-5)` y `reverse()` — mostrar los últimos 5 intentos recientes en orden inverso.

- `join('')` — unir strings para generar HTML.

Objetos y JSON

- Cada jugador es un objeto `{ name: string, games: Array }`.
- Cada juego guardado es un objeto `{ date: string, difficulty: string, attempts: number, number: numRand }`.
- Para persistencia en `localStorage` conviertes a JSON con `JSON.stringify(players)` y recuperas con `JSON.parse(...)`.

Eventos y DOM

- `document.addEventListener('DOMContentLoaded', ...)` — inicialización al cargar.
- `element.addEventListener('click', ...)` — por ejemplo en las tarjetas de dificultad.
- `form.addEventListener('submit', ...)` — captura envío del formulario; usas `e.preventDefault()` para evitar recarga de página.
- `document.addEventListener('keydown', ...)` — escuchar tecla `Escape` para volver al menú de dificultad.
  Métodos DOM: `getElementById`, `querySelectorAll`, `createElement`, `appendChild`, `classList.add/remove`, `focus()`.
  Modificación de atributos del input: `guessInput.min`, `guessInput.max`, `guessInput.placeholder`.
- Manipulación de `innerHTML` para cambiar mensajes y listas dinámicamente.

Validación y control de errores

- `isNaN(num)` y comprobación del rango (`num < 1 || num > maxNumber`).

Persistencia: localStorage

- Cargar jugadores al inicio: `JSON.parse(localStorage.getItem('numberGuessPlayers')) || []`.
- Guardar el nombre del jugador actual: `localStorage.setItem('currentPlayerName', currentPlayer)`.
- Guardar el array completo `players` tras cada cambio con `savePlayers()`.

Fechas

Al guardar una partida usas `new Date().toISOString()` para almacenar la fecha en formato ISO (útil para ordenaciones o mostrar después).

Animaciones y temporización

- `lanzarConfeti()` usa `requestAnimationFrame` para lanzar confeti de forma controlada durante 3 segundos (`Date.now() + duration`) y `cancelAnimationFrame` para detener animaciones previas.
- Evitas bucles infinitos: el bucle de animación termina cuando `Date.now() >= end`.

Interacciones con el usuario (UX)

- `prompt()` para pedir nombre.
- `confirm()` para confirmar cambio de jugador.
- Mensajes dinámicos en la UI con `innerHTML` (pistas, felicitación con botón de reinicio, contador de intentos).
- Botón de reinicio (`reset-btn`) insertado dinámicamente con `onclick="resetGame()"` (esto funciona porque `resetGame` es global).
- Botones añadidos dinámicamente en el menú con `addMenuButtons()`.

Cálculo del ranking

En `updateRanking()` se construye un array `rankedPlayers` con:

- `totalAttempts` por jugador usando `reduce`.
- `avgAttempts = totalAttempts / player.games.length` (redondeado con `toFixed(1)`).
- Filtrar jugadores sin partidas (`filter`), y ordenas por `avgAttempts` (`sort((a,b) => a.avgAttempts - b.avgAttempts)`).
- Generar HTML con `map()` que muestra posición, nombre, promedio e indicación si es el jugador actual.

Pequeño ejemplo de flujo

1. Usuario hace clic en dificultad → `selectDifficulty()` pide nombre si no hay.
2. Se fija `maxNumber` y se llaman `startGame()` y `showGameScreen()`.

3. Usuario introduce número y envía formulario → `submit` event → `evaluateGuess(num)`.
4. `evaluateGuess` incrementa `count`, compara, muestra PISTA o mensaje de acierto.
5. Si acierta: `saveGame(count)` añade la partida al jugador y se llama a `lanzarConfeti()`.
6. Si quiere ver ranking → `showRanking()` → `updateRanking()` muestra orden por promedio.

