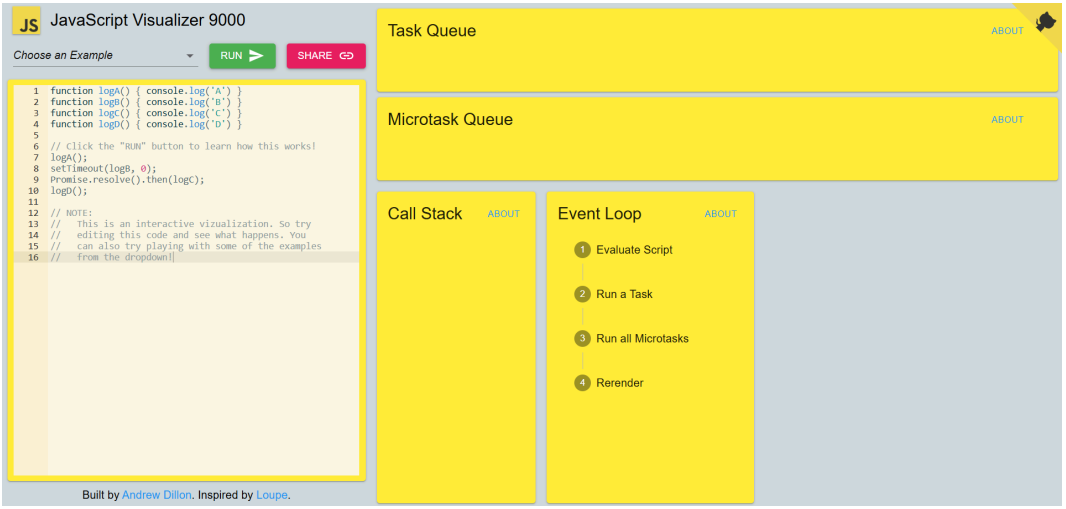
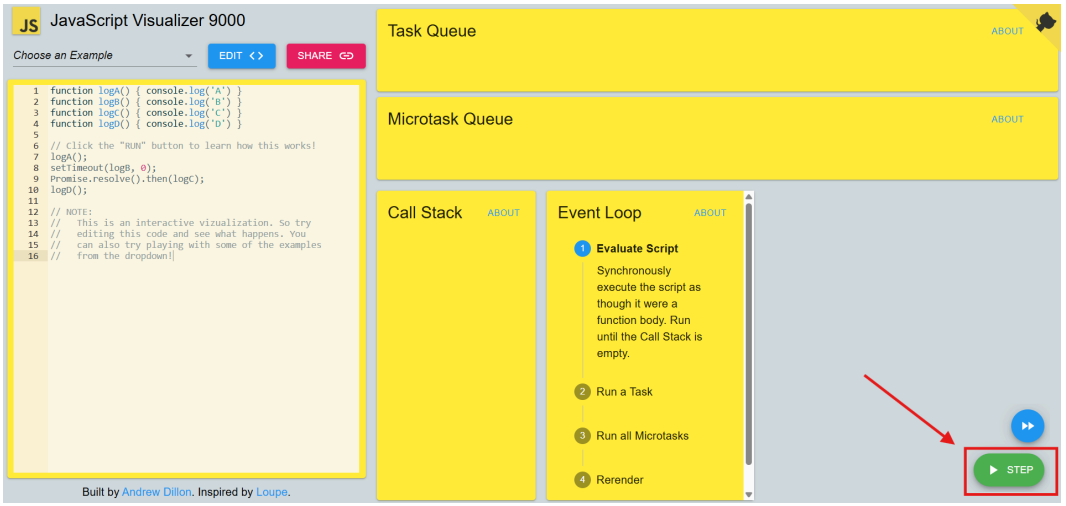
Retos de fin de semana con JavaScript Visualizer 9000 (JSV9000)

Objetivo general: Comprender a fondo cómo funciona el event loop en JavaScript mediante visualización dinámica.



Nota: Cuando des click en el botón "RUN" ve paso a paso (step) para entender la ejecución 😀



Reto 1 – Reconocer la anatomía del Event Loop

Objetivo: Identificar visualmente los componentes clave del event loop: call stack, Web APIs, task queue (macro-tareas), micro-tasks.

