

Implementación: Consumo Automático de Totems

Fecha de Implementación

7 de Noviembre, 2025

Descripción

Se ha implementado la funcionalidad de consumo automático de totems cuando el temporizador de cuenta llega a cero.

Cambios Realizados

1. Modificación de `hooks/useSimulation.ts`

Función modificada: `useTotem()`

Cambios:

- Ahora decrementa correctamente el contador de totems cuando se usa uno
- Actualiza el localStorage con el nuevo valor de totems
- Crea una notificación más específica indicando que el totem fue consumido automáticamente
- Retorna `true` si se pudo consumir un totem, `false` si no hay totems disponibles

Código anterior:

```
const useTotem = useCallback(() => {
  if (simulationState.totemCount > 0) {
    const updatedState = {
      ...simulationState,
      hasTimeExtension: true,
    };
    setSimulationState(updatedState);
    // No decrementaba el contador de totems
    return true;
  }
  return false;
}, [simulationState]);
```

Código nuevo:

```

const useTotem = useCallback(() => {
  if (simulationState.totemCount > 0) {
    // MODIFICADO: Decrementar el contador de totems
    const newTotemCount = simulationState.totemCount - 1;

    const newNotification: SimulationNotification = {
      id: Date.now(),
      type: 'totem',
      title: 'Tótem Consumido Automáticamente',
      message: 'Se ha consumido 1 tótem para reactivar tu cuenta automáticamente y
reiniciar el temporizador',
      timestamp: new Date().toISOString(),
      read: false,
      icon: 'ri-shield-line',
      color: 'purple',
    };

    const updatedNotifications = [newNotification, ...simulationState.notifications];

    const updatedState = {
      ...simulationState,
      totemCount: newTotemCount,
      notifications: updatedNotifications,
      unreadCount: simulationState.unreadCount + 1,
      hasTimeExtension: true,
    };

    setSimulationState(updatedState);

    // MODIFICADO: Actualizar localStorage con el nuevo contador de totems
    const userData = JSON.parse(localStorage.getItem('user_simulation_data') || '{}');
    userData.totems = newTotemCount;
    userData.hasTimeExtension = true;
    localStorage.setItem('user_simulation_data', JSON.stringify(userData));
    localStorage.setItem('simulation_notifications', JSON.stringi-
fy(updatedNotifications));

    return true;
  }
  return false;
}, [simulationState]);

```

2. Modificación de hooks/useTimer.ts

Cambios en la interface:

```

export interface UseTimerOptions {
  onTimerExpired?: () => void | boolean | Promise<void> | Promise<boolean>;
}

```

- Se agregó la posibilidad de que el callback devuelva un valor booleano
- `false` indica que se consumió un totem y no se debe bloquear la página
- `true` o `void` indica que se debe bloquear la página normalmente

Cambios en la lógica del temporizador:

```
// Bloquear página cuando llegue a 0
if (newTimer === 0) {
    // Call onTimerExpired callback if provided
    // Si el callback devuelve false, significa que se consumió un totem
    // y no debemos bloquear la página, sino reiniciar el temporizador
    let shouldBlock = true;

    if (options?.onTimerExpired) {
        const result = options.onTimerExpired();
        // Si el callback devuelve false, se consumió un totem
        if (result === false) {
            shouldBlock = false;
            // Reiniciar el temporizador automáticamente
            setTimer(initialTimer);
            setShowFloatingTimer(false);
            // No retornar el newTimer de 0, sino reiniciar
            return initialTimer;
        }
    }

    if (shouldBlock) {
        setIsPageBlocked(true);
        setShowFloatingTimer(false);
    }
}
```

Funcionalidad:

- Cuando el timer llega a 0, llama al callback `onTimerExpired`
- Si el callback devuelve `false`, significa que se consumió un totem:
 - * No bloquea la página
 - * Reinicia automáticamente el temporizador al valor inicial
 - * Oculta el temporizador flotante
- Si el callback devuelve `true` o no hay callback, ejecuta el comportamiento normal de bloqueo

3. Modificación de `app/page.tsx`

Nuevo callback agregado:

```
// NUEVO: Callback para manejar la expiración del temporizador
const handleTimerExpired = () => {
    // Verificar si el usuario tiene al menos 1 totem disponible
    if (simulationState.totemCount > 0) {
        // Consumir el totem
        const totemConsumed = useTotem();

        if (totemConsumed) {
            // Retornar false para indicar que NO se debe bloquear la página
            // El timer se reiniciará automáticamente desde useTimer
            return false;
        }
    }

    // Si no hay totems o no se pudo consumir, permitir el bloqueo normal
    // Retornar true para bloquear la página
    return true;
};

const timerState = useTimer(timerDuration, { onTimerExpired: handleTimerExpired });
```

Funcionalidad:

- Verifica si hay totems disponibles cuando el temporizador llega a 0
- Si hay al menos 1 totem:
 - * Lo consume automáticamente
 - * Retorna `false` para evitar el bloqueo de la página
 - * El temporizador se reinicia automáticamente
- Si no hay totems:
 - * Retorna `true`
 - * La página se bloquea normalmente
 - * Se muestra el modal de cuenta suspendida

Flujo de Funcionamiento

1. Usuario con totems disponibles:

- Timer llega a 0
- `handleTimerExpired()` verifica que hay totems
- `useTotem()` decrementa el contador de totems
- Se crea una notificación informando del consumo
- Se retorna `false` al hook `useTimer`
- El timer se reinicia automáticamente al valor inicial
- La página NO se bloquea
- El usuario puede seguir usando la plataforma

2. Usuario sin totems disponibles:

- Timer llega a 0
- `handleTimerExpired()` verifica que NO hay totems
- Se retorna `true` al hook `useTimer`
- La página se bloquea (`isPageBlocked = true`)
- Se muestra el modal de cuenta suspendida
- Comportamiento normal del sistema

Consumo Múltiple de Totems

- Si el usuario tiene múltiples totems (2, 3, 4, etc.), el sistema los consumirá uno por uno
- Cada vez que el temporizador llegue a 0 y haya totems disponibles, se consumirá 1 totem
- El usuario recibirá una notificación cada vez que se consuma un totem
- El contador de totems visible en la UI se actualizará automáticamente

Beneficios de la Implementación

1. **Automatización completa:** No requiere intervención del usuario
2. **Protección contra suspensión:** Los usuarios con totems están protegidos automáticamente
3. **Transparencia:** El usuario recibe notificaciones de cada consumo
4. **Robustez:** Mantiene toda la funcionalidad existente del sistema
5. **Escalabilidad:** Funciona con cualquier cantidad de totems

Compatibilidad

- Mantiene la funcionalidad existente del sistema de temporizadores

- Mantiene la funcionalidad del modal de suspensión de cuenta
- Mantiene el sistema de notificaciones
- Compatible con el sistema de rangos y beneficios
- Compatible con el sistema de puntos

Testing Recomendado

1. Caso 1: Usuario con 1 totém

- Esperar a que el timer llegue a 0
- Verificar que el totém se consume
- Verificar que el timer se reinicia
- Verificar que aparece la notificación
- Verificar que NO se muestra el modal de suspensión

2. Caso 2: Usuario con múltiples totéms

- Esperar a que el timer llegue a 0 múltiples veces
- Verificar que cada vez se consume 1 totém
- Verificar que el contador de totéms disminuye correctamente

3. Caso 3: Usuario sin totéms

- Esperar a que el timer llegue a 0
- Verificar que se muestra el modal de suspensión
- Verificar el comportamiento normal del sistema

4. Caso 4: Verificación de localStorage

- Verificar que el contador de totéms en localStorage se actualiza correctamente
- Verificar que las notificaciones se guardan correctamente

Notas Técnicas

- El sistema utiliza el callback `onTimerExpired` del hook `useTimer`
- El retorno booleano del callback determina si se debe bloquear o no la página
- El reinicio del temporizador se maneja automáticamente dentro del hook
- Las notificaciones se agregan al estado global y se persisten en localStorage
- El contador de totéms se sincroniza entre el estado de React y localStorage