Justificación del diseño

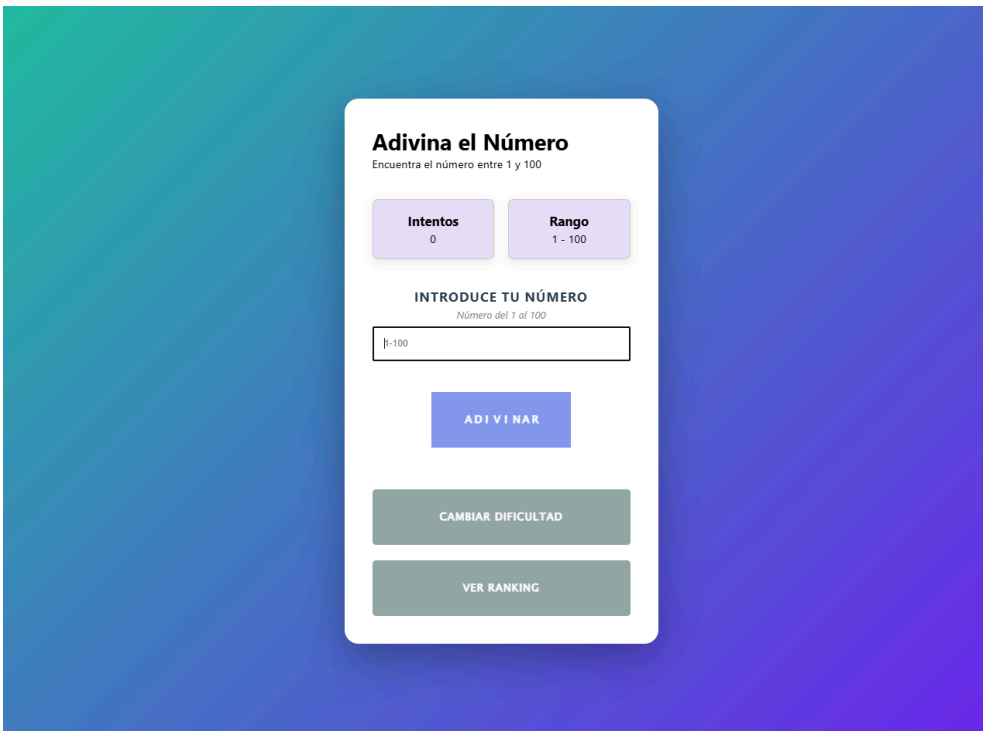
Para este proyecto hemos elegido un diseño moderno y visualmente atractivo basado en **claridad**, **gamificación** y **retroalimentación visual inmediata**, con el objetivo de mejorar la experiencia del usuario mientras juega.

