Based on the codebase context, you're asking about converting Leon AI into a desktop application. Leon is currently a web-based personal assistant that runs as a server with a web client interface.

## Current Architecture

Leon currently operates as a web application with these main components:

- **Server**: Node.js/TypeScript backend [1](#0-0)

- **Web Client**: HTML/JavaScript frontend served at `http://localhost:1337` [2](#0-1)

- **TCP Server**: Python-based voice processing component [3](#0-2)

## Desktop Application Options

Leon already has infrastructure that could support desktop deployment:

### 1. Electron Wrapper
The existing web client [4](#0-3) could be wrapped in Electron to create a native desktop app. The client uses Socket.IO for real-time communication with the server.

### 2. Standalone Binaries
Leon already builds standalone binaries for its components using cx_Freeze:

- **TCP Server Binary**: [3](#0-2)

- **Python Bridge Binary**: [5](#0-4)

- **Build System**: [6](#0-5)

### 3. Self-Contained Package

The build system [7](#0-6) already supports creating production builds that could be packaged as a desktop application.

## Implementation Approach

To convert Leon to a desktop app, you would need to:

1. **Package the server** as a background service
2. **Wrap the web client** in Electron or similar framework
3. **Bundle the TCP server** and Python bridge binaries
4. **Handle local file paths** and configuration management

The existing binary building infrastructure [8](#0-7) already detects when running from a binary and adjusts paths accordingly.

## Notes

Leon's modular architecture with separate HTTP/Socket servers, TCP server, and web client makes it well-suited for desktop packaging. The existing build scripts and binary creation tools provide a foundation for creating a self-contained desktop application.

Wiki pages you might want to explore:
- [HTTP & Socket Servers (leon-ai/leon)](/wiki/leon-ai/leon#2.2)
- [TCP Server (leon-ai/leon)](/wiki/leon-ai/leon#4.1)
- [Client Application (leon-ai/leon)](/wiki/leon-ai/leon#5)

It seems you're asking about changing the design of Leon's desktop application interface. Based on the codebase context, this would involve modifying the client application's visual components and styling.

## Design Components You Can Modify

The main design elements are located in Leon's web client application:

### 1. HTML Structure

The core layout is defined in [1](#2-0) , which includes:

- Main chat feed container
- Voice overlay interface
- Input container with microphone button
- Suggestions area
- Footer with branding

### 2. CSS Styling

The visual styling is controlled by [2](#2-1) , which defines:

- Chat bubble colors and positioning
- Suggestion button styling
- Input container layout
- Voice interface animations

### 3. Voice Interface Design

The voice mode has its own visual components in [3](#2-2) , creating animated particles and energy visualization effects.

## Key Areas for Design Changes

### Chat Interface

You can modify the chat bubble appearance by updating the CSS classes `.me .bubble` and `.leon .bubble` which control user and Leon's message styling respectively.

### Voice Visualization

The voice energy system creates 32 animated particles with blue and pink colors. You can customize the particle count, colors, and animations by modifying the particle creation logic.

### Overall Theme

The main color scheme uses CSS variables like `--white-color`, `--black-color`, and `--light-black-color` defined in the SCSS files.

## Implementation in Desktop App

Since you're creating a desktop version using Electron, you have several options:

1. **Modify the existing web assets** before packaging them into your Electron app
2. **Override styles** in your Electron main process by injecting custom CSS
3. **Create a completely custom interface** while keeping the same Socket.IO communication with Leon's backend

The client initialization in [4](#2-3) shows how the interface connects to Leon's server, so you'd maintain this connection pattern while customizing the visual presentation.

Would you like me to elaborate on any specific design aspect you want to change?

