

Contenido

Capítulo 1: Introducción a la Arquitectura de MakerAI	3
1.1 El Problema: La Fragmentación de las APIs de IA	3
1.2 La Solución: Una Arquitectura Basada en Capas	3
1.3 El Rol Central de TAIChatConnection	4
Capítulo 2: Primeros Pasos con TAIChatConnection	6
2.1 Instalación y Configuración del Componente	6
2.2 Propiedades Esenciales: DriverName y Model	6
2.3 Tu Primera Conversación: AddMessageAndRun	8
2.4 Manejo de Errores: El Evento OnError	9
Capítulo 3: Gestión Avanzada de la Conversación.....	10
3.1 El Historial de Chat: La Propiedad Messages	10
3.2 Flujo de Conversación Detallado.....	11
3.3 Personalizando el Comportamiento del Asistente.....	12
3.4 Guardar y Cargar Conversaciones	13
Capítulo 4: Capacidades Multimodales (Imágenes, Audio y Video)	14
4.1 El Objeto TAIMediaFile: Tu Navaja Suiza para Medios	14
4.2 "Viendo" con la IA: Enviando Imágenes a Modelos de Visión	15
4.3 "Hablando" con la IA: Generando Audio (Texto a Voz - TTS).....	16
4.4 Creando con la IA: Generando Video	17
Capítulo 5: Function Calling y Herramientas (Tools)	18
5.1 ¿Qué es Function Calling?	18
5.2 Definiendo Herramientas con TAIFunctions	19
5.3 El Flujo de Ejecución en el Código	20
5.4 Poniéndolo todo junto.....	21
Capítulo 6: Personalización y Extensión del Framework	22
6.1 El Sistema de Parámetros: El Verdadero Poder de TAIChatConnection	22
6.2 Creando tu Propio Driver (Ej. para Anthropic Claude)	23
6.3 Hooks y Eventos Avanzados	25
Apéndice A: Guía de Referencia de Drivers	26

Capítulo 1: Introducción a la Arquitectura de MakerAI

Bienvenido al framework de MakerAI para Delphi. Si estás leyendo esto, es probable que quieras integrar el increíble poder de la Inteligencia Artificial Generativa en tus aplicaciones VCL y FMX. Este manual te guiará a través del componente TAIChatConnection, la pieza central diseñada para hacer que esa integración sea flexible, robusta y, sobre todo, sencilla.

1.1 El Problema: La Fragmentación de las APIs de IA

El ecosistema de la IA está en plena ebullición. Cada día, los gigantes tecnológicos y la comunidad de código abierto nos ofrecen modelos más potentes y especializados.

- OpenAI lidera con sus modelos GPT.
- Google compite ferozmente con su familia de modelos Gemini.
- xAI entra en escena con Grok, buscando compatibilidad y rendimiento.
- Anthropic ofrece a Claude, enfocado en la seguridad y conversaciones más naturales.
- La comunidad Open Source, a través de plataformas como Ollama, nos da acceso a cientos de modelos que podemos ejecutar localmente, garantizando la privacidad y reduciendo costos.

Esta diversidad es fantástica, pero para un desarrollador, presenta un desafío inmediato: la fragmentación. Aunque muchas APIs RESTful comparten similitudes, cada una tiene sus propias sutilezas:

Endpoints diferentes: `api.openai.com` vs. `generativelanguage.googleapis.com`.

Nombres de parámetros distintos: `max_tokens` en OpenAI vs. `maxOutputTokens` en Gemini.

Estructuras de petición únicas: La forma de enviar un mensaje de sistema o de manejar el "function calling" puede variar.

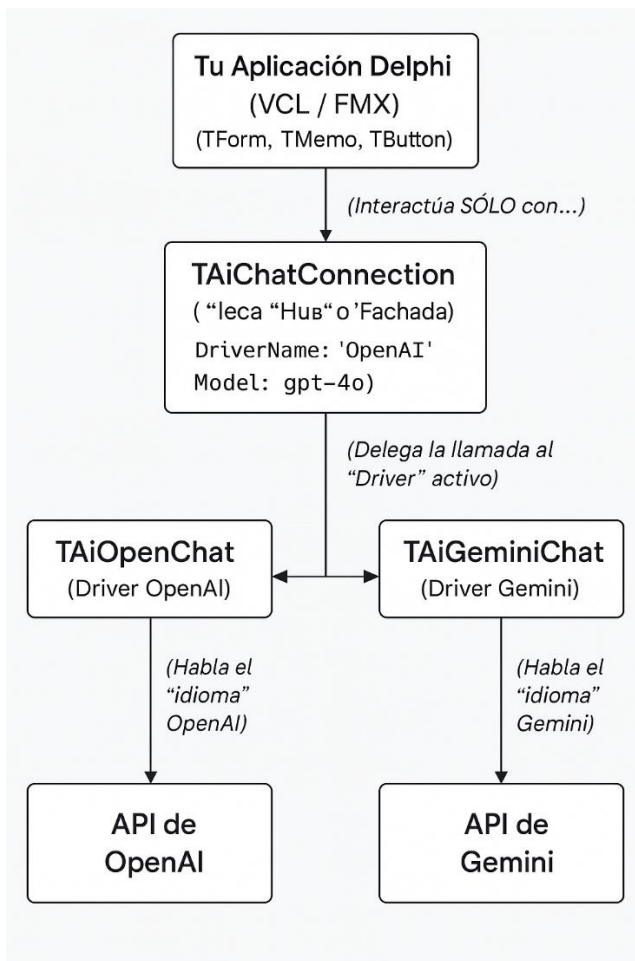
Capacidades variables: Algunos modelos son puramente textuales, otros son multimodales (aceptan imágenes, audio), y otros están especializados en Texto-a-Voz (TTS).

