

Manual de Usuario

Nombre del Proyecto: Plataforma Web Inteligente para Enseñanza Personalizada y Evaluación con IA.

Versión: 1.0

Autores: Gutiérrez Mateo, Moya Dilan

Fecha: 21/05/2025

Institución: Universidad de las Fuerzas Armadas ESPE – Sede Latacunga

Índice

1. Introducción
2. Requisitos del Sistema
3. Acceso al Sistema
4. Descripción de la Interfaz
5. Funcionalidades del Sistema
6. Procedimientos Comunes
7. Preguntas Frecuentes (FAQ)
8. Contacto de Soporte
9. Glosario (opcional)

1. Introducción

Este documento técnico describe cómo fue implementado desde cero el proyecto Plataforma Web Inteligente para la Enseñanza Personalizada y la Evaluación Automática, utilizando tecnologías modernas del lado del cliente como HTML, JavaScript y Tailwind CSS. Esta plataforma está orientada a ofrecer un entorno de aprendizaje digital adaptable, que incorpora principios de inteligencia artificial, interfaces dinámicas y visión por computadora en fases futuras.

La plataforma permite registrar estudiantes y docentes, asignar clases, gestionar contenidos y visualizar información educativa personalizada.

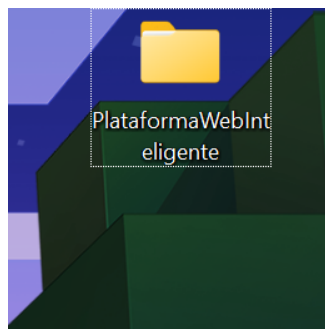
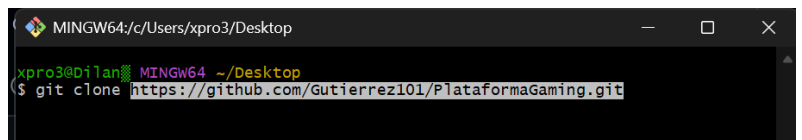
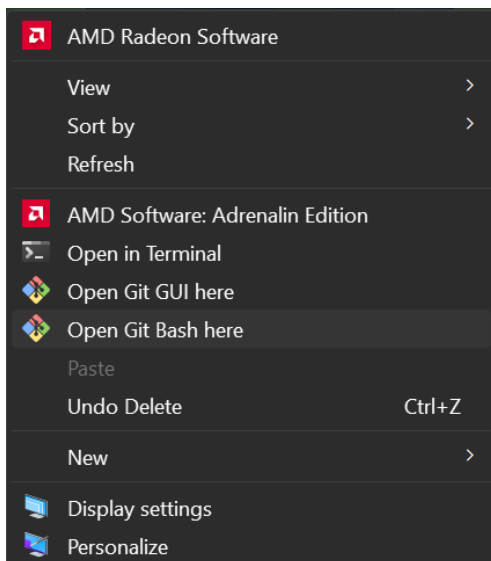
2. Requisitos del Sistema

Para instalar y/o acceder al sistema se necesita:

- Navegador web moderno (Chrome, Firefox, Edge)
- Editor de código (VSCode recomendado)
- Tener instalado Git.

3. Acceso al Sistema

- **Obtener el proyecto Hecho**
 1. **Hacer un git en el escritorio y luego abrirlo en el Studio Code.**
 2. **git clone <https://github.com/Gutierrez101/PlataformaGaming.git>**



- **Creación del Proyecto desde Cero**

1. Crear una carpeta principal:

[PlataformaWebInteligente/](#)

2. Dentro de ella, crear la subcarpeta con los siguiente contenidos:

```
src/  
├─ index.html  
├─ login.html  
├─ register.html  
├─ teacher.html  
├─ class.html  
├─ js/  
│   ├─ models/  
│   ├─ views/  
│   └─ controllers/  
└─ utils/  
└─ images/
```

3. También puedes crear un archivo de estilos personalizados (opcional):

[style.css](#)