Wiki pages you might want to explore:

- [HTTP & Socket Servers (leon-ai/leon)](/wiki/leon-ai/leon#2.2)

- [Voice Processing (leon-ai/leon)](/wiki/leon-ai/leon#4)

- [Client Application (leon-ai/leon)](/wiki/leon-ai/leon#5)

Based on your request for step-by-step instructions to convert Leon into a desktop application, here's a comprehensive guide using Leon's existing infrastructure:

## Step-by-Step Desktop Application Creation

### Step 1: Set Up Development Environment

First, ensure you have the required dependencies installed:

- Node.js ≥ 22.13.1 and npm ≥ 10.9.2

- Python 3.11.9 with Pipenv 2024.0.1 [1](#1-0)

### Step 2: Build All Required Binaries

Leon already has build scripts for creating standalone binaries. Build all components:

~~~bash
# Build the server
npm run build:server

# Build all binaries
npm run build:nodejs-bridge
npm run build:python-bridge
npm run build:tcp-server
``` [2](#1-1)
~~~

The build system uses [3](#1-2) to create platform-specific binaries for each OS.

### Step 3: Create Electron Wrapper

Create a new Electron application to wrap Leon's web interface:

1. **Initialize Electron project:**
```bash
mkdir leon-desktop
cd leon-desktop
npm init -y
npm install electron --save-dev
```

2. **Create main Electron process** (`main.js`):
```javascript
const { app, BrowserWindow, spawn } = require('electron')
const path = require('path')

let leonServerProcess = null
let mainWindow = null

function createWindow() {
  mainWindow = new BrowserWindow({
    width: 1200,
    height: 800,
    webPreferences: {
      nodeIntegration: false,
      contextIsolation: true
    }
  })

  // Load Leon's web interface
  mainWindow.loadURL('http://localhost:1337')
}

function startLeonServer() {
  // Start Leon server as background process
  leonServerProcess = spawn('node', ['server/dist/index.js'], {
    env: { ...process.env, LEON_NODE_ENV: 'production' }
  })
}

app.whenReady().then(() => {
  startLeonServer()
  setTimeout(createWindow, 3000) // Wait for server to start
})

app.on('before-quit', () => {
  if (leonServerProcess) {
    leonServerProcess.kill()
  }
})
```

### Step 4: Bundle Leon Components

Create a packaging structure that includes all Leon components:

```
leon-desktop/
├── resources/
│   ├── server/        # Built Leon server
│   ├── app/           # Built web client
│   ├── binaries/      # All binary components
│   │   ├── leon-tcp-server
│   │   ├── leon-python-bridge
│   │   └── leon-nodejs-bridge
│   └── core/          # Models and data
└── main.js
```

### Step 5: Handle Binary Execution

Leon's binaries already detect when running from a packaged state. The TCP server uses [4](#1-3) to adjust paths accordingly.

Update your Electron main process to start the TCP server:

```javascript
function startTCPServer() {
  const tcpServerPath = path.join(__dirname, 'resources/binaries/leon-tcp-server')
  const tcpServerProcess = spawn(tcpServerPath, ['en'])
  return tcpServerProcess
}
```

### Step 6: Configure Server for Desktop Mode

Modify Leon's server startup to work in desktop mode by setting appropriate environment variables:

```javascript
function startLeonServer() {
  leonServerProcess = spawn('node', ['resources/server/index.js'], {
    env: {
      ...process.env,
      LEON_NODE_ENV: 'production',
      LEON_HOST: 'localhost',
      LEON_PORT: '1337'
    }
  })
}
```

The server initialization process [5](#1-4) already handles spawning the TCP server automatically.

### Step 7: Package the Desktop Application

Use Electron Builder to create distributable packages:

```bash
```

```
npm install electron-builder --save-dev
```

Add to `package.json`:

```json
{
  "build": {
    "appId": "ai.leon.desktop",
    "productName": "Leon AI",
    "directories": {
      "output": "dist"
    },
    "files": [
      "main.js",
      "resources/**/*"
    ],
    "extraResources": [
      "resources/**/*"
    ]
  }
}
```

### Step 8: Build and Test