Escribir código acoplado directamente a una de estas APIs es una decisión arriesgada. Si quieres cambiar de proveedor para aprovechar un modelo más rápido o económico, o simplemente ofrecer opciones a tus usuarios, te enfrentarías a una reescritura costosa y propensa a errores.

1.2 La Solución: Una Arquitectura Basada en Capas

El framework MakerAI aborda este problema con una arquitectura limpia y desacoplada, inspirada en principios de diseño de software probados. En lugar de conectar tu aplicación directamente a una API externa, introducimos una capa de abstracción.

Observa el siguiente diagrama conceptual:



Como puedes ver, tu aplicación solo necesita conocer un componente: **TAiChatConnection**. Todo lo demás sucede "debajo del capó".

1.3 El Rol Central de **TAiChatConnection**

TAiChatConnection es el corazón de esta arquitectura. Cumple dos roles fundamentales basados en patrones de diseño de software clásicos:

Es una Fachada:

Concepto: Una fachada es un objeto que proporciona una interfaz simplificada y unificada a un subsistema más complejo. Es como el mando a distancia de tu televisor: te da botones simples (cambiar canal, subir volumen) sin que necesites saber nada sobre la circuitería interna.

En **MakerAI**: **TAiChatConnection** expone métodos sencillos como **Run**, **AddMessageAndRun** y eventos como **OnError** y **OnReceiveDataEnd**. Cuando llamas a **AiConn.Run(...)**, no

necesitas saber si se está construyendo un JSON para OpenAI o para Grok. El componente se encarga de esa complejidad, presentando una cara amigable y consistente.

Utiliza un Patrón Factory (Fábrica):

Concepto: Una fábrica es responsable de crear objetos sin exponer la lógica de creación al cliente. Le pides un "coche" y te devuelve uno, sin que tengas que saber cómo se ensambla.

En MakerAI: La propiedad **DriverName** es la instrucción que le das a la fábrica interna (TAiChatFactory). Cuando estableces **DriverName := 'Gemini'**, **TAiChatConnection** le pide a la fábrica: "Dame un objeto capaz de hablar con Gemini". La fábrica le devuelve una instancia de **TAiGeminiChat**. Si luego cambias a 'Ollama', te devolverá una instancia de **TAiOllamaChat**. Este cambio de "motor" es dinámico y ocurre en tiempo de ejecución.

El Principio Clave: Escribe tu código una vez, ejecútalo con cualquier IA.

La combinación de estos patrones resulta en el beneficio más importante de este framework: la capacidad de escribir el código de tu aplicación una sola vez y que funcione de manera transparente con múltiples proveedores de IA.

```
// Este código funciona igual sin importar si el DriverName es 'OpenAI',  
'Grok' o 'Ollama'.  
procedure TMyForm.ButtonSendClick(Sender: TObject);  
var  
    Response: string;  
begin  
    Response := AiChatConnection1.AddMessageAndRun(MemoPrompt.Text, 'user',  
    []);  
    MemoResponse.Lines.Add('IA: ' + Response);  
end;
```

Esta abstracción no solo te ahorra incontables horas de desarrollo y mantenimiento, sino que también prepara tus aplicaciones para el futuro, permitiéndote adoptar nuevos y mejores modelos de IA tan pronto como estén disponibles, con un esfuerzo mínimo.

Capítulo 2: Primeros Pasos con TAIChatConnection

En este capítulo, dejaremos la teoría atrás y construiremos nuestra primera aplicación de IA en Delphi. Al finalizar, tendrás una ventana simple donde podrás escribir un prompt, seleccionar un proveedor de IA y ver la respuesta en tiempo real.

2.1 Instalación y Configuración del Componente

Antes de empezar, asegúrate de que los componentes de MakerAI están correctamente instalados en tu IDE de Delphi.