Paso a paso:

1. Entra a https://www.jsv9000.app.

2. Pega este código en el recuadro lateral izquierdo (justo debajo de los botones “Run” y “Share”) 😁

```
console.log('Start');

setTimeout(() => {
  console.log('Timeout');
}, 1000);

Promise.resolve().then(() => {
  console.log('Promise');
});

console.log('End');
```

5. Reemplaza el tiempo de `setTimeout` por:

    - **`0`**
    - **`100`**
    - **`2000`**

6. Observa en cada caso:

    - ¿Qué entra primero a la cola de tareas (**Task Queue**)? 👀
    - ¿Qué se ejecuta antes: el `setTimeout` o el `Promise`? 🤔

7. Agrega otra línea con:

```
Promise.resolve().then(() => console.log('Another promise'));
```

8. Observa si ambas promesas se ejecutan juntas o en orden.

Reto 2 – Contrastar tareas macro vs micro

Objetivo: Verificar visualmente cómo se priorizan las tareas en JavaScript (tareas asincrónicas macro vs micro).

Paso a paso:

1. Borra el contenido del editor en **JSV9000**.

2. Pega el siguiente snippet:

```
console.log('A');
setTimeout(() => console.log('B'), 0);
Promise.resolve().then(() => console.log('C'));
console.log('D');
```

3. Antes de correrlo:

    - Anota en una hoja o archivo tu predicción: ¿en qué orden aparecerán las letras? 👀

4. Ejecuta el código.

5. Observa el flujo: ¿en qué orden aparecen en el call stack y colas (**Task Queue y Microtask Queue**)?.

Realiza la siguiente tabla comparativa como apoyo para sacar tus conclusiones 😁

Evento	Predicción	Resultado
1	A	A
2	B	D
3	C	C
4	D	B

Reto 3 – Visualizar asincronía (versión Promises)

Objetivo: Entender cómo se ejecutan las promesas de forma asíncrona y cómo afectan el orden de ejecución.

Paso a paso:

1. Abre JSV9000 y limpia el editor.

2. Pega este código:

```
function foo() {
  console.log('1');
  Promise.resolve().then(() => {
    console.log('2');
  });
}

foo();
console.log('3');
```

3. Antes de correrlo:

    - Escribe tu predicción del orden en que se imprimirán los números.

4. Ejecuta, observa y compara contra tus predicciones.

Reto 4 – Buenas prácticas de código asíncrono con Promesas encadenadas

Objetivo: Reescribir código anidado de callbacks usando **promesas encadenadas** para mejorar claridad, seguimiento del flujo y mantenimiento.

Paso a paso:

1. Simula estas funciones en JSV9000 (puedes copiar y pegar):

```
function getUser(id) {
  console.log('Fetching user');
  return Promise.resolve({ id: id, name: 'Ana' });
}

function getOrders(userId) {
  console.log('Fetching orders for', userId);
  return Promise.resolve(['order1', 'order2']);
}

function getTotal(orders) {
  console.log('Calculating total for', orders.length, 'orders');
  return Promise.resolve(orders.length * 100);
}
```

2. Crea y prueba dos versiones diferentes (añade después del código anterior):

Versión con callback hell (simulada):

```
getUser(1).then(user => {
  getOrders(user.id).then(orders => {
    getTotal(orders).then(total => {
      console.log('Total:', total);
    });
  });
});
```

Versión con promesas encadenadas (buena práctica):

```
getUser(1)
  .then(user => getOrders(user.id))
  .then(orders => getTotal(orders))
  .then(total => console.log('Total:', total));
```

3. Ejecuta ambos en JSV9000 y observa cómo se visualizan los `.then()` en la cola de microtareas.

Aspecto	Callback Hell	Promesas Encadenadas
Legibilidad	Baja	Alta
Flujo claro	No	Sí
Escalabilidad	Complicada	Sencilla

¿Por qué las promesas encadenadas mejoran el código asincrónico? 🤔

Checa esto:
https://www.tiktok.com/@adriansieber/video/7062820046506052869?q=callback%20hell&t=1746153537707

Reto 5 – Medir impacto de operaciones CPU-intensivas

Objetivo: Visualizar cómo operaciones que bloquean el CPU (como bucles pesados) pueden **obstruir el event loop**, retrasando incluso tareas programadas.

Paso a paso:

1. Abre JSV9000 y pega este código:

```
console.log('Inicio');

setTimeout(() => {
  console.log('Antes del bucle');

  const start = performance.now();
  for (let i = 0; i < 1e8; i++) {} // Simulación de tarea pesada
  const end = performance.now();

  console.log('Fin del bucle, tomó', end - start, 'ms');
}, 0);

console.log('Fin');
```

2. Observa en JSV9000:
    - ¿Cuándo entra el `setTimeout`?
    - ¿Qué pasa mientras se ejecuta el `for`?
    - ¿Se bloquea el flujo de eventos?
3. Calcula cuánto tiempo real toma la operación con `performance.now()`.

Nota 2: Anota todas las dudas para resolverlas y envíalas al canal de discord para discutirlas con las compañeras y compañeros. 😁

Nota 3: Explora todos los ejemplos de la plataforma, además de buscar los conceptos que no entiendas 👀