4. Inclusión de Tailwind CSS

Tailwind se incluyó mediante CDN, lo que significa que no es necesario instalar ni compilar nada. Se añadió directamente en el <head> de cada archivo HTML como:

```
<link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.19/dist/tailwind.min.css"  
rel="stylesheet">
```

5. Organización del Proyecto (Modelo-Vista-Controlador)

```
PlataformaWebInteligente/  
├─ src/  
│   ├─ index.html  
│   └─ js/  
│       ├─ models/  
│       │   ├─ userModel.js  
│       │   └─ classModel.js  
│       └─ controllers/  
│           ├─ userController.js  
│           ├─ classController.js  
│           └─ teacherController.js  
│       └─ views/  
│           ├─ renderClasses.js  
│           ├─ renderClassDetail.js  
│           └─ renderTeacherDashboard.js  
│       └─ utils/  
│           ├─ validation.js  
│           └─ crypto.js
```

6. Componentes del Patrón MVC

Modelos (carpeta **models/**)

- **userModel.js**

Este archivo define la estructura y operaciones para el manejo de usuarios.

Se implementaron funciones como `addUser`, `getUser`, y `loginUser`, que permiten registrar, obtener y autenticar usuarios desde el `localStorage`.

Se diseñó así para simular una base de datos local, permitiendo pruebas sin backend.

- Registrar nuevos usuarios
- Valida credenciales en el login
- Usa `localStorage` para persistencia simulada

```
// userModel.js      Gutierrez101, yesterday * ajustes, cambios e implementacion del
// Gestión "en memoria" de usuarios

const users = [
  { username: "teacher1", password: "1234", role: "teacher" },
  { username: "student1", password: "abcd", role: "student" },
];

// Devuelve el array entero
export function getUsers() {
  return users;
}

// Añade un usuario (único username)
export function addUser({ username, password, role }) {
  if (users.find(u => u.username === username)) {
    throw new Error("Usuario ya existe");
  }
  users.push({ username, password, role });
}

// Autentica y devuelve el usuario o null
export function authenticate(username, password) {
  return users.find(u => u.username === username && u.password === password) || null;
}
```

- **classModel.js**

Gestiona la lógica de creación y manipulación de clases.

Aquí se crean objetos con nombre, descripción, ID, y se almacenan en `localStorage`.

El modelo incluye funciones como `addClass`, `getAllClasses`, y `getClassById`, pensadas para que tanto docentes como estudiantes puedan interactuar con las clases registradas en la plataforma.

- Crea objetos con título, descripción y lista de alumnos
- Permite búsqueda, almacenamiento y edición

```
const classes = [
  {
    id: 1,
    name: "Marcos de Trabajo",
    image: "images/marco-de-referencia.png",
    retos: [
      { id: 1, title: "React" },
      { id: 2, title: "Next.js" },
    ],
    students: ["student1", "student2"]
  },
  {
    id: 2,
    name: "Desarrollo web",
    image: "images/software.png",
    retos: [
      { id: 1, title: "HTML5" },
      { id: 2, title: "CSS" }
    ],
    students: ["student1"]
  },
]
```

Controladores (carpeta [controllers/](#))

- **UserController.js**

Este controlador conecta los formularios de login y registro con el modelo de usuarios.

Maneja eventos como el envío de formularios y valida los datos ingresados antes de pasarlos a `userModel.js`

- Captura eventos del formulario de login y registro
- Llama a funciones del `userModel`
- Realiza validaciones usando `utils/`[validation.js](#)

```

// userController.js
import { authenticate, addUser } from "../models/userModel.js";

export function initAuthPages() {
  // Para login.html
  const loginForm = document.getElementById("login-form");
  if (loginForm) {
    loginForm.addEventListener("submit", e => {
      e.preventDefault();
      const u = e.target.username.value;
      const p = e.target.password.value;
      const user = authenticate(u, p);
      if (!user) {
        document.getElementById("login-error").textContent = "Credenciales inválidas";
      } else {
        localStorage.setItem("currentUser", JSON.stringify(user));
        // Redirige según rol
        window.location = user.role === "teacher" ? "teacher.html" : "index.html";
      }
    });
  }
  // Para logout.html
  // Para register.html
  const regForm = document.getElementById("register-form");
  if (regForm) {
    regForm.addEventListener("submit", e => {
      e.preventDefault();
      const u = e.target.username.value;
      const p = e.target.password.value;
      const role = e.target.role.value;
      try {
        addUser({ username: u, password: p, role });
        window.location = "login.html";
      } catch (err) {
        document.getElementById("register-error").textContent = err.message;
      }
    });
  }
}

```

- **classController.js**

Controla la visualización general de las clases.