```bash
# Build for current platform
npx electron-builder
```

```
# Build for all platforms
npx electron-builder --mac --win --linux
```

## Key Considerations

1. **Binary Compatibility**: Leon's build system [6](#1-5) already handles platform-specific requirements like `patchelf` on Linux.

2. **Path Resolution**: The existing binary detection logic [7](#1-6) will automatically adjust paths when running from the packaged desktop app.

3. **Process Management**: The server startup code [8](#1-7) already includes process cleanup to prevent multiple instances.

## Notes

This approach leverages Leon's existing binary building infrastructure and modular architecture. The web client [9](#1-8) already uses Socket.IO for real-time communication, which will work seamlessly within the Electron wrapper. The binary setup scripts [10](#1-9) can be adapted to handle the desktop packaging workflow.

Wiki pages you might want to explore:

- [Overview (leon-ai/leon)](/wiki/leon-ai/leon#1)

Based on your request to create a settings screen for the Leon desktop application, I'll provide step-by-step instructions leveraging Leon's existing settings infrastructure.

## Step-by-Step Settings Screen Creation

### Step 1: Create Settings Window in Electron

Add a settings window function to your main Electron process:

```javascript
const { Menu } = require('electron')

function createSettingsWindow() {
  const settingsWindow = new BrowserWindow({
    width: 900,
    height: 700,
    parent: mainWindow,
    modal: true,
    resizable: false,
    webPreferences: {
      nodeIntegration: true,
      contextIsolation: false
    }
  })

  settingsWindow.loadFile('settings.html')
  settingsWindow.setMenuBarVisibility(false)

  return settingsWindow
}

// Add menu item to open settings
const template = [
 {
   label: 'Leon',
   submenu: [
    {
      label: 'Settings...',
      accelerator: 'CmdOrCtrl+,',
      click: () => createSettingsWindow()
    }
   ]
 }
]

Menu.setApplicationMenu(Menu.buildFromTemplate(template))
```

### Step 2: Create Settings HTML Interface

Create `settings.html` with organized sections based on Leon's configuration system [1](#4-0) :

```html
<!DOCTYPE html>
<html>
<head>
   <title>Leon Settings</title>
   <link rel="stylesheet" href="settings.css">
</head>
<body>
```

```
<div class="settings-container">
  <nav class="settings-nav">
    <ul>
      <li><a href="#general" class="nav-item active">General</a></li>
      <li><a href="#voice" class="nav-item">Voice</a></li>
      <li><a href="#llm" class="nav-item">LLM</a></li>
      <li><a href="#server" class="nav-item">Server</a></li>
      <li><a href="#skills" class="nav-item">Skills</a></li>
    </ul>
  </nav>

  <main class="settings-content">
    <section id="general" class="settings-section active">
      <h2>General Settings</h2>
      <div class="setting-group">
        <label for="language">Language</label>
        <select id="language">
          <option value="en-US">English (US)</option>
          <option value="fr-FR">Français</option>
        </select>
      </div>
      <div class="setting-group">
        <label for="timezone">Time Zone</label>
        <input type="text" id="timezone" placeholder="Auto-detect">
      </div>
      <div class="setting-group">
        <label>
          <input type="checkbox" id="telemetry">
          Enable telemetry
        </label>
      </div>
    </section>

    <section id="voice" class="settings-section">
      <h2>Voice Settings</h2>
      <div class="setting-group">
        <label>
          <input type="checkbox" id="stt">
          Enable Speech-to-Text
        </label>
      </div>
      <div class="setting-group">
        <label>
          <input type="checkbox" id="tts">
          Enable Text-to-Speech
        </label>
      </div>
      <div class="setting-group">
        <label>
          <input type="checkbox" id="wake-word">
          Enable Wake Word ("Hey Leon")
        </label>
      </div>
    </section>

    <!-- Add other sections similarly -->
```

```
    </main>
  </div>

  <div class="settings-footer">
    <button id="cancel">Cancel</button>
    <button id="save" class="primary">Save</button>
  </div>

  <script src="settings.js"></script>
</body>
</html>
```

### Step 3: Implement Settings Logic

Create `settings.js` to handle the settings interface:

```javascript
const fs = require('fs')
const path = require('path')

class SettingsManager {
  constructor() {
    this.envPath = path.join(__dirname, '../.env')
    this.currentSettings = {}
    this.loadSettings()
  }
```

```
 async loadSettings() {
  try {
   // Load current environment settings
   const envContent = fs.readFileSync(this.envPath, 'utf8')
   const envLines = envContent.split('\n')

   envLines.forEach(line => {
    if (line.includes('=') && !line.startsWith('#')) {
     const [key, value] = line.split('=')
     this.currentSettings[key] = value
    }
   })

   // Populate UI with current values
   this.populateUI()
  } catch (error) {
   console.error('Error loading settings:', error)
  }
 }

 populateUI() {
  // General settings
  document.getElementById('language').value = this.currentSettings.LEON_LANG || 'en-
US'
  document.getElementById('timezone').value = this.currentSettings.LEON_TIME_ZONE
|| ''
  document.getElementById('telemetry').checked =
this.currentSettings.LEON_TELEMETRY === 'true'
```

```
    // Voice settings
    document.getElementById('stt').checked = this.currentSettings.LEON_STT === 'true'
    document.getElementById('tts').checked = this.currentSettings.LEON_TTS === 'true'
    document.getElementById('wake-word').checked =
this.currentSettings.LEON_WAKE_WORD === 'true'
  }

  async saveSettings() {
    // Collect values from UI
    const newSettings = {
      LEON_LANG: document.getElementById('language').value,
      LEON_TIME_ZONE: document.getElementById('timezone').value,
      LEON_TELEMETRY: document.getElementById('telemetry').checked.toString(),
      LEON_STT: document.getElementById('stt').checked.toString(),
      LEON_TTS: document.getElementById('tts').checked.toString(),
      LEON_WAKE_WORD: document.getElementById('wake-word').checked.toString()
    }

    // Update .env file
    let envContent = fs.readFileSync(this.envPath, 'utf8')

    Object.entries(newSettings).forEach(([key, value]) => {
      const regex = new RegExp(`^${key}=.*$`, 'm')
      if (envContent.match(regex)) {
        envContent = envContent.replace(regex, `${key}=${value}`)
      } else {
        envContent += `\n${key}=${value}`
      }
    })

    fs.writeFileSync(this.envPath, envContent)

    // Show restart notification
    this.showRestartNotification()
  }

  showRestartNotification() {
    const notification = document.createElement('div')
    notification.className = 'notification'
    notification.textContent = 'Settings saved. Restart Leon to apply changes.'
    document.body.appendChild(notification)

    setTimeout(() => notification.remove(), 3000)
  }
}

// Initialize settings manager
const settingsManager = new SettingsManager()

// Navigation handling
document.querySelectorAll('.nav-item').forEach(item => {
  item.addEventListener('click', (e) => {
    e.preventDefault()
    const target = e.target.getAttribute('href').substring(1)

    // Update navigation
    document.querySelectorAll('.nav-item').forEach(nav => nav.classList.remove('active'))
    document.querySelectorAll('.settings-section').forEach(section => section.classList.remove('active'))

    e.target.classList.add('active')
    document.getElementById(target).classList.add('active')
  })
})

// Button handlers
document.getElementById('save').addEventListener('click', () => {
  settingsManager.saveSettings()
})

document.getElementById('cancel').addEventListener('click', () => {
  window.close()
})
```

### Step 4: Add Settings API Integration

Create an API endpoint to fetch current Leon configuration, extending the existing info endpoint [2](#4-1) :

```javascript
// In your Electron main process, add IPC communication
const { ipcMain } = require('electron')

ipcMain.handle('get-leon-config', async () => {
  try {
    const response = await fetch('http://localhost:1337/api/v1/info')
    return await response.json()
  } catch (error) {
    console.error('Failed to fetch Leon config:', error)
    return null
  }
})

ipcMain.handle('restart-leon', async () => {
  // Restart Leon server process
  if (leonServerProcess) {
    leonServerProcess.kill()
    setTimeout(() => startLeonServer(), 2000)
  }
})
```

### Step 5: Handle Skill-Specific Settings

For individual skill settings, leverage Leon's existing skill settings system [3](#4-2) :

```javascript
async loadSkillSettings() {
  const skillsPath = path.join(__dirname, '../skills')
  const domains = fs.readdirSync(skillsPath)

  domains.forEach(domain => {
    const domainPath = path.join(skillsPath, domain)
    const skills = fs.readdirSync(domainPath)

    skills.forEach(skill => {
      const settingsPath = path.join(domainPath, skill, 'src', 'settings.json')
      const settingsSamplePath = path.join(domainPath, skill, 'src', 'settings.sample.json')

      if (fs.existsSync(settingsPath) && fs.existsSync(settingsSamplePath)) {
        // Load and display skill-specific settings
        this.loadIndividualSkillSettings(domain, skill, settingsPath, settingsSamplePath)
      }
    })
  })
}
```

### Step 6: Add CSS Styling

Create `settings.css` for a clean interface:

```css
.settings-container {
  display: flex;
  height: 100vh;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
}

.settings-nav {
  width: 200px;

  background: #f5f5f5;

  border-right: 1px solid #ddd;

}


.settings-nav ul {

  list-style: none;

  padding: 20px 0;

  margin: 0;

}


.settings-nav .nav-item {

  display: block;

  padding: 12px 20px;

  text-decoration: none;

  color: #333;

  border-left: 3px solid transparent;

}


.settings-nav .nav-item.active {

  background: white;

  border-left-color: #007acc;

  color: #007acc;

}


.settings-content {

  flex: 1;

  padding: 30px;
  overflow-y: auto;
}

.settings-section {
  display: none;
}

.settings-section.active {
  display: block;
}

.setting-group {
  margin-bottom: 20px;
}

.setting-group label {
  display: block;
  margin-bottom: 5px;
  font-weight: 500;
}

.settings-footer {
  position: fixed;
  bottom: 0;
  right: 0;
  left: 200px;
  padding: 15px 30px;
  background: white;
```

```css
  border-top: 1px solid #ddd;
  text-align: right;
}

.settings-footer button {
  margin-left: 10px;
  padding: 8px 16px;
  border: 1px solid #ddd;
  background: white;
  cursor: pointer;
}

.settings-footer button.primary {
  background: #007acc;
  color: white;
  border-color: #007acc;
}
```

## Notes

This implementation leverages Leon's existing configuration system [4](#4-3) and skill settings infrastructure [5](#4-4) . The settings screen reads from and writes to the same `.env` file and skill configuration files that Leon uses, ensuring consistency with the existing system.

Some settings changes (like voice providers or LLM configuration) will require restarting Leon to take effect, while others (like UI preferences) can be applied immediately.

Wiki pages you might want to explore:

- [HTTP & Socket Servers (leon-ai/leon)](/wiki/leon-ai/leon#2.2)

- [Voice Processing (leon-ai/leon)](/wiki/leon-ai/leon#4)

Based on your request to create a chat screen for the Leon desktop application, I can see you're asking about implementing the chat interface component that would handle the conversational UI between users and Leon AI.

## Current Chat Implementation

Leon already has a comprehensive chat system implemented in the web client. The main chat functionality is handled by the `Chatbot` class [1](#5-0) , which manages the conversation interface including message bubbles, typing indicators, and widget rendering.

## Key Chat Components

### 1. Chat Feed Container

The chat interface is structured around a main feed container [2](#5-1) that displays the conversation history with a placeholder message for new users.

### 2. Message Bubble System

The chat uses a bubble-based design where user messages and Leon's responses are displayed as styled bubbles [3](#5-2) . User messages appear with a blue background on the right, while Leon's responses use a dark background on the left.

### 3. Real-time Communication

The chat integrates with Leon's Socket.IO system for real-time messaging [4](#5-3) . The client establishes a WebSocket connection to handle live conversation flow.

## Implementation for Desktop App

### Step 1: Create Chat Window Component

For your desktop application, you can leverage the existing chat structure by creating a dedicated chat window:

```javascript
function createChatWindow() {
  const chatWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true,
      contextIsolation: false
    }
  })

  chatWindow.loadFile('chat.html')
  return chatWindow
}
```

### Step 2: Adapt Existing Chat Logic

The `Chatbot` class already handles core chat functionality including:

- **Message Creation**: [5](#5-4) - Creates message bubbles for both user and Leon messages

- **Typing Indicators**: [6](#5-5) - Shows when Leon is processing responses

- **History Management**: [7](#5-6) - Loads previous conversation history from localStorage

### Step 3: Socket Integration

The chat system relies on Socket.IO events for real-time communication [8](#5-7) . Your desktop chat would connect to the same socket server that handles utterance processing and response delivery.

### Step 4: Message Flow

The complete message flow works as follows:

1. User types message in chat input

2. Client sends utterance via socket [9](#5-8)

3. Server processes through NLU and Brain systems

4. Leon responds with answer event [10](#5-9)

5. Chat interface displays the response bubble

### Step 5: Advanced Features

The existing chat system also supports:

- **LLM Token Streaming**: [11](#5-10) - Real-time token display for LLM responses

- **Widget Rendering**: Interactive UI components within chat bubbles

- **Suggestions**: [12](#5-11) - Clickable suggestion buttons

- **Voice Integration**: Chat works seamlessly with voice input/output

## Notes

The existing chat implementation in Leon is quite comprehensive and well-architected. For your desktop application, you can essentially reuse most of the existing `Chatbot` and `Client` classes with minimal modifications. The main changes would be adapting the HTML structure and CSS styling to fit your desktop application's design requirements while maintaining the same Socket.IO communication patterns with Leon's backend services.

The chat system is tightly integrated with Leon's core Brain and NLU systems, so maintaining this architecture ensures full compatibility with all of Leon's AI capabilities including skills, widgets, and voice processing.

Wiki pages you might want to explore:

- [HTTP & Socket Servers (leon-ai/leon)](/wiki/leon-ai/leon#2.2)

- [Client Application (leon-ai/leon)](/wiki/leon-ai/leon#5)

Based on your request to add MCP (Model Context Protocol) server ability to Leon AI, I can see you want to integrate a standardized protocol for connecting AI models with external resources and tools.

## Current Leon Architecture Context

Leon already has a sophisticated server architecture that could support MCP integration:

### Existing Server Infrastructure

Leon operates with multiple server components that handle different aspects of AI interaction:

- **HTTP Server**: [1](#7-0) - Provides REST API endpoints and serves the web application

- **Socket Server**: [2](#7-1) - Handles real-time bidirectional communication

- **TCP Server**: [3](#7-2) - Manages voice processing and Python-based services

### LLM Integration Points

Leon already has LLM provider infrastructure that could be extended for MCP: [4](#7-3) shows how external LLM services are integrated.

## MCP Server Implementation Approach

### 1. Create MCP Server Component

You could add a new MCP server component alongside Leon's existing servers. The core server initialization in [5](#7-4) shows how Leon bootstraps multiple server components.

### 2. Integrate with Skill System

Leon's skill system [6](#7-5) could be extended to support MCP-enabled tools and resources, allowing skills to access external data sources through the standardized protocol.

### 3. TCP Client Extension

The existing TCP client architecture [7](#7-6) could be adapted to communicate with MCP servers, similar to how it currently handles Python TCP server communication.

## Implementation Steps

1. **Add MCP Server Module**: Create a new server component in the core system alongside the existing HTTP and Socket servers

2. **Extend Bridge System**: Modify the Node.js and Python bridges to support MCP protocol communication

3. **Update Skill Framework**: Allow skills to declare MCP resource dependencies and access them through the protocol

4. **Integrate with NLU**: Extend the entity recognition system to understand MCP resource queries

The modular architecture shown in [8](#7-7) demonstrates how Leon registers and manages multiple core components, making it well-suited for adding MCP server capabilities.

This would enable Leon to access external tools, databases, and services through the standardized MCP protocol while maintaining its existing conversational AI capabilities.

Wiki pages you might want to explore:

- [Overview (leon-ai/leon)](/wiki/leon-ai/leon#1)

- [Core System (leon-ai/leon)](/wiki/leon-ai/leon#2)

- [HTTP & Socket Servers (leon-ai/leon)](/wiki/leon-ai/leon#2.2)