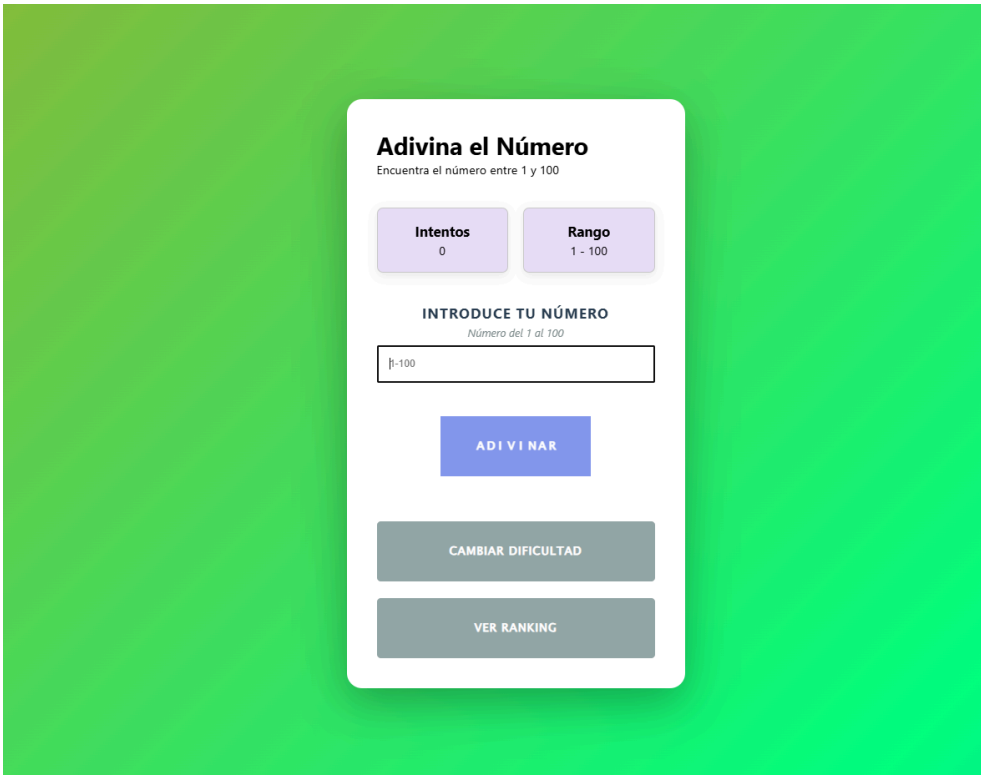
Fondo animado: sensación de "juego dinámico"

El fondo usa un `linear-gradient` animado (`@keyframes gradient`) que va cambiando suavemente los colores.
Esto transmite:

- Energía / ambiente lúdico
- No es una web "estática"
- Acompaña al juego sin distraer

Además se complementa con la animación `lock-reveal`, que usa `mask-image` para hacer un efecto de "apertura" como si se revelara la pantalla inicial. Esto da una **primera impresión mucho más profesional y elegante**.





Caja central (#box): claridad y foco

Se coloca el contenido principal dentro de una *card blanca* centrada, con borde redondeado y sombra (`box-shadow`) para que destaque totalmente sobre el fondo.

Se eligió este estilo porque:

- Aísla la zona de juego del fondo animado

- Facilita la lectura
- Guía visualmente al usuario
- Da sensación de “componente” moderno (estilo app móvil)

Tipografías, botones y “feedback visual”

Los botones tienen:

- Animaciones personalizadas
- Hover states con movimiento de las letras en "Adivinar"
- Colores llamativos

Esto refuerza el **feedback inmediato**, que es clave en juegos.

También el input de número está estilizado para **quitar las flechas de los number inputs**, haciendo que parezca más un campo de texto “propio” del juego → estética más limpia.

Uso de colores según estado del jugador

Se aplican **estados visuales** en `#box`:

Estado	Color / efecto	Significado
Normal	borde neutro	esperando intento
Error	borde rojo + shake	número incorrecto
Success	borde verde + "celebrate"	adivinado

Esto ayuda al jugador a entender rápidamente si va por buen camino **sin necesidad de leer mucho texto** → accesibilidad cognitiva.

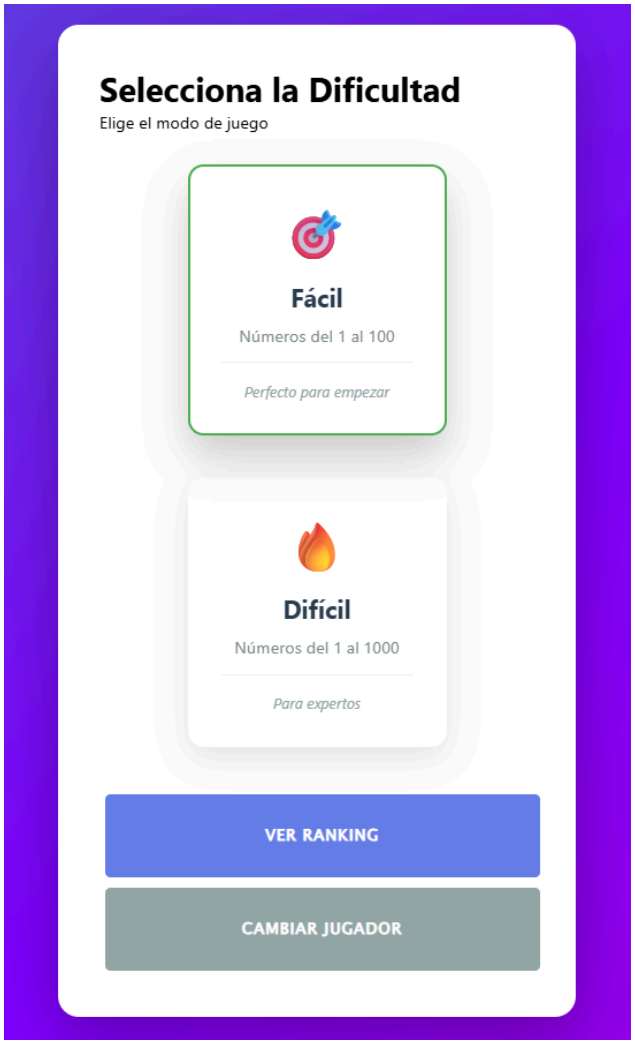
Interfaz modular / pantallas

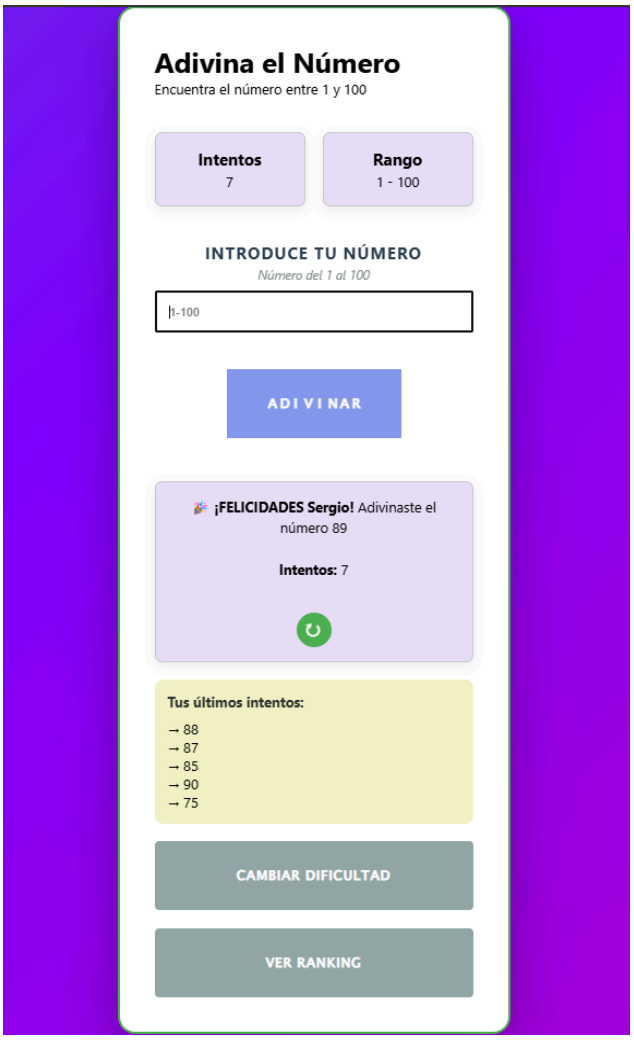
Usamos clases `.screen` y `.active` en lugar de cargar nuevas páginas.
Esto permite:

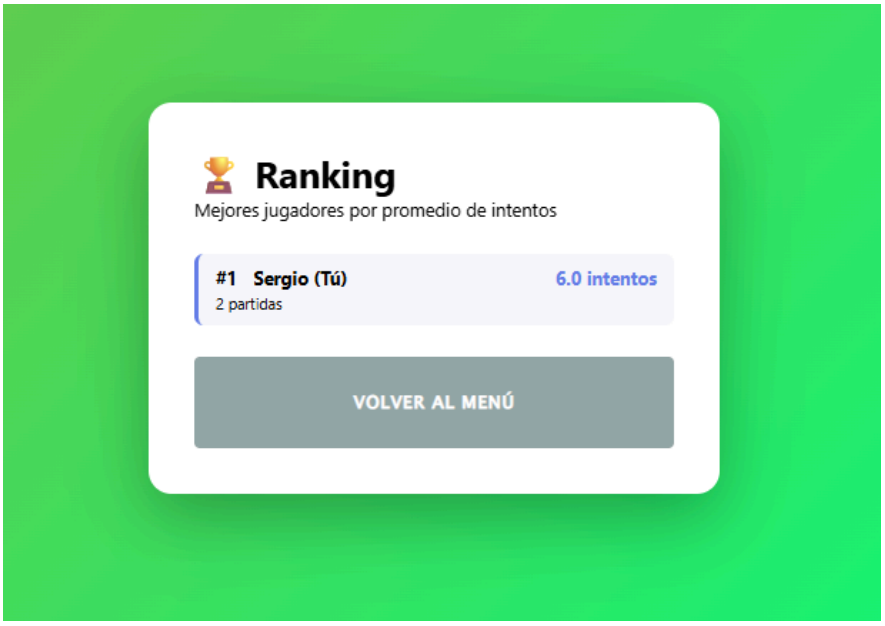
- Transiciones rápidas
- Mejor experiencia tipo aplicación
- Sin recargas
- Flujo guiado: menú → juego → ranking

Estilo de ranking y tarjetas

Las tarjetas de dificultad (`.difficulty-card`) se diseñaron estilo “cards” con sombra y hover para simular botones táctiles (usabilidad móvil).
Para el ranking se usa un borde lateral de color para *escala visual* y destacar al jugador.







Enlace del repositorio GitHub

https://github.com/justo147/GuessItGrupo6

Jira

El trabajo del proyecto GuessIt se organizó en cinco partes principales, siguiendo una secuencia lógica de construcción del producto desde su estructura hasta la presentación visual final.

En primer lugar, se estableció una base sólida mediante la creación de la estructura HTML, que definió la arquitectura del contenido y los elementos esenciales de la interfaz. Esta tarea, desarrollada por Justo Puerto, sirvió como punto de partida sobre el cual se construyeron las siguientes etapas.

Posteriormente, se implementó una capa de estilos inicial con CSS básico, también a cargo de Justo Puerto, con el objetivo de dar formato general al contenido, definir tipografías, colores y primeras reglas de disposición.

Paralelamente, se desarrolló la lógica funcional del proyecto mediante JavaScript, permitiendo la interacción con el usuario y la manipulación dinámica de los elementos de la página. Esta parte fue igualmente responsabilidad de Justo Puerto, quien se encargó de integrar la funcionalidad con la estructura base.

A continuación, se profundizó en la parte estética mediante la implementación de CSS intermedio, tarea realizada por Sergio Perea. En esta fase se optimizaron los estilos, mejorando la disposición visual y aplicando técnicas más avanzadas de diseño, como el uso de flexbox, grid o para dotar al sitio de una apariencia más coherente y profesional.

Finalmente, Juan Jesús García se encargó del desarrollo de CSS avanzado, centrado en la refinación del diseño, la incorporación de animaciones, transiciones y la adaptación del sitio a distintos tamaños de pantalla (diseño responsive), garantizando una experiencia de usuario fluida y moderna.

Gracias a esta división del trabajo, el equipo consiguió un desarrollo colaborativo, progresivo y estructurado, donde cada fase dependía de la anterior, permitiendo consolidar un producto final coherente, funcional y visualmente atractivo.