UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

GENERALITAT VALENCIANA
Conselleria d'Educació, Investigació, Cultura i Esport

I.E.S. SERRA PERENXISA

DWEC 2º DAW

VUE. FUNDAMENTOS

CONTROL DE GASTOS

1. INTRODUCCIÓN RESUMEN

Vamos a crear una aplicación de control de gastos.
Definiremos un presupuesto (usaremos validación).
Podremos añadir gastos para llevar el control y nos irá diciendo el importe gastado y disponible. Además lo veremos en una barra de progreso.
Podremos filtrar los gastos según categoría.
Usaremos formularios para añadir los gastos (con validación).
Esta aplicación será persistente.

Estado inicial:

Pantalla cuando hay presupuesto válido:

2. DESARROLLO → CREACIÓN PROYECTO

Vamos a crear un proyecto con Vite siguiendo los pasos de la unidad.
El proyecto se llamará 'admin-gastos'

3. DESARROLLO → PREPARACIÓN index.html

Sustituimos las siguientes líneas por la de <title> en el fichero 'index.html'

- <title>Control de Gastos Vue + Vite</title>
- <link rel="stylesheet" ref="https://necolas.github.io/normalize.css/8.0.1/normalize.css">
- <link rel="preconnect" href="https://fonts.googleapis.com">
- <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
- <link ref="https://fonts.googleapis.com/css2?family=Lato:wght@400;700;900&display=swap" rel="stylesheet">

4. DESARROLLO → PREPARACIÓN PROYECTO

1. Preparamos el fichero App.vue para introducir nuestro código (borramos todo el contenido para dejar únicamente las tres secciones).
2. En la parte de la vista pondremos un <header> que incluirá un <h1> con el texto 'Planificador de Gastos'
3. Eliminamos el fichero del componente 'HelloWorld' y también el fichero 'style.css'.
4. Eliminamos del fichero 'main.js' el import del fichero style.css.

5. DESARROLLO → PRIMEROS ESTILOS

VER APARTADO 17 UNIDAD DIDÁCTICA: STYLE

Incluimos estilos ***globales*** en el componente principal ***(App.vue***).

```
:root{
    --azul: #3b82f6;
    --blanco: #FFF;
    --gris-claro: #F5F5F5;
    --gris: #94a3b8;
```

```
    --gris-oscuro: #64748b;
    --negro: #000;
  }
html{
    font-size: 62.5%;
    box-sizing: border-box;
  }
  *,
  *:before,
  *:after{
    box-sizing: inherit;
  }
body{
    font-size: 1.6rem;
    font-family: "Lato", sans-serif;
    background-color: var(--gris-claro);
  }
  h1{
    font-size: 4rem;
  }
  h2{
    font-size: 3rem;
  }
  header{
    background-color: var(--azul);
  }
  header h1{
    padding: 3rem 0;
    margin: 0;
    color: var(--blanco);
    text-align: center;
  }
```

6. DESARROLLO → COMPONENTE PRESUPUESTO.VUE

Creamos un nuevo componente que será el que tendrá el formulario inicial de la app.

Debemos:

1. Crear en el <template> un **formulario**. Tendrá la clase "presupuesto". Dentro tendrá:
   - Un con clase "campo" y dentro:
     - Un label con el atributo for="nuevo-presupuesto" y el texto de la etiqueta será 'Definir Presupuesto'
     - Un input con los siguientes atributos y valores:
       - id="nuevo-presupuesto"
       - class="nuevo-presupuesto"
       - placeholder="Indica tu presupuesto"
       - type="number"
   - Un input de tipo "submit" con el valor "Aceptar"

2. Creamos estilos con CSS, que serán ***propios*** del componente. Serán estos:

```css
.presupuesto{
    width: 100%;
}
.campo {
    display: grid;
    gap: 2rem;
}
.presupuesto label {
    font-size: 2.2rem;
    text-align: center;
    color: var(--azul);
}
.presupuesto input[type="number"] {
    background-color: var(--gris-claro);
    border-radius: 1rem;
    padding: 1rem;
    border: none;
    font-size: 2.2rem;
    text-align: center;
}
.presupuesto input[type="submit"] {
    background-color: var(--azul);
    border: none;
    padding: 1rem;
    font-size: 2rem;
    text-align: center;
```

```
        margin-top: 2rem;
        color: var(--blanco);
        font-weight: 900;
        width: 100%;
        transition: background-color 300ms ease;
    }
    .presupuesto input[type="submit"]:hover {
        background-color: #1048A4;
        cursor: pointer;
    }
```

3. ***Renderizamos*** el componente Presupuesto.vue en el componente principal (App.vue): Lo haremos **dentro de un *div*** (justo debajo del h1), que tendrá tres **clases**:
    - contenedor-header
    - contenedor
    - sombra

4. Los **estilos** css serán estos:

```
.contenedor{
      width: 90%;
      max-width: 80rem;
      margin: 0 auto;
    }
    .contenedor-header{
      margin-top: -5rem;
      transform: translateY(5rem);
      padding: 5rem;
    }
    .sombra{
      box-shadow: 0px 10px 15px -3px rgba(0,0,0,0.1);
      background-color: var(--blanco);
      border-radius: 1.2rem;
      padding: 5rem;
    }
```

Si ejecutamos, vemos esto:

7. DESARROLLO → COMPONENTE ALERTA.VUE

Creamos un nuevo componente llamado "Alerta.vue". Este componente nos servirá para definir los mensaje de alerta que vamos a mostrar al usuario. No va a tener parte de lógica. Los estilos serán de css.

Creamos en <template> un div con la clase "alerta"

Esto son estilos de la clase "alerta" (serán **propios** del componente):

```
.alerta{
    padding: 1rem;
    font-weight: 900;
    text-transform: uppercase;
    font-size: 1.6rem;
    text-align: center;
    background-color: var(--blanco);
    border-left: 0.5rem solid #B91C1C;
    color: #B91C1C;
    margin-bottom: 2rem;
}
```

8. DESARROLLO → FORMULARIO COMPONENTE PRESUPUESTO.VUE

En este apartado vamos a dinamizar el formulario que recoge el presupuesto inicial.

1. Creamos un ***State*** llamado **presupuestoIncial** y lo inicializamos a 0. El valor de este State será el presupuesto con el que contará el usuario. Este State servirá únicamente para hacer verificaciones. Después, se trabajará con otro State presupuesto en App.vue.
2. Creamos otro ***State*** llamado **error** y lo inicializamos a cadena vacía. Lo usaremos para mostrar mensajes.

VER APARTADO 18 UNIDAD DIDÁCTICA: FORMULARIOS

3. Añadimos la **directiva** correspondiente en el input para poder "recoger" el valor del ***State*** de presupuesto del formulario de forma dinámica, sin eventos (usamos ***v-model***). En este caso, NO necesitamos Emits ni Props porque los datos se gestionan desde el mismo componente.
4. Añadimos en la etiqueta <form> el evento correspondiente (Method Handler) para **validar** el formulario (a través del **evento submit**). Después de class="presupuesto". La función será 'validarPresupuesto'. La función la haremos en el mismo componente.
5. La **lógica** de la función es: Si la cantidad recogida en el input, es decir, el valor del ***State* presupuestoInicial** no es correcta, el ***State* error** cambia de valor (con el mensaje que consideremos). A los tres segundos, el ***State* presupuestoInicial**, volverá a ser 0.

VER APARTADO 19 UNIDAD DIDÁCTICA: SLOTS

6. Queremos mostrar el valor del ***State* error**, es decir, el mensaje de error en pantalla. El valor del State lo tenemos en ***Presupuesto.vue*** y queremos **inyectarlo** en el componente ***Alerta.vue***. Este, es un componente muy sencillo, por lo que NO usaremos Props, sino ***<slots>***

   En el componente Alerta.vue, incluimos el <slot></slot> debajo de , de esa forma nuestros mensajes tendrán los estilos incorporados al componente.