Llama a funciones del modelo classModel.js para obtener las clases y usa vistas como renderClasses.js para mostrarlas.

La idea fue separar claramente la lógica de obtención de datos de la presentación visual, como lo dicta MVC.

- Controla la visualización de clases
- Se apoya en classModel.js y renderClasses.js

```
// classController.js
import { getClasses } from "../models/classModel.js";
import { renderClasses } from "../views/renderClasses.js";

export function initClassDisplay() {
  // Si no hay user logueado, envío a login
  if (!localStorage.getItem("currentUser")) {
    return window.location = "login.html";
  }
  const user = JSON.parse(localStorage.getItem("currentUser"));
  renderClasses(getClasses(), user);
}
```

- **teacherController.js**

Diseñado para gestionar funciones exclusivas del rol docente.

Permite crear nuevas clases, editarlas y visualizar el dashboard personalizado.

Este controlador conecta los formularios del docente con classModel.js y actualiza la interfaz con renderTeacherDashboard.js.

- Funcionalidades exclusivas del perfil docente
- Maneja creación de clases y visualización en el dashboard

```
1 // teacherController.js
2 import { getClassesByTeacher } from "../models/classModel.js";
3 import { renderTeacherDashboard } from "../views/renderTeacherDashboard.js";
4
5 export function initTeacherDashboard() {
6   const user = JSON.parse(localStorage.getItem("currentUser") || "{}");
7   if (user.role !== "teacher") {
8     return window.location = "login.html";
9   }
10  renderTeacherDashboard(getClassesByTeacher(user.username));
11 }
```

- **classDetailController.js**

Se encarga de cargar los detalles completos de una clase específica cuando el usuario la selecciona.

Toma el ID de la clase desde el localStorage, busca los datos en el modelo, y los envía a la vista correspondiente.

Este controlador facilita la navegación profunda sin recargar toda la interfaz.


```
// classDetailController.js
import { getClassById } from "../models/classModel.js";
import { renderClassDetail } from "../views/renderClassDetail.js";

export function initClassDetail() {
  const clsId = localStorage.getItem("selectedClassId");
  if (!clsId) return window.location = "index.html";
  renderClassDetail(getClassById(clsId));
}
```

Vistas (carpeta [views/](#))

- **renderClasses.js**

Este archivo toma un arreglo de clases y lo renderiza visualmente en el HTML.

Se usaron componentes Tailwind para que cada clase se vea como una tarjeta con su título y descripción.

Fue implementado así para que la interfaz se construyera dinámicamente según los datos reales cargados.

- Renderiza lista de clases en pantalla
- Usa clases de Tailwind para el diseño

```
// renderClasses.js
export function renderClasses(classes, user) {
  const container = document.getElementById("class-container");
  if (!container) return;
  container.innerHTML = "";

  classes.forEach(c => {
    const card = document.createElement("div");
    card.className = "bg-white shadow-md rounded p-4 mb-4 cursor-pointer";
    card.innerHTML = `
      
      <h3 class="text-xl font-bold">${c.name}</h3>
    `;
    card.addEventListener("click", () => {
      localStorage.setItem("selectedClassId", c.id);
      window.location = "class.html";
    });
    container.appendChild(card);
  });

  // Muestra avatar del usuario arriba si quieres
  const avatar = document.getElementById("user-avatar");
  if (avatar) avatar.src = "jugador.png";
}
```

- **renderClassDetail.js**

Renderiza la información detallada de una clase seleccionada por el usuario.

Muestra nombre, descripción, y otras propiedades adicionales si se expanden en futuras versiones.

Permite que los usuarios tengan una experiencia contextualizada sin salir del flujo actual.

- Muestra información detallada de una clase seleccionada

```
// renderClassDetail.js
export function renderClassDetail(cls) {
  if (!cls) return;
  document.getElementById("class-title").textContent = cls.name;
  document.getElementById("class-teacher").textContent = "Docente: " + cls.teacher;

  const retosList = document.getElementById("retos-list");
  cls.retos.forEach(r => {
    const li = document.createElement("li");
    li.textContent = r.title;
    retosList.appendChild(li);
  });
}
```

- **renderTeacherDashboard.js**

Esta vista organiza y muestra las funciones que tiene el docente al iniciar sesión.

Aquí se listan las clases creadas, con botones para crear nuevas, editar o eliminar.

Se usó para brindar un espacio exclusivo y práctico para el docente, mejorando la experiencia de administración.

- Panel de administración del docente, con estadísticas y opciones

```
// renderTeacherDashboard.js
export function renderTeacherDashboard(classes) {
  const container = document.getElementById("teacher-container");
  classes.forEach(c => {
    const section = document.createElement("section");
    section.className = "bg-white shadow-md rounded p-4 mb-6";
    section.innerHTML = `
      <h2 class="text-2xl font-semibold mb-2">${c.name}</h2>
      <h3 class="font-bold">Alumnos:</h3>
      <ul id="list-${c.id}" class="list-disc list-inside"></ul>
    `;
    container.appendChild(section);
    const ul = section.querySelector("ul");
    c.students.forEach(s => {
      const li = document.createElement("li");
      li.textContent = s;
      ul.appendChild(li);
    });
  });
}
```

Utilidades (utils/)

- **validation.js**

Funciones para validar formularios: correos, contraseñas, etc.

```
export function validateUsername(username) {
  const sanitized = username.trim();
  if (!sanitized) throw new Error("El nombre de usuario no puede estar vacío.");
  if (sanitized.length < 3) throw new Error("El nombre de usuario debe tener al menos 3 caracteres.");
  if (!/^[a-zA-Z0-9_]+$/.test(sanitized)) throw new Error("El nombre de usuario solo puede contener letras, números y guiones bajos.");
  return sanitized;
}

export function validatePassword(password) {
  if (!password) throw new Error("La contraseña no puede estar vacía.");
  if (password.length < 6) throw new Error("La contraseña debe tener al menos 6 caracteres.");
  return password;
}

export function sanitizeText(text) {
  const div = document.createElement('div');
  div.textContent = text;
  return div.innerHTML;
}
```

- **crypto.js**

Hashing básico de contraseñas (simulado)

```
// Funciones para hashear con SHA-256 usando SubtleCrypto

// Convierte ArrayBuffer a cadena hex
function bufferToHex(buffer) {
  const bytes = new Uint8Array(buffer);
  return Array.from(bytes).map(b => b.toString(16).padStart(2, '0')).join('');
}

// Hashea una cadena (p. ej. contraseña) y devuelve su hex digest
export async function hashSHA256(text) {
  const encoder = new TextEncoder();
  const data = encoder.encode(text);
  const hashBuffer = await crypto.subtle.digest('SHA-256', data);
  return bufferToHex(hashBuffer);
}
```

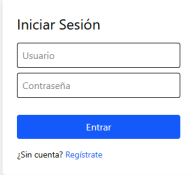
7. Conexión entre los Componentes

El flujo del sistema sigue este orden:

1. El **usuario** interactúa con el **HTML**.
2. El archivo HTML incluye por `<script>` el controlador correspondiente (ej. `UserController.js`).
3. El controlador se comunica con el modelo para obtener o guardar datos.
4. El modelo responde y el controlador actualiza la vista usando funciones de `views/`.
5. Toda la lógica está separada por roles y responsabilidades, facilitando la ampliación del sistema.

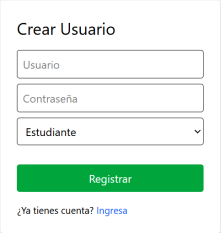
4. Descripción de la Interfaz

Página de inicio: Login



The image shows a login form centered on a light gray background. The form is titled "Iniciar Sesión" and contains two input fields: "Usuario" and "Contraseña". Below these fields is a blue button labeled "Entrar". At the bottom of the form, there is a link that says "¿Sin cuenta? Register".

Registro de Usuario: Iniciar Sesión y Crear Cuenta



Crear Usuario

Usuario

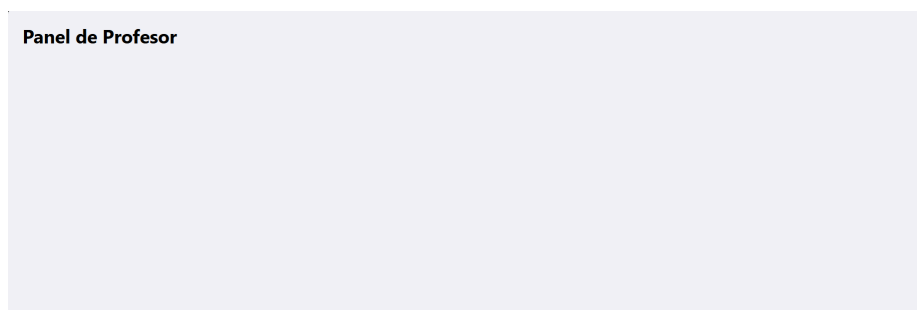
Contraseña

Estudiante

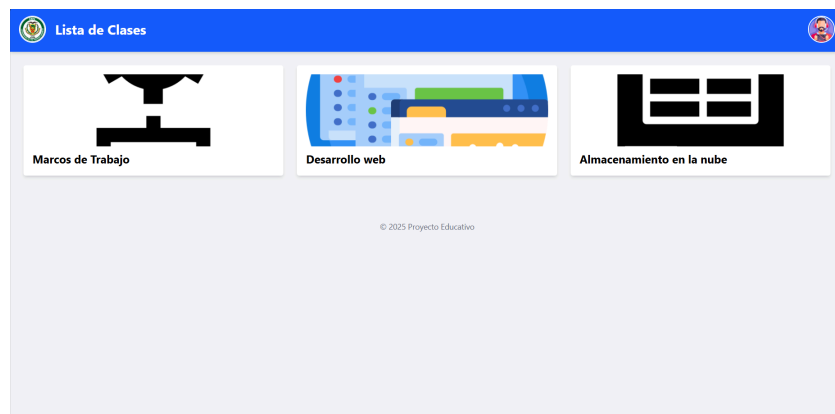
¿Ya tienes cuenta? [Ingresar](#)

Página principal : Introducción al sistema, enlaces al login y al registro.

Dashboard del docente: Panel para gestionar clases, crear nuevas y revisar alumnos registrados.



Panel de clases: Muestra todas las clases disponibles para el usuario. Cada clase incluye detalles y funciones asociadas.



5. Funcionalidades del Sistema

- **Registro de usuarios**
Permite a nuevos usuarios registrarse como estudiantes o docentes.
- **Inicio de sesión con validación**
Los usuarios registrados pueden iniciar sesión según su rol.
- **Gestión de clases**
 - Los docentes pueden crear nuevas clases con descripción e información relevante.
 - Los estudiantes pueden visualizar las clases disponibles.
- **Panel inteligente para docentes**
Sección exclusiva con opciones como: ver estadísticas, controlar clases, añadir contenido educativo.
- **Renderizado dinámico**
Las clases y detalles se cargan y renderizar en tiempo real desde estructuras definidas en JavaScript.

