



Clase 06

Diseño y Programación Web

Materia:

Aplicaciones para
Dispositivos Móviles

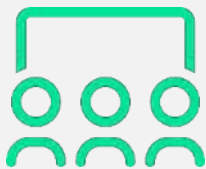
Docente contenidista: GARCIA, Mabel

Revisión: Coordinación

Contenido

Enfoque Offline First	04
LocalStorage en Javascript.....	05
Métodos y Propiedad de localStorage	06
Serializar y Deserializar (JSON)	07
Funciones de Parseo	09
LocalStorage con Vue y componentes	14
Filtros en Vuejs	21
Bibliografía	26

Clase 6



iTe damos la bienvenida a la materia
Aplicaciones para Dispositivos Móviles!

En esta clase vamos a ver los siguientes temas:

- Enfoque Offline First.
- LocalStorage en Javascript.
- Métodos y propiedad de localStorage.
- Serializar y Deserializar (JSON).
- Funciones de parseo.
- LocalStorage con Vue y componentes.
- Filtros en Vuejs.

Enfoque Offline First

El enfoque "**offline first**" es una estrategia de diseño y desarrollo de aplicaciones que **prioriza la funcionalidad y la disponibilidad de datos incluso cuando el dispositivo del usuario no está conectado a internet**. Esta filosofía busca mejorar la experiencia del usuario al permitirle utilizar la aplicación de manera fluida y sin interrupciones, independientemente de su conexión a la red.

Algunos de los beneficios del enfoque "offline first" incluyen:

- **Acceso continuo a datos y funcionalidades:** Con el almacenamiento local como **localStorage** y **sessionStorage**, las aplicaciones pueden almacenar datos en el dispositivo del usuario. Esto significa que incluso si la conexión a internet es intermitente o está ausente, la aplicación puede seguir funcionando y ofreciendo acceso a los datos almacenados.
- **Mejora de la velocidad y la respuesta:** Al guardar ciertos datos y recursos en la memoria local del dispositivo, pueden cargar y responder más rápido a las interacciones del usuario, creando una experiencia más fluida y agradable, incluso en condiciones de red menos óptimas.
- **Reducción de la dependencia de la conectividad:** Resulta especialmente beneficioso en entornos donde la conectividad puede ser inestable o limitada, como áreas rurales, viajes en transporte público o lugares con mala cobertura de red.
- **Mayor robustez y fiabilidad:** Los usuarios pueden continuar trabajando o interactuando con la aplicación incluso en situaciones donde la conexión a internet no está disponible, evitando interrupciones en su flujo de trabajo o actividades.
- **Optimización del uso de datos:** Al almacenar datos localmente y sincronizarlos de manera inteligente cuando se restaura la conexión, estas aplicaciones pueden optimizar el uso de datos y reducir la cantidad de transferencia de información necesaria. Esto puede ser muy útil en dispositivos móviles con planes de datos limitados.

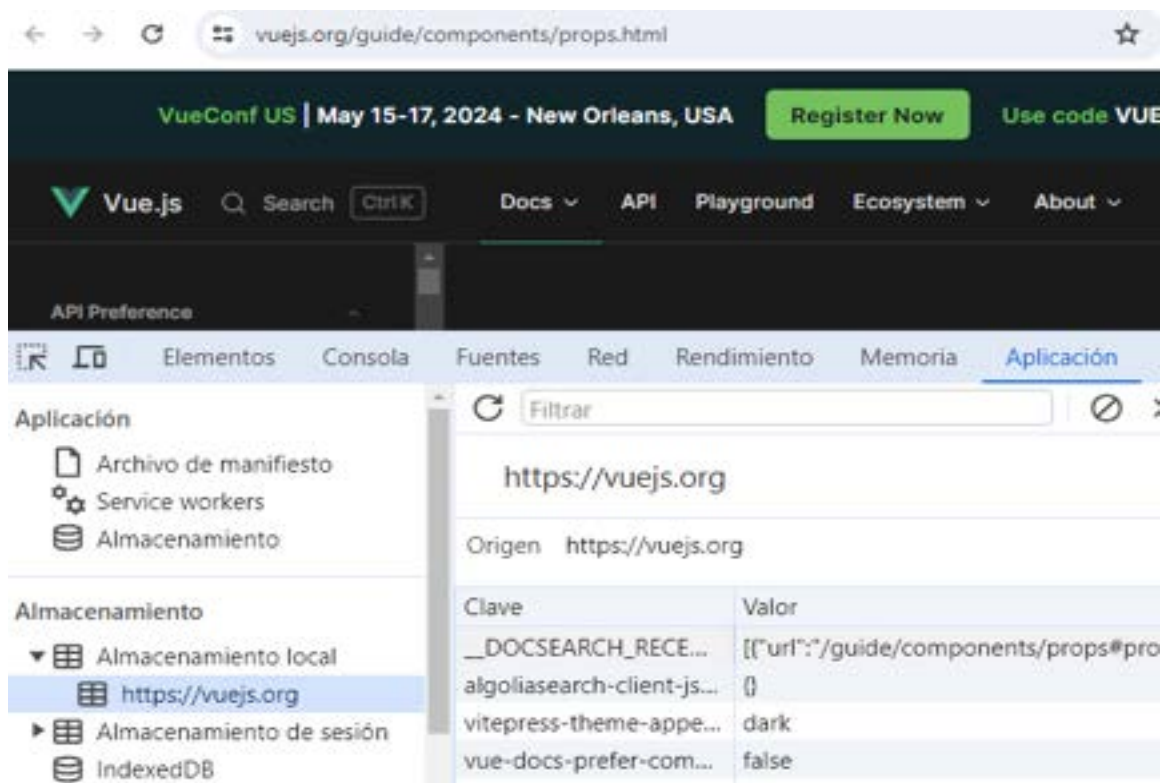
LocalStorage en Javascript

LocalStorage y **sessionStorage** son parte de las **APIs HTML5** que permiten la **persistencia de datos** en una aplicación web.

LocalStorage, específicamente, se usa para almacenar información en forma de **clave y valor**, de manera que los datos persisten incluso después de cerrar la ventana o pestaña del navegador.

Por otro lado, **sessionStorage** funciona de manera similar a **localStorage**, pero con una diferencia fundamental: **los datos almacenados en sessionStorage se mantienen sólo durante la duración de la sesión del navegador**. Por lo tanto, cuando se cierra la ventana o pestaña del navegador, los datos almacenados se borran automáticamente.

Es importante tener en cuenta, cómo se accede desde la consola del navegador cuando trabajamos con estas tecnologías. Dentro de la solapa **Aplicación**, en el menú lateral izquierdo, hay una sección llamada **Almacenamiento** y dentro de este menú encontramos la opción **Almacenamiento Local**. Al hacer click nos muestra cuáles son los elementos que se están almacenado para el sitio que estamos visualizando.



Métodos y Propiedad de localStorage

Si bien localStorage y sessionStorage ofrecen **métodos** similares para la gestión de datos, como setItem, getItem, removeItem, clear, key(n), y la propiedad length. Nosotros nos enfocaremos en **localStorage**.

- **localStorage.setItem(clave, valor);** : Cuando se le proporciona una clave y un valor, añadirá estos al almacenamiento, o actualizará el valor si la clave ya existe.
- **localStorage.getItem(clave);** : Devuelve el valor asociado a la clave especificada. Si la clave no existe, devuelve null.
- **localStorage.removeItem(clave);** : Elimina la clave y su valor asociado del almacenamiento.
- **localStorage.clear();** : Elimina todas las claves y valores que haya en almacenamiento.
- **localStorage.key(n);** : Nos permite obtener la clave en la posición n. Como argumento, se especifica el número de elemento de la clave.

Y la propiedad:

- **localStorage.length;** : Devuelve el número total de elementos guardados en localStorage.

Es importante tener en cuenta que **localStorage tiene una limitación: sólo puede almacenar datos de tipo string.**

Para guardar otros tipos de datos, como booleanos o números, es necesario utilizar **funciones de parseo** de datos para manipular estas estructuras.

JSON (JavaScript Object Notation) es un formato comúnmente utilizado para este propósito, ya que permite representar datos como atributos con valores, facilitando la encapsulación de datos entre el cliente y el servidor.

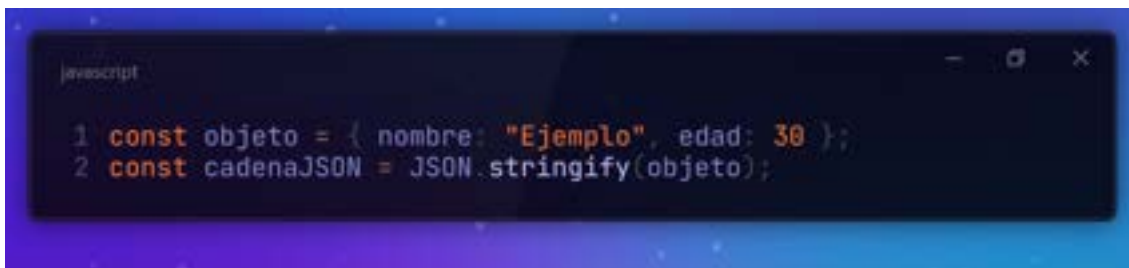
Serializar y Deserializar (JSON)

La **serialización** es el proceso de **convertir un objeto en una cadena de caracteres**, lo que facilita su almacenamiento o transferencia en un medio de almacenamiento, como un archivo o el almacenamiento local del navegador (como localStorage).

Por otro lado, la **deserialización** es la acción opuesta, que consiste en **convertir una cadena de caracteres en un objeto**, permitiendo que la información sea utilizada nuevamente en la aplicación.

JSON (JavaScript Object Notation) es una forma popular de serialización de datos utilizada en aplicaciones web. Proporciona un formato legible por humanos y fácilmente interpretable por las máquinas.

La función **JSON.stringify()** se utiliza para **convertir un objeto JavaScript en una cadena JSON**, lo que facilita su almacenamiento o transmisión. Por ejemplo:

A screenshot of a JavaScript console window with a dark background and blue border. It shows two lines of code: 1. `const objeto = { nombre: "Ejemplo", edad: 30 };` and 2. `const cadenaJSON = JSON.stringify(objeto);`.

```
javascript
1 const objeto = { nombre: "Ejemplo", edad: 30 };
2 const cadenaJSON = JSON.stringify(objeto);
```

En este caso, **cadenaJSON** contendrá la cadena JSON resultante, que sería **`{"nombre":"Ejemplo","edad":30}`**.

Por otro lado, la función **JSON.parse()** realiza la **deserialización**, **convirtiendo una cadena JSON en un objeto JavaScript**. Por ejemplo:

A screenshot of a JavaScript console window with a dark background and blue border. It shows two lines of code: 1. `const cadenaJSON = '{"nombre":"Ejemplo","edad":30}';` and 2. `const objeto = JSON.parse(cadenaJSON);`.

```
javascript
1 const cadenaJSON = '{"nombre":"Ejemplo","edad":30}';
2 const objeto = JSON.parse(cadenaJSON);
```

Ahora, **objeto** contendrá el **objeto JavaScript reconstruido a partir de la cadena JSON**.

Ambas funciones son fundamentales para el manejo de datos en formato JSON, lo que facilita la comunicación y el intercambio de información entre diferentes partes de una aplicación o entre aplicaciones distintas.

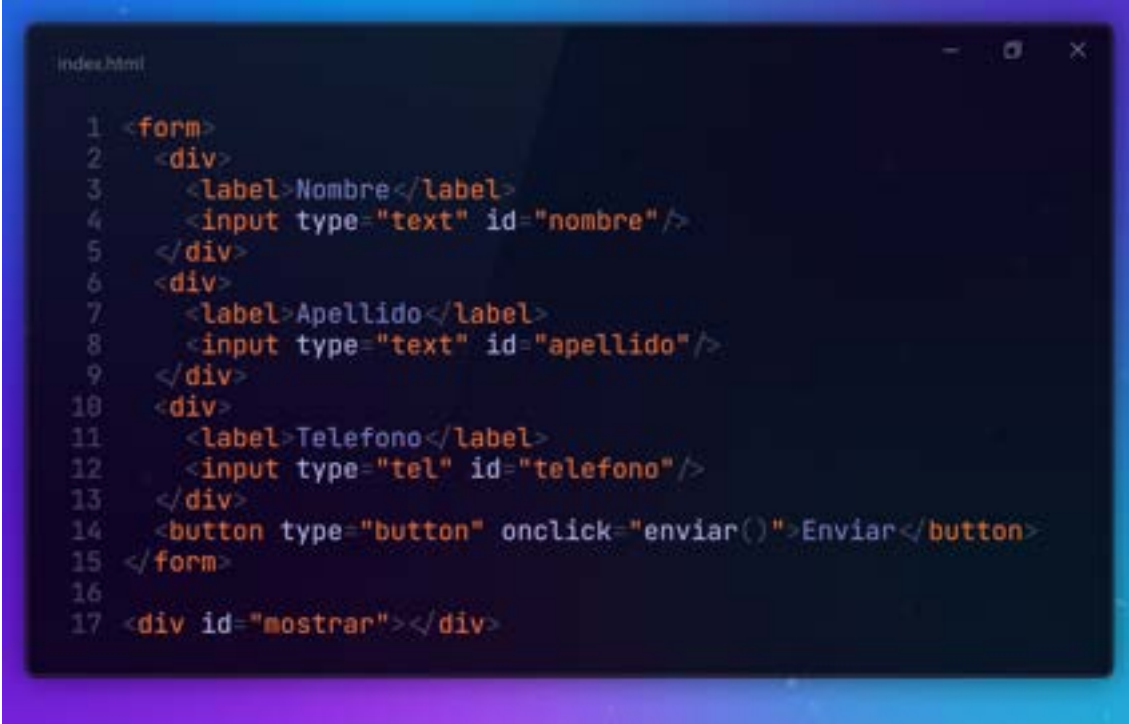
Funciones de Parseo

JSON.stringify() -> Recibe por parámetro un array y devuelve un string en formato JSON - (datos en string).

JSON.parse() -> Recibe por parámetro el string y devuelve un array.

Entonces, si queremos guardar en localStorage datos tenemos que aplicar JSON.stringify() (setter) pero si queremos mostrar la información (requiere iterar), necesitamos usar JSON.parse(). (getter).

En este caso, tenemos un formulario que va a almacenar una agenda de contactos, por lo tanto tendremos que ver qué estructura vamos a implementar desde la lógica para que cada vez que el usuario ejecute la **función enviar**, pueda ir almacenando en localStorage sin errores. Nos centraremos en cómo lo hacíamos habitualmente con Javascript para después poder llevarlo a la práctica con Vue, acorde a la sintaxis y herramientas que nos brinda.

A screenshot of a code editor window titled 'index.html'. The code is written in HTML and defines a form for adding contacts. It includes three input fields for 'Nombre', 'Apellido', and 'Telefono', each with a corresponding label. A button labeled 'Enviar' is attached to the form with an 'onclick' event that calls a function named 'enviar()'. Below the form, there is a placeholder for a 'mostrar' section. The code is as follows:

```
1 <form>
2   <div>
3     <label>Nombre</label>
4     <input type="text" id="nombre" />
5   </div>
6   <div>
7     <label>Apellido</label>
8     <input type="text" id="apellido" />
9   </div>
10  <div>
11    <label>Telefono</label>
12    <input type="tel" id="telefono" />
13  </div>
14  <button type="button" onclick="enviar()">Enviar</button>
15 </form>
16
17 <div id="mostrar"></div>
```

En nuestro archivo javascript, plantearemos la lógica necesaria para ir almacenando cada contacto como un objeto que guardaremos dentro de un array.

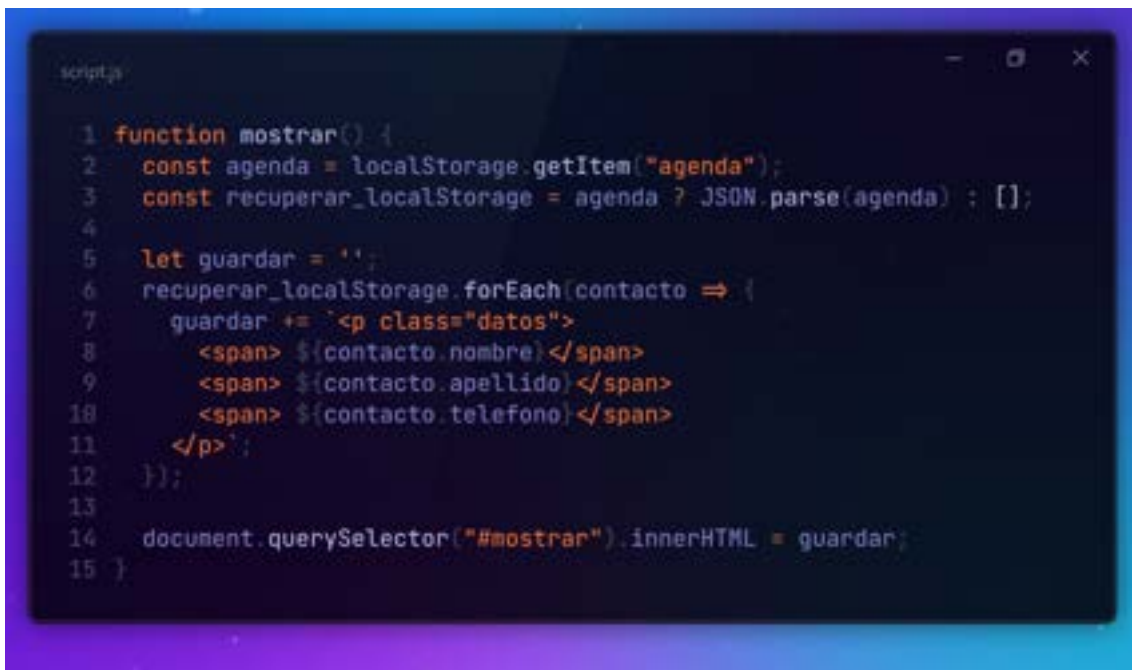
```
script.js
1 function enviar() {
2   const agenda = localStorage.getItem("agenda");
3   const array_para_data = agenda ? JSON.parse(agenda) : [];
4
5   const nombre = document.querySelector("#nombre").value;
6   const apellido = document.querySelector("#apellido").value;
7   const telefono = document.querySelector("#telefono").value;
8
9   const data = { nombre, apellido, telefono, };
10
11  array_para_data.push(data);
12  localStorage.setItem("agenda", JSON.stringify(array_para_data));
13
14  document.querySelector("#form").reset();
15
16  mostrar();
17 }
```

Aclaremos que hace cada línea de código:

- **const agenda = localStorage.getItem("agenda");**: Esta línea obtiene el valor almacenado en localStorage bajo la clave "agenda" y lo asigna a la variable agenda. Si no hay ningún valor almacenado con esa clave, agenda será null.
- **const array_para_data = agenda ? JSON.parse(agenda) : [];**: usamos el **operador ternario para verificar si agenda tiene un valor** (es decir, si hay datos almacenados en "agenda" en localStorage). **Si agenda tiene un valor, se utiliza JSON.parse() para convertir el valor de agenda de formato JSON a un objeto JavaScript y se asigna a array_para_data.** Si agenda es null, se asigna un array vacío [] a array_para_data.
- Se obtienen los **valores de nombre, apellido y teléfono** del formulario utilizando:
document.querySelector("#nombre").value,
document.querySelector("#apellido").value y
document.querySelector("#telefono").value respectivamente.

- Se crea un **objeto** data con las propiedades **nombre**, **apellido** y **teléfono**, utilizando los **valores obtenidos del formulario**.
- **array_para_data.push(data);**: Se **añade el objeto data al array array_para_data**. Esto significa que se está agregando un nuevo contacto a la lista de contactos existente en **array_para_data**.
- **localStorage.setItem("agenda",JSON.stringify(array_para_data));** : Se utiliza JSON.stringify() para **convertir el array array_para_data en una cadena JSON y se guarda en localStorage bajo la clave "agenda"**. Esto **actualiza** la lista de contactos almacenada en localStorage **con el nuevo contacto agregado**.
- Se **restablecen los valores del formulario con la función reset del objeto formulario** para que el usuario pueda ingresar nuevos contactos fácilmente.
- **mostrar();**: Finalmente, se llama a la función mostrar() para **actualizar la visualización de los contactos** en la página, mostrando el nuevo contacto agregado.

La **función mostrar** se encargará de la muestra de los datos en el HTML que se recuperarán desde localStorage:



```

1 function mostrar() {
2   const agenda = localStorage.getItem("agenda");
3   const recuperar_localStorage = agenda ? JSON.parse(agenda) : [];
4
5   let guardar = '';
6   recuperar_localStorage.forEach(contacto => {
7     guardar += '<p class="datos">
8       <span> ${contacto.nombre}</span>
9       <span> ${contacto.apellido}</span>
10      <span> ${contacto.telefono}</span>
11    </p>';
12  });
13
14  document.querySelector("#mostrar").innerHTML = guardar;
15 }

```

Examinemos qué hace cada línea de código:

- **const agenda = localStorage.getItem("agenda");**: Esta línea **obtiene el valor almacenado en localStorage bajo la clave "agenda"** y lo asigna a la variable agenda. Si no hay ningún valor almacenado con esa clave, agenda será null.
- **const recuperar_localStorage = agenda ? JSON.parse(agenda) : [];**: Usamos el **operador ternario** para **verificar si agenda tiene un valor** (es decir, si hay datos almacenados en "agenda" en localStorage). **Si agenda tiene un valor, se utiliza JSON.parse() para convertir el valor de agenda de formato JSON a un objeto JavaScript y se asigna a recuperar_localStorage. Si agenda es null, se asigna un array vacío [] a recuperar_localStorage.**
- Se **inicializa la variable guardar** como una **cadena vacía** para **almacenar el HTML de los contactos que se mostrará en la página.**
- **recuperar_localStorage.forEach(contacto => { ... });**: Usamos el método **forEach()** para **iterar sobre cada contacto en el array recuperar_localStorage**. Para cada contacto, **se concatena una cadena HTML a la variable guardar, que representa la información del contacto en un párrafo con las etiquetas span para cada atributo (nombre, apellido y teléfono).**
- **document.querySelector("#mostrar").innerHTML = guardar;**: Se selecciona el elemento con el **id "mostrar" en el HTML y se le asigna el contenido HTML almacenado en la variable guardar**. Esto actualiza la visualización de los contactos en la página.

En el navegador podemos observar efectivamente la muestra de cada contacto.

LocalStorage

Nombre

Apellido

Telefono

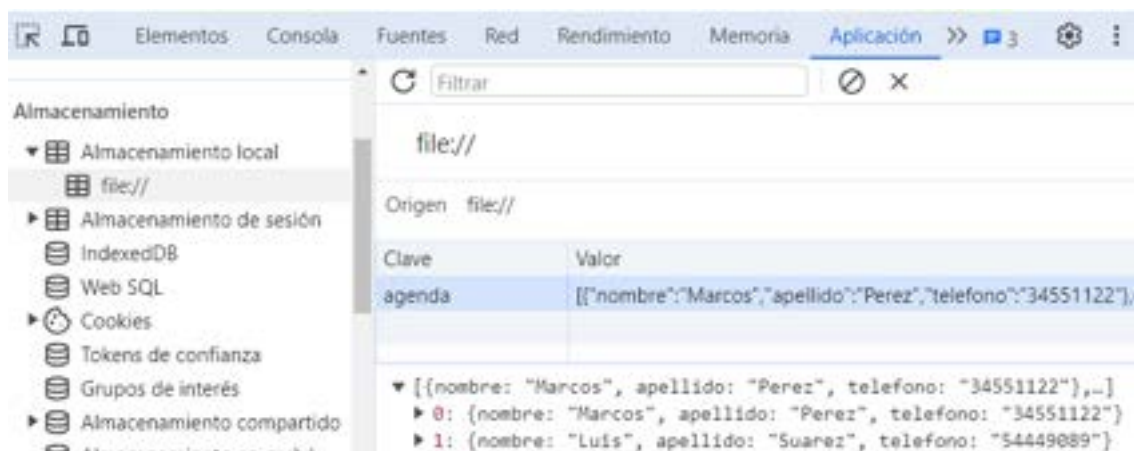
[Enviar](#)

Datos desde LocalStorage

Marcos Perez 34551122

Luis Suarez 54449089

En la consola vemos que los datos se ingresan correctamente, a medida que se agregan nuevos contactos al key **agenda**.



Una **estructura de datos bien definida** y una **lógica de almacenamiento adecuada** son fundamentales para gestionar de forma correcta **la información en localStorage**.

El uso de **JSON.stringify()** y **JSON.parse()** en conjunto con una estructura de datos coherente permite almacenar y recuperar objetos complejos de manera efectiva.

LocalStorage con Vue y componentes

En el desarrollo de aplicaciones web, el uso de localStorage para la persistencia de datos es una práctica común. En el ejemplo anterior en JavaScript, vimos cómo almacenar y recuperar datos utilizando JSON.stringify() y JSON.parse() en conjunto con localStorage.

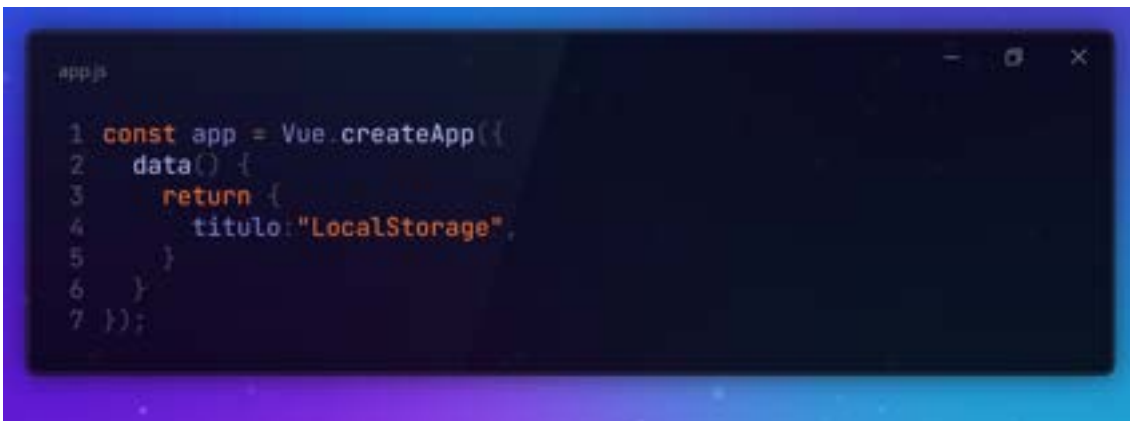
Ahora, trasladaremos esa lógica a Vue, aprovechando las capacidades que nos brinda este framework para la creación de componentes y la gestión de datos de manera más estructurada y eficiente.

En el código hemos creado dos componentes: **form-dato**, que actúa como el **componente padre** y **mostrar-dato**, que es el **componente hijo**.

El componente padre recibe los datos del formulario, los guarda en localStorage y los pasa al componente hijo a través de props para su visualización.

Vamos a analizar cómo esta lógica se implementa en Vue y cómo podemos aprovechar las ventajas de este framework para mejorar la gestión de datos en nuestra aplicación.

Recordemos que siempre tenemos la **instancia principal de Vue**, en este caso sólo, inicializa la propiedad **título**. Luego usaremos la **interpolación** para **mostrar su contenido en el index**.



```
1 const app = Vue.createApp({
2   data() {
3     return {
4       titulo: "LocalStorage",
5     }
6   }
7 });
```

```
APP.PX
1 //REGISTRO DEL COMPONENTE FORM-DATO -PADRE DEL COMPONENTE
  MOSTRAR-DATO Y PASA ARR PARA SU HIJO POR PROPS
2 app.component('form-dato', {
3   data:function(){
4     return {
5       dato:{
6         nombre:"",
7         apellido:"",
8         telefono:""
9       },
10      arr:[],
11    }
12  },
```

Analicemos el código :

- En principio **registramos** el **componente form-dato**.
- Dentro de la **función data** retornamos un **objeto** llamado **dato**, que posee las propiedades: **nombre**, **apellido** y **teléfono**. (necesario para cada contacto). Estos campos se inicializan vacíos al principio. Además, tenemos un **array** llamado **arr**, que servirá para contener a cada uno de los objetos que se van a ir agregando.


```
app.js
1 template:
2   `<div class="form-data">
3     <h2>Agenda Vue</h2>
4     <form @submit.prevent>
5       <div>
6         <label>Nombre</label>
7         <input type="text" v-model= "dato.nombre" />
8       </div>
9       <div>
10        <label>Apellido</label>
11        <input type="text" v-model= "dato.apellido" />
12      </div>
13      <div>
14        <label>Teléfono</label>
15        <input type="tel" v-model= "dato.telefono" />
16      </div>
17      <button @click="guardar(dato)">Guardar</button>
18    </form>
19    <mostrar-dato v-bind:arr ="this.arr"></mostrar-dato>
20  </div>`
```

- El **template** define el HTML que **representa el contenido del componente, el formulario para la carga de datos**. La etiqueta form @submit.prevent va a escuchar al evento submit y evitar gracias al modificador prevent la recarga de la página.

Dentro del formulario tendremos los **campos para nombre, apellido y teléfono** y estarán vinculados a las **propiedades dato.nombre, dato.apellido, dato.telefono** por medio de la **directiva v-model**.

- Además el formulario, posee un botón para guardar los datos, que llama al **método guardar(dato)** al hacer click.
- Por último, **<mostrar-dato v-bind:arr ="this.arr"></mostrar-dato>**: Llama al componente 'mostrar-dato' y le pasa la propiedad **arr** como un **prop** para que pueda **mostrar los datos ingresados**.

Y dentro de la propiedad methods desarrollamos la lógica necesaria para ejecutar la función guardar:


```
#PP28
1  methods:{
2    guardar(dato){
3      var localData = localStorage.getItem("local");
4      this.arr = localData ? JSON.parse(localData) : [];
5
6      this.arr.push({
7        nombre: dato.nombre,
8        apellido: dato.apellido,
9        telefono: dato.telefono
10     });
11
12     localStorage.setItem("local", JSON.stringify(this.arr));
13
14     this.dato = {nombre:"", apellido:"", telefono:""};
15   },
16 }
```

Repasemos el código y conceptos:

- **methods: { ... }:** Define los métodos disponibles en el componente. En este caso, solo hay un método llamado **guardar** que se ejecuta cuando **se hace click en el botón "Guardar" del formulario**.
- **guardar(dato) { ... }:** Recibe un objeto dato como argumento, que contiene los datos ingresados por el usuario (nombre, apellido y teléfono). **Obtiene los datos almacenados en localStorage bajo la clave "local" y los asigna a la variable localData.**
- **Verifica si localData tiene algún valor:** Si tiene valor, significa que ya hay datos guardados en localStorage. Entonces, utiliza **JSON.parse** para convertir esos datos de JSON a un array y lo asigna a **this.arr**. Si no tiene valor, **inicializa this.arr como un array vacío []**.
- **Agrega un nuevo objeto al array this.arr** con los datos del contacto ingresado por el usuario (nombre, apellido y teléfono). Convierte **this.arr de nuevo a formato JSON utilizando JSON.stringify** y lo **guarda en localStorage bajo la clave "local"**.
- **Reinicia los campos de entrada del formulario** (dato.nombre, dato.apellido y dato.telefono) a sus valores

iniciales (vacíos) para **permitir el ingreso de un nuevo contacto.**

Finalmente, **registramos el componente mostrar-dato, responsable de recibir los datos del formulario por props.**

```
app.js
1 //REGISTRO DEL COMPONENTE MOSTRAR-DATO- HIJO QUE RECIBE POR
  PROPS ARR
2 app.component("mostrar-dato", {
3   props: ["arr"],
4   template: `
5     <div class="ver" > <h1>Datos ingresados</h1>
6     <p v-for="x in arr">
7       {{x.nombre}}: {{x.apellido}}: {{x.telefono}}
8     </p>
9   </div>`
10 })
```

En la propiedad **template** tenemos el HTML que representa el contenido del componente **mostrar-dato**. Aplicamos la directiva **v-for** para iterar sobre cada elemento del arreglo **arr** que se pasa al componente como **prop**.

Dentro de cada párrafo, mostramos el nombre, apellido y teléfono del objeto utilizando las variables **x.nombre**, **x.apellido** y **x.telefono**, por medio de las **interpolaciones**.

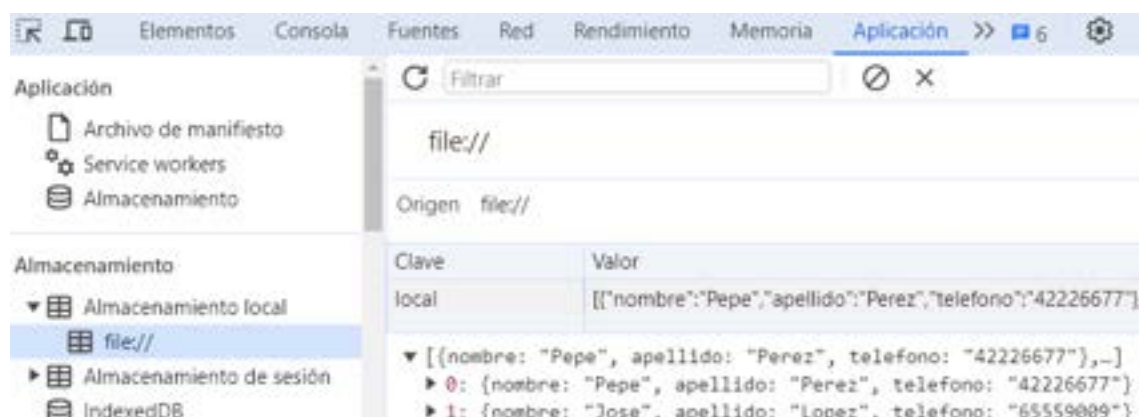
```
index.html
1 <div class="contenedor">
2   <h1>{{titulo}}</h1>
3   ← Componente padre →
4   <form-dato></form-dato>
5 </div>
```

En el **index** incorporamos el componente **form-dato** que contiene internamente todas las funcionalidades del **padre** y las funcionalidades del hijo **mostrar datos**.

Desde el navegador, si accedemos a la consola de **Vuejs devtools** podemos observar cómo se han estructurado los componentes.



Desde la consola en el menú **Aplicación** podemos verificar que localStorage puede almacenar los datos correctamente, manteniendo la persistencia de los datos a nivel local.



Como podemos observar, a lo largo del apunte **hemos trasladado la lógica de almacenamiento y recuperación de datos utilizando localStorage y JSON en JavaScript a un enfoque más estructurado y modular en Vue.js.**

Creamos componentes separados para la gestión del formulario de ingreso de datos y la visualización de los mismos, lo que nos permite **organizar nuestro código de manera más clara y reutilizable.**

Filtros en Vuejs

En Vue.js, los filtros son una característica que permite aplicar transformaciones simples a los datos en la capa de presentación antes de mostrarlos a la vista. Esto facilita la manipulación y formateo de datos **directamente en el template, mejorando la legibilidad y la claridad del código.**

En las **primeras versiones de Vue.js (1.x)**, los filtros eran usados para formatear datos de manera rápida y sencilla en los templates. Se podían aplicar directamente en la interpolación de datos dentro de las etiquetas HTML, y formaban parte del archivo core de Vue. Su sintaxis se representaba habitualmente dentro de una interpolación a una variable, luego se escribía el símbolo pipe (|) y luego se colocaba el nombre del filtro a aplicar sobre la variable.

A screenshot of a code editor window titled 'index.html'. It shows a single line of code: `<p>{{ message | uppercase }}</p>`. The code is highlighted with a light blue background.

En este caso **uppercase** era un **filtro definido en el componente Vue para** convertir el texto a mayúsculas.

En **Vue.js 2.x**, los filtros fueron revisados y sacados del archivo core de vue, para ser considerados como una herramienta externa que había que agregar si se quería usar esta funcionalidad por medio de otro archivo javascript. A nivel sintáctico los filtros tenían una **propiedad filter** para que pudiéramos desarrollar una función con nuestra propia lógica.

A screenshot of a code editor window titled 'app.js'. It shows a JavaScript object defining a filter:

```
1 filters: {
2   uppercase(value) {
3     return value.toUpperCase();
4   }
5 }
```

 The code is highlighted with a light blue background.

Más allá de esto, también teníamos la posibilidad de añadir filtros desarrollados por otros desarrolladores de forma sencilla, sin tener

que generar nosotros la lógica para los resultados visuales.

En **Vue 3**, los **filtros se han eliminado y ya no se pueden utilizar de la misma manera que en Vue 2**. Sin embargo, podemos **replicar la funcionalidad** de los filtros usando **métodos o propiedades computadas**. Más adelante profundizaremos sobre las propiedades computadas.

Por ejemplo, en nuestro caso inicializamos un **dato numérico** para que nuestro **método formatoUSD(valor)** le cambie su representación a una cadena de texto en formato de **moneda USD** (dólares estadounidenses). Usamos el **método toFixed(2)** para asegurarnos que el número tenga exactamente **dos decimales y luego agrega el símbolo "\$" al principio de la cadena**.

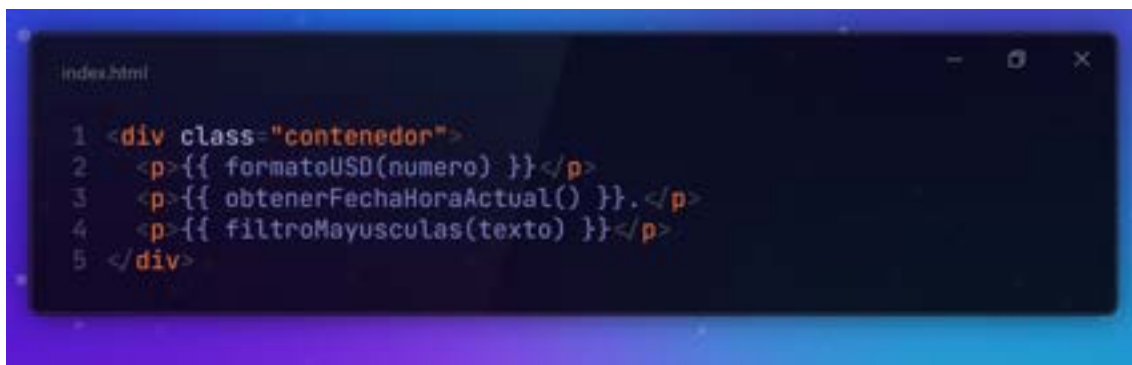


```
1 const app = Vue.createApp({
2   data() {
3     return {
4       texto: "este es un TEXTO de prueba para aplicar filtro",
5       numero: 1234,
6     }
7   },
8   methods: {
9     formatoUSD(valor) {
10       return '$' + valor.toFixed(2);
11     },
12     filtroMayusculas(valor) {
13       return valor.toUpperCase();
14     },
15
16     obtenerFechaHoraActual() {
17       const fecha = new Date();
18       const dias = ['domingo', 'lunes', 'martes', 'miércoles',
19 'jueves', 'viernes', 'sábado'];
20       const meses = ['enero', 'febrero', 'marzo', 'abril',
21 'mayo', 'junio', 'julio', 'agosto', 'septiembre', 'octubre',
22 'noviembre', 'diciembre'];
23
24       const diaSemana = dias[fecha.getDay()];
25       const día = fecha.getDate();
26       const mes = meses[fecha.getMonth()];
27       const año = fecha.getFullYear();
28
29       return `Hoy es ${diaSemana} ${día} de ${mes} del año
30 ${año}`;
31     }
32   }
33 })
34 app.mount('.contenedor');
```

El **segundo método se llama filtroMayusculas(valor)**, y toma una cadena de texto y la convierte completamente a **mayúsculas** usando el **método toUpperCase()**.

Nuestro **tercer método obtenerFechaHoraActual()**, se encarga de obtener la fecha y hora actuales del sistema y las formatea en una cadena de texto que indica el día de la semana, el día del mes, el mes y el año. Usa el objeto **Date** para **obtener la fecha actual, luego extrae los componentes necesarios (día de la semana, día, mes, año) y los combina en una cadena legible.**

En nuestro index tendríamos la siguiente sintaxis, usando los métodos definidos en Vue.js para formatear datos (números, fechas, texto) antes de mostrarlos en la página web, mejorando la presentación y la legibilidad de la información para el usuario.

A screenshot of a code editor window titled 'index.html'. The code is written in a dark theme with syntax highlighting. It shows a Vue.js template snippet within a <div class="contenedor"> block. The snippet contains three <p> elements, each using a different Vue.js method: 'formatoUSD(numero)', 'obtenerFechaHoraActual()', and 'filtroMayusculas(texto)'. The code is as follows:

```
1 <div class="contenedor">
2   <p>{{ formatoUSD(numero) }}</p>
3   <p>{{ obtenerFechaHoraActual() }}</p>
4   <p>{{ filtroMayusculas(texto) }}</p>
5 </div>
```

En el navegador veremos el resultado visual :

\$1234.00

Hoy es miércoles 14 de febrero del año 2024.

ESTE ES UN TEXTO DE PRUEBA PARA APLICAR FILTRO

Y si examinamos la consola desde **Vuejs devtools**, veremos que los datos sólo están cambiando su representación visual, el dato en sí, no cambia.



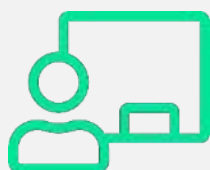
Es muy importante tener presente cuando trabajamos con frameworks, entender estos tipos de cambios para no cometer errores en la implementación. Cada versión puede mantener o eliminar comportamientos en sus distintas versiones, por tal motivo, siempre es recomendable leer la documentación oficial acorde a la versión que corresponde en el desarrollo.

¡Es hora de poner en práctica lo aprendido y crear componentes que aprovechen al máximo las capacidades de Vue.js!



Hemos llegado así al final de esta clase en la que vimos:

- Enfoque Offline First.
- LocalStorage en Javascript.
- Métodos y propiedad de localStorage.
- Serializar y Deserializar (JSON).
- Funciones de parseo.
- LocalStorage con Vue y componentes.
- Filtros en Vue.js.



Te esperamos en la **clase en vivo** de esta semana.
No olvides realizar el **desafío semanal**.

¡Hasta la próxima clase!

Bibliografía

Mozilla Developers Network. (s.f.). Documentación sobre Web Storage. En Mozilla Developers Network. Recuperado de https://developer.mozilla.org/es/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API/

Vue.js. (s.f.). Guía de Migración de Vue3: Documentación sobre filtros. En Vue.js. Recuperado de <https://v3-migration.vuejs.org/breaking-changes/filters.html>