7. **Renderizamos** el componente ***Alerta.vue*** en ***Presupuesto.vue***, justo debajo de la etiqueta <form> → No lo queremos renderizar siempre, solo cuando el State error no es cadena vacía.

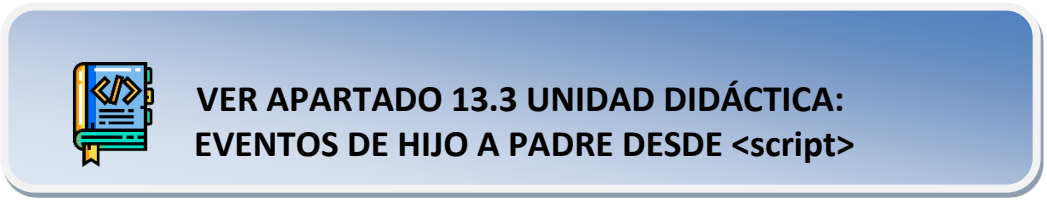
Ejemplo de partida y con el error:

9. DESARROLLO → STATE PRESUPUESTO

Cuando el usuario introduce un importe, lo recoge el componente Presupuesto.vue (es el presupuesto inicial) y cuando valida el formulario, verifica (en la función 'validarPresupuesto'), que lo introducido es correcto.

Si el usuario ha introducido una cantidad válida, ésta se convierte en su **presupuesto** y, tendrá todo ese dinero **disponible** directamente. Se muestra la imagen solo para poner en contexto al alumno:

1. Creamos en el **componente principal** un ***State*** llamado **presupuesto**. Lo inicializamos a 0.



2. Creamos en el **componente principal**, una **función** llamada 'definirPresupuesto' que recibe una cantidad y el State presupuesto tomará el valor de esa cantidad.

   A esta función se le llamará desde la propia función 'validarPresupuesto' del componente ***Presupuesto.vue***, por lo que añadiremos un ***Custom Event*** para que la llame. Debemos, en el componente Presupuesto.vue:

   - Crear una constante que igualaremos a la macro DefineEmits con el array de eventos.

```
const emit = defineEmits(['definir-presupuesto'])
```

   - Añadir el emit en la función con la llamada.

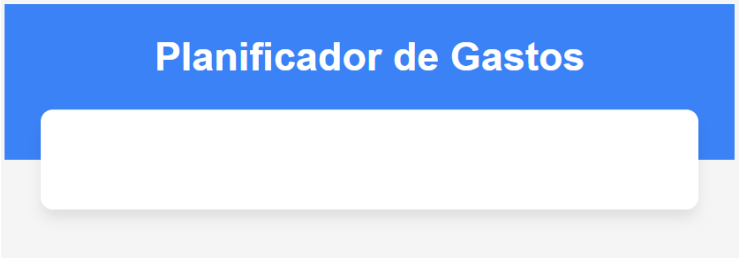
```
const validarPresupuesto = () => {
    if(presupuestoInicial.value <=0 || presupuestoInicial.value === ''){
        error.value = '¡PRESUPUESTO NO VALIDO!'

        setTimeout(() => {
            error.value ='';
        }, 3000);

        //para evitar seguir con el código y emitir el valor del presupuesto erróneo
        return
    }
    //emitimos desde aquí
    emit('definir-presupuesto', presupuestoInicial.value);
}
```

- En el componente principal, debemos incluir el evento.

10. DESARROLLO → MOSTRAR PRESUPUESTO

Una vez definimos el presupuesto no tiene sentido seguir mostrando el formulario inicial, ya que, a partir de ahí, debemos trabajar con otros componentes.

Solo mostraremos el componente Presupuesto.vue si el valor del State presupuesto es 0.



11. DESARROLLO → COMPONENTE CONTROLPRESUPUESTO.VUE

Creamos el componente con el que vamos a controlar el presupuesto. Siempre se estará mostrando, o el componente Presupuesto (inicial) o este componente. Se llamará ***ControlPresupuesto.vue***. Acciones:

1. Importamos la imagen (llamándola 'imagen') llamada 'grafico.jpg' de la carpeta assets.

```
import imagen from '../assets/img/grafico.jpg'
```

2. En el <template>, tendremos un div class="dos-columnas" y dentro de él:

   → div class="contenedor-grafico" que incluirá una etiqueta img y, como **atributo dinámico** (src) la imagen importada

   → div class="contenedor-presupuesto". Dentro de este div, tendremos:

   - Un botón class="reset-app" con el texto "Resetear app"
   - 3 párrafos (Presupuesto, disponible y gastado). El importe será sustituido más adelante por el real

```
<p>
    <span>Presupuesto:</span>
    0€
</p>
```

3. Los estilos propios de CSS serán:

```
.dos-columnas{
    display: flex;
    flex-direction: column;
}
```

```css
.dos-columnas > :first-child{
    margin-bottom: 3rem;
}

@media (min-width: 768px) {
    .dos-columnas {
        flex-direction: row;
        gap: 4rem;
        align-items: center;
    }
    .dos-columnas > :first-child{
    margin-bottom: 0;
    }
}

.reset-app {
    background-color: #DB2777;
    border: none;
    padding: 1rem;
    width: 100%;
    color: var(--blanco);
    font-weight: 900;
    text-transform: uppercase;
    border-radius: 1rem;
    transition-property: background-color;
    transition-duration: 300ms;
}
.reset-app:hover {
    cursor: pointer;
    background-color: #c11d67;
}
.contenedor-presupuesto {
    width: 100%;
}

.contenedor-presupuesto p {
    font-size: 2.4rem;
    text-align:center;
    color: var(--gris-oscuro);
}
@media (min-width: 768px){
    .contenedor-presupuesto p {
        text-align:left;
    }
}

.contenedor-presupuesto span {
    font-weight: 900;
    color: var(--azul);
}
```

4. **Renderizamos** este componente justo debajo del componente ***Presupuesto.vue*** en ***App.vue*** (recordemos que, si no están juntos, las directivas de selección anidadas NO funcionan). **Este componente se mostrará solo cuando no se muestre el de Presupuesto**. Recordatorio: Estos dos componentes estarán dentro de

```
<div class="contenedor-header contenedor sombra">
```

Veamos las dos opciones que, de momento podemos ver en pantalla:

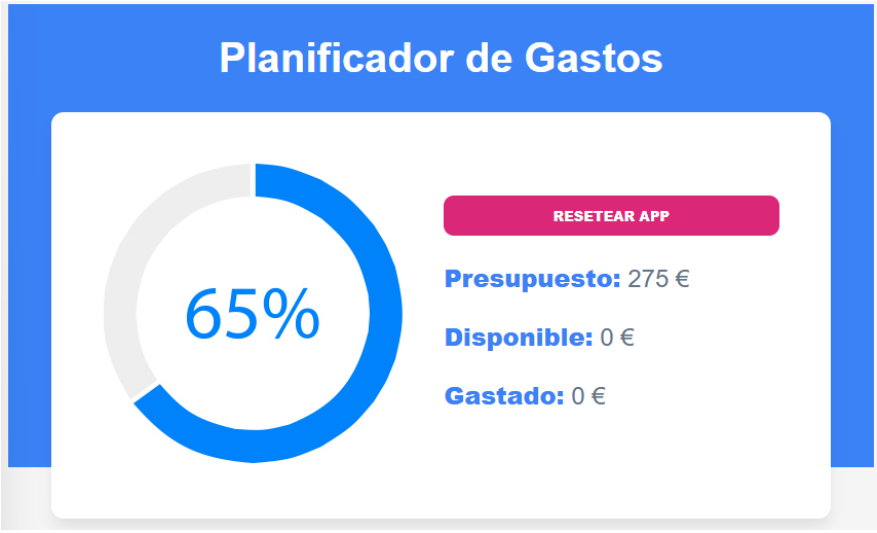
5. Queremos, desde el componente ***ControlPresupuesto.vue*** mostrar el valor del ***State* presupuesto** del usuario. Debemos:

   Desde el componente principal, pasamos con un **Prop** el valor del ***State presupuesto*** al componente ***ControlPresupuesto.vue*** (que es el que estamos renderizando)

   En el componente ***ControlPresupuesto.vue*** definimos los ***Props***

   Sustituimos el valor donde corresponda.

   Se vería así:



6. Queremos **mostrar el importe 'Disponible'**. Este importe irá cambiando según vayan añadiendo, editanto o borrando gastos:

   Creamos un ***State*** llamado **disponible** en el componente principal. Lo inicializamos a 0.

Hacemos los mismos pasos que con el State presupuesto para que se pueda mostrar.

Le damos valor en la función **'definirPresupuesto'**. Su valor debe ser el mismo que el de **presupuesto**.

7. **Formateamos las cantidades**. Acciones:

   Creamos una ***carpeta*** en **src** llamada ***helpers*** y dentro un ***fichero*** llamado ***index.js***. En este fichero crearemos distintas funciones generales que utilzaremos en el proyecto.

   Función para dar formato moneda → se usará el método ***toLocaleString***. Parámetros:

   - Código país →'es-ES'
   - Objeto opciones → style: 'currency y currency: 'EUR'

```
const number = 123456.789;

// request a currency format
console.log(
  number.toLocaleString("de-DE", { style: "currency", currency: "EUR" }),
);
// 123.456,79 €
```

   Importaremos y llamaremos a esta función desde el componente ***ControlPresupuesto.vue*** para ir mostrando desde el ***<template>*** los importes de ***presupuesto*** y ***disponible***.

   Ahora se vería así:

12. DESARROLLO → AÑADIR NUEVO GASTO

Vamos a añadir un nuevo gasto. Para ello definiremos un botón para añadir el gasto (será una imagen). Cuando se haga click sobre el botón, se abrirá una ventana emergente (será un nuevo componente) en la que el usuario podrá clasificar el gasto y asignarle una cantidad.

Acciones que realizar en el **componente principal**:

1. **Importamos** una imagen de la carpeta ***assets*** (nuevo-gasto.svg). La llamaremos ***'iconoNuevoGasto',***
2. En el ***<template>***, añadirmos después de </header>, un <main> con un . Dentro de ese div, tenemos:
   - ***Etiqueta imagen*** y hacemos dinámico el atributo ***src*** con la imagen que hemos importado.
   - ***Etiqueta alt***, cuyo valor será "icono nuevo gasto"

3. Añadimos **estilos** al componente (teniendo en cuenta la nueva clase)

```
.crear-gasto{
  position:fixed;
  bottom: 5rem;
  right: 5rem;
}
.crear-gasto img {
    width: 5rem;
    cursor:pointer;
}
```

4. En este momento, la imagen de añadir gasto siempre está visible y eso no es lógico. Solo se verá cuando estamos en condiciones de poder añadir un gasto (utilizaremos una directiva de selección).

5. Creamos un nuevo ***State*** llamado **modal**. En este caso, lo creamos llamando a reactive(). Recordamos que se debe importar de vue.

   Este State, tendrá dos propiedades inicializadas a false: 'mostrar' y 'animar' (la ventana tendrá una pequeña animación al crearse).

6. Cuando hagamos **click** sobre la imagen de añadir gasto, se deberá activar un **evento** que llamará a una **función** ('mostrarModal'), que lo que hará será modificar el valor de las propiedades del ***State* modal** a true.

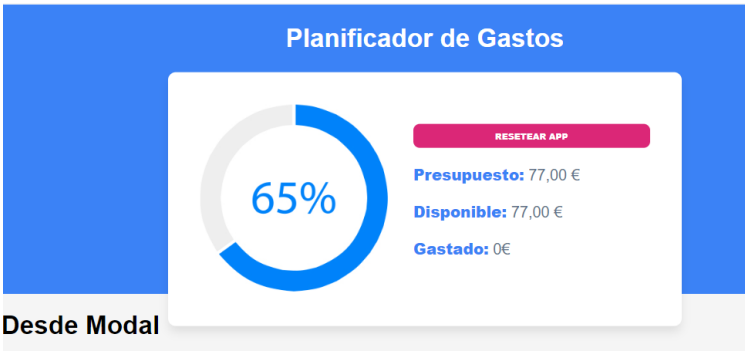
   Recordamos que no hay emits porque todo se hace desde el mismo componente.

13. DESARROLLO → COMPONENTE MODAL.VUE

1. Creamos un **nuevo componente Modal.vue**. Podemos hacer una prueba para ver que se muestra poniendo un <h1> en el <template> con un texto 'Desde Modal'.
2. En App.vue **renderizamos** nuestro nuevo componente en <main>, debajo del y, a través de una directiva de selección, mostraremos el componente si la propiedad del State modal, mostrar, es igual a true.

   Recordamos pasar con ***Props*** el **State modal** desde App.vue.

   Se vería así:



3. En la vista del componente Modal, ponemos un div class="modal"
4. Le damos **estilos propios** al componente para que parezca una ventana modal:

```
.modal {
        position:absolute;
        background-color: rgb(0 0 0 / 0.9);
        top:0;
        left: 0;
        right: 0;
        bottom: 0;
    }
```

Al ejecutar veremos la pantalla más oscura y arriba a la izquierda el texto de prueba.

5. Para cerrar la ventana modal, usaremos una imagen (la podemos llamar 'cerrarModal') con forma de aspa. La importamos ('cerrar.svg')

   En la vista, quitamos el <h1> de prueba y creamos la siguiente estructura:

    → este es el que hemos creado antes… Dentro de él:

   - Div con clase="cerrar-modal". Dentro:
     - Imagen con el atributo src dinámico con el valor de la imagen 'cerrarModal'>

6. Aplicamos estilos al "botón" de cerrar (la imagen del aspa):

```
.cerrar-modal{
    position:absolute;
    right: 3rem;
    top: 3rem;
}
.cerrar-modal img{
    width: 3rem;
    cursor: pointer;
}
```

7. Esta imagen tendrá asociado un Custom Event ('ocultar-modal') a través de la directiva de eventos.

   Llamará a la función 'ocultarModal' que la crearemos en App.vue (junto a 'mostrarModal').

   La función 'ocultarModal' pone las propiedades del State modal a false.

14. DESARROLLO → FORMULARIO EN MODAL.VUE

- Para poder añadir un gasto, vamos a crear un formulario. Tendrá varios campos: Nombre del gasto, cantidad y categoría.

  Lo haremos a la misma altura de

Tendrá la siguiente estructura:

```
<div class="contenedor">
    <form class="nuevo-gasto">
        <legend>Añadir Gasto</legend>

        <div class=campo>
            <label for="nombre">Nombre Gasto:</label>
            <input
                type="text"
                id ="nombre"
                placeholder="Añade el nombre del Gasto"
            />
        </div>
    </form>
</div>
```

- Añadimos otro con la misma estructura para la **cantidad**. En este caso, type="number"
- Añadimos un tercer que será un selector de **Categorías**:

```
<div class=campo>
    <label for="categoria">Categoría:</label>
    <select id="categoria">
        <option value="">-- Selecciona --</option>
        <option value="ahorro">Ahorro</option>
        <option value="comida">Cesta compra</option>
        <option value="casa">Casa</option>
        <option value="ocio">Ocio</option>
        <option value="salud">Salud</option>
        <option value="suscripciones">Suscripciones</option>
        <option value="gastos">Gastos Varios</option>
    </select>
</div>
```

- La última parte antes de cerrar el formulario será el input:

```
    <input
        type="submit"
        value="Añadir Gasto"
    >
</form>
```

- Aplicamos estilos al formulario:

```
.nuevo-gasto {
        margin: 10rem auto 0 auto;
        display: grid;
        gap: 2rem;
    }
    .nuevo-gasto legend {
        text-align: center;
        color: var(--blanco);
        font-size: 3rem;
        font-weight: 700;
```

```
}
.campo {
    display: grid;
    gap: 2rem;
}
.nuevo-gasto input,
.nuevo-gasto select {
    background-color: var(--gris-claro);
    border-radius: 1rem;
    padding: 1rem;
    border: none;
    font-size:2.2rem;
}
.nuevo-gasto label {
    color: var(--blanco);
    font-size: 3rem;
}
.nuevo-gasto input[type="submit"]{
    background-color: var(--azul);
    color: var(--blanco);
    font-weight: 700;
    cursor: pointer;
}
```

Se vería así:

15. DESARROLLO → ANIMACIÓN EN MODAL.VUE

Vamos a crear una **pequeña animación** a la pantalla modal. Se hará cuando hagamos click sobre la imagen de añadir gasto.

1. En Modal.vue, añadimos una clase al div:

```
<div class="contenedor contenedor-formulario">
```

2. Añadimos dos clases (animar y cerrar) en el div class="contenedor-formulario".

   No podemos añadir las dos a la vez, es decir, tendremos en cuenta el valor de la propiedad 'animar' del State modal:

   - Si es true → añadimos la clase 'animar'
   - Si es false → añadimos la clase 'cerrar'

   Para ello, utilizaremos la directiva de enlace para dinamizar el atributo 'class'

```
<div
    class="contenedor contenedor-formulario"
    :class="[condidion ? 'claseSiTrue' : 'claseSiFalse']"
>
```

3. Añadimos estilos:

```
.contenedor-formulario {
      transition-property: all;
      transition-duration: 300ms;
      transition-timing-function: ease-in;
      opacity: 0;
  }
  .contenedor-formulario.animar {
      opacity: 1;
  }
  .contenedor-formulario.cerrar {
      opacity: 0;
  }
```

4. Modicamos las funciones 'mostrarModal' y 'ocultarModal' de App.vue:
   - mostrarModal → El atributo 'animar' cambia a true pasado medio segundo
   - ocultarModal → El atributo'mostrar' cambia a false pasado medio segundo

16. DESARROLLO → STATE GASTO

Necesitamos una estructura que recoja los datos de cada gasto que el usuario vaya generando.

Puesto que vamos a agrupar datos, crearemos un nuevo ***State*** llamado **gasto** en App.vue con reactive().

Los atributos y sus valores iniciales serán:

- nombre: string vacío

- cantidad: string vacío
- categoría: string vacío
- id: null,
- fecha: la fecha actual

17. DESARROLLO → DATOS DEL FORMULARIO MODAL.VUE

Vamos a recoger los **datos del formulario** de forma directa, sin eventos, a través de la directiva **v-model**.

VER APARTADO 18.3 UNIDAD DIDÁCTICA: FORMULARIOS. Enviar datos v-model

Los recogemos en **App.vue** cuando renderizamos el componente **Modal**.

Únicamente recogemos, nombre, cantidad y categoría.

1. En App.vue, pasamos, a través de Props, con v-model los tres atributos del State gasto. Recordamos que los debemos diferenciar.

```
v-model:nombre="gasto.nombre"
```

2. En Modal.vue, los añadimos a la macro defineProps. Todos son String y required true a excepción de cantidad que, se inicializa a String pero después es Number.

   Lo lógico en este caso es que este Prop sea directamente Number pero aprovechamos para ver la sintaxis de posibles valores para un mismo Prop.

```
cantidad: {
    type: [String, Number],
    required: true,
},
```

3. En Modal.vue, debemos declarar los eventos de edición en cada campo del formulario. Recordamos que el atributo :value es OPCIONAL!!!!!

```
<input
    type="text"
    id ="nombre"
    placeholder="Añade el nombre del Gasto"
    :value="nombre"
    @input="$emit('update:nombre', $event.target.value)"
/>
```

   En el cado de cantidad, le añadimos el signo + delante de $event.target.value para convertir lo recogido en Number.

```
@input="$emit('update:cantidad', +$event.target.value)"
```

4. En Modal.vue, debemos declarar los eventos de edición en la macro ***defineEmits()***

```
defineEmits(['update:nombre','update:cantidad','update:categoria']);
```

18. DESARROLLO → VALIDACIÓN DEL FORMULARIO MODAL.VUE

A nuestro formulario le falta el evento submit

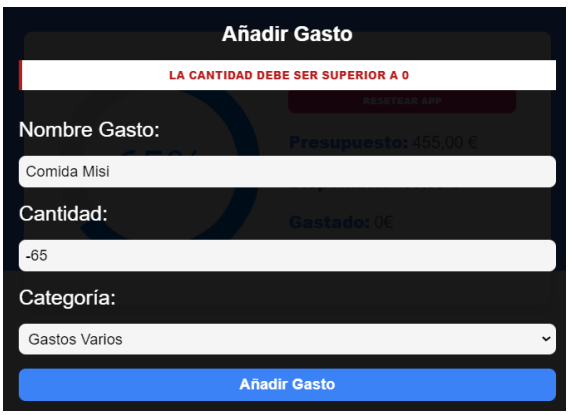
Cuando añadimos un gasto, necesitamos recoger: nombre, cantidad y categoría.

Tendremos que hacer validaciones y dar feedback al usuario si no ha introducido correctamente los datos antes de incluir el gasto en un array de gastos.

1. En ***Modal.vue***, incluimos un ***evento de tipo submit*** en la etiqueta del ***form***.

   Llamará a una función que podemos llamar 'agregarGasto'.

2. Creamos la **función 'agregarGasto' en el propio componente**. Lo que hará esta función es mostrar una alerta si algún campo está vacío o la cantidad no es correcta. Haremos distinción en el mensaje según sea el error. Para probar la lógica hacemos un console.log del mensaje.
3. Para mostrar las alertas haremos uso del componente Alerta.vue. Debemos pues importar el componente Alerta.vue y crear un State llamado error, que lo inicializamos a cadena vacía.
4. Volviendo a la función 'agregarGasto' → Si hay algún error, aparecerá una alerta durante un par de segundos. Modificaremos el State error de la siguiente manera:
   - Algún campo vacío → 'TODOS LOS CAMPOS SON OBLIGATORIOS'
   - Cantidad inferior a 1 →'LA CANTIDAD DEBE SER SUPERIOR A 0'
5. Renderizamos el componente Alerta.vue en Modal.vue tras el elemento <legend>Añadir Gasto</legend>



19. DESARROLLO → GUARDAR EL NUEVO GASTO

Una vez hemos validado el formulario, debemos guardar el nuevo gasto.

Lo que haremos será llamar desde la función 'validarGasto' a una función que vamos a crear en App.vue que se llamará 'guardarGasto'. Esta función lo que hará será almacenar el gasto en un array de gastos.

1. Creamos en ***App.vue*** una ***función*** llamada ***'guardarGasto'***. NO recibe argumentos. Para probar, hacemos un console.log del ***State gasto***.
2. Hacemos la **llamada** a esa función desde ***Modal.vue*** ('validarGasto'). Recordamos utilizar ***emit***.
3. Donde vamos a almacenar el gasto es en un array. Será un ***State*** llamado ***gastos*** (en App.vue), inicializado a array vacío.
4. La función 'guardarGasto' añadirá el gasto al State gastos. El id de cada gasto lo hemos inicializado a null por lo que ahora debemos darle valor. Para ello, en el fichero index.js de la carpeta helpers definimos la función generarId que debemos exportar. Como ejemplo para generar números aleatorios:

```
export const generarId = () => {
    const fecha = Date.now().toString(36);
    const random = Math.random().toString(36).substring(2);
    return fecha + random;
}
```

```
▾ props
  ▾ gasto: Reactive
      nombre: "Cena Navidad"
      cantidad: 45
      categoria: "ocio"
      id: "lrt4q8s3gdtx5p0lars"
      fecha: 1706180709146
```

VER APARTADO 20 UNIDAD DIDÁCTICA:
OBJETOS REACTIVOS DENTRO DE ARRAYS

5. Esta función la importamos y la usaremos en la función **'guardarGasto'**
6. En esta misma función, **cerraremos la ventana modal**. Además reiniciaremos los valores del **State gasto** para que cuando volvamos a introducir otro no aparezca en el formulario

20. DESARROLLO → LISTAR LOS GASTOS

Vamos a ir listando los gastos. Lo que se pretende es que, en la misma ventana de Control de Presupuesto, el usuario pueda ver la relación de sus gastos. Se muestra imagen para contextualización del alumnado.

