Guía de Implementación - Google Calendar + Meet

Proyecto: SpeaklyPlan - Prácticas 1 a 1
Objetivo: Agendamiento automático con Google Calendar y Meet
Costo: $0 (100% gratuito)

Índice

Configuración Inicial

Paso 1: Crear Proyecto en Google Cloud Console

1. Ir a Google Cloud Console
2. Crear nuevo proyecto: "SpeaklyPlan Calendar"
3. Habilitar APIs:
   - Google Calendar API
   - Google Meet API (opcional, viene con Calendar)

Paso 2: Crear Credenciales OAuth 2.0

1. En Google Cloud Console → "Credentials"
2. "Create Credentials" → "OAuth 2.0 Client ID"
3. Application type: "Web application"
4. Nombre: "SpeaklyPlan Calendar Integration"
5. Authorized redirect URIs:

   ```
   https://speaklyplan.abacusai.app/api/auth/google-calendar/callback
   http://localhost:3000/api/auth/google-calendar/callback
   ```

6. Copiar:
   - Client ID
   - Client Secret

Paso 3: Configurar Variables de Entorno

```
# .env
GOOGLE_CLIENT_ID=123456789-abcdefg.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=GOCSPX-AbCdEfGhIjKlMnOpQrStUvWxYz
NEXTAUTH_URL=https://speaklyplan.abacusai.app
```

Paso 4: Instalar Dependencias

```
cd nextjs_space
yarn add googleapis date-fns date-fns-tz
```

