



HTML5, CSS3 y ES6 RegistroLogin localStorage

Centro público integrado de
formación profesional
Alan Turing





Índice

¿Qué veremos en esta sección?

Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Bloque 2 – Login de usuario usando localStorage

Bloque 3 – Visualizar sesión iniciada (localStorage)

Bloque 4 – Paso 3: Flujo general y conclusiones



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Objetivo del bloque

En este primer bloque, vamos a adaptar nuestro formulario de registro para almacenar los datos del usuario en localStorage y permitir que suba una imagen de perfil. Aprovecharemos esta implementación para aprender:

- Cómo se crean objetos en JavaScript
- Cómo funciona localStorage
- Cómo se convierte una imagen a base64 usando FileReader
- Qué es la programación asíncrona en JavaScript: async/await y Promise
- Por qué usamos [0] para acceder al archivo subido



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Objetivo final

Al rellenar el formulario y pulsar “Registrar”, se guarda en localStorage un objeto de usuario con:

- Nombre completo
- Nombre de usuario
- Contraseña
- Imagen de perfil en formato base64

Archivos a modificar o crear

Archivo	Carpeta	Acción
registro.html	raíz	Modificar formulario
registro.js	js/	Reescribir completamente



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Paso 1 – Modificar el formulario en registro.html

Abre el archivo registro.html y reemplaza el contenido del formulario dentro de `<div class="cardBody">` por lo siguiente:

```
<form id="formRegistro">
  <label for="nombre">Nombre real:</label><br>
  <input type="text" id="nombre" name="nombre" required><br><br>

  <label for="usuario">Usuario:</label><br>
  <input type="text" id="usuario" name="usuario" required><br><br>

  <label for="password">Contraseña:</label><br>
  <input type="password" id="password" name="password" required><br><br>

  <label for="imagen">Imagen de perfil:</label><br>
  <input type="file" id="imagen" name="imagen" accept="image/*"><br><br>

  <button type="submit">Registrar</button>
</form>

<p id="mensaje"></p>
```



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

```
<input type="file" id="imagen" accept="image/*">
```

¿Qué hace exactamente?

- Abre el explorador de archivos del sistema.
- Solo permite seleccionar archivos de tipo imagen (.jpg, .png, .gif, .webp, etc.).
- No impide completamente que el usuario intente subir otro tipo de archivo (por ejemplo, pegando una ruta a mano en navegadores antiguos), pero ayuda a prevenir errores.



¿Por qué usamos image/* y no un tipo concreto?

- image/* significa "cualquier tipo de imagen".
- Si quisieras restringirlo más, podrías hacer:

```
accept="image/png, image/jpeg"
```

Pero en este caso, como queremos permitir cualquier tipo de imagen, usamos image/*.



Bloque 1 – Registro de usuario con localStorage y carga de imagen

Paso 2 – Guardar el usuario en localStorage

Reemplazamos todo el código del addEventListener de envío del formulario en registro.js por:

```
// Función para convertir una imagen a base64
async function convertirA64(file) {
  return new Promise((resolve, reject) => {
    const lector = new FileReader();

    lector.onload = () => resolve(lector.result); // Cuando termina de leer
    lector.onerror = () => reject("❌ Error al leer la imagen");

    lector.readAsDataURL(file); // Método para obtener base64
  });
}

formulario.addEventListener("submit", async (e) => {
  e.preventDefault();

  const nombre = document.getElementById("nombre").value.trim();
  const usuario = document.getElementById("usuario").value.trim();
  const password = document.getElementById("password").value.trim();
  const imagenInput = document.getElementById("imagen");

  // Validación básica
  if (!nombre || !usuario || !password) {
    mensaje.textContent = "❗ Rellena todos los campos.";
    return;
  }
}
```

```
// Convertimos la imagen a base64 si el usuario ha subido una
let imagenBase64 = "";
if (imagenInput.files.length > 0) {
  imagenBase64 = await convertirA64(imagenInput.files[0]);
}

// Creamos un objeto con los datos del usuario
const nuevoUsuario = {
  nombre,
  usuario,
  password,
  imagen: imagenBase64
};

// Guardamos el objeto en localStorage (convertido a texto)
localStorage.setItem(usuario, JSON.stringify(nuevoUsuario));
```




Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué es **FileReader** y para qué usamos base64?