1. En el <template> de App.vue, añadimos un div justo debajo de renderizar Modal.vue

```
<div class="listado-gastos contenedor">
    <h2>Gastos:</h2>
</div>
```

2. Añadimos estilos:

```
.listado-gastos{
        margin-top: 10rem;
    }
    .listado-gastos h2{
        font-weight: 900;
        color:var(--gris-oscuro);
    }
```

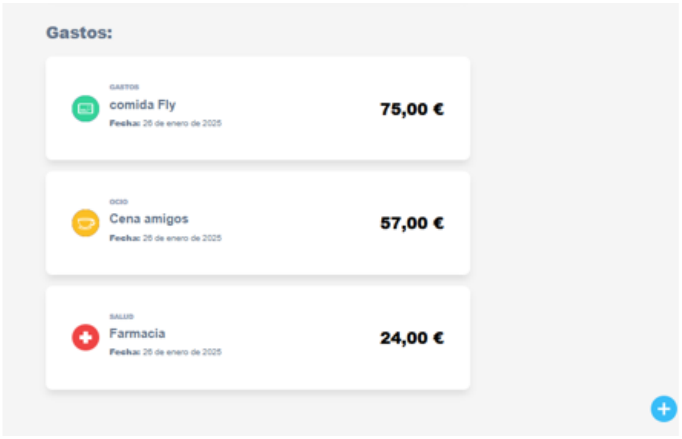
3. En el <h2> que hemos creado se mostrarán distintos mensajes, dependiendo de, si hay gastos o no hay.

   Lo haremos a través de un ternario, igual que cuando asignamos las clases 'animar' o 'cerrar'. Esta vez con la sintaxis de la doble llave porque vamos a poner código JS en un elemento del template (h2):

   {{ condición ? OpciónTrue : OpciónFalse }}

21. DESARROLLO → COMPONENTE GASTO.VUE

Este componente recogerá el formato del código de cada gasto que se va a listar en App.vue.



1. Creamos un nuevo componente llamado Gasto.vue. Podemos probar con un <h1> que ponga Gasto….</h1> para ver que se listan.
2. En ***App.vue*** lo renderizamos debajo del <h2>. Debemos:
    - Recorrer el array
    - Pasar el State gasto al componente Gasto.vue para que lo muestre
    - Pasaremos un identificador único de cada gasto, a través de key → `:key="gasto.id"`
3. En ***Gasto.vue*** definimos el ***Prop*** (gasto: tipo objeto, requerido)
4. En ***Gasto.vue*** creamos la siguiente estructura:
    - Div class="gasto sombra". Dentro:
        - Div class="contenido". Dentro haremos otro div="detalles". Este incluirá:
            - Párrafo class="categoría" → pondremos la categoría del gasto
            - Párrafo class="nombre" → pondremos el nombre del gasto
            - Párrafo class="fecha" → pondremos Fecha:

- Span → pondremos la fecha del gasto

  - Párrafo class="cantidad" → pondremos la cantidad del gasto

5. Formateamos la fecha → Para ello, haremos una función en index.js (formatearFecha) en la que podremos usar el método de fechas toLocaleDateString('es-ES',opciones). El parámetro opciones es un objeto que debemos crear con la siguiente información:

```
const opciones = {
    year: 'numeric',
    month:'long',
    day: '2-digit'
}
```

6. Para cada categoría de gasto, se mostrará un icono distinto. Los Importamos.

```
import IconoAhorro from '../assets/img/icono_ahorro.svg'
import IconoCasa from '../assets/img/icono_casa.svg'
import IconoComida from '../assets/img/icono_comida.svg'
import IconoGastos from '../assets/img/icono_gastos.svg'
import IconoOcio from '../assets/img/icono_ocio.svg'
import IconoSalud from '../assets/img/icono_salud.svg'
import IconoSuscripciones from '../assets/img/icono_suscripciones.svg'
```

7. Creamos un **objeto** como 'diccionarioIconos' para **relacionar las categorías** (las llamaremos como están en Modal.vue) **con los iconos**. La categoría será el atributo y el icono el valor.

   Esto lo usaremos para dinamizar la imagen del gasto.

8. Vamos a dinamizar la imagen del gasto. En el <template>, justo antes del (los dos elementos a la misma altura) añadimos una imagen con los siguientes atributos:

   - src dinámico → `"diccionarioIconos[gasto.categoria]"`
   - alt → texo 'Icono gasto'
   - class → "icono"

9. Aplicamos estilos propios a Gasto.vue

```
.gasto{
    display:flex;
    justify-content: space-between;
    align-items:center;
    margin-bottom:2rem;
}
.contenido{
    display:flex;
    align-items: center;
    gap:2rem;
}
```

```css
.icono{
    width:5rem;
}
.detalles p{
    margin: 0 0 1rem 0;
}
.categoria{
    color: var(--gris);
    font-size: 1.2rem;
    text-transform: uppercase;
    font-weight: 900;
}
.nombre{
    color: var(--gris-oscuro);
    font-size: 2.4rem;
    font-weight: 700;
    cursor: pointer;
}
.fecha{
    color: var(--gris-oscuro);
    font-size: 1.6rem;
    font-weight: 900;
}
.fecha span{
    font-weight:400;
}
.cantidad{
    font-size: 3rem;
    font-weight: 900;
    margin: 0;
}
```

Se vería así:

22. DESARROLLO → VENTANA MODAL CORTADA

Si tenemos muchos gastos y queremos añadir uno más, al hacer click sobre la imagen (+), la ventana modal aparece cortada. Tenemos que hacer scroll para verla completa en pantalla. Lo solucionaremos con estilos.

1. Añadimos la clase "fijar" al div principal de App.vue (el que está debajo de <template>), cuando el modal esté visible.

   Lo que estamos haciendo es poner al atributo class de forma dinámica. Podemos hacerlo con dos sintaxis distintas:

   ```
   :class="{fijar:modal.mostrar}"
   ```

   ```
   :class="[modal.mostrar ? 'fijar': '']"
   ```

2. Aplicamos estilos:

   ```
   .fijar{
     overflow: hidden;
     height: 100vh;
   }
   ```

23. DESARROLLO → CALCULAR TOTAL GASTADO Y DISPONIBLE

Vamos a mostrar en la pantalla de Control de Gastos, los importes 'Disponible' y 'Gastado'.

Según vayamos introduciendo gastos, editándolos o borrándolos, estos importes se irán modificando.

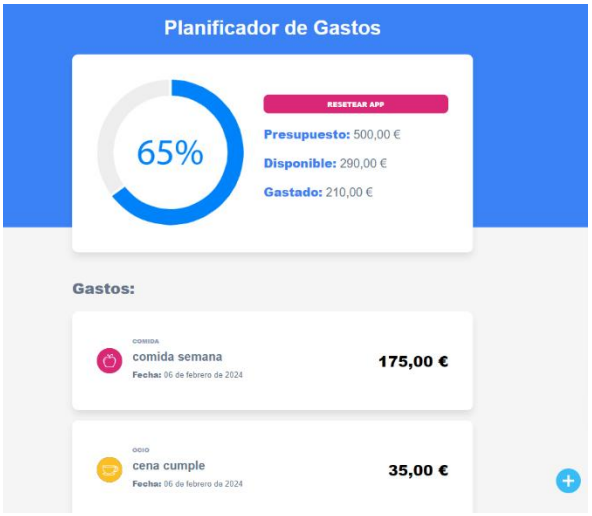
1. En App.vue creamos un ***State*** llamado 'gastado' y lo inicializamos a 0
2. Vamos a utilizar un ***watch()*** que estará pendiente de los cambios del State gastos (array que almacena los gastos). Recordamos que se debe importar y su sintaxis (no es obligatorio llamar a una función, podemos desarrollarla en el propio watch()) :

```
watch(carrito, ()=>{
    guardarLocalStorage()
    },{
    deep:true
    }
)
```

En nuestra función deberemos calcular el importe de los gastos totales y será el valor del State 'gastado'.

La diferencia entre el State 'presupuesto' y el State 'gastado' será el valor del State 'disponible'.

3. El nuevo State 'gastado' lo tendremos que pasar por ***Props*** al componente ***ControlPresupuesto.vue*** y en él, lo añadiremos a los Props y además lo mostraresmos formateado en el template.



24. DESARROLLO → NO GASTAMOS MÁS DEL PRESUPUESTO

Si tenemos una aplicación de control de gastos, lo esperable es que nos de un error si nos pasamos del presupuesto.

Esto lo avisaremos en la propia pantalla modal → en cada gasto nuevo, comprobaremos que el importe del gasto no supere el disponible.

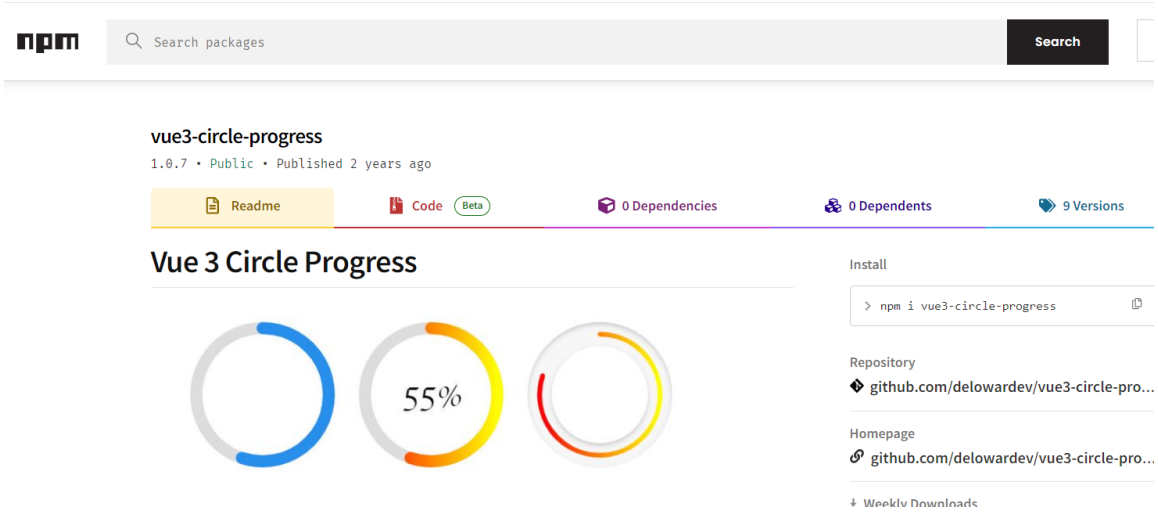
1. En App.vue, pasamos con un Prop el State 'disponible' a Modal.vue. En Modal.vue, lo incorporamos (type number, required)
2. En Modal.vue, en la función agregarGasto, haremos la validación necesaria y, si el gasto supera el disponible, aparecerá un mensaje de 'HAS EXCEDIDO EL PRESUPUESTO'. Se comportará como el resto de alertas.

25. DESARROLLO → GRÁFICO DE % DE GASTOS

Hasta ahora, estamos viendo los importes en la pantalla de control de presupuesto y siempre tenemos un 65% gastado. Es una imagen.

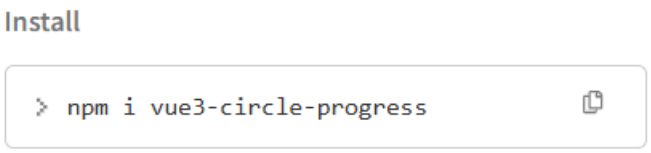
Podemos dar vida a este gráfico. Para ello, necesitamos una dependencia de Vue3.

1. En la página npmjs.com seleccionamos vue3 circle progress

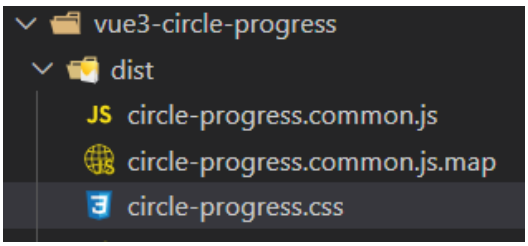


Podemos ver las opciones y formas de personalización y los Props que podremos pasar.

2. Copiamos el código de instalación.



Una vez instalado, veremos que se ha añadido una carpeta en node-modules:



3. En el componente donde mostramos la imagen del % (ControlPresupuesto.vue), eliminamos el import de la imagen estática y el atributo dinámico que hacía referencia a esa imagen.
4. Importamos el componente 'CircleProgress' y su fichero de estilos, tal y como indica su documentación.

```
import "vue3-circle-progress/dist/circle-progress.css";
import CircleProgress from "vue3-circle-progress";
```

5. Renderizamos el componente donde teníamos el atributo :src y pasamos un Prop de porcentaje, según el valor del Prop, se verá el componente de una forma u otra.

```
<div class="contenedor-grafico">
    <CircleProgress
        :percent='25'
    />
</div>
```

```
<div class="contenedor-grafico">
    <CircleProgress
        :percent='75'
    />
</div>
```

6. Le pasamos otros Props y nos fijamos en que, si el valor contienen "#", NO usaremos la directiva de enlace. Esto es porque Vue identifica el valor de los Props como elementos de JS. Si usamos la directiva de enlace, daría error.

   Cuando NO se usa la directiva de enlace, Vue lo interpreta como una cadena estática.

```
<CircleProgress
    :percent="50"
    :size="250"
    :border-width="30"
    :border-bg-width="30"
    fill-color="#3b82f6"
    empty-color="#e1e1e1"
/>
```

   Como ejemplo para usar la directiva de enlace en todos los Props:

```
<script setup>
const colorDinamico = "#00ff00";
</script>

<template>
  <Componente :color="colorDinamico" />
</template>
```

De esta forma, si cambiamos el color, se cambiaría en todas sus referencias.

7. Vamos a hacer real el valor de %. Utilizaremos una ***Computed Property***. Recordamos que se tiene que importar. Recordamos también su sintaxis:

```
const totalPagar = computed(() =>{
    return props.carrito.reduce((ac
})
```

OJO!! Sin decimales.

Sustituimos el valor fijo del ***Prop 'percent'*** por el ***Computed Property***

8. Para que el % se vea dentro del círculo, **añadimos un párrafo** justo antes del componente CircleProgress (a la misma altura, son hermanos). Este párrafo tendrá una clase 'porcentaje' y el valor será el del Computed Property.

9. Añadimos **estilos**.

```
.contenedor-grafico{
    position:relative
}
.porcentaje{
    position: absolute;
    margin: auto;
    top:calc(50% - 1.5rem);
    left: 0;
    right: 0;
    text-align:center;
    z-index: 100;
    font-size: 3rem;
    font-weight: 900;
    color: var(--gris-oscuro);
}
```

26. DESARROLLO → EDITAR UN GASTO I

El usuario se puede equivocar al introducir un gasto. No toda la información es errónea

pero sí alguno de los datos.

Le vamos a dar la oportunidad de editar el gasto.

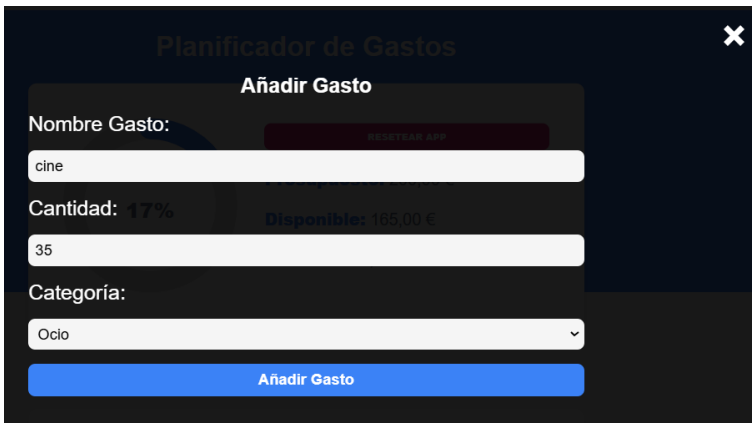
Cuando haga ***click sobre el nombre del gasto***, aparecerá de nuevo la pantalla modal.