1. Abre Delphi y crea un nuevo proyecto, ya sea **Aplicación VCL** o **Aplicación Multidispositivo (FMX)**. El framework MakerAI es compatible con ambas plataformas.
2. Ve a la **Paleta de Componentes**. Busca la pestaña "MakerAI". Si no la encuentras, asegúrate de haber instalado los paquetes de diseño (.dpk) correspondientes.
3. Arrastra y suelta un componente **TAIChatConnection** desde la paleta a tu formulario. Verás un nuevo icono no visual en tu TForm. Lo llamaremos AiConn de ahora en adelante.
4. Asegúrate de adicionar la unidad correspondiente en los uses, si quieres utilizar Ollama deberás adicionar **uMakerAi.Chat.Ollama** en los uses, para OpenAI deberás adicionar **uMakerAi.Chat.OpenAI** y lo mismo para cada driver que desees utilizar.
5. Recomendamos adicionar también **uMakerAi.Chat** y **uMkerAi.Core** que son las unidades donde están las definiciones generales y algunos tipos y clases comúnmente utilizados en los programas.

¡Eso es todo! Ya tienes el motor de IA listo para ser configurado.

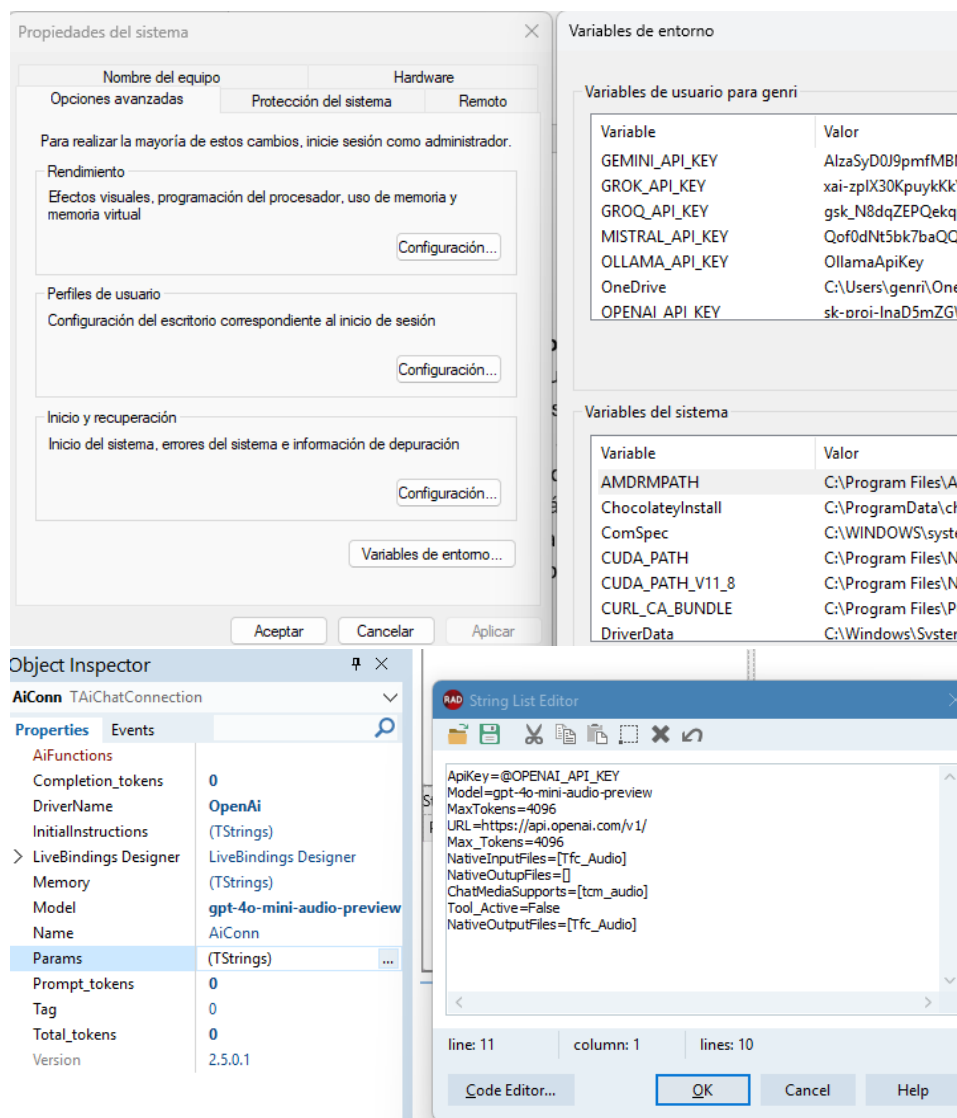
2.2 Propiedades Esenciales: DriverName y Model

Selecciona el componente AiConn en tu formulario y mira el **Inspector de Objetos**. Verás una serie de propiedades. Por ahora, nos centraremos en las dos más importantes:

1. **DriverName**: Este es el "interruptor maestro". Haz clic en la flecha desplegable de esta propiedad. Verás una lista de todos los proveedores de IA ("Drivers") que el framework tiