Soshogle CRM - Integration Guide for Developers

🎯 Purpose

This guide helps developers integrate with Soshogle CRM, either to:
1. **Extend functionality** - Add new features to the CRM
2. **Integrate external systems** - Connect other apps to Soshogle
3. **Build on top of Soshogle** - Use Soshogle as a backend for custom apps

---

🚀 Quick Start

1. Clone & Setup

```
# Clone repository
git clone <repository-url>
cd go_high_or_show_google_crm/nextjs_space

# Install dependencies
yarn install

# Setup environment variables
cp .env.example .env
# Edit .env with your credentials

# Generate Prisma client
yarn prisma generate

# Run migrations
yarn prisma migrate deploy

# Start development server
yarn dev
```

The app will be available at `http://localhost:3000`

---

2. Authentication Setup

Soshogle uses **NextAuth.js** with credentials provider.

Get Session in Server Components:

```
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";

export default async function MyPage() {
  const session = await getServerSession(authOptions);

  if (!session) {
    redirect("/auth/signin");
  }

  return <div>Hello {session.user.name}</div>;
}
```

Get Session in Client Components:

```
"use client";
import { useSession } from "next-auth/react";

export default function MyComponent() {
  const { data: session, status } = useSession() || {};

  if (status === "loading") return <div>Loading...</div>;
  if (!session) return <div>Please sign in</div>;

  return <div>Hello {session.user.name}</div>;
}
```

Protect API Routes:

```
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";
import { NextResponse } from "next/server";

export async function GET(request: Request) {
  const session = await getServerSession(authOptions);

  if (!session) {
    return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
  }

  const userId = session.user.id;

  // Your logic here
  return NextResponse.json({ data: "success" });
}
```

---

3. Database Access

Soshogle uses **Prisma ORM** with PostgreSQL.

Prisma Client Singleton:

```
// lib/db.ts
import { PrismaClient } from '@prisma/client';

declare global {
  var prisma: PrismaClient | undefined;
}

export const prisma = globalThis.prisma || new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  globalThis.prisma = prisma;
}
```

Query Examples:

```
import { prisma } from "@/lib/db";

// Get all contacts for a user
const contacts = await prisma.contact.findMany({
  where: { userId },
  include: {
    messages: true,
    appointments: true
  },
  orderBy: { createdAt: 'desc' }
});

// Create a contact
const newContact = await prisma.contact.create({
  data: {
    userId,
    firstName: "John",
    lastName: "Doe",
    email: "john@example.com",
    tags: ["Prospect"]
  }
});

// Update a contact
const updated = await prisma.contact.update({
  where: { id: contactId },
  data: {
    tags: ["Customer", "VIP"]
  }
});

// Delete a contact (cascade deletes related records)
await prisma.contact.delete({
  where: { id: contactId }
});
```

🔌 Integration Patterns

Pattern 1: Adding a New API Endpoint

File: `app/api/my-endpoint/route.ts`

```
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";
import { NextResponse } from "next/server";
import { prisma } from "@/lib/db";

export async function GET(request: Request) {
  try {
    // 1. Authenticate
    const session = await getServerSession(authOptions);
    if (!session) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const userId = session.user.id;

    // 2. Parse query parameters
    const { searchParams } = new URL(request.url);
    const search = searchParams.get("search") || "";

    // 3. Query database
    const results = await prisma.contact.findMany({
      where: {
        userId,
        OR: [
          { firstName: { contains: search, mode: 'insensitive' } },
          { lastName: { contains: search, mode: 'insensitive' } },
          { email: { contains: search, mode: 'insensitive' } }
        ]
      },
      take: 20
    });

    // 4. Return response
    return NextResponse.json({ results });

  } catch (error) {
    console.error("API Error:", error);
    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 }
    );
  }
}

export async function POST(request: Request) {
  try {
    const session = await getServerSession(authOptions);
    if (!session) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const userId = session.user.id;
    const body = await request.json();

    // Validate input (use Zod for complex validation)
    if (!body.firstName) {
      return NextResponse.json(
        { error: "firstName is required" },
        { status: 400 }
      );
    }
```

```
    // Create record
    const contact = await prisma.contact.create({
      data: {
        userId,
        firstName: body.firstName,
        lastName: body.lastName,
        email: body.email,
        tags: body.tags || []
      }
    });

    return NextResponse.json(contact, { status: 201 });

  } catch (error) {
    console.error("API Error:", error);
    return NextResponse.json(
      { error: "Internal server error" },
      { status: 500 }
    );
  }
}
```

Pattern 2: Creating a New Page

File: `app/dashboard/my-page/page.tsx`

```
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";
import { redirect } from "next/navigation";
import { prisma } from "@/lib/db";
import MyClientComponent from "@/components/my-client-component";

export default async function MyPage() {
  // Server-side authentication
  const session = await getServerSession(authOptions);

  if (!session) {
    redirect("/auth/signin");
  }

  // Fetch data on server
  const contacts = await prisma.contact.findMany({
    where: { userId: session.user.id },
    take: 10
  });

  return (
    <div>
      <h1>My Custom Page</h1>
      <MyClientComponent initialData={contacts} userId={session.user.id} />
    </div>
  );
}
```

File: `components/my-client-component.tsx`

```tsx
"use client";
import { useState, useEffect } from "react";
import { toast } from "sonner";

interface Props {
  initialData: any[];
  userId: string;
}

export default function MyClientComponent({ initialData, userId }: Props) {
  const [data, setData] = useState(initialData);
  const [loading, setLoading] = useState(false);

  const fetchData = async () => {
    setLoading(true);
    try {
      const res = await fetch("/api/my-endpoint");
      const json = await res.json();
      setData(json.results);
    } catch (error) {
      toast.error("Failed to fetch data");
    } finally {
      setLoading(false);
    }
  };

  return (
    <div>
      <button onClick={fetchData} disabled={loading}>
        {loading ? "Loading..." : "Refresh"}
      </button>

      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.firstName}</li>
        ))}
      </ul>
    </div>
  );
}
```

Pattern 3: Adding a New Service

File: `lib/my-service.ts`

```
import { prisma } from "./db";

export class MyService {
  async processContact(contactId: string) {
    // Fetch contact
    const contact = await prisma.contact.findUnique({
      where: { id: contactId },
      include: {
        messages: true,
        appointments: true
      }
    });

    if (!contact) {
      throw new Error("Contact not found");
    }

    // Your business logic
    const result = this.doSomething(contact);

    // Update contact
    await prisma.contact.update({
      where: { id: contactId },
      data: {
        customFields: {
          ...contact.customFields,
          processedAt: new Date().toISOString()
        }
      }
    });

    return result;
  }

  private doSomething(contact: any) {
    // Your logic here
    return { success: true };
  }
}

export const myService = new MyService();
```

Usage:

```
import { myService } from "@/lib/my-service";

// In API route or server component
const result = await myService.processContact(contactId);
```

🎨 UI Components

Soshogle uses **shadcn/ui** components built on **Radix UI** and **Tailwind CSS**.

Using Existing Components:

```
import { Button } from "@/components/ui/button";
import { Input } from "@/components/ui/input";
import { Dialog, DialogContent, DialogTrigger } from "@/components/ui/dialog";
import { toast } from "sonner";

export default function MyForm() {
  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();

    try {
      const res = await fetch("/api/my-endpoint", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ data: "example" })
      });

      if (res.ok) {
        toast.success("Success!");
      } else {
        toast.error("Failed");
      }
    } catch (error) {
      toast.error("Network error");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <Input placeholder="Enter name" />
      <Button type="submit">Submit</Button>
    </form>
  );
}
```

Creating New shadcn/ui Components:

```
# Add a new component from shadcn/ui
npx shadcn-ui@latest add <component-name>

# Example:
npx shadcn-ui@latest add dropdown-menu
```

🔗 External Service Integration

Example: Integrating with Slack

1. Create Service

File: `lib/integrations/slack-service.ts`

```
export class SlackService {
  private webhookUrl: string;

  constructor() {
    this.webhookUrl = process.env.SLACK_WEBHOOK_URL || "";
  }

  async sendNotification(message: string) {
    try {
      const res = await fetch(this.webhookUrl, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ text: message })
      });

      if (!res.ok) {
        throw new Error("Slack API error");
      }

      return { success: true };
    } catch (error) {
      console.error("Slack notification error:", error);
      throw error;
    }
  }
}

export const slackService = new SlackService();
```

2. Create API Endpoint

File: `app/api/integrations/slack/notify/route.ts`

```
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";
import { NextResponse } from "next/server";
import { slackService } from "@/lib/integrations/slack-service";

export async function POST(request: Request) {
  try {
    const session = await getServerSession(authOptions);
    if (!session) {
      return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
    }

    const { message } = await request.json();

    await slackService.sendNotification(message);

    return NextResponse.json({ success: true });
  } catch (error) {
    return NextResponse.json(
      { error: "Failed to send notification" },
      { status: 500 }
    );
  }
}
```

3. Environment Variable

Add to `.env`:

```
SLACK_WEBHOOK_URL="https://hooks.slack.com/services/YOUR/WEBHOOK/URL"
```

🤖 ElevenLabs Voice Agent Integration

Creating a Voice Agent Programmatically

```
import { ElevenLabsProvisioningService } from "@/lib/elevenlabs-provisioning";

const service = new ElevenLabsProvisioningService();

// Create agent
const agent = await service.createAgent({
  name: "My Sales Agent",
  firstMessage: "Hi! How can I help you today?",
  systemPrompt: "You are a helpful sales assistant."
});

// Import phone number
const phoneImport = await service.importPhoneNumber("+15551234567");

// Assign phone to agent
await service.assignPhoneToAgent(agent.agent_id, phoneImport.phone_number_id);

// Save to database
await prisma.voiceAgent.create({
  data: {
    userId,
    name: "My Sales Agent",
    elevenLabsAgentId: agent.agent_id,
    phoneNumberId: phoneImport.phone_number_id,
    twilioPhoneNumber: "+15551234567",
    status: "ACTIVE"
  }
});
```

📊 Workflow Automation Integration

Creating a Custom Workflow Trigger

File: `lib/workflow-triggers/custom-trigger.ts`

```
import { prisma } from "@/lib/db";
import { executeWorkflow } from "@/lib/workflow-engine";

export async function triggerCustomEvent(userId: string, data: any) {
  // Find workflows with custom trigger
  const workflows = await prisma.workflow.findMany({
    where: {
      userId,
      status: "ACTIVE",
      trigger: "CUSTOM_EVENT" // Add to enum in schema
    }
  });

  // Execute each workflow
  for (const workflow of workflows) {
    try {
      await executeWorkflow(workflow.id, data);
    } catch (error) {
      console.error(`Workflow ${workflow.id} failed:`, error);
    }
  }
}
```

Usage:

```
// In your API route
import { triggerCustomEvent } from "@/lib/workflow-triggers/custom-trigger";

// After some event
await triggerCustomEvent(userId, {
  eventType: "purchase_completed",
  amount: 150.00,
  contactId: "clx123"
});
```

🔒 Security Best Practices

1. Always Validate User Ownership

```
// ❌ BAD - No ownership check
const contact = await prisma.contact.findUnique({
  where: { id: contactId }
});

// ✅ GOOD - Verify user owns the contact
const contact = await prisma.contact.findFirst({
  where: {
    id: contactId,
    userId: session.user.id
  }
});
```

2. Use Prepared Statements (Prisma handles this)

```
// ✅ Prisma automatically prevents SQL injection
const contacts = await prisma.contact.findMany({
  where: {
    email: { contains: userInput } // Safe
  }
});
```

3. Rate Limiting (Example with middleware)

```
// middleware.ts
import { NextResponse } from "next/server";
import type { NextRequest } from "next/server";

const rateLimitMap = new Map<string, { count: number; resetTime: number }>();

export function middleware(request: NextRequest) {
  const ip = request.ip || "127.0.0.1";
  const now = Date.now();
  const limit = 100; // requests
  const window = 60 * 1000; // 1 minute

  const rateLimitData = rateLimitMap.get(ip) || { count: 0, resetTime: now + window };

  if (now > rateLimitData.resetTime) {
    rateLimitData.count = 0;
    rateLimitData.resetTime = now + window;
  }

  rateLimitData.count++;
  rateLimitMap.set(ip, rateLimitData);

  if (rateLimitData.count > limit) {
    return NextResponse.json(
      { error: "Too many requests" },
      { status: 429 }
    );
  }

  return NextResponse.next();
}
```

4. Environment Variables

```
// ❌ BAD - Exposing server secret to client
const ELEVENLABS_API_KEY = process.env.NEXT_PUBLIC_ELEVENLABS_API_KEY;

// ✅ GOOD - Server-only variable
const ELEVENLABS_API_KEY = process.env.ELEVENLABS_API_KEY;
```

🧪 Testing

API Route Testing Example

File: `__tests__/api/contacts.test.ts`

```
import { GET, POST } from "@/app/api/contacts/route";
import { prisma } from "@/lib/db";

// Mock NextAuth
jest.mock("next-auth", () => ({
  getServerSession: jest.fn(() => ({
    user: { id: "test-user-123", email: "test@example.com" }
  }))
}));

describe("/api/contacts", () => {
  afterAll(async () => {
    await prisma.$disconnect();
  });

  it("should return contacts for authenticated user", async () => {
    const request = new Request("http://localhost:3000/api/contacts");
    const response = await GET(request);
    const data = await response.json();

    expect(response.status).toBe(200);
    expect(Array.isArray(data.contacts)).toBe(true);
  });

  it("should create a new contact", async () => {
    const request = new Request("http://localhost:3000/api/contacts", {
      method: "POST",
      body: JSON.stringify({
        firstName: "John",
        lastName: "Doe",
        email: "john@example.com"
      })
    });

    const response = await POST(request);
    const data = await response.json();

    expect(response.status).toBe(201);
    expect(data.firstName).toBe("John");

    // Cleanup
    await prisma.contact.delete({ where: { id: data.id } });
  });
});
```

📚 Common Integration Scenarios

Scenario 1: Add a New Contact via External Form

Webhook endpoint: `POST /api/webhooks/contact-form`

```
import { NextResponse } from "next/server";
import { prisma } from "@/lib/db";

export async function POST(request: Request) {
  try {
    // Verify webhook signature (if using services like Typeform, Zapier)
    const signature = request.headers.get("x-webhook-signature");
    // ... verify signature logic

    const body = await request.json();

    // Find user by API key or subdomain
    const user = await prisma.user.findFirst({
      where: { company: body.companyId }
    });

    if (!user) {
      return NextResponse.json({ error: "User not found" }, { status: 404 });
    }

    // Create contact
    const contact = await prisma.contact.create({
      data: {
        userId: user.id,
        firstName: body.firstName,
        lastName: body.lastName,
        email: body.email,
        phone: body.phone,
        source: "external_form",
        tags: ["Lead", "Website"]
      }
    });

    // Trigger workflows
    const workflows = await prisma.workflow.findMany({
      where: {
        userId: user.id,
        trigger: "CONTACT_CREATED",
        status: "ACTIVE"
      }
    });

    for (const workflow of workflows) {
      // Execute workflow (send welcome email, SMS, etc.)
      await executeWorkflow(workflow.id, { contactId: contact.id });
    }

    return NextResponse.json({ success: true, contactId: contact.id });
  } catch (error) {
    console.error("Webhook error:", error);
    return NextResponse.json({ error: "Processing failed" }, { status: 500 });
  }
}
```

Scenario 2: Export Contacts to External CRM

```
import { prisma } from "@/lib/db";

export async function exportContactsToHubSpot(userId: string) {
  const contacts = await prisma.contact.findMany({
    where: { userId }
  });

  const hubspotApiKey = process.env.HUBSPOT_API_KEY;

  for (const contact of contacts) {
    try {
      const res = await fetch("https://api.hubapi.com/crm/v3/objects/contacts", {
        method: "POST",
        headers: {
          "Authorization": `Bearer ${hubspotApiKey}`,
          "Content-Type": "application/json"
        },
        body: JSON.stringify({
          properties: {
            email: contact.email,
            firstname: contact.firstName,
            lastname: contact.lastName,
            phone: contact.phone
          }
        })
      });

      if (!res.ok) {
        console.error(`Failed to export contact ${contact.id}`);
      }
    } catch (error) {
      console.error(`Error exporting contact ${contact.id}:`, error);
    }
  }
}
```

🔄 Real-time Updates (WebSockets - Optional)

For real-time features like live chat or notifications, you can integrate WebSockets using **Pusher** or **Socket.IO**.

Example with Pusher:

```
// lib/pusher.ts
import Pusher from "pusher";

export const pusher = new Pusher({
  appId: process.env.PUSHER_APP_ID!,
  key: process.env.NEXT_PUBLIC_PUSHER_KEY!,
  secret: process.env.PUSHER_SECRET!,
  cluster: process.env.PUSHER_CLUSTER!,
  useTLS: true
});
```

Trigger event:

```
import { pusher } from "@/lib/pusher";

// After creating a message
await pusher.trigger(
  `private-user-${userId}`,
  "new-message",
  {
    messageId: message.id,
    content: message.content,
    from: message.from
  }
);
```

Client-side listener:

```
"use client";
import { useEffect } from "react";
import Pusher from "pusher-js";

export default function useRealtimeMessages(userId: string) {
  useEffect(() => {
    const pusher = new Pusher(process.env.NEXT_PUBLIC_PUSHER_KEY!, {
      cluster: process.env.NEXT_PUBLIC_PUSHER_CLUSTER!
    });

    const channel = pusher.subscribe(`private-user-${userId}`);

    channel.bind("new-message", (data: any) => {
      console.log("New message received:", data);
      // Update UI
    });

    return () => {
      channel.unbind_all();
      channel.unsubscribe();
    };
  }, [userId]);
}
```

📖 Additional Resources

Documentation Files

- `README.md` - Project overview
- `API_DOCUMENTATION.md` - Complete API reference
- `DATABASE_SCHEMA.md` - Database structure
- `ELEVENLABS_SETUP_GUIDE.md` - Voice agent setup
- `PHONE_NUMBER_SYNC_FIX.md` - Twilio/ElevenLabs integration

External Documentation

- Next.js Docs (https://nextjs.org/docs)
- Prisma Docs (https://www.prisma.io/docs)

- NextAuth.js Docs (https://next-auth.js.org)
- shadcn/ui Components (https://ui.shadcn.com)
- ElevenLabs API Docs (https://elevenlabs.io/docs)
- Twilio API Docs (https://www.twilio.com/docs)

🤦 Support

For integration questions or issues:
1. Check the documentation files in project root
2. Review API endpoint responses for error details
3. Use diagnostic scripts (`tsx check_*.ts`) for troubleshooting
4. Inspect browser console for frontend errors
5. Check server logs for backend errors

Last Updated: November 2025
Integration Guide Version: 1.0