1. En App.vue crearemos una ***función*** llamada seleccionarGasto que recibirá un id y nos devolverá el gasto concreto que ha seleccionado el usuario para editar.

   Esa función será llamada en el ***Custom Event*** del componente ***Gasto.vue***, en el nombre del gasto (para que sea más intuitivo, podemos poner entre paréntesis 'haz click para editar').

   En la misma función editamos el state gasto. Ahora sus valores corresponderán a los del gasto a editar.

   Después de editar el State, llamamos a la función que muestra la pantalla modal para editar el gasto. Esa pantalla se deberá mostrar con los datos del gasto. Recordamos que necesitaremos la directiva de enlace en los inputs del form (para hacer dinámicos los atributos del formulario) de forma que, una vez se haya refrescado la pantalla, los valores de los input permanezcan.



PROBLEMA!!!

Si cerramos la pantalla modal sin editar (nos hemos equivocado por ejemplo) y queremos añadir otro gasto, veremos que aparece la pantalla con la información rellenada del último gasto introducido.

Necesitamos que, cada vez que se cierre la pantalla modal, el State se reinicie.

Lo haremos a través de un watch() que estará pendiente del State modal. Si su atributo 'mostrar' es false, se reinicia el State gasto.

Podemos ver que, tanto aquí como al añadir un gasto, estamos reiniciando el mismo State, se recomienda, como buenas prácticas, hacer una función que lo reinicie y llamarla desde la función 'guardarGasto()' y 'watch()' de modal.

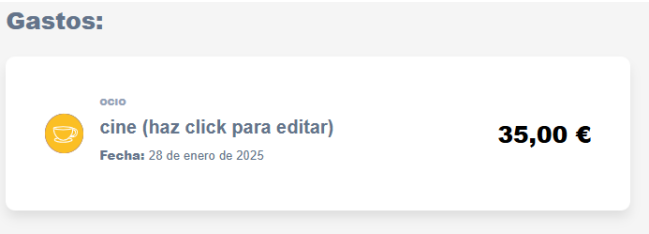
27. DESARROLLO → EDITAR UN GASTO II

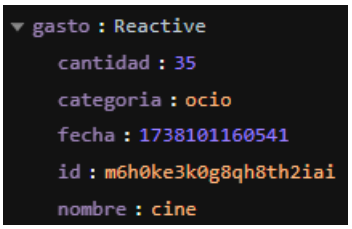
Vamos a utilizar para guardar los cambios del gasto editado la misma función que para añadir un nuevo gasto: 'guardarGasto'.

Debemos diferenciar cuándo estamos añadiendo y cuándo estamos editando.

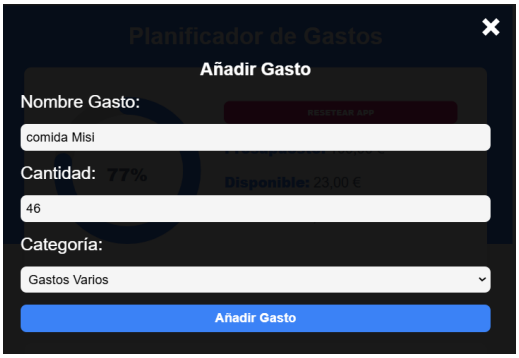
Si utilizamos la extensión de Vue Developer Tools podremos ver que hay una diferencia entre los dos gastos, el id. Esto es porque se genera cuando, desde la función 'agregarGasto' llamamos a la función 'guardarGasto' → La primera hace la validación de los datos. Si todo es correcto, llama a la segunda, que será la que asigne el id al gasto nuevo.

Gasto ya generado





Gasto en proceso de creación



1. En la función 'agregarGasto' verificamos si tiene id (estamos editando)o si no lo tiene, es un gasto nuevo.

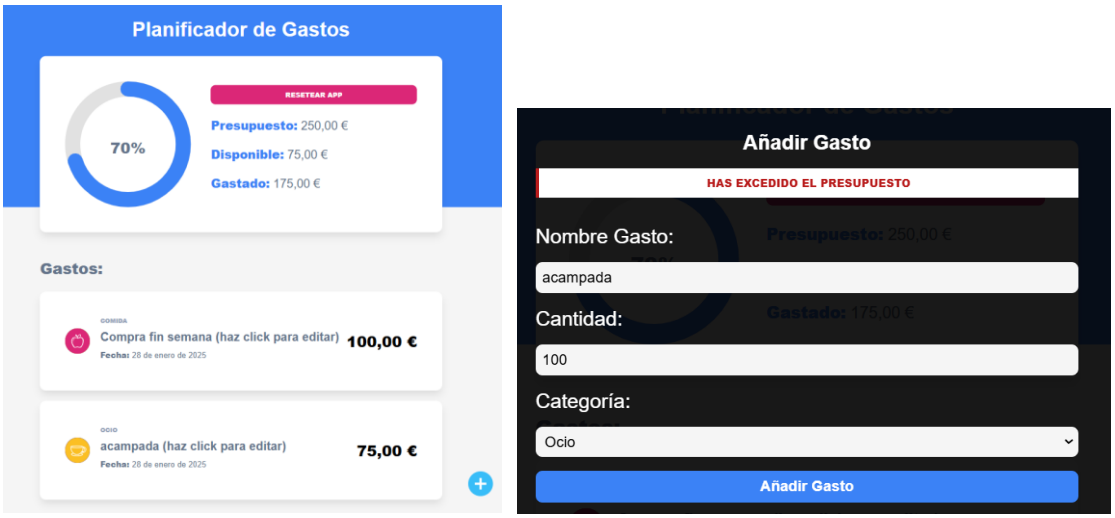
Arreglamos la función: En el caso en que estemos editando, debemos sobreescribir el elemento del State gastos concreto con los nuevos datos. En el caso en que estemos añadiendo, dejamos lo que teníamos antes en la función.

28. DESARROLLO → AJUSTAR IMPORTE DISPONIBLE EN EDICIÓN DE GASTOS

Supongamos esta situación: Tenemos un presupuesto de 250€, hemos gastado 175€, por lo que nos queda un disponible de 75€.

Si editamos el gasto 'acampada' y le pongo como cantidad 100€, me salta un error, porque solo tenía disponible 75€, pero eso NO es real → con los dos gastos registrados, aún tendría 50€ disponbles.

Debemos recalcular la cabecera (los States), cada vez que un gasto se edite.



1. La validación de si el usuario sobrepasa el presupuesto la hacemos en el componente Modal.vue, en la función 'agregarGasto'.

   Necesitamos el id del gasto para saber si estamos añadiendo o editando, así que, desde App.vue, lo pasaremos a Modal.vue a través de Props.

2. En el componente Modal.vue, lo registramos en la macro defineProps como id(tipo: [String, null] y required: true)

3. En la función, dejamos las validaciones como están hasta llegar a la parte del disponible: si tenemos id estamos editando, si el id es null, estamos añadiendo (lo dejaremos como estaba).

   En el caso de edición, nos debemos guardar el atributo cantidad del State gasto que vemos en la pantalla modal. Hasta que no escribamos sobre el input, ese valor de cantidad NO va a cambiar.

   Si la nueva cantidad es mayor que la cantidad antigua más el disponible, nos habremos excedido del presupuesto.