- FileReader es una clase que permite leer archivos locales (como imágenes).
- **.readAsDataURL()** convierte la imagen a base64, una cadena de texto segura para almacenar o enviar por red.

¿Por qué `imagenInput.files[0]`?

Chicos, esto es muy importante:

- `imagenInput.files` es una lista de archivos (aunque solo subamos uno).
- Siempre hay que usar el índice `[0]` para acceder al primer archivo.

```
const archivo = imagenInput.files[0]; // Primer archivo seleccionado
```

Si no escribes `[0]`, el navegador no sabrá qué archivo convertir.



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué es un **objeto** en JavaScript?

Un objeto es una estructura que agrupa varios datos bajo un mismo nombre. Es muy útil cuando queremos representar "cosas" del mundo real: una persona, un coche, una cuenta de usuario...

Sintaxis básica:

```
const persona = {  
  nombre: "Laura",  
  edad: 25,  
  ciudad: "Málaga"  
};
```

Cada dato dentro del objeto se llama propiedad, y se define con una clave (nombre, edad, ciudad) y su valor correspondiente.



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué es **localStorage**?

localStorage es una base de datos muy sencilla que tienen todos los navegadores modernos. Sirve para guardar datos en el navegador del usuario, sin fecha de caducidad.

Funciona como un sistema de pares clave/valor.

¿Qué significa clave/valor?

Imagina una tabla con dos columnas:

Clave Valor

```
"juan" {"nombre": "Juan Pérez", "usuario": "juan", ...}
```

```
"lucia" {"nombre": "Lucía Ramos", "usuario": "lucia", ...}
```

- La clave es un identificador único (por ejemplo, el nombre de usuario).
- El valor es un texto que representa todos los datos asociados.



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Por qué usamos **JSON.stringify()**?

JavaScript no puede guardar directamente objetos en localStorage. Solo puede guardar cadenas de texto.

Por eso usamos **JSON.stringify(objeto)**, que convierte un objeto JavaScript a texto:

```
const datos = {  
  nombre: "Mario",  
  usuario: "mario123",  
  password: "1234"  
};  
localStorage.setItem("mario123", JSON.stringify(datos));
```

Luego, cuando queramos leerlo, lo hacemos así:

```
const texto = localStorage.getItem("mario123");  
const objeto = JSON.parse(texto); // Volvemos a tener un objeto
```




Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué se guarda finalmente en el navegador?

Después de registrar a un usuario, si abres las herramientas del navegador (pulsando F12) y vas a la pestaña Almacenamiento (Storage), verás algo así:

Clave	Valor
laura123	{"nombre":"Laura","usuario":"laura123","password":"1234","imagen":"data:image/..."}

El navegador guarda todo eso de forma permanente, hasta que tú mismo lo borres con `localStorage.removeItem()` o `localStorage.clear()`.

¿Por qué esta estructura es útil?

- Nos permite organizar los datos como si fueran fichas de usuario.
- Podemos buscar rápidamente por nombre de usuario.
- Podemos acceder a todas las propiedades del usuario tras hacer `JSON.parse(...)`.
- Es la base para el login, mostrar perfil, recuperar imagen, etc.



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Antes de nada: ¿qué es la **programación asíncrona**?

La programación asíncrona permite ejecutar tareas que tardan un tiempo (como leer un archivo, consultar una base de datos o pedir datos a una API) sin bloquear el resto del programa.

¿Qué hace **async**?

Cuando declaras una función como async, estás diciendo:

“Esta función va a tener tareas que tardan un tiempo. Quiero que me devuelvas una promesa automáticamente.”

```
async function procesarImagen() {  
    const resultado = await convertirA64(archivo);  
    console.log(resultado);  
}
```



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué hace **await**?

await espera el resultado de una promesa, pero sin bloquear el resto del programa.

```
const resultado = await convertirA64(archivo);
```

- Aquí el programa se detiene solo dentro de esta función async, hasta que `convertirA64(archivo)` termine (es decir, hasta que la promesa se resuelva).
- Una vez que termina, `resultado` contiene el valor que devolvió la promesa.

Flujo

Paso 1: Llamamos a la función asíncrona desde el submit
await detiene la ejecución hasta que se obtenga el resultado de `convertirA64(...)`.

Paso 2: Ejecutamos `convertirA64(file)` que devuelve una promesa
Cuando el `FileReader` termine de leer el archivo, resolvemos la promesa con el valor (imagen en base64).



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

¿Qué es new Promise(...)?

Una promesa es un objeto que representa una operación asíncrona que puede:

- cumplirse con éxito → resolve(...)
- fallar con un error → reject(...) (aunque en este caso no lo usamos)

En esta función, devolvemos una promesa con:

```
return new Promise((resolve) => { ... });
```

Esto permite que quien llame a esta función pueda usar await o .then() para esperar a que termine la operación.



Bloque 1 – Registro de usuario con localStorage y carga de imagen en base64

Resultado esperado

Al registrar un usuario:

- Se guarda en localStorage con su nombre, usuario, contraseña e imagen en base64.
- No se necesita servidor, todo es local.

The screenshot displays a web application interface on the left and a browser's developer tools on the right.

Formulario de Registro (Left):

- Nombre real:** Input field containing "jose".
- Usuario:** Input field containing "jota".
- Contraseña:** Input field with masked characters "*****".
- Imagen de perfil:** A button labeled "Seleccionar archivo" followed by the text "perfil.jpg".
- Registrar:** A button to submit the form.
- Feedback:** A green checkmark icon followed by the text "Usuario 'jota' registrado correctamente."
- Navigation:** A link "← Volver a la página principal".

Storage (Right):

The "Storage" tab in the developer tools shows the following data:

Key	Value
jota	{ "nombre": "jose", "usuario": "jota", "password": "dejameya", "imagen": "... (truncated)" }

Below the table, a collapsed entry shows the full JSON object:

```
{ nombre: "jose", usuario: "jota", password: "dejameya", ... }  
imagen: "...  
nombre: "jose"  
password: "dejameya"  
usuario: "jota"
```



Bloque 2 – Login de usuario usando localStorage

Bloque 2 – Login de usuario usando localStorage



Bloque 2 – Login de usuario usando localStorage

Objetivo del bloque

Modificar el comportamiento del formulario de login para que valide los datos usando localStorage en lugar de cookies.

Archivos implicados

Carpeta	Archivo	Acción a realizar
/	login.html	⚠ No hay que modificar nada aquí.
/js/	login.js	✎ Modificar contenido del archivo.



Bloque 2 – Login de usuario usando localStorage

Contexto del cambio

Hasta ahora:

- Validábamos el usuario y la contraseña comprobando si existían en las cookies.

Ahora:

- Vamos a buscar en localStorage si hay un usuario guardado y si la contraseña coincide.

Esto nos permitirá usar los objetos que almacenamos en el Bloque 1, incluyendo:

- nombre,
- usuario,
- contraseña,
- y la imagen en base64.



Bloque 2 – Login de usuario usando localStorage

Paso 1. Cambios en login.js

Ubicación del archivo: /js/login.js

Qué cambia: Reemplazamos la lógica de validación con cookies por una validación basada en objetos almacenados en localStorage.

login.js

```
"use strict";

document.addEventListener("DOMContentLoaded", () => {
  const formulario = document.getElementById("formLogin");
  const mensaje = document.getElementById("mensaje");

  formulario.addEventListener("submit", (e) => {
    e.preventDefault();

    const usuario = document.getElementById("usuario").value.trim();
    const password = document.getElementById("password").value.trim();

    if (!usuario || !password) {
      mensaje.textContent = "❌ Por favor, rellena todos los campos.";
      return;
    }

    // 🧠 Recuperamos el objeto de localStorage (está en formato JSON string)
    const datosJSON = localStorage.getItem(usuario);

    if (!datosJSON) {
      mensaje.textContent = "❌ El usuario no existe.";
      return;
    }

    // 🌱 Convertimos el string a objeto
    const datos = JSON.parse(datosJSON);

    // 🗝 Validamos contraseña
    if (datos.password !== password) {
      mensaje.textContent = "❌ Contraseña incorrecta.";
      return;
    }

    // ✅ Guardamos los datos en localStorage para la sesión actual
    localStorage.setItem("usuarioActivo", usuario);

    mensaje.textContent = `✅ Bienvenido, ${datos.nombre}. Redirigiendo...`;

    setTimeout(() => {
      window.location.href = "index.html";
    }, 1500);
  });
});
```



Bloque 2 – Login de usuario usando localStorage

Qué es `localStorage.getItem(usuario)`?

Busca en `localStorage` un valor asociado a la clave que coincide con el nombre de usuario.

El valor está en formato string (JSON).