Código Backend

1. Servicio de Google Calendar

```typescript
// lib/services/google-calendar-service.ts

import { google, calendar_v3 } from 'googleapis'
import { OAuth2Client } from 'google-auth-library'
import prisma from '@/lib/db'
import { addMinutes, format } from 'date-fns'
import { utcToZonedTime, zonedTimeToUtc } from 'date-fns-tz'

// Cliente OAuth2
function createOAuth2Client() {
  return new google.auth.OAuth2(
    process.env.GOOGLE_CLIENT_ID,
    process.env.GOOGLE_CLIENT_SECRET,
    `${process.env.NEXTAUTH_URL}/api/auth/google-calendar/callback`
  )
}

// Obtener cliente autenticado para un usuario
async function getAuthenticatedClient(userId: string): Promise<OAuth2Client> {
  // Buscar tokens del usuario en la DB
  const integration = await prisma.calendarIntegration.findUnique({
    where: { userId }
  })

  if (!integration) {
    throw new Error('Usuario no ha conectado Google Calendar')
  }

  const oauth2Client = createOAuth2Client()

  // Setear tokens
  oauth2Client.setCredentials({
    access_token: integration.accessToken,
    refresh_token: integration.refreshToken,
    expiry_date: integration.expiresAt.getTime()
  })

  // Si el token está expirado, refrescarlo
  if (integration.expiresAt < new Date()) {
    const { credentials } = await oauth2Client.refreshAccessToken()

    // Actualizar en DB
    await prisma.calendarIntegration.update({
      where: { userId },
      data: {
        accessToken: credentials.access_token!,
        expiresAt: new Date(credentials.expiry_date!)
      }
    })
```

```
    oauth2Client.setCredentials(credentials)
  }

  return oauth2Client
}

// CREAR EVENTO DE PRÁCTICA
export async function createPracticeEvent({
  userId,
  partnerId,
  scheduledFor,
  topic,
  durationMinutes = 30
}: {
  userId: string
  partnerId: string
  scheduledFor: Date
  topic: string
  durationMinutes?: number
}) {
  try {
    // 1. Obtener cliente autenticado
    const auth = await getAuthenticatedClient(userId)
    const calendar = google.calendar({ version: 'v3', auth })

    // 2. Obtener información de ambos usuarios
    const [user, partner] = await Promise.all([
      prisma.user.findUnique({ where: { id: userId } }),
      prisma.user.findUnique({ where: { id: partnerId } })
    ])

    if (!user || !partner) {
      throw new Error('Usuario no encontrado')
    }

    // 3. Calcular tiempos
    const startTime = scheduledFor
    const endTime = addMinutes(startTime, durationMinutes)

    // 4. Crear evento
    const event: calendar_v3.Schema$Event = {
      summary: ` Práctica de Inglés: ${topic}`,
      description: `
Sesión 1 a 1 con ${partner.name || partner.email} en SpeaklyPlan

 Tema: ${topic}
 Duración: ${durationMinutes} minutos

 Tips:
• Llega 2 minutos antes para verificar audio/video
• Ten listo el tema de conversación
• Practica con confianza, ¡los errores son parte del aprendizaje!
```

```
Enlace a la plataforma: ${process.env.NEXTAUTH_URL}/practice
    `.trim(),
    start: {
      dateTime: startTime.toISOString(),
      timeZone: 'America/Bogota'
    },
    end: {
      dateTime: endTime.toISOString(),
      timeZone: 'America/Bogota'
    },
    attendees: [
      {
        email: user.email!,
        displayName: user.name || undefined,
        responseStatus: 'accepted' // Organizador auto-acepta
      },
      {
        email: partner.email!,
        displayName: partner.name || undefined
      }
    ],
    conferenceData: {
      createRequest: {
        requestId: `practice-${userId}-${partnerId}-${Date.now()}`,
        conferenceSolutionKey: { type: 'hangoutsMeet' }
      }
    },
    reminders: {
      useDefault: false,
      overrides: [
        { method: 'email', minutes: 24 * 60 },  // 24 horas antes
        { method: 'popup', minutes: 60 },       // 1 hora antes
        { method: 'popup', minutes: 10 }        // 10 minutos antes
      ]
    },
    guestsCanModify: true,
    guestsCanSeeOtherGuests: true
  }

  // 5. Insertar evento
  const response = await calendar.events.insert({
    calendarId: 'primary',
    conferenceDataVersion: 1,
    sendUpdates: 'all', // Enviar emails a todos los invitados
    requestBody: event
  })

  // 6. Extraer información
  const createdEvent = response.data

  return {
    success: true,
    eventId: createdEvent.id!,
    eventLink: createdEvent.htmlLink!,
```

```
      meetLink: createdEvent.hangoutLink || createdEvent.conferenceData?.entryPoints?.[0]?.uri,
      startTime: createdEvent.start?.dateTime!,
      endTime: createdEvent.end?.dateTime!
    }
  } catch (error: any) {
    console.error('Error creando evento de Calendar:', error)

    // Manejo de errores específicos
    if (error.code === 401) {
      throw new Error('Sesión de Google Calendar expirada. Por favor reconecta tu cuenta.')
    }

    throw new Error('Error al crear evento: ' + error.message)
  }
}

// ACTUALIZAR EVENTO
export async function updatePracticeEvent({
  userId,
  eventId,
  scheduledFor,
  topic
}: {
  userId: string
  eventId: string
  scheduledFor?: Date
  topic?: string
}) {
  try {
    const auth = await getAuthenticatedClient(userId)
    const calendar = google.calendar({ version: 'v3', auth })

    const updateData: calendar_v3.Schema$Event = {}

    if (topic) {
      updateData.summary = ` Práctica de Inglés: ${topic}`
    }

    if (scheduledFor) {
      updateData.start = {
        dateTime: scheduledFor.toISOString(),
        timeZone: 'America/Bogota'
      }
      updateData.end = {
        dateTime: addMinutes(scheduledFor, 30).toISOString(),
        timeZone: 'America/Bogota'
      }
    }

    const response = await calendar.events.patch({
      calendarId: 'primary',
      eventId,
      sendUpdates: 'all',
      requestBody: updateData
```

```
    })

    return {
      success: true,
      event: response.data
    }
  } catch (error: any) {
    console.error('Error actualizando evento:', error)
    throw new Error('Error al actualizar evento: ' + error.message)
  }
}

// CANCELAR EVENTO
export async function cancelPracticeEvent({
  userId,
  eventId
}: {
  userId: string
  eventId: string
}) {
  try {
    const auth = await getAuthenticatedClient(userId)
    const calendar = google.calendar({ version: 'v3', auth })

    await calendar.events.delete({
      calendarId: 'primary',
      eventId,
      sendUpdates: 'all'
    })

    return {
      success: true,
      message: 'Evento cancelado correctamente'
    }
  } catch (error: any) {
    console.error('Error cancelando evento:', error)
    throw new Error('Error al cancelar evento: ' + error.message)
  }
}

// OBTENER DISPONIBILIDAD DE UN USUARIO
export async function getUserAvailability({
  userId,
  date
}: {
  userId: string
  date: Date
}) {
  try {
    const auth = await getAuthenticatedClient(userId)
    const calendar = google.calendar({ version: 'v3', auth })

    // Obtener eventos del día
    const startOfDay = new Date(date)
```

```
startOfDay.setHours(0, 0, 0, 0)

const endOfDay = new Date(date)
endOfDay.setHours(23, 59, 59, 999)

const response = await calendar.events.list({
  calendarId: 'primary',
  timeMin: startOfDay.toISOString(),
  timeMax: endOfDay.toISOString(),
  singleEvents: true,
  orderBy: 'startTime'
})

const busyEvents = response.data.items || []

// Generar slots de 30 minutos entre 8am y 10pm
const slots: Array<{ start: Date; end: Date; available: boolean }> = []
const startHour = 8
const endHour = 22

for (let hour = startHour; hour < endHour; hour++) {
  for (let minute of [0, 30]) {
    const slotStart = new Date(date)
    slotStart.setHours(hour, minute, 0, 0)

    const slotEnd = addMinutes(slotStart, 30)

    // Verificar si el slot está ocupado
    const isOccupied = busyEvents.some(event => {
      const eventStart = new Date(event.start?.dateTime || event.start?.date!)
      const eventEnd = new Date(event.end?.dateTime || event.end?.date!)

      return (
        (slotStart >= eventStart && slotStart < eventEnd) ||
        (slotEnd > eventStart && slotEnd <= eventEnd) ||
        (slotStart <= eventStart && slotEnd >= eventEnd)
      )
    })

    slots.push({
      start: slotStart,
      end: slotEnd,
      available: !isOccupied && slotStart > new Date() // No puede ser en el pasado
    })
  }
}

return {
  success: true,
  date,
  slots,
  busyEvents: busyEvents.map(e => ({
    summary: e.summary,
    start: e.start?.dateTime || e.start?.date,
```

```
        end: e.end?.dateTime || e.end?.date
      }))
    }
  } catch (error: any) {
    console.error('Error obteniendo disponibilidad:', error)
    throw new Error('Error al obtener disponibilidad: ' + error.message)
  }
}
```

2. API Routes

```
// app/api/auth/google-calendar/route.ts

import { NextResponse } from 'next/server'
import { getServerSession } from 'next-auth'
import { authOptions } from '@/lib/auth'
import { google } from 'googleapis'

const oauth2Client = new google.auth.OAuth2(
  process.env.GOOGLE_CLIENT_ID,
  process.env.GOOGLE_CLIENT_SECRET,
  `${process.env.NEXTAUTH_URL}/api/auth/google-calendar/callback`
)

// INICIAR AUTORIZACIÓN
export async function GET(request: Request) {
  try {
    const session = await getServerSession(authOptions)

    if (!session?.user?.id) {
      return NextResponse.json(
        { error: 'No autenticado' },
        { status: 401 }
      )
    }

    // Generar URL de autorización
    const scopes = [
      'https://www.googleapis.com/auth/calendar',
      'https://www.googleapis.com/auth/calendar.events'
    ]

    const authUrl = oauth2Client.generateAuthUrl({
      access_type: 'offline',
      scope: scopes,
      prompt: 'consent', // Fuerza a obtener refresh token
      state: session.user.id // Pasamos el userId en el state
    })

    return NextResponse.json({ authUrl })
  } catch (error: any) {
    console.error('Error generando auth URL:', error)
    return NextResponse.json(
      { error: error.message },
```

```
      { status: 500 }
    )
  }
}
```

```
// app/api/auth/google-calendar/callback/route.ts

import { NextResponse } from 'next/server'
import { google } from 'googleapis'
import prisma from '@/lib/db'

const oauth2Client = new google.auth.OAuth2(
  process.env.GOOGLE_CLIENT_ID,
  process.env.GOOGLE_CLIENT_SECRET,
  `${process.env.NEXTAUTH_URL}/api/auth/google-calendar/callback`
)

// CALLBACK DESPUÉS DE AUTORIZAR
export async function GET(request: Request) {
  try {
    const { searchParams } = new URL(request.url)
    const code = searchParams.get('code')
    const state = searchParams.get('state') // userId
    const error = searchParams.get('error')

    // Usuario canceló
    if (error) {
      return NextResponse.redirect(
        `${process.env.NEXTAUTH_URL}/practice?calendar=cancelled`
      )
    }

    if (!code || !state) {
      throw new Error('Código o estado faltante')
    }

    // Intercambiar código por tokens
    const { tokens } = await oauth2Client.getToken(code)

    if (!tokens.access_token || !tokens.refresh_token) {
      throw new Error('Tokens inválidos')
    }

    // Guardar en DB
    await prisma.calendarIntegration.upsert({
      where: { userId: state },
      create: {
        userId: state,
        provider: 'google',
        accessToken: tokens.access_token,
        refreshToken: tokens.refresh_token,
        expiresAt: new Date(tokens.expiry_date!)
      },
      update: {
```

```
        accessToken: tokens.access_token,
        refreshToken: tokens.refresh_token,
        expiresAt: new Date(tokens.expiry_date!)
      }
    })

    // Redirigir al usuario
    return NextResponse.redirect(
      `${process.env.NEXTAUTH_URL}/practice?calendar=connected`
    )
  } catch (error: any) {
    console.error('Error en callback de Calendar:', error)
    return NextResponse.redirect(
      `${process.env.NEXTAUTH_URL}/practice?calendar=error`
    )
  }
}
```

```
// app/api/practice/sessions/route.ts (ACTUALIZADO)

import { NextResponse } from 'next/server'
import { getServerSession } from 'next-auth'
import { authOptions } from '@/lib/auth'
import { createMeeting } from '@/lib/services/practice-service'
import { notifySessionScheduled } from '@/lib/services/practice-notification-service'
import { createPracticeEvent } from '@/lib/services/google-calendar-service'
import prisma from '@/lib/db'

export async function POST(request: Request) {
  try {
    const session = await getServerSession(authOptions)

    if (!session?.user?.id) {
      return NextResponse.json(
        { error: 'No autenticado' },
        { status: 401 }
      )
    }

    const body = await request.json()
    const { partnerId, scheduledFor, topic, useGoogleCalendar } = body

    if (!partnerId || !scheduledFor || !topic) {
      return NextResponse.json(
        { error: 'Faltan datos requeridos' },
        { status: 400 }
      )
    }

    let meetLink: string | undefined
    let calendarEventId: string | undefined

    // Si el usuario quiere usar Google Calendar
    if (useGoogleCalendar) {
```

```
  try {
    // Verificar que el usuario tiene Calendar conectado
    const integration = await prisma.calendarIntegration.findUnique({
      where: { userId: session.user.id }
    })

    if (!integration) {
      return NextResponse.json(
        { error: 'Conecta Google Calendar primero' },
        { status: 400 }
      )
    }

    // Crear evento en Calendar + Meet link
    const calendarResult = await createPracticeEvent({
      userId: session.user.id,
      partnerId,
      scheduledFor: new Date(scheduledFor),
      topic
    })

    meetLink = calendarResult.meetLink
    calendarEventId = calendarResult.eventId
  } catch (calendarError: any) {
    console.error('Error con Calendar, continuando sin él:', calendarError)
    // Continuar sin Calendar si falla
  }
}

// Crear meeting en nuestra DB
const meeting = await createMeeting({
  initiatorId: session.user.id,
  partnerId,
  scheduledFor: new Date(scheduledFor),
  topic,
  externalLink: meetLink,
  calendarEventId
})

// Notificar al partner
await notifySessionScheduled(
  partnerId,
  session.user.name || 'Un usuario',
  meeting.id,
  new Date(scheduledFor)
)

return NextResponse.json({
  success: true,
  session: meeting,
  meetLink,
  message: meetLink
    ? 'Sesión programada con Google Meet'
    : 'Sesión programada correctamente'
```

```
    })
  } catch (error: any) {
    console.error('Error creating session:', error)
    return NextResponse.json(
      { error: error.message || 'Error al crear sesión' },
      { status: 400 }
    )
  }
}
```

Código Frontend

1. Botón de Conectar Calendar

```tsx
// components/practice/connect-calendar-button.tsx

'use client'

import { useState } from 'react'
import { Button } from '@/components/ui/button'
import { Calendar, CheckCircle2, Loader2 } from 'lucide-react'
import { useToast } from '@/hooks/use-toast'

export function ConnectCalendarButton({ isConnected }: { isConnected: boolean }) {
  const [loading, setLoading] = useState(false)
  const { toast } = useToast()

  const handleConnect = async () => {
    try {
      setLoading(true)

      // Obtener URL de autorización
      const response = await fetch('/api/auth/google-calendar')
      const data = await response.json()

      if (!data.authUrl) {
        throw new Error('Error obteniendo URL de autorización')
      }

      // Abrir popup de Google
      const width = 600
      const height = 700
      const left = window.screen.width / 2 - width / 2
      const top = window.screen.height / 2 - height / 2

      const popup = window.open(
        data.authUrl,
        'Google Calendar Authorization',
        `width=${width},height=${height},left=${left},top=${top}`
      )

      // Esperar a que el usuario autorice
      const checkClosed = setInterval(() => {
```

```
        if (popup?.closed) {
          clearInterval(checkClosed)
          setLoading(false)
          window.location.reload() // Recargar para actualizar estado
        }
      }, 500)
    } catch (error: any) {
      console.error('Error conectando Calendar:', error)
      toast({
        title: 'Error',
        description: error.message,
        variant: 'destructive'
      })
      setLoading(false)
    }
  }

  if (isConnected) {
    return (
      <Button variant="outline" disabled className="w-full">
        <CheckCircle2 className="mr-2 h-4 w-4 text-green-600" />
        Google Calendar Conectado
      </Button>
    )
  }

  return (
    <Button
      onClick={handleConnect}
      disabled={loading}
      className="w-full"
    >
      {loading ? (
        <>
          <Loader2 className="mr-2 h-4 w-4 animate-spin" />
          Conectando...
        </>
      ) : (
        <>
          <Calendar className="mr-2 h-4 w-4" />
          Conectar Google Calendar
        </>
      )}
    </Button>
  )
}
```

2. Selector de Disponibilidad

```
// components/practice/availability-picker.tsx

'use client'

import { useState, useEffect } from 'react'
```

```tsx
import { Button } from '@/components/ui/button'
import { Calendar } from '@/components/ui/calendar'
import { Loader2 } from 'lucide-react'
import { format, addDays } from 'date-fns'
import { es } from 'date-fns/locale'

interface Slot {
  start: Date
  end: Date
  available: boolean
}

export function AvailabilityPicker({
  partnerId,
  onSelectSlot
}: {
  partnerId: string
  onSelectSlot: (slot: Slot) => void
}) {
  const [selectedDate, setSelectedDate] = useState<Date>(new Date())
  const [slots, setSlots] = useState<Slot[]>([])
  const [loading, setLoading] = useState(false)

  useEffect(() => {
    loadAvailability()
  }, [selectedDate])

  const loadAvailability = async () => {
    try {
      setLoading(true)
      const response = await fetch(
        `/api/practice/availability/${partnerId}?date=${selectedDate.toISOString()}`
      )
      const data = await response.json()

      if (data.success) {
        setSlots(data.slots.map((s: any) => ({
          ...s,
          start: new Date(s.start),
          end: new Date(s.end)
        })))
      }
    } catch (error) {
      console.error('Error cargando disponibilidad:', error)
    } finally {
      setLoading(false)
    }
  }

  return (
    <div className="space-y-4">
      {/* Calendario */}
      <Calendar
        mode="single"
```

```
          selected={selectedDate}
          onSelect={(date) => date && setSelectedDate(date)}
          disabled={(date) => date < new Date()}
          locale={es}
          className="rounded-md border"
        />

        {/* Slots de hora */}
        <div>
          <h4 className="font-medium mb-2">
            Horarios disponibles - {format(selectedDate, 'PPPP', { locale: es })}
          </h4>

          {loading ? (
            <div className="flex items-center justify-center py-8">
              <Loader2 className="h-6 w-6 animate-spin" />
            </div>
          ) : (
            <div className="grid grid-cols-3 gap-2 max-h-[300px] overflow-y-auto">
              {slots.map((slot, index) => (
                <Button
                  key={index}
                  variant={slot.available ? 'outline' : 'ghost'}
                  disabled={!slot.available}
                  onClick={() => onSelectSlot(slot)}
                  className="text-sm"
                >
                  {format(slot.start, 'HH:mm')}
                </Button>
              ))}
            </div>
          )}
        </div>
      </div>
    )
}
```

3. Modal de Agendar Sesión

```
// components/practice/schedule-session-modal.tsx

'use client'

import { useState } from 'react'
import { Button } from '@/components/ui/button'
import { Input } from '@/components/ui/input'
import { Label } from '@/components/ui/label'
import { Checkbox } from '@/components/ui/checkbox'
import {
  Dialog,
  DialogContent,
  DialogDescription,
  DialogFooter,
  DialogHeader,
```

```
  DialogTitle,
} from '@/components/ui/dialog'
import { AvailabilityPicker } from './availability-picker'
import { useToast } from '@/hooks/use-toast'
import { Loader2, Calendar, Video } from 'lucide-react'

export function ScheduleSessionModal({
  open,
  onClose,
  partnerId,
  partnerName,
  hasGoogleCalendar
}: {
  open: boolean
  onClose: () => void
  partnerId: string
  partnerName: string
  hasGoogleCalendar: boolean
}) {
  const [loading, setLoading] = useState(false)
  const [topic, setTopic] = useState('')
  const [selectedSlot, setSelectedSlot] = useState<any>(null)
  const [useGoogleCalendar, setUseGoogleCalendar] = useState(hasGoogleCalendar)
  const { toast } = useToast()

  const handleSchedule = async () => {
    try {
      if (!topic.trim()) {
        toast({
          title: 'Error',
          description: 'Por favor escribe un tema para la sesión',
          variant: 'destructive'
        })
        return
      }

      if (!selectedSlot) {
        toast({
          title: 'Error',
          description: 'Por favor selecciona un horario',
          variant: 'destructive'
        })
        return
      }

      setLoading(true)

      const response = await fetch('/api/practice/sessions', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          partnerId,
          scheduledFor: selectedSlot.start.toISOString(),
          topic,
```

```
        useGoogleCalendar
      })
    })

    const data = await response.json()

    if (!data.success) {
      throw new Error(data.error || 'Error al programar sesión')
    }

    toast({
      title: '¡Sesión programada! ',
      description: data.meetLink
        ? 'Evento agregado a tu Calendar con link de Meet'
        : `Sesión confirmada con ${partnerName}`
    })

    onClose()
    window.location.reload()
  } catch (error: any) {
    console.error('Error:', error)
    toast({
      title: 'Error',
      description: error.message,
      variant: 'destructive'
    })
  } finally {
    setLoading(false)
  }
}

return (
  <Dialog open={open} onOpenChange={onClose}>
    <DialogContent className="max-w-2xl max-h-[90vh] overflow-y-auto">
      <DialogHeader>
        <DialogTitle>
          Programar sesión con {partnerName}
        </DialogTitle>
        <DialogDescription>
          Selecciona un horario disponible y el tema de conversación
        </DialogDescription>
      </DialogHeader>

      <div className="space-y-4 py-4">
        {/* Tema */}
        <div>
          <Label htmlFor="topic">Tema de conversación</Label>
          <Input
            id="topic"
            placeholder="Ej: Business presentations, Travel conversations..."
            value={topic}
            onChange={(e) => setTopic(e.target.value)}
            className="mt-1"
          />
```

```
      </div>

      {/* Selector de horario */}
      <div>
        <Label>Fecha y hora</Label>
        <AvailabilityPicker
          partnerId={partnerId}
          onSelectSlot={setSelectedSlot}
        />
        {selectedSlot && (
          <div className="mt-2 p-3 bg-blue-50 rounded-lg border border-blue-200">
            <p className="text-sm font-medium text-blue-900">
                Seleccionado: {selectedSlot.start.toLocaleString('es-CO', {
                 dateStyle: 'full',
                 timeStyle: 'short'
               })}
            </p>
          </div>
        )}
      </div>

      {/* Google Calendar */}
      {hasGoogleCalendar && (
        <div className="flex items-start space-x-2 p-4 bg-green-50 rounded-lg border border-green-2
          <Checkbox
            id="useCalendar"
            checked={useGoogleCalendar}
            onCheckedChange={(checked) => setUseGoogleCalendar(!!checked)}
          />
          <div className="space-y-1">
            <label
              htmlFor="useCalendar"
              className="text-sm font-medium leading-none peer-disabled:cursor-not-allowed peer-disa
            >
              <Calendar className="inline h-4 w-4 mr-1" />
              Crear evento en Google Calendar
            </label>
            <p className="text-xs text-muted-foreground">
              Se generará un link de Google Meet automáticamente y ambos recibirán recordatorios
            </p>
          </div>
        </div>
      )}
    </div>

    <DialogFooter>
      <Button variant="outline" onClick={onClose} disabled={loading}>
        Cancelar
      </Button>
      <Button onClick={handleSchedule} disabled={loading}>
        {loading ? (
          <>
            <Loader2 className="mr-2 h-4 w-4 animate-spin" />
            Programando...
```

```
              </>
            ) : (
              <>
                <Video className="mr-2 h-4 w-4" />
                Programar Sesión
              </>
            )}
          </Button>
        </DialogFooter>
      </DialogContent>
    </Dialog>
  )
}
```

Testing

Test Manual

```
// scripts/test-google-calendar.ts

import { createPracticeEvent } from '@/lib/services/google-calendar-service'

async function test() {
  const result = await createPracticeEvent({
    userId: 'tu_user_id',
    partnerId: 'partner_user_id',
    scheduledFor: new Date('2025-10-20T15:00:00-05:00'),
    topic: 'Business English - Presentations',
    durationMinutes: 30
  })

  console.log(' Evento creado:', result)
  console.log(' Meet link:', result.meetLink)
}

test()
```

Verificaciones

1. Evento aparece en Google Calendar
2. Invitado recibe email
3. Link de Meet funciona
4. Recordatorios llegan 24h/1h antes
5. Timezone es correcto

Troubleshooting

Error: “Access not configured”

- Verificar que Calendar API está habilitada en Google Cloud Console

Error: "Invalid token"

- Token expiró → El sistema debería refrescarlo automáticamente
- Si persiste, pedir al usuario que reconecte

Error: "Redirect URI mismatch"

- Verificar que la URI en Google Cloud Console coincide exactamente con la del código

Meet link no se genera

- Verificar que `conferenceDataVersion: 1` está presente
- Verificar que `conferenceSolutionKey: { type: 'hangoutsMeet' }` está correcto

Modelo de Prisma Actualizado

```
model CalendarIntegration {
  id            String   @id @default(cuid())
  userId        String   @unique
  user          User     @relation(fields: [userId], references: [id], onDelete: Cascade)

  provider      String   // "google", "outlook"
  accessToken   String   @db.Text
  refreshToken  String   @db.Text
  expiresAt     DateTime

  createdAt     DateTime @default(now())
  updatedAt     DateTime @updatedAt

  @@index([userId])
}

model PracticeMeeting {
  id          String   @id @default(cuid())

  // ... campos existentes ...

  // NUEVO: ID del evento de Calendar
  calendarEventId String?

  // ... resto de campos ...
}
```

```
# Aplicar migración
npx prisma migrate dev --name add_calendar_integration
```

Checklist de Implementación

- ☐ Crear proyecto en Google Cloud Console
- ☐ Habilitar Calendar API
- ☐ Crear credenciales OAuth 2.0
- ☐ Agregar variables de entorno
- ☐ Instalar dependencias (`googleapis`)

☐ Crear modelo `CalendarIntegration` en Prisma
☐ Aplicar migración
☐ Implementar `google-calendar-service.ts`
☐ Crear API routes OAuth
☐ Actualizar API de sesiones
☐ Crear componentes frontend
☐ Testear flujo completo
☐ Deploy a producción

Resultado Final

Usuario A invita a Usuario B: 1. Click en "Programar sesión" 2. Selecciona fecha/hora del calendario visual 3. Escribe tema: "Business English" 4. Crear evento en Google Calendar 5. Click "Programar"

Sistema automáticamente: - Crea evento en Calendar de A - Envía invitación a email de B - Genera link de Google Meet - Programa recordatorios 24h/1h antes - Guarda todo en BD de SpeaklyPlan

Ambos usuarios: - Reciben email de Google con detalles - Evento aparece en sus Calendars - Reciben notificaciones automáticas - Tienen link de Meet listo para usar

Costo: $0 **Tiempo de implementación:** 1-2 días **Mantenimiento:** Cero (Google maneja todo)

¿Listo para implementar?