29. DESARROLLO → MODIFICAR TEXTO BOTÓN MODAL.VUE

En la pantalla modal, si estamos editando un gasto, el texto del botón deberá ser 'Guardar cambios'

Lo debemos cambiar en <legend> y en <input>. Se muestra un ejemplo. También se puede usar un Computed Property

```
<form class="nuevo-gasto"
      @submit.prevent="agregarGasto"
>
    <legend>{{id ? 'Guardar Cambios' : 'Añadir Gasto'}}</legend>
```

```
<input
    type="submit"
    :value="[id ? 'Guardar Cambios': 'Añadir Gasto']"
>
```

30. DESARROLLO → ELIMINAR UN GASTO

Vamos a hacer un botón en el componente Modal.vue que nos permitirá borrar el gasto.

1. Creamos el botón justo debajo del formulario

```
<button>
    type="button"
    class="btn-eliminar"
</button>
```

2. Le damos estilos:

```
.btn-eliminar{
        border-radius: 1rem;
        border:none;
        padding: 1rem;
        width: 100%;
        background-color: #ef4444;
        font-weight: 700;
```

```
        font-size: 2.2rem;
        color: var(--blanco);
        margin-top: 10rem;
        cursor: pointer;
    }
```

3. El botón solo aparecerá cuando estemos editando…

4. Creamos una función en App.vue y la comunicamos con Modal.vue para emitir el evento cuando hagamos click sobre el botón.

   Esta función editará el State gastos filtrando los elementos y quitando el que tenga el id seleccionado. Recordemos que cuando vayamos a realizar esta acción tenemos en pantalla un State gasto activo. Después, cerraremos la pantalla moda.

31. DESARROLLO → COMPONENTE FILTRO.VUE

Vamos a darle un valor añadido a nuestra página.

Vamos a crear un componente para que el usuario pueda filtrar por categoría de gastos.

1. Creamos el componente Filtro.vue
2. Esta será su estructura en el <template>

```
<template>
    <div class="filtros sombra contenedor">
        <form>
            <div class="campo">
                <label for="filtro">Filtrar Gastos</label>
                <select
```

```
                        id="filtro"
                        :value="filtro"
                        @input="$emit('update:filtro, $event.target.value)"
                    >
                        <option value="">-- Selecciona --</option>
                        <option value="ahorro">Ahorro</option>
                        <option value="comida">Cesta compra</option>
                        <option value="casa">Casa</option>
                        <option value="ocio">Ocio</option>
                        <option value="salud">Salud</option>
                        <option value="suscripciones">Suscripciones</option>
                        <option value="gastos">Gastos Varios</option>
                    </select>
            </div>
        </form>
    </div>
</template>
```

3. Lo renderizamos en App.vue justo debajo de la etiqueta <main>

4. Le damos estilos propios:

```
.filtros {
        margin-top: 10rem;
    }
    .filtros .campo {
        display:flex;
        align-items: center;
        gap: 2rem;
    }
    .filtros label {
        font-size: 3rem;
        font-weight: 900;
        color: var(--gris-oscuro);
    }
    .filtros select {
        flex:1;
        padding: 1rem;
        border: none;
        border-radius: 1rem;
        background-color: var(--gris-claro);
        text-align: center;
    }
```

5. Creamos en App.vue un nuevo State llamado filtro, lo inicializamos a cadena vacía
6. En la renderización del componente, le pasamos el State como Props a través de la directiva v-model

```
<Filtro
    v-model:filtro="filtro"
/>
```

7. En el componente Filtro.vue registramos el Prop (lo hacemos con una sintaxis abreviada. También definimos los Emits

```
defineProps(['filtro'])
defineEmits(['update:filtro'])
```

32. DESARROLLO → MOSTRAR GASTOS SEGÚN CATEGORÍA

Ya tenemos el componente, renderizado en App.vue, y están comunicados, a través de Props y evento.

Queremos que, cuando el usuario elija una categoría, o cambies se listen los gastos de esa categoría.

Lo vamos a hacer a través de un ***Computed Property***. Nos permitirá agrupar la lógica y nuestra vista únicamente usará su nombre manteniendo el código limpio.

1. Lo vamos a crear en App.vue. Recordamos que hay que importarlo.
2. Creamos el Computed Property que lo que hará será filtrar los gastos del State gastos por el valor del State filtro.

   Si el State tiene valor, devolverá el array nuevo, si no lo tiene, el array de State gastos.
3. Cuando renderizamos el componente Gasto.vue, iteramos sobre el State gastos, ahora debemos hacerlo sobre el Computed Property

33. DESARROLLO → LOCALSTORAGE

Vamos a almacenar el State presupuesto y gastos en localStorage.

Lo haremos con watch()

Presupuesto solo cambia una vez, inicia en 0 y el usuario lo modifica.

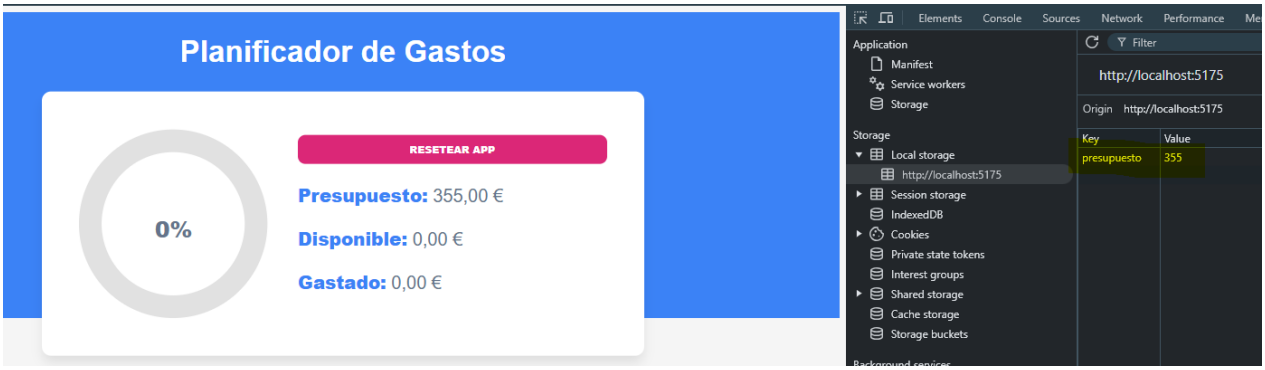
Si ponemos un valor en localStorage (con watch()) y reiniciamos la página, el State prespuesto se reinicia a 0 pero en localStorage sigue teniendo el valor que guardamos.

Lo que debemos hacer es verificar si ya hay un presupuesto cuando el componente se crea a través de una función de ciclo de vida de los componentes. De esa forma, si hay presupuesto, NO volverá a salir la pantalla inicial.

1. Vamos a utilizar onMounted(). Recordamos que se debe importar.

   En la función onMounted, lo que hacemos es verificar si hay presupuesto. Si lo hay, modificamos el State presupuesto con el valor de localStorage.

   Si recargamos, no nos muestra la pantalla de solicitar presupuesto porque ya tiene



2. Vemos que, al recargar, mantenemos nuestro presupuesto, pero perdemos el valor del State disponible. Lo que haremos es darle valor también en la misma función a ese State (el valor será el mismo que el State presupuesto).
3. Vamos a hacer lo mismo con los gastos. Ya tenemos un watch() para gastos. Cada vez que cambien los gastos, los guardaremos en localStorage.

   Para recuperarlos, utilizaremos la función onMounted().

34. DESARROLLO → RESETEAR APP

Vamos a darle vida al botón 'RESETEAR APP'.

Lo que haremos es reiniciar presupuesto y eliminar los gastos.

1. En App.vue creamos la función 'resetearApp'. Podemos poner un console.log de prueba con algún texto. En la renderización del componente ControlPresupuesto registramos el evento ('reset-app') y lo igualamos a la función que hemos hecho
2. En ControlPresupuesto.vue, utilizamos la macro defineEmits y, en el botón:
   - Añadimos type="button"
   - Registramos el evento 'click'

3. En la función 'resetearApp', lo que hacemos es, si el usuario confirma que quiere reiniciar presupuesto y gastos, lo hacemos → reiniciamos los States de presupuesto y gastos y, directamente se mostrará la pantalla inicial de presupuesto (no tenemos que hacer nada nosotros).