Taller de Svelte 5 y Visualización de Datos con D3

Guía de Referencia y Conceptos Clave

Juan Manuel Flores de la Cruz

29 de noviembre de 2025

Introducción

Este documento sirve como guía de apoyo para el desarrollo del dashboard. A continuación, se detallan los conceptos técnicos , organizados por componentes y funcionalidades, utilizando la sintaxis moderna de Svelte 5 (Runes).

Índice

1. Estilado y Layout (Componente `BaseCard`)

El componente `BaseCard` actúa como el contenedor visual de nuestra aplicación. Su objetivo es encapsular el estilo base (bordes, sombras, espaciado) para reutilizarlo en todo el dashboard.

Esta sección sirve como guía de referencia rápida para estilizar los componentes (`BaseCard`, botones, inputs) y organizar el layout del dashboard.

1.1. Posicionamiento y Espaciado (Mover cosas)

Propiedades para organizar elementos en la pantalla y darles aire.

- **Estructuras (Grid vs Flex):**
  - `display: grid;` → Úsalo para el **layout principal** (rejilla 2D). Permite definir columnas y áreas.
  - `display: flex;` → Úsalo para **componentes internos** (alinear iconos con texto, botones en fila).
- **Alineación (Dentro de Flex/Grid):**
  - `justify-content: center | space-between;` → Distribuye el espacio en el eje principal (horizontal por defecto).
  - `align-items: center;` → Alinea verticalmente al centro.
  - `gap: 1rem;` → Crea un hueco seguro entre elementos (mejor que usar márgenes manuales).
- **Espaciado (Modelo de Caja):**
  - `padding: 20px;` → Espacio **interno** (hacia adentro del borde). .Engorda"la caja.
  - `margin: 20px;` → Espacio **externo** (hacia afuera del borde). Separa la caja de las vecinas.

1.2. 2. Decoración de Contenedores (Cajas y Botones)

Propiedades para dar aspecto visual a `div`, `button` o `BaseCard`.

```
.contenedor-ejemplo {
    /* FONDO */
    background-color: white; /* O c digos hex: #f5f5f5 */

    /* BORDES */
    border: 1px solid #ccc; /* Grosor, estilo y color */

    /* REDONDEO (Esquinas) */
    border-radius: 8px; /* 0px es cuadrado, 50% es c rculo */

    /* SOMBRAS (Eje X, Eje Y, Difuminado, Color) */
    box-shadow: 0 4px 6px rgba(0,0,0,0.1);

    /* CURSOR (Para botones) */
    cursor: pointer; /* Cambia el rat n a "manita" */
}
```

1.3. 3. Formato de Texto (Tipografía)

Propiedades para jerarquizar la información (Títulos vs Párrafos).

- **Color:**
  - `color: #333;` → Gris oscuro (mejor que negro absoluto para lectura).
  - `color: rgb(255, 99, 132);` → Colores específicos para datos/alertas.
- **Peso y Tamaño:**
  - `font-size: 1.5rem;` → Tamaño de la letra (1rem ≈ 16px).
  - `font-weight: bold | 600;` → Grosor (negrita). Útil para destacar totales o títulos.
- **Alineación y Estilo:**
  - `text-align: center | right;` → Para centrar títulos o alinear números a la derecha.
  - `text-transform: uppercase;` → Pone todo en MAYÚSCULAS automáticamente.
  - `letter-spacing: 1px;` → Separa las letras (estético en títulos cortos).

1.4. A realizar

- Crear grid de 3 columnas
- Mostrar 4 BaseCard en la pantalla que se vean de la siguiente manera:

2. Interactividad y Estado (Componente `CardControl`)

En este componente haremos que la interfaz reaccione al usuario. Crearemos un formulario simple para añadir nuevos gastos a un array.

2.1. Teoría: Reactividad y Eventos

2.1.1. 1. Estado Local (`$state`)

En Svelte 5, si quieres que una variable actualice la pantalla cuando cambia su valor, no basta con usar `let`. Debes usar la runa `$state`.

```
let contador = $state(0); // Reactivo
let nombre = $state("Hola");
```

2.1.2. 2. Enlace Bidireccional (`bind:value`)

Para conectar una variable de JavaScript con un `<input>` de HTML de forma que si uno cambia, el otro también se actualice automáticamente, usamos la directiva `bind:value`.

```
<input type="text" bind:value={nombre} />
```

2.1.3. 3. Manejo de Eventos (`onclick`, `onkeydown`)

En Svelte 5 usamos los atributos estándar de HTML para escuchar eventos (clics, teclas, etc.).

