[Manual del Programador 2](#_Toc195191910)

[1. Introducción 2](#_Toc195191911)

[2. Estructura del Código 2](#_Toc195191912)

[Audio.js 2](#_Toc195191913)

[CampoDeBatalla.js 3](#_Toc195191914)

[Combate.js 5](#_Toc195191915)

[CreadorPersonaje.js 7](#_Toc195191916)

[Enemigo.js 8](#_Toc195191917)

[Exploracion.js 10](#_Toc195191918)

[Inventario.js 11](#_Toc195191919)

[Lobby.js 13](#_Toc195191920)

[Main.js 15](#_Toc195191921)

[Menu.js 17](#_Toc195191922)

[Personaje.js 19](#_Toc195191923)

[Storage.js 21](#_Toc195191924)

[Tienda.js 23](#_Toc195191925)

[index.html 24](#_Toc195191926)

[estilo.css 26](#_Toc195191927)

# Manual del Programador

## 1. Introducción

**Este manual está dirigido a programadores que deseen modificar o ampliar el código de Merida Street Fighter. Aquí encontrarás una descripción de la estructura del código, las clases y funciones principales, y cómo realizar modificaciones o ampliaciones.**

## 2. Estructura del Código

**El código está organizado en varios archivos, cada uno encargado de una parte específica del juego:**

### Audio.js

**Descripción del funcionamiento de AudioManager.js**

**El archivo AudioManager.js está diseñado para gestionar el audio dentro del juego "Merida Street Fighter", específicamente para controlar la música de fondo y ciertos efectos de sonido.**

**Clase AudioManager**

1. **Inicialización del contexto de audio: La clase AudioManager utiliza la API de AudioContext de los navegadores modernos para manejar el audio del juego. Se crea un contexto de audio que permite gestionar la reproducción de sonidos y música de fondo.**
2. **Reproducción de música de fondo: Dentro de la clase, se define un método que permite reproducir la música de fondo del juego. Este método hace uso del elemento <audio> del HTML, el cual tiene un archivo de música asignado. El volumen de la música se ajusta a un valor de 40% para evitar que el sonido sea demasiado alto.**
3. **Pausar la música: Además de reproducir música, el código permite pausarla en cualquier momento. Esto es útil para gestionar los sonidos del juego y adaptarlos a las acciones del usuario o a cambios en la escena del juego.**

**Funcionalidad del Código**

1. **Reproducción al interactuar con la página: Debido a las restricciones de los navegadores, la reproducción automática de música solo es posible después de que el usuario interactúe con la página, como al hacer clic en cualquier parte del sitio. Por lo tanto, el código está diseñado para esperar un clic en el cuerpo del documento para empezar a reproducir la música. Una vez que se produce el primer clic, el evento que detecta ese clic se elimina para evitar que se repita.**
2. **Gestión de efectos de sonido: El código también gestiona un efecto de sonido para reproducirlo al hacer clic en el cuerpo del documento. Este sonido se prepara utilizando la clase Audio, pero no se reproduce automáticamente hasta que se lo indique el código.**

### CampoDeBatalla.js

**El archivo CampoDeBatalla.js se encarga de gestionar el combate entre un personaje (jugador) y un enemigo, así como de gestionar los eventos aleatorios que ocurren durante la batalla. Además, incluye funciones para realizar acciones de combate (como atacar, defender, usar objetos, etc.), actualizar la interfaz de usuario, y finalizar el combate cuando uno de los dos personajes (jugador o enemigo) pierde toda su vida.**

**Clases y Funciones Principales**

1. **Clase Combate**
   * **Atributos:**
     + **personaje: Referencia al objeto del jugador.**
     + **enemigo: Referencia al objeto del enemigo.**
   * **Métodos:**
     + **turnoJugador(): Representa el turno del jugador. Llama al método atacar del jugador, pasando al enemigo como argumento.**
     + **turnoEnemigo(): Representa el turno del enemigo. Llama al método atacar del enemigo, pasando al jugador como argumento.**
     + **haTerminado(): Verifica si el combate ha terminado, comprobando si alguno de los dos personajes tiene vida igual o menor a cero.**
     + **eventoAleatorio(): Ejecuta un evento aleatorio de una lista predefinida de eventos que pueden afectar tanto al jugador como al enemigo. Los eventos son definidos en un array de funciones. Cada evento aplica un modificador de ataque o defensa, y muestra un mensaje al jugador.**
     + **aplicarEvento(mensaje, modificadorAtaque, modificadorDefensa): Aplica el evento aleatorio, modificando las estadísticas de ataque del jugador y defensa del enemigo, además de mostrar un mensaje al jugador.**
2. **Función iniciarCombate(enemigo, zona): Esta función se encarga de inicializar el combate, asignando el enemigo actual y la zona en la que se encuentra. También se asegura de que la zona y los enemigos estén definidos antes de proceder a mostrar el campo de batalla.**
3. **Función realizarAccion(accion): Esta función maneja las diferentes acciones que el jugador puede realizar durante el combate, como atacar, defender, huir, curarse o utilizar objetos. Según la acción seleccionada, la función ejecuta el comportamiento adecuado.**
4. **Función ejecutarAccionDeAtaque(): Esta función ejecuta el ataque del jugador sobre el enemigo. Después de ejecutar el ataque, actualiza la vida del jugador y del enemigo. Si el enemigo sigue con vida, realiza su propio ataque al jugador.**
5. **Función ejecutarAccionDeAtaqueFuerte(): Funciona de manera similar a ejecutarAccionDeAtaque(), pero con un ataque fuerte. El ataque fuerte causa un daño mayor, pero el jugador no podrá atacar en el siguiente turno. Después de realizar el ataque, la función también maneja el ataque del enemigo.**
6. **Función usarObjeto(accion): Permite al jugador usar un objeto de su inventario. La acción se pasa como un parámetro, y la función extrae el índice del objeto desde el string de la acción. Luego, aplica el efecto del objeto, como aumentar el ataque o la defensa, y actualiza la interfaz para reflejar estos cambios.**
7. **Función finalizarCombate(): Esta función se ejecuta cuando el enemigo ha sido derrotado. El jugador recibe experiencia y dinero como recompensa. Además, se guarda el estado de la partida y el enemigo es eliminado de la lista de enemigos de la zona actual.**
8. **Función actualizarVida(): Esta función actualiza la interfaz de usuario, mostrando el estado actual de las barras de vida tanto del jugador como del enemigo. También actualiza la visualización de la experiencia del jugador.**
9. **Función actualizarExperiencia(): Actualiza la barra de experiencia del jugador en función de su experiencia actual y su nivel. La barra se ajusta proporcionalmente al progreso del jugador hacia el siguiente nivel.**

**Estructuras Clave**

1. **Arrays y Funciones de Eventos Aleatorios: La función eventoAleatorio utiliza un array de funciones (eventos) que representa diferentes efectos que pueden ocurrir aleatoriamente durante el combate. Cada función en el array llama a aplicarEvento con diferentes parámetros, como el mensaje del evento y los modificadores de ataque o defensa.**
2. **Inventario del Jugador: El jugador tiene un inventario que está representado como un array de objetos (jugador.inventario). Cada objeto tiene atributos como ataque o defensa que pueden ser modificados cuando el jugador usa un objeto en combate.**
3. **Zonas y Enemigos: Las zonas (zonaActual) tienen una lista de enemigos. Los enemigos que son derrotados se eliminan de esta lista utilizando filter, lo que garantiza que solo los enemigos vivos sigan en la lista.**

### Combate.js

**Clase Combate**

* **Constructor constructor(personaje, enemigo): Esta función inicializa un combate entre el personaje (jugador) y el enemigo. Guarda ambos objetos para su uso posterior en las funciones del combate.**
* **Método turnoJugador(): Se encarga de ejecutar el turno del jugador llamando a su método de ataque sobre el enemigo. Utiliza un bloque try-catch para capturar y manejar cualquier error durante el ataque.**
* **Método turnoEnemigo(): Similar a turnoJugador(), pero para el enemigo, donde el enemigo ataca al jugador. También usa try-catch para manejo de errores.**
* **Método haTerminado(): Verifica si el combate ha terminado, lo cual ocurre cuando la vida de alguno de los combatientes (jugador o enemigo) llega a 0 o menos.**
* **Método eventoAleatorio(): Este método contiene un array de eventos aleatorios que pueden afectar el combate. Los eventos modifican características del jugador o del enemigo (ataque, defensa) de forma aleatoria. Elige un evento al azar y lo ejecuta. Luego, se actualizan los valores de vida del combate.**
* **Método aplicarEvento(mensaje, modificadorAtaque, modificadorDefensa): Aplica los efectos de un evento en el combate. Modifica el ataque del jugador y la defensa del enemigo, y muestra un mensaje relacionado con el evento.**

**Funciones Fuera de la Clase**

* **iniciarCombate(enemigo, zona): Inicializa un combate seleccionando un enemigo y una zona. Se encarga de verificar que la zona y la lista de enemigos estén definidos antes de iniciar el combate. Llama a la función mostrarCampoDeBatalla() para representar el inicio del combate en la interfaz.**
* **realizarAccion(accion): Permite al jugador realizar una acción en su turno (atacar, defender, huir, curarse, usar un objeto, etc.). Cada acción se maneja con una lógica específica. Si el jugador está descansando, no podrá realizar ninguna acción. Dependiendo de la acción seleccionada, se ejecuta el método correspondiente (como ejecutarAccionDeAtaque() para un ataque normal).**
* **ejecutarAccionDeAtaque(): Ejecuta la acción de atacar. Se realiza el ataque del jugador al enemigo, y si el enemigo sobrevive, éste devuelve su ataque al jugador. Se realizan actualizaciones de la vida después de cada acción, y si el enemigo es derrotado, se finaliza el combate.**
* **ejecutarAccionDeAtaqueFuerte(): Similar a ejecutarAccionDeAtaque(), pero ejecuta un ataque fuerte del jugador. Tras realizar el ataque, el enemigo responde con su ataque. Al igual que en el ataque normal, se actualiza la vida y se verifica si el jugador ha sido derrotado o si el combate finaliza.**
* **usarObjeto(accion): Permite al jugador usar un objeto de su inventario. La acción es seleccionada a través de una cadena de texto (accion), que indica el índice del objeto a utilizar. Dependiendo del tipo de objeto (ataque o defensa), se modifican los atributos del jugador o enemigo. Después de usarlo, el objeto se elimina del inventario y se actualiza la interfaz.**
* **finalizarCombate(): Se ejecuta al finalizar el combate, ya sea porque el jugador ha ganado o el enemigo ha sido derrotado. El jugador gana experiencia y dinero como recompensa, y se actualiza el estado de la zona eliminando al enemigo derrotado. Después de finalizar el combate, se muestra el menú principal o el lobby, según el estado del juego.**
* **actualizarVida(): Actualiza visualmente las barras de vida tanto del jugador como del enemigo en la interfaz. Calcula el porcentaje de vida restante de cada uno, y actualiza los valores de vida en pantalla.**
* **actualizarExperiencia(): Actualiza la barra de experiencia del jugador, calculando el porcentaje de progreso hacia el siguiente nivel. Se usa un cálculo basado en el nivel actual del jugador y su experiencia total para mostrar el progreso visualmente.**

**Tipos de Estructuras Utilizadas**

* **Arrays (Listas): En la función eventoAleatorio(), se utiliza un array para almacenar varios eventos aleatorios posibles que se pueden activar durante el combate. Este array es seleccionado al azar mediante Math.random().**
* **Objetos: Los objetos jugador y enemigo son utilizados ampliamente en el código. Estos objetos tienen métodos y propiedades como ataque, defensa, vida, etc., que modifican el flujo del combate.**
* **Strings y Templates Literales: En varias partes del código, se usan strings y template literals (con ${}) para mostrar mensajes dinámicos al jugador, como el ataque realizado, los resultados de las acciones, y el progreso en la vida o experiencia.**
* **Manejo de Errores: Se utilizan bloques try-catch en varias funciones para atrapar errores que puedan ocurrir durante el combate o las acciones del jugador, permitiendo una gestión más robusta de posibles fallos.**

### CreadorPersonaje.js

Este código define una clase llamada **CreadorPersonaje**, que se encarga de gestionar el proceso de creación de un personaje en un juego. La clase proporciona una interfaz interactiva para que el jugador pueda personalizar su personaje, incluyendo su nombre, apariencia, color de cabello, tipo de armadura, estadísticas y arma básica. A continuación se explican los detalles de cada parte del código:

1. **Propiedades de la clase**:
   * **nombre**: Almacena el nombre del personaje.
   * **apariencia**: Define la raza o tipo de apariencia del personaje (Humano, Elfo, Enano).
   * **colorCabello**: Establece el color del cabello del personaje (Castaño, Rubio, Pelirrojo, Negro).
   * **tipoArmadura**: Indica el tipo de armadura que usa el personaje (Cuero, Malla, Placas).
   * **estadisticas**: Un objeto que contiene las estadísticas del personaje, como la vida, el ataque, la defensa y los puntos disponibles para distribuir entre estas características.
   * **armaBasica**: Define el arma básica que el personaje utilizará, inicialmente es null.
2. **Métodos de la clase**:
   * **mostrarCreador()**: Este método genera el HTML necesario para mostrar la interfaz de usuario del creador de personajes. En esta interfaz, el jugador puede seleccionar opciones para personalizar su personaje, como la raza, el color de cabello, la armadura y las estadísticas. Además, el método incluye botones para modificar las estadísticas (vida, ataque, defensa) y un botón para crear el personaje.
   * **modificarEstadistica()**: Permite modificar las estadísticas del personaje (vida, ataque, defensa) al hacer clic en los botones correspondientes. Los puntos disponibles se reducen cada vez que se aumentan las estadísticas.
   * **crearPersonaje()**: Este método se llama cuando el jugador hace clic en el botón "Crear Personaje". Verifica que el nombre del personaje esté ingresado y que todos los puntos disponibles hayan sido distribuidos correctamente. Si todo es correcto, se crea una nueva instancia del objeto Personaje con las estadísticas asignadas, se le equipa un arma y se redirige al jugador al lobby del juego.
   * **asignarCaracteristicasIniciales()**: Según la apariencia elegida (Humano, Elfo, Enano), este método asigna un conjunto predeterminado de estadísticas iniciales (vida, ataque, defensa). Por ejemplo, un Elfo tiene mayor ataque pero menos vida, mientras que un Enano tiene mayor defensa y vida.
   * **actualizarImagenPersonaje()**: Actualiza la imagen del personaje según la apariencia seleccionada. Si el jugador elige "Humano", se mostrará una imagen de un humano; si elige "Elfo", se actualizará a la imagen de un Elfo, y lo mismo para el Enano.
   * **actualizarEstadisticas()**: Este método actualiza la interfaz de usuario para mostrar las estadísticas actuales del personaje y la cantidad de puntos disponibles. Esto asegura que el jugador vea siempre el estado actualizado de su personaje mientras modifica las estadísticas.
3. **Eventos**: El código incluye varios **eventos de cambio** que escuchan los cambios en los campos de la interfaz (como la selección de apariencia, color de cabello y tipo de armadura). Cuando el jugador cambia alguna opción, se actualizan las estadísticas, la imagen del personaje y las características correspondientes.

### Enemigo.js

**Clase Enemigo**

* **Constructor constructor(nombre, vida, ataque, defensa, recompensa, habilidadEspecial): Esta función inicializa un objeto Enemigo con los atributos nombre, vida, ataque, defensa, recompensa y habilidadEspecial. El atributo habilidadEspecial es una función que define el ataque especial del enemigo, que puede variar entre diferentes enemigos.**
* **Método atacar(personaje): Esta función realiza un ataque del enemigo al personaje. Calcula el daño infligido al personaje, tomando en cuenta la defensa del mismo. El daño es al menos 1 (no puede ser negativo). Además, se imprime un mensaje de ataque en consola y se devuelve ese mensaje como un string. Si ocurre un error, se captura y se muestra en consola.**
* **Método usarHabilidadEspecial(personaje): Permite al enemigo usar su habilidad especial durante el combate. La habilidad especial es una función que recibe al personaje como argumento y le inflige daño, según lo definido por el enemigo. Si el enemigo tiene una habilidad especial definida, se ejecuta; si no, no pasa nada. Al igual que el método atacar(), se usa un bloque try-catch para manejar posibles errores.**

**Array enemigosDisponibles: Este array contiene una lista de enemigos disponibles en el juego, cada uno representado como un objeto de la clase Enemigo. Los enemigos tienen diferentes atributos: nombre, vida, ataque, defensa, recompensa y habilidadEspecial.**

* **Ejemplos de Enemigos:**
  + **"🧟 Goblin": Tiene una habilidad especial llamada "Garra Salvaje".**
  + **"Orco": Tiene una habilidad especial llamada "Golpe Brutal".**
  + **"Dragón": Tiene una habilidad especial llamada "Aliento de Fuego".**

**Estructuras Utilizadas**

* **Objetos: Cada enemigo es una instancia de la clase Enemigo con propiedades como nombre, vida, ataque, defensa, recompensa, y habilidadEspecial.**
* **Funciones de Callback (Funciones como habilidades especiales): La habilidad especial de cada enemigo es una función que se pasa como un callback a través del atributo habilidadEspecial. Esta función recibe al personaje como argumento y le inflige un daño determinado. Cada habilidad tiene un efecto único según el enemigo.**
* **Arrays: El array enemigosDisponibles contiene una lista de instancias de enemigos. Esto permite almacenar y acceder a los diferentes enemigos del juego de forma organizada y sencilla.**

**Manejo de Errores: Se usan bloques try-catch en los métodos atacar() y usarHabilidadEspecial() para manejar posibles errores que pudieran ocurrir durante el proceso de ataque o cuando se usa la habilidad especial. Si hay un error, se imprime un mensaje en la consola.**

### Exploracion.js

**Clase Zona**

* **Constructor constructor(nombre, descripcion, enemigos): Esta función crea una nueva instancia de la clase Zona, la cual representa una localización en el mundo del juego. Los parámetros que toma son:**
  + **nombre: El nombre de la zona, que suele estar acompañado de un emoji para representarla visualmente (Ej. "🌳 Emeritus Arboledus").**
  + **descripcion: Una breve descripción del ambiente o características de la zona, como "Un bosque lleno de criaturas mágicas y peligros desconocidos."**
  + **enemigos: Una lista de enemigos disponibles que pueden encontrarse al explorar esta zona, que proviene del array enemigosDisponibles.**

**Array zonas: Este array contiene las diferentes zonas del juego. Cada zona es un objeto creado a partir de la clase Zona.**

* **Ejemplos de Zonas:**
  + **"🌳 Emeritus Arboledus": Un bosque lleno de criaturas mágicas.**
  + **"🌑 Cueva Oscura": Una cueva peligrosa con trampas y enemigos poderosos.**
  + **"🏜 Desierto Árido": Un desierto vasto y peligroso.**

**Función explorarZona(zona): Esta función simula la acción de explorar una zona en el juego.**

* **Tareas que realiza:**
  1. **Muestra una alerta al jugador indicando que está explorando la zona seleccionada y proporciona una breve descripción de la misma.**
  2. **Selecciona aleatoriamente un enemigo de la lista de enemigos de la zona, usando Math.random() y Math.floor() para obtener un índice aleatorio del array zona.enemigos.**
  3. **Inicia el combate con el enemigo seleccionado, llamando a la función iniciarCombate(enemigo, zona).**

**Función mostrarMapa(): Esta función permite al jugador elegir una zona para explorar.**

* **Tareas que realiza:**
  1. **Muestra un mensaje con una lista numerada de zonas disponibles, utilizando zonas.forEach() para iterar sobre el array zonas y mostrar los nombres de cada zona.**
  2. **Solicita al jugador que elija una zona usando prompt(). Si la entrada del jugador es válida (un número entre 1 y el número de zonas disponibles), se llama a explorarZona() para iniciar la exploración de esa zona.**
  3. **Si la entrada del jugador es inválida (fuera de los límites de las zonas disponibles), muestra un mensaje de alerta indicando que la selección fue inválida.**

**Estructuras Utilizadas**

* **Objetos: Cada zona es un objeto de la clase Zona con atributos como nombre, descripcion y enemigos. Cada enemigo en la zona es un objeto de la clase Enemigo, lo que permite crear diferentes tipos de zonas con una variedad de enemigos.**
* **Arrays: El array zonas almacena todas las zonas disponibles en el juego. El array zona.enemigos es utilizado en cada zona para almacenar los enemigos que pueden aparecer en esa zona, seleccionándolos aleatoriamente durante la exploración.**
* **Funciones de Callback: La función iniciarCombate(enemigo, zona) es llamada dentro de la función explorarZona() para iniciar un combate con el enemigo seleccionado en la zona.**

**Manejo de Errores:**

* **Selección Inválida: En la función mostrarMapa(), si el jugador ingresa un número fuera del rango de las zonas disponibles, se muestra un mensaje de alerta indicando que la selección es inválida.**

### Inventario.js

**Función mostrarInventario(): Esta función se encarga de mostrar el inventario del jugador en la interfaz, permitiendo que el jugador vea los objetos que posee y los use o equipe según corresponda.**

**Estructura HTML Generada Dinámicamente**

* **Dentro de la función, se utiliza document.body.innerHTML para actualizar el contenido de la página y mostrar una nueva sección que representa el inventario del jugador. El contenido generado es:**
  + **Un título <h1>Inventario</h1> para identificar la sección.**
  + **Un contenedor <div id="listaInventario"></div>, donde se mostrarán los objetos que el jugador posee.**
  + **Un botón "Volver", que permite al jugador regresar a otra pantalla (en este caso, al lobby) usando la función mostrarLobby().**

**Lista de Objetos del Inventario**

* **Se obtiene el contenedor de inventario con let listaInventario = document.getElementById("listaInventario").**
* **Si el inventario está vacío (jugador.inventario.length === 0), se muestra un mensaje "(Vacío)".**
* **Si hay objetos en el inventario, se recorre la lista de objetos (jugador.inventario.forEach(...)), creando dinámicamente un botón por cada objeto.**

**Creación de Botones para Usar o Equipar Objetos**

* **Para cada objeto en el inventario, se crea un botón (botonUsar) con las siguientes características:**
  + **El botón tiene una clase btn para aplicarle estilos CSS.**
  + **Se establece el contenido del botón, que incluye una imagen representativa del objeto (cuyo nombre se convierte en un formato de nombre de archivo válido) y el nombre del objeto.**
  + **Acción al hacer clic en el botón:**
    - **Si el objeto es una Poción de Curación (su nombre incluye "Poción de Curación"), se ejecuta la función jugador.curar() y se elimina el objeto del inventario usando splice(index, 1).**
    - **Si el objeto tiene una propiedad ataque, significa que es un arma. Se llama a jugador.equiparArma(item), y el objeto se elimina del inventario.**
    - **Si el objeto tiene una propiedad defensa, se considera una armadura y se equipa con jugador.equiparArmadura(item), eliminándolo luego del inventario.**
    - **Si el objeto no es un arma ni una armadura ni una poción, se muestra una alerta diciendo "Objeto no equipable."**

**Actualizar la Vista del Inventario**

* **Después de realizar cualquier acción (usar o equipar un objeto), se vuelve a llamar a mostrarInventario() para actualizar la interfaz y reflejar los cambios, como la eliminación de objetos del inventario.**

**Manejo de Errores**

* **La función está envuelta en un bloque try-catch, lo que permite capturar y mostrar errores que puedan ocurrir durante la ejecución del código. Si algo falla, se imprime el error en la consola con console.error("Error al mostrar el inventario:", error).**

**Interacción con el Jugador**

* **Botón Volver: El botón "Volver" tiene un icono de imagen y llama a la función mostrarLobby(), que probablemente maneja la transición de vuelta al lobby o pantalla principal del juego.**
* **Botones de Uso de Objetos: Los objetos en el inventario tienen un botón que, al hacer clic, ejecuta un proceso dependiendo del tipo de objeto. Ya sea curar, equipar un arma o armadura, o mostrar un mensaje si el objeto no es equipable.**

**Resumen de lo que hace la función: La función mostrarInventario() permite al jugador interactuar con su inventario, mostrar los objetos que posee y hacer uso de ellos, ya sea curándose con pociones, equipando armas o armaduras, o recibiendo un mensaje de alerta si intenta usar un objeto no equipable. Además, al interactuar con los objetos, el inventario se actualiza automáticamente para reflejar los cambios.**

### Lobby.js

**Función mostrarLobby(): Esta función se encarga de mostrar la pantalla principal o "lobby" del juego, donde el jugador puede ver su estado actual (nivel, vida, dinero, experiencia) y elegir qué hacer a continuación, ya sea ir a la tienda, explorar el mapa, ver el inventario o regresar al menú principal.**

**Cálculo de la Experiencia Necesaria para el Siguiente Nivel**

* **const nivelSiguiente = jugador.nivel \* 100: El siguiente nivel se calcula multiplicando el nivel del jugador por 100. Esto implica que, para pasar al siguiente nivel, el jugador debe ganar 100 puntos de experiencia por cada nivel.**
* **const experienciaNecesaria = nivelSiguiente - jugador.experiencia: Se calcula cuánta experiencia falta para alcanzar el siguiente nivel, restando la experiencia actual del jugador de la experiencia necesaria para el siguiente nivel.**

**Actualización de la Interfaz**

* **document.body.innerHTML: Se actualiza el contenido de la página (específicamente el cuerpo del HTML) para mostrar la interfaz de usuario del lobby.**
  + **<h1>🏡 Inicio 🏡</h1>: Un título que identifica la pantalla como el "Inicio" o "Lobby".**
  + **Información del Jugador: Muestra los siguientes datos:**
    - **Nombre y nivel del jugador.**
    - **Vida actual del jugador.**
    - **Dinero disponible en monedas.**
    - **Experiencia actual y la experiencia necesaria para alcanzar el siguiente nivel.**

**Botones de Interacción**

**Los botones permiten al jugador elegir qué hacer a continuación. Cada botón tiene un icono representativo y llama a diferentes funciones cuando se hace clic:**

* **Ir a la Tienda: Al hacer clic en el botón "Ir a la Tienda", se llama a la función tienda.mostrarTienda(), que probablemente maneja la visualización de la tienda donde el jugador puede comprar objetos o mejoras.**
* **Explorar el Mapa: Al hacer clic en "Explorar Mapa", se llama a mostrarMapa(), que probablemente muestra las diferentes zonas que el jugador puede explorar en el juego.**
* **Ver Inventario: Al hacer clic en "Ver Inventario", se llama a mostrarInventario(), que muestra el inventario del jugador y permite usar o equipar objetos.**
* **Volver al Menú Principal: Este botón lleva al jugador al menú principal, llamando a mostrarMenuPrincipal(), que probablemente muestra la pantalla de inicio o menú de opciones.**

**Estructura HTML Generada Dinámicamente**

* **La estructura de la interfaz está compuesta por una serie de elementos HTML generados dinámicamente:**
  + **Un contenedor principal (<div class="container">).**
  + **Un título y párrafos con la información del jugador, como nivel, vida, dinero y experiencia.**
  + **Los botones están agrupados dentro de un contenedor <div class="botones">, con un botón para cada acción que el jugador puede realizar.**
  + **Cada botón tiene un icono correspondiente (<img src="iconos/tienda.png" ...>), que proporciona una representación visual de lo que hace el botón.**

**Manejo de Errores**

* **La función está envuelta en un bloque try-catch, lo que permite capturar y manejar cualquier error que pueda ocurrir durante la ejecución de la función. Si ocurre un error, se imprime un mensaje en la consola con el error: console.error("Error al mostrar el lobby:", error).**

**Interacción con el Jugador**

* **Vista del Jugador: El jugador puede ver su progreso en el juego (nivel, vida, dinero, experiencia).**
* **Acciones Disponibles: El jugador tiene varias opciones de interacción, como ir a la tienda, explorar el mapa, ver su inventario o regresar al menú principal.**
* **Acciones Interactivas: Cada botón es interactivo y realiza una acción diferente al ser presionado, lo que permite al jugador elegir qué hacer a continuación.**

**Resumen de la Función mostrarLobby(): La función mostrarLobby() es crucial porque presenta la pantalla principal del juego, donde el jugador puede ver su estado y elegir qué hacer a continuación. Ofrece botones para interactuar con diferentes aspectos del juego como la tienda, el mapa, el inventario y regresar al menú principal. La función también muestra información relevante del jugador, como el nivel, la vida y la experiencia.**

### Main.js

**Variables Globales**

* **let jugador: Declara la variable jugador, que se supone almacenará los datos del personaje controlado por el jugador (aunque no se ha inicializado aquí).**
* **let indiceEnemigoActual = 0: Define un índice para rastrear al enemigo actual en combate.**
* **let enemigosEnCola = [...enemigosDisponibles]: Crea una copia del arreglo de enemigos disponibles y lo almacena en enemigosEnCola. Esto podría usarse para manejar una cola de enemigos por enfrentar (aunque no se usa activamente en este fragmento de código).**

**Carga Inicial (DOMContentLoaded)**

* **document.addEventListener('DOMContentLoaded', () => { ... }): Este evento asegura que el código dentro de la función solo se ejecute una vez que el documento HTML haya sido completamente cargado y procesado.**
  + **mostrarMenuPrincipal(): Llama a la función mostrarMenuPrincipal() para cargar la pantalla principal del juego (probablemente en otra parte del código, la función mostrarMenuPrincipal se encarga de mostrar la interfaz inicial al jugador). Si hay un error, se captura y muestra en la consola.**

**Función seleccionarEnemigo(): Esta función permite al jugador elegir un enemigo de la lista de enemigos disponibles para comenzar un combate.**

* **let mensaje = "Elige un enemigo:\n": Inicia un mensaje que luego será mostrado al jugador, detallando los enemigos disponibles.**
* **enemigosDisponibles.forEach((enemigo, index) => { ... }): Itera sobre todos los enemigos disponibles, creando una lista de los mismos con su nombre, vida, ataque y defensa.**
* **let eleccion = prompt(mensaje): Muestra una ventana emergente (prompt) para que el jugador seleccione un enemigo, recibiendo la entrada del jugador.**
  + **Si la elección es válida (dentro del rango de los enemigos disponibles), se establece al enemigo actual (enemigoActual = enemigosDisponibles[eleccion - 1]).**
  + **Luego, se llama a mostrarCampoDeBatalla(enemigoActual) para iniciar el combate y mostrar el campo de batalla.**
  + **Si la elección es inválida, muestra una alerta y retorna al Lobby llamando a mostrarLobby().**

**Función iniciarCombate(enemigo, zona): Esta función es llamada cuando se inicia un combate contra un enemigo.**

* **enemigoActual = enemigo: Asigna el enemigo seleccionado a la variable enemigoActual, lo que permite rastrear qué enemigo está siendo enfrentado.**
* **console.log("Iniciando combate con:", enemigoActual.nombre): Imprime en la consola un mensaje de log que indica que el combate está comenzando.**
* **mostrarCampoDeBatalla(enemigoActual): Llama a la función mostrarCampoDeBatalla para mostrar la interfaz de combate con el enemigo seleccionado.**
* **Si ocurre un error durante el proceso, se captura y se imprime en la consola.**

**Interacción del Jugador**

* **Selección de Enemigos: A través de la función seleccionarEnemigo, el jugador puede escoger a qué enemigo enfrentarse. La función presenta una lista de enemigos con estadísticas (vida, ataque y defensa).**
* **Iniciar Combate: Cuando se inicia el combate, la función iniciarCombate prepara el entorno de batalla al seleccionar un enemigo y mostrar el campo de batalla.**

### Menu.js

**Función mostrarMenuPrincipal(): Esta función genera y muestra el contenido HTML del menú principal.**

* **document.body.innerHTML = ...: Establece el HTML de la página con los botones para interactuar con el jugador.**
  + **Botón "Cargar Partida": Al hacer clic en este botón, se llama a continuarPartida() para cargar una partida guardada.**
  + **Botón "Eliminar Datos": Al hacer clic en este botón, se llama a eliminarDatos() para eliminar la partida guardada del almacenamiento local.**
  + **Botón "Crear Personaje": Al hacer clic en este botón, se llama a creadorPersonaje.mostrarCreador() para permitir que el jugador cree un nuevo personaje.**
  + **Botón "Repositorio": Este botón redirige al jugador al repositorio de GitHub donde se encuentra el proyecto del juego.**

**Manejo de errores: Si ocurre algún error durante la ejecución de la función, se captura y muestra en la consola.**

**Función continuarPartida(): Esta función permite al jugador continuar una partida previamente guardada.**

* **const personajeCargado = cargarPartida(): Llama a la función cargarPartida(), que debería devolver el objeto jugador guardado en el almacenamiento local. Si la partida está guardada, el jugador puede continuar desde donde lo dejó.**
* **if (personajeCargado) { ... }: Si se encuentra una partida guardada, asigna el jugador cargado a la variable jugador y muestra un mensaje de bienvenida con el nombre del personaje.**
* **mostrarLobby(): Llama a la función mostrarLobby() para cargar la pantalla de lobby donde se puede gestionar el personaje.**
* **Si no se encuentra una partida guardada, muestra una alerta indicando que no hay ninguna partida guardada.**

**Función eliminarDatos(): Esta función elimina los datos de la partida guardada.**

* **localStorage.removeItem("partida"): Elimina la partida guardada en el almacenamiento local usando la clave "partida".**
* **console.log("🗑 Eliminando partida..."): Muestra un mensaje en la consola indicando que la partida está siendo eliminada.**
* **alert("Partida eliminada."): Muestra una alerta al jugador indicando que la partida ha sido eliminada.**
* **mostrarMenuPrincipal(): Vuelve a mostrar el menú principal después de eliminar la partida.**

**Interacciones del Jugador**

* **Cargar Partida: Si el jugador tiene una partida guardada, puede continuarla haciendo clic en el botón "Cargar Partida". Si no hay una partida guardada, se muestra una alerta.**
* **Eliminar Partida: El jugador puede eliminar la partida guardada haciendo clic en el botón "Eliminar Datos". Esto eliminará cualquier dato de la partida almacenado localmente.**
* **Crear Personaje: Si el jugador quiere crear un nuevo personaje, puede hacer clic en el botón "Crear Personaje", lo que abrirá la función encargada de crear el personaje.**
* **Repositorio de GitHub: El jugador puede acceder al repositorio del proyecto del juego en GitHub haciendo clic en el botón "Repositorio", lo que lo redirige al enlace proporcionado.**

### Personaje.js

**Constructor constructor(): La clase Personaje se inicializa con los siguientes atributos:**

* **nombre: Nombre del personaje.**
* **vida: Vida actual del personaje.**
* **vidaMaxima: La vida máxima del personaje.**
* **ataque: El poder de ataque base del personaje.**
* **defensa: El poder de defensa base del personaje.**
* **dinero: Cantidad de dinero que el personaje tiene.**
* **experiencia: Puntos de experiencia del personaje (comienza en 0).**
* **nivel: Nivel del personaje (comienza en 1).**
* **inventario: Lista de objetos que el personaje tiene en su inventario.**
* **armaEquipada: El arma actualmente equipada, si la hay.**
* **armaduraEquipada: La armadura actualmente equipada, si la hay.**
* **descanso: Si el personaje está descansando (utilizado en el ataque fuerte).**
* **escudoTemporal: Un valor para el escudo temporal que absorbe daño.**
* **contraataque: Si el personaje puede realizar un contraataque después de defenderse.**
* **capacidadInventario: La capacidad máxima del inventario (en este caso, 10).**

**Métodos del Personaje**

* **atacar(enemigo): Este método permite que el personaje ataque a un enemigo.**
  + **Calcula el daño base del ataque del personaje (con o sin arma equipada).**
  + **El daño se ajusta según la defensa del enemigo.**
  + **Muestra un mensaje en la consola con el daño causado.**
* **ataqueFuerte(enemigo): Este método permite al personaje realizar un ataque fuerte.**
  + **Multiplica el ataque base por 1.5 para generar mayor daño.**
  + **Realiza un segundo golpe normal después del primer ataque fuerte.**
  + **Activa el estado de descanso en el personaje (para evitar un ataque inmediato tras el ataque fuerte).**
  + **Muestra los mensajes correspondientes en la consola.**
* **defender(): Este método permite al personaje defenderse.**
  + **Aumenta la defensa del personaje en 5 puntos.**
  + **Activa un escudo temporal que absorbe 10 puntos de daño.**
  + **Habilita el contraataque, lo que permite al personaje responder después de recibir daño.**
  + **Muestra un mensaje de defensa activada.**
* **usarObjeto(objeto): Este método permite al personaje usar un objeto de su inventario.**
  + **Si el objeto es una Poción de Curación, se aumenta la vida del personaje.**
  + **Si el objeto es un arma, el personaje la equipa.**
  + **Si el objeto es una armadura, el personaje la equipa.**
  + **Muestra un mensaje adecuado para cada tipo de objeto utilizado.**
* **curar(): Este método permite al personaje curarse.**
  + **Aumenta la vida del personaje en 20 puntos, asegurándose de no superar la vida máxima.**
  + **Muestra un mensaje de curación.**
* **recibirDanio(danio): Este método maneja el daño que recibe el personaje.**
  + **Si tiene un escudo temporal activo, absorbe parte del daño.**
  + **Si el personaje está defendiendo, el daño se reduce a la mitad.**
  + **Actualiza la vida del personaje y muestra un mensaje con el daño recibido.**
* **subirNivel(): Este método permite al personaje subir de nivel.**
  + **Aumenta el nivel del personaje.**
  + **Incrementa los atributos de vida, ataque y defensa.**
  + **Muestra un mensaje indicando que el personaje ha subido de nivel.**
* **ganarExperiencia(cantidad): Este método permite al personaje ganar experiencia.**
  + **Aumenta la cantidad de experiencia del personaje.**
  + **Si alcanza o supera 100 puntos de experiencia, el personaje sube de nivel.**
  + **Muestra un mensaje con los puntos de experiencia ganados.**
* **agregarAlInventario(objeto): Este método permite al personaje agregar un objeto al inventario.**
  + **Si el inventario no está lleno, el objeto se añade al inventario.**
  + **Si el inventario está lleno, muestra una alerta informando al jugador.**
* **equiparArma(arma): Este método permite al personaje equipar un arma.**
  + **Si ya tiene un arma equipada, la devuelve al inventario.**
  + **Equipar el nuevo arma y actualizar el mensaje correspondiente.**
* **equiparArmadura(armadura): Este método permite al personaje equipar una armadura.**
  + **Equipar la nueva armadura y mostrar un mensaje indicando el cambio.**
* **contraataque(enemigo): Este método permite al personaje realizar un contraataque después de haberse defendido.**
  + **Si el personaje tiene la habilidad de contraataque activada, realiza un golpe al enemigo.**
  + **El contraataque se desactiva después de ser utilizado.**

**Manejo de Errores**

**Cada método tiene un bloque de manejo de errores para capturar y mostrar en consola cualquier problema que pueda surgir durante su ejecución, lo que facilita el diagnóstico de problemas durante el desarrollo.**

### Storage.js

**Función guardarPartida(jugador): Esta función guarda el estado actual del jugador en el localStorage para que pueda ser recuperado en otro momento.**

* **Entrada: Un objeto jugador (instancia de la clase Personaje).**
* **Proceso:**
  + **Verifica si el objeto jugador está definido.**
  + **Crea un objeto datos que contiene las propiedades relevantes del jugador (nombre, vida, ataque, defensa, dinero, nivel, experiencia, inventario, arma equipada y armadura equipada).**
  + **Convierte este objeto datos en una cadena JSON usando JSON.stringify().**
  + **Guarda la cadena JSON en el localStorage bajo la clave "partida".**
* **Salida: No devuelve nada, pero imprime un mensaje de éxito en la consola si la partida se guarda correctamente.**
* **Manejo de errores: Si no se puede guardar la partida (por ejemplo, si el objeto jugador no está definido), se muestra una advertencia en la consola. También maneja cualquier error que ocurra durante el proceso de almacenamiento.**

**Función cargarPartida(): Esta función carga el estado guardado de la partida desde el localStorage.**

* **Entrada: No requiere parámetros de entrada.**
* **Proceso:**
  + **Intenta recuperar los datos almacenados bajo la clave "partida" del localStorage usando localStorage.getItem().**
  + **Si los datos están presentes, los convierte de vuelta a un objeto JavaScript usando JSON.parse().**
  + **Crea una nueva instancia de la clase Personaje con los datos recuperados (como nombre, vida, ataque, defensa, dinero, nivel, experiencia, inventario, arma equipada y armadura equipada).**
  + **Si no se encuentran datos guardados, muestra una advertencia en la consola.**
* **Salida: Devuelve el objeto personajeCargado si la carga fue exitosa, o null si no hay partida guardada.**
* **Manejo de errores: Si ocurre un error al intentar cargar la partida (por ejemplo, problemas con el almacenamiento o la conversión de JSON), se muestra un mensaje de error en la consola.**

**Manejo de Errores**

**Ambos métodos (guardar y cargar) cuentan con bloques try...catch para capturar errores en el proceso de guardar o cargar los datos:**

* **Si se intenta guardar una partida sin un objeto jugador válido, o si el proceso de almacenamiento falla, se muestra un mensaje de advertencia o error.**
* **Si no se puede cargar la partida debido a un problema en el almacenamiento o la conversión de JSON, se muestra un mensaje de error en la consola.**

### Tienda.js

**Clase Tienda: Esta clase representa la tienda dentro del juego y tiene dos métodos principales: mostrarTienda() y comprarArticulo(). Además, tiene un conjunto de artículos disponibles para la compra.**

**Propiedades de la clase Tienda**

* **this.articulos: Un arreglo de objetos que representan los artículos disponibles para la compra en la tienda. Cada artículo tiene:**
  + **nombre: El nombre del artículo.**
  + **ataque: Incremento en el atributo de ataque, si aplica.**
  + **defensa: Incremento en el atributo de defensa, si aplica.**
  + **curacion: Valor de curación, si aplica.**
  + **fuerza: Aumento en la fuerza, si aplica.**
  + **precio: El costo del artículo en monedas.**
  + **icono: El nombre del archivo de imagen que representa el artículo.**

**Método mostrarTienda()**

* **Objetivo: Este método muestra la interfaz de la tienda en la página web.**
* **Proceso:**
  + **Limpia el contenido de la página (document.body.innerHTML) y agrega el HTML necesario para mostrar la tienda.**
  + **Muestra la cantidad de dinero que el jugador tiene actualmente (jugador.dinero).**
  + **Crea un botón para cada artículo en el array articulos y lo agrega al contenedor de la tienda.**
  + **Para cada artículo, se muestra su nombre, descripción (si tiene atributos como ataque, curación, etc.), precio y su icono correspondiente.**
  + **Se genera un evento de clic en el botón de cada artículo que ejecuta la compra.**
* **Manejo de errores: Si ocurre un error durante el proceso de mostrar la tienda, se captura y se muestra en la consola.**

**Método comprarArticulo(articulo)**

* **Objetivo: Este método maneja el proceso de compra de un artículo en la tienda.**
* **Proceso:**
  + **Verifica si el jugador tiene suficiente dinero para comprar el artículo.**
  + **Si tiene suficiente dinero:**
    - **Resta el costo del artículo al dinero del jugador.**
    - **Agrega el artículo al inventario del jugador llamando al método agregarAlInventario().**
    - **Muestra un mensaje de éxito mediante una alerta y actualiza la cantidad de dinero en la interfaz.**
    - **Guarda la partida usando la función guardarPartida(jugador) para mantener el progreso.**
    - **Vuelve a mostrar la tienda con los nuevos valores.**
  + **Si no tiene suficiente dinero, muestra una alerta indicando que no puede comprar el artículo.**
* **Manejo de errores: Si ocurre un error durante la compra, se captura y muestra en la consola.**

**Manejo de Errores**

**Ambos métodos (mostrarTienda y comprarArticulo) cuentan con bloques try-catch para manejar errores:**

* **Si ocurre un error al mostrar los artículos o al comprar, el error se captura y se imprime en la consola.**
* **También se manejan errores relacionados con la falta de dinero del jugador.**

### index.html

**Estructura del Documento HTML**

**El archivo index.html está estructurado de la siguiente manera:**

**Encabezado (<head>)**

* **<meta charset="UTF-8">: Define el conjunto de caracteres como UTF-8, que es adecuado para la mayoría de los lenguajes.**
* **<meta name="viewport" content="width=device-width, initial-scale=1.0">: Configura la escala y el tamaño de la página para que sea responsiva en dispositivos móviles.**
* **<title>Merida Street Fighter</title>: El título de la página, que aparecerá en la pestaña del navegador.**
* **<link rel="icon" href="favicon.ico" type="image/x-icon">: Define el icono de la página (favicon).**
* **Enlaces a hojas de estilo y scripts:**
  + **<link rel="stylesheet" href="css/estilo.css">: Vincula el archivo CSS para la estilización de la página.**
  + **Vinculación de archivos JavaScript: Todos los archivos JS necesarios para el funcionamiento del juego están vinculados en el <head>. Todos los archivos están usando el atributo defer para que se carguen después de que se haya cargado el contenido HTML.**

**Cuerpo (<body>)**

**El contenido de la página está organizado dentro del elemento <div id="pantalla-principal" class="container">. Dentro de este contenedor, se encuentran varios botones y un elemento <audio> para manejar la música de fondo.**

* **Botones:**
  + **Cargar Partida: Un botón para cargar la partida guardada. Este botón ejecuta la función continuarPartida() cuando se hace clic en él.**
  + **Eliminar Datos: Un botón para eliminar los datos guardados del juego. Este botón ejecuta la función eliminarDatos() cuando se hace clic en él.**
  + **Crear Personaje: Un botón para acceder a la función de creación de personajes mediante el objeto creadorPersonaje.mostrarCreador().**
  + **Repositorio: Un botón que lleva al usuario al repositorio del juego en GitHub al hacer clic. Está vinculado a un enlace externo mediante el atributo <a>.**
* **Música de Fondo:**
  + **Elemento <audio>: Se incluye un reproductor de audio que se reproduce en bucle. El archivo de audio de fondo es lobby.mp3 y si el navegador no soporta el elemento de audio, se muestra el mensaje "Tu navegador no soporta el elemento de audio."**

**Características del Archivo HTML**

* **Interactividad: Los botones permiten al jugador interactuar con el juego, cargando la partida, eliminando datos o creando un personaje nuevo.**
* **Enlace a GitHub: El botón "Repositorio" lleva al usuario al repositorio en GitHub, lo que facilita el acceso al código fuente del juego.**
* **Música de fondo: La música se reproduce en bucle mientras el jugador está en la pantalla principal, creando una atmósfera envolvente.**

### estilo.css

**Este archivo estilo.css define el diseño visual del juego Merida Street Fighter, utilizando una combinación de fuentes, colores, efectos de animación y estilo de interfaz para mejorar la experiencia del usuario.**

**Estructura General**

* **Fuente y Fondo:**
  + **Se importa la fuente Cinzel desde Google Fonts, una fuente elegante que le da un toque de distinción al juego.**
  + **El fondo de la página está compuesto por un color oscuro con una textura sutil, creada mediante la propiedad background-image.**
* **Contenedor Principal:**
  + **Se usa un contenedor con un fondo semitransparente (rgba(0, 0, 0, 0.7)), bordes redondeados y una sombra para darle profundidad al área central donde se muestran los botones y la interfaz.**

**Estilos de Componentes**

* **Encabezado (h1):**
  + **El título principal tiene un color dorado con un borde inferior para destacarse. Además, se le da un sombra para crear un efecto de profundidad y atraer la atención del usuario.**
* **Botones:**
  + **Los botones (.btn, .btn-accion, .btn-repositorio, etc.) tienen un diseño cohesivo con bordes redondeados y transiciones de color para proporcionar retroalimentación visual al pasar el cursor (hover) y al hacer clic (active).**
  + **Los botones también incluyen iconos alineados con el texto, utilizando una propiedad gap para espaciar ambos elementos.**
  + **Cuando el usuario pasa el ratón sobre un botón, se realiza un cambio en el color de fondo y un ligero aumento de tamaño, lo que mejora la experiencia de interacción.**
* **Contenedores de Combate:**
  + **Los contenedores que albergan los elementos del combate tienen un fondo oscuro con un patrón sutil. Estos contenedores también tienen bordes redondeados y sombras para destacar el área de combate en la interfaz.**
* **Barras de Vida y Experiencia:**
  + **Las barras de vida y experiencia se muestran con un fondo oscuro y las barras se llenan de un color dorado. La animación en la propiedad width hace que el cambio en la cantidad de vida o experiencia se vea fluido, mejorando la experiencia visual del jugador.**
* **Animaciones:**
  + **shake: Se utiliza para hacer que los elementos tiemblen ligeramente, como una forma de notificar al jugador algún cambio o efecto visual importante.**
  + **fadeIn: Se emplea para hacer que los elementos aparezcan suavemente, mejorando la transición visual.**

**Diseño Adaptable**

* **Media Queries: Para dispositivos con pantallas más pequeñas, como teléfonos móviles, se ajustan el tamaño de los botones, la fuente y el espaciado para asegurar que la interfaz sea funcional y atractiva en pantallas más pequeñas.**

**Componentes de Estilo Específicos**

* **Formularios:**
  + **Los campos de texto (input[type="text"]) y los selectores (select.select-style) tienen un diseño limpio con bordes redondeados y sombras suaves para que sean fáciles de usar.**
* **Contenedores de Estadísticas:**
  + **La interfaz de estadísticas (.stat-container) permite al usuario ver y ajustar las estadísticas de su personaje, con botones que tienen una transición de color en el hover.**
* **Botón Crear Personaje:**
  + **Un botón grande (.btn-crear) para crear personajes, con una sombra de caja y una transición de color para que resalte en la interfaz.**
* **Imagen del Personaje:**
  + **Las imágenes de los personajes tienen un tamaño máximo para evitar que se distorsionen y se redondean con bordes suaves. También se les da una sombra para agregarles profundidad visual.**