6. Procedimientos Comunes

Iniciar sesión

1. Acceder a login.html
2. Ingresar correo y contraseña
3. Hacer clic en "Iniciar sesión"
4. Accede según el rol (estudiante o docente)

Registrar un nuevo usuario

1. Acceder a register.html
2. Llenar los campos obligatorios
3. Seleccionar tipo de usuario (estudiante/docente)
4. Confirmar y guardar

Crear una nueva clase (solo docentes)

1. Iniciar sesión como docente
2. Acceder a teacher.html
3. Hacer clic en "Crear clase"
4. Completar el formulario de nueva clase
5. Guardar y visualizar en el panel

Ver listado de clases

1. Iniciar sesión
2. Acceder a class.html
3. Se mostrarán todas las clases registradas con detalles como nombre y descripción

7. Preguntas Frecuentes (FAQ)

¿El sistema almacena los datos permanentemente?

Actualmente, el sistema utiliza `localStorage`, por lo tanto los datos se mantienen en el navegador del usuario. No se borra al recargar, pero sí al limpiar el caché.

¿Se puede registrar más de un tipo de usuario?

Sí. El sistema admite registros para estudiantes y docentes, y el contenido cambia según el rol.

¿El sistema está conectado a una base de datos real?

En esta versión no. Todo se maneja localmente. Se recomienda una futura integración con una base de datos externa (Firebase, MongoDB, etc.).

¿Cómo se protege la contraseña?

Se usa un hash simulado en JavaScript (crypto.js) para ofuscar la contraseña antes de guardarla. No es cifrado seguro para producción.

¿Funciona en celulares?

Sí. Gracias a Tailwind, la plataforma es completamente responsive y se adapta a cualquier dispositivo.

8. Contacto de Soporte

Para asistencia técnica o problemas con el sistema:

Nombres: Gutiérrez Mateo, Moya Dilan

Correo: damoya3@espe.edu.ec

Teléfono: 0995834206

9. Glosario

HTML

Es el lenguaje de marcado que se utiliza para estructurar el contenido de las páginas web. Define los elementos básicos como títulos, párrafos, imágenes, formularios, etc.

JavaScript (JS)

Lenguaje de programación del lado del cliente que permite dotar de interactividad a las páginas web. Se encarga de la lógica del sistema, validaciones, eventos y actualizaciones dinámicas del contenido.

Tailwind CSS

Framework de diseño basado en clases utilitarias. Permite construir interfaces modernas directamente desde el HTML sin escribir CSS personalizado. En este proyecto se usó a través de un enlace CDN.

CDN (Content Delivery Network)

Red de distribución de contenido. Se utiliza para cargar bibliotecas como Tailwind CSS desde servidores externos, sin necesidad de instalación local.

Modelo-Vista-Controlador (MVC)

Patrón de arquitectura que divide el sistema en tres componentes: modelo (datos), vista (interfaz) y controlador (lógica de control). Facilita el mantenimiento, escalabilidad y organización del código.

Modelo

Parte del código que gestiona la estructura de datos, como usuarios o clases. Contiene las funciones de creación, almacenamiento y recuperación de datos.

Vista

Componente encargado de mostrar los datos al usuario. Define cómo se presenta la información en pantalla y cómo se actualiza el contenido visual.

Controlador

Gestiona la interacción entre el usuario, los modelos y las vistas. Se encarga de interpretar acciones (como clics o formularios) y decidir cómo responder.

localStorage

Funcionalidad del navegador que permite almacenar datos de forma local en el dispositivo del usuario. Se usa en este proyecto para simular una base de datos.

Responsive Design

Técnica de diseño web que permite que la interfaz se adapte correctamente a diferentes tamaños de pantalla: computadoras, tabletas y teléfonos móviles.

Hashing

Técnica utilizada para proteger datos sensibles, como contraseñas. En este proyecto se simula el proceso de hashing como práctica de seguridad.

Renderizado

Proceso de generar y mostrar elementos en la interfaz visual a partir de los datos. Es dinámico y controlado por JavaScript en este proyecto.

Docente

Rol de usuario con acceso al panel de control para crear clases, visualizar estadísticas y gestionar el entorno educativo.

Estudiante

Usuario que se registra para acceder a clases, contenidos educativos y participar en la plataforma.

Visión por Computadora

Tecnología basada en inteligencia artificial que permite analizar imágenes y videos. En futuras versiones del sistema se implementará para automatizar evaluaciones visuales.