- **onclick:** Para ejecutar una función al hacer clic en un botón.
- **onkeydown:** Para detectar qué tecla se ha pulsado (útil para detectar ."Enter").

2.1.4. 4. Modificar el Estado Global (Arrays)

Como nuestros datos globales (`datos`) están definidos con `$state` en un archivo externo, podemos modificarlos directamente como un array normal de JavaScript.

```
datos.push(nuevoValor); // Funciona y actualiza todos los componentes
```

2.2. A Realizar: Lógica del Formulario

Sigue estos pasos para completar el componente `CardControl.svelte`.

2.2.1. 1. Definición de Variables ($state)

Crea dos variables reactivas dentro del `<script>`:

- `nuevoGasto`: Inicialízala en `null`. Esta guardará el número que escriba el usuario.
- `mensajeError`: Inicialízala como string vacío .

2.2.2. 2. Implementar la función `guardar()`

Crea una función que realice lo siguiente:

1. **Validar:** Comprueba si `nuevoGasto` es `null`.
2. **Si es inválido:** Asigna un texto de advertencia a `mensajeError` y detén la función (`return`).
3. **Si es válido:**
   - Borra el mensaje de error (asigna).
   - Añade el valor al array global `datos` usando `.push()`.
   - Limpia el input reseteando `nuevoGasto` a `null`.

2.2.3. 3. Conectar el HTML

En el bloque `template`, busca los comentarios `TODO` y:

- Enlaza el input con tu variable usando `bind:value`.
- Añade el evento `onclick` al botón para llamar a tu función.
- (Opcional) Añade `onkeydown` al input para que funcione al pulsar ."Enter".
- Muestra la variable `mensajeError` dentro del `div` de error.

Para comprobar que se ha hecho de forma correcta, puedes hacer un log de los datos y ver como cambian al guardar datos nuevos.

3. Datos Globales y Derivados (Componente `CardBalance`)

Este componente será el cerebro visual. No tiene datos propios; su trabajo es escuchar al almacén global (`datos`), sumar todas las cantidades y mostrar el total.

3.1. Teoría: Reactividad Calculada

3.1.1. 1. Importación de Stores (La Fuente de la Verdad)

En el componente anterior (`CardControl`) escribíamos datos. Aquí los leemos. Al importar `datos` desde `stores.svelte.js`, ambos componentes comparten **exactamente el mismo array en memoria**.

3.1.2. 2. Estado Derivado (`$derived`)

¿Qué pasa si queremos mostrar la suma total? Podríamos sumarlo manualmente cada vez que cambia el array, pero Svelte 5 nos da una herramienta mejor: **las Runas Derivadas**.

```
// total es una formula, no un valor fijo.
// Si precio o cantidad cambian, total se recalcula solo.
let total = $derived(precio * cantidad);
```

Es igual que una celda con fórmula en Excel. Tú no dices actualiza la celda C1, tú defines la fórmula y Excel se encarga del resto.

3.1.3. 3. Sumar Arrays (`.reduce`)

Para sumar todos los números de un array en una sola línea, usamos el método `.reduce()` de JavaScript.

```
// (acumulador, valorActual) => nuevaSuma, valorInicial
datos.reduce((acc, val) => acc + val, 0);
```

Si tenemos `[10, 20]`, hace: $0 + 10 = 10$, luego $10 + 20 = 30$.

3.2. A Realizar: Cálculo del Balance

Sigue estos pasos en `CardBalance.svelte`:

3.2.1. 1. Importar los Datos

Importa la variable `datos` desde tu archivo de stores.

3.2.2. 2. Crear el Derivado

Declara una variable `total` usando la runa `$derived(...)`. Dentro del paréntesis, ejecuta la lógica para sumar el array `datos` (usa `.reduce` como vimos en la teoría).

3.2.3. 3. Visualización

En el HTML, sustituye el texto estático por tu variable `{total}`.

4. Datos Globales y Derivados (Componente `CardBalance`)

El componente Balance no tiene estado propio, sino que escucha los datos globales y calcula un total.

4.1. Importación de Stores

En el componente anterior hemos declarado la variable datos que este componente también necesita, para compartir esta variable creamos un archivo stores.svelte.js e importamos datos en ambos archivos desde ahí.

```
import { datos } from '../lib/stores.svelte.js';
```

4.2. Estado Derivado: `$derived`

Si necesitamos calcular un valor basado en otro estado (por ejemplo, sumar todos los gastos), usamos la runa **`$derived`**. Svelte recalculará este valor automáticamente cuando cambien los datos originales.

```
<script>
    // Calcula la suma total autom ticamente
    // .reduce es un m todo de Array para acumular valores
    let total = $derived( suma de los datos );
</script>
```

5. Visualización de Datos con D3 (Componente `CardChart`)

En esta sección construiremos el gráfico. A diferencia de los componentes anteriores, aquí no solo maquetamos, sino que calculamos geometría.

5.1. Teoría: La Caja de Herramientas D3

Para dibujar un gráfico, necesitamos traducir números (datos) a píxeles (pantalla). D3 nos ofrece funciones específicas para ello.

5.1.1. 1. El Sistema de Coordenadas SVG

Es vital recordar que en el SVG el origen **(0,0)** está en la **esquina superior izquierda**.

- **Eje X:** Aumenta hacia la derecha $\rightarrow$.
- **Eje Y:** Aumenta hacia **abajo** $\downarrow$.

Importante: Si quieres dibujar algo .ªrriba"(valor alto), necesitas una coordenada Y pequeña (cerca del 0).

5.1.2. 2. Escalas (`scaleLinear`)

Son funciones que transforman un valor del *mundo de los datos* al *mundo de los píxeles.*

- **domain([min, max]):** El rango de tus datos de entrada (ej: de 0€ a 100€).
- **range([minPx, maxPx]):** El espacio disponible en pantalla (ej: de 0px a 300px).

```
// Sintaxis Gen rica
const miEscala = scaleLinear()
    .domain([valorMinimoDato, valorMaximoDato])
    .range([pixelInicio, pixelFin]);

// Uso: miEscala(50) -> devuelve los p xeles correspondientes
```

5.1.3. 3. Análisis de Arrays (`min, max`)

Para no escribir los límites del dominio a mano, D3 ofrece funciones para analizar arrays.

- `min(array)`: Devuelve el valor más bajo.
- `max(array)`: Devuelve el valor más alto.

5.1.4. 4. Generador de Formas (`line`)

El elemento SVG `<path>` necesita un atributo `d` (un string con coordenadas). Calcularlo a mano es imposible. La función `line()` lo hace por nosotros. Se configura definiendo dos "accessors"(accesores):

- **.x((d, i) =>...)**: Cómo calcular la posición X para un dato.
- **.y((d) =>...)**: Cómo calcular la posición Y para un dato.
- **.curve(...)**: (Opcional) Suaviza las líneas rectas.

```
// Sintaxis Gen rica
const generador = line()
    .x((dato, indice) => escalaX(indice))
    .y((dato) => escalaY(dato));

const pathString = generador(miArrayDeDatos);
```

5.1.5. 5. Ejes Automáticos (`.ticks()`)

Las escalas creadas en el punto 2 tienen un método llamado `.ticks(n)`. Este método devuelve un array de n números distribuidos uniformemente, ideales para dibujar las etiquetas del eje.

5.2. A Realizar: Construcción del Gráfico

Utilizando los conceptos anteriores, implementa la lógica en `<script>` y el dibujo en el `template`.

5.2.1. 1. Preparación de Constantes

Define dentro del script las dimensiones del gráfico: una variable reactiva para el ancho (`containerWidth`), una constante para el alto (`height`) y un objeto para los márgenes (`margin`) (top, right, bottom, left).

5.2.2. 2. Creación de Escalas Reactivas ($derived)

Define dos variables derivadas, `xScale` y `yScale`:

- **xScale:** Debe usar `scaleLinear`.
  - *Dominio:* Desde 0 hasta el último índice del array de datos (`datos.length - 1`).
  - *Rango:* Desde el margen izquierdo hasta el ancho total menos el margen derecho.
- **yScale:** Debe usar `scaleLinear`.
  - *Dominio:* Desde el `min` de los datos hasta el `max`. (Tip: Añade valores por defecto `||` `0` por si el array está vacío).
  - *Rango:* **¡Cuidado!** Recuerda que la Y crece hacia abajo. El rango debe ir desde `height - margin.bottom` (abajo) hasta `margin.top` (arriba).

5.2.3. 3. Generación del Path ($derived)

Crea una variable derivada `pathD` utilizando el generador `line()`. Configura sus propiedades `.x()` e `.y()` utilizando las escalas que acabas de crear. Pásale tus datos para obtener el string resultante.

5.2.4. 4. Renderizado en el HTML (Capas del SVG)

Ahora vamos a dibujar dentro del bloque `<svg>`. Es importante entender tres etiquetas nuevas de SVG que no existen en HTML normal:

- `<g>` **(Group):** Es como un `<div>` invisible. Sirve para agrupar varios elementos.
- `transform="translate(x, y)"`**:** Mueve el sistema de coordenadas. Si aplicas esto a un grupo, todo lo que dibujes dentro empezará desde esa nueva posición (x, y) en lugar de desde (0,0). Esto simplifica mucho las matemáticas.
- `<circle>`**:** Dibuja un punto. Requiere tres atributos: `cx` (centro X), `cy` (centro Y) y `r` (radio).

Instrucciones paso a paso:

1. **Capa 1: Eje Y y Rejilla (Grid Lines)**
   Queremos líneas horizontales de referencia.
   - Crea un bloque `#each` iterando sobre `yScale.ticks(5)`.
   - Por cada valor (*tick*), crea un grupo `<g>` y muévelo verticalmente usando `transform="translate(0, {yScale(tick)})"`.
   - **Dentro del grupo:**
     - Dibuja una `<line>` que vaya desde el margen izquierdo (`x1`) hasta el margen derecho (`x2`). *Estilo:* Usa un color gris muy claro (`#eee`) y hazla discontinua con `stroke-dasharray="4"`.
     - Dibuja un `<text>` con el valor del *tick*. *Estilo:* Alinealo al final (`text-anchor=.endd"`), usa un gris medio (`#999`) y ajústalo verticalmente con `dy="0.32em"` para que quede centrado respecto a la línea.
2. **Capa 2: Eje X (Etiquetas)**
   Queremos poner los días debajo de cada punto de datos.
   - Crea un bloque `#each` iterando sobre tus `datos` (usando `d, i`).
   - Crea un grupo `<g>` y muévelo horizontalmente a la posición del dato actual: `transform="translate({ {height - margin.bottom})"`.
   - **Dentro del grupo:**
     - Dibuja una pequeña línea vertical (`<line y2="6»`) como "marca"del eje.
     - Dibuja el `<text>` debajo. Para elegir el texto correcto, accede a tu array de etiquetas usando el módulo: `etiquetas[i % etiquetas.length]`. *Estilo:* Usa `text-anchor="middle"` para que el texto se centre justo debajo de la marca.
3. **Capa 3: La Línea de Datos (Path)**
   El trazo principal del gráfico.
   - Añade un elemento `<path>` al final del SVG (para que quede encima de la rejilla).
   - Asigna tu variable derivada `d={pathD}`.

- **Estilo obligatorio:** `fill="none"` (para que no se rellene de negro), `stroke=çolor"` (elige un color destacado) y `stroke-width="3"` (grosor). Usa `stroke-linecap=` `ound"` para bordes suaves.

4. **Capa 4: Puntos Interactivos**
   Círculos para resaltar cada dato.

   - Itera de nuevo sobre los `datos`.
   - Dibuja un `<circle>` para cada uno.
   - **Posición:** `cx={xScale(i)}` y `cy={yScale(d)}`.
   - **Estilo:** Radio `r="4"`. Rellénalos de blanco (`fill="white"`) y ponles el mismo borde que la línea (`stroke`) para crear un efecto de .ªnillo".