Ejemplo:

```
localStorage.getItem("pepe");  
// → '{ "nombre": "Pepe  
Pérez", "usuario": "pepe", "password": "1234", "imagen": "data:image/png;base64,..."  
'
```

¿Por qué usamos `JSON.parse(...)`?

Porque los objetos solo se pueden almacenar como texto plano (string).

Necesitamos convertir el texto a objeto para poder acceder a sus propiedades:

```
datos.password  
datos.nombre  
datos.imagen
```



Bloque 2 – Login de usuario usando localStorage

¿Dónde se guarda el usuario activo?

En esta línea:

```
localStorage.setItem("usuarioActivo", usuario);
```

Esto nos permitirá después recuperar los datos del usuario activo para:

- mostrar su nombre,
- mostrar su foto de perfil,
- ocultar/mostrar accesos privados...

Todo eso lo haremos en el archivo auth.js.



Bloque 2 – Login de usuario usando localStorage

Resultado esperado

1. Si el usuario y la contraseña son correctos, se guarda el nombre de usuario en localStorage bajo la clave usuarioActivo.
2. Se muestra un mensaje de bienvenida con el nombre del usuario (no el "usuario").
3. Se redirige automáticamente a index.html.

The screenshot shows a web application titled "Registro + Login (con localStorage)". The header has links for "w3schools", "javascript.info", "MDN", "Registro", and "Login". The main content area displays "Bienvenido" followed by the instruction "Selecciona un ejercicio desde el menú superior para empezar." and a copyright notice "© José García ©".

On the right, the browser's developer tools are open to the "Storage" tab. The "Local storage" section shows two items:

Key	Value
jota	{ "nombre": "jose", "usuario": "jota", "password": "..." }
usuarioActivo	jota

Below this, an expanded view of the "jota" item shows a JSON object:

```
{ "nombre": "jose", "usuario": "jota", "password": "dejameya", ... }  
imagen: "..."  
nombre: "jose"  
password: "dejameya"  
usuario: "jota"
```



Bloque 3 – Visualizar sesión iniciada (localStorage)

Bloque 3 – Visualizar sesión iniciada (localStorage)



Bloque 3 – Visualizar sesión iniciada (localStorage)

Objetivo del bloque

Tras iniciar sesión, queremos que el sistema:

- Oculte los enlaces públicos (Registro/Login).
- Muestre el nombre real del usuario.
- Muestre su imagen en el header.
- Habilite el botón de cerrar sesión.
- Muestre el menú de ejercicios (privado).



Bloque 3 – Visualizar sesión iniciada (localStorage)

Paso 1 – Modificar auth.js

Explicamos y escribimos solo el código que cambia o se añade.

Añade esta función al final de auth.js (fuera de DOMContentLoaded si existe uno anterior)

```
function obtenerUsuarioActivo() {  
    const usuarioId = localStorage.getItem("usuarioActivo");  
    if (!usuarioId) return null;  
  
    const datos = localStorage.getItem(usuarioId);  
    if (!datos) return null;  
  
    return JSON.parse(datos); // objeto con nombre, usuario, imagen...  
}
```

Explicación:

- Recuperamos el nombre de usuario activo desde localStorage.
- Con esa clave buscamos su objeto de datos (nombre, imagen, etc.).
- Convertimos el string (guardado con JSON.stringify) a objeto JS.



Bloque 3 – Visualizar sesión iniciada (localStorage)

Paso 2 – Reescribir el DOMContentLoaded completo de auth.js

Reemplaza el DOMContentLoaded actual con este:

menu.js

```
"use strict";

document.addEventListener("DOMContentLoaded", () => {
  const enlacesPrivados = document.querySelectorAll(".privado");
  const enlacesPublicos = document.querySelectorAll(".publico");
  const nodoUsuario = document.getElementById("usuario");
  const cerrarSesion = document.getElementById("cerrarSesion");
  const fotoPerfil = document.querySelector(".header-profile img");

  const usuarioActivo = obtenerUsuarioActivo();

  if (usuarioActivo) {
    // Mostrar enlaces privados
    enlacesPrivados.forEach(el => el.style.display = "inline-block");
    enlacesPublicos.forEach(el => el.style.display = "none");

    // Mostrar nombre real del usuario
    nodoUsuario.innerHTML = `<a href="#">👤 ${usuarioActivo.nombre}</a>`;

    // Mostrar imagen personalizada
    if (usuarioActivo.imagen) {
      fotoPerfil.src = usuarioActivo.imagen;
    }

    // Habilitar cierre de sesión
    cerrarSesion.addEventListener("click", (e) => {
      e.preventDefault();
      localStorage.removeItem("usuarioActivo");
      location.reload();
    });
  } else {
    // Usuario NO logueado
    enlacesPrivados.forEach(el => el.style.display = "none");
    enlacesPublicos.forEach(el => el.style.display = "inline-block");
    fotoPerfil.src = "./img/user.png";
  }
});
```




Bloque 3 – Visualizar sesión iniciada (localStorage)

Explicación pedagógica

Chicos, fijaos en lo siguiente:

1. usuarioActivo es solo una clave ("jose123", por ejemplo).
2. Con esa clave buscamos los datos reales del usuario en localStorage (que se guardaron en el registro).
3. Si todo va bien, mostramos:
 - El nombre real del usuario (no el identificador).
 - Su imagen en formato base64.
4. Si no hay sesión:
 - Volvemos a la imagen user.png.
 - Mostramos solo los enlaces Registro y Login.



Bloque 3 – Visualizar sesión iniciada (localStorage)

Resultado esperado

Al iniciar sesión:

- Verás el nombre del usuario (no el identificador).
- Verás la imagen que subió al registrarse.
- El menú mostrará los ejercicios.
- Aparecerá el botón "Cerrar sesión" funcional.





Bloque 4 – Paso 3: Flujo general y conclusiones

Bloque 4 – Paso 3: Flujo general y conclusiones



Bloque 4 – Paso 3: Flujo general y conclusiones

Objetivo del bloque

No vamos a modificar ningún archivo. Vamos a entender cómo fluye la información desde que el usuario se registra hasta que cierra sesión, repasando todos los puntos clave y consolidando el aprendizaje.

¿Qué ocurre en cada momento?

Acción del usuario	Archivo afectado	Qué sucede
Rellena el formulario de registro	registro.html y registro.js	Se crea un objeto usuario con nombre, usuario, contraseña e imagen en base64. Se almacena en localStorage.
Va al formulario de login	login.html y login.js	Se comprueba si el usuario existe en localStorage y si coincide la contraseña. Si todo va bien, se guarda usuarioLogueado en localStorage y se redirige a index.html.
Llega a la página principal	index.html y menu.js	Se lee el objeto usuarioLogueado, se muestra su nombre en el menú, se activa el acceso al área privada (DOM) y se actualiza la foto de perfil.
Pulsa " Cerrar sesión "	index.html y menu.js	Se elimina usuarioLogueado del localStorage, se recarga la página y vuelve a verse el menú público.



Bloque 4 – Paso 3: Flujo general y conclusiones

¿Cómo se comunican los archivos?

registro.html → registra al usuario → registro.js (guarda en localStorage)

login.html → inicia sesión → login.js (verifica usuario → guarda
usuarioLogueado)

index.html → muestra contenido → menu.js (lee usuarioLogueado → personaliza menú)
→ cierra sesión → menu.js (borra usuarioLogueado → recarga la vista)



Bloque 4 – Paso 3: Flujo general y conclusiones

¿Qué se guarda exactamente en localStorage?

Todos los usuarios registrados:

```
{
  "usuario1": {
    "nombre": "Pepe Gómez",
    "usuario": "pepeg",
    "password": "1234",
    "imagen": "data:image/png;base64,..."
  },
  ...
}
```

El usuario que ha iniciado sesión:

```
{
  "nombre": "Pepe Gómez",
  "usuario": "pepeg",
  "password": "1234",
  "imagen": "data:image/png;base64,..."
}
```

Se guarda bajo la clave "usuarioLogueado".



Bloque 4 – Paso 3: Flujo general y conclusiones

Recordatorio técnico importante

- localStorage permite almacenar objetos, pero solo como texto plano (por eso usamos JSON.stringify() y JSON.parse()).
- La imagen se guarda como una cadena base64 usando FileReader.
- Usamos async/await para esperar a que se convierta la imagen antes de guardar.
- En el menú principal, usamos DOMContentLoaded para esperar a que todo esté cargado antes de manipular el DOM.



Ejercicios

HTML5 - CSS3 - JS6 – RegistroLogin-localStorage



Ejercicios



Ejercicios. Enunciado



Ejercicio

Implementa los ejemplos anteriores en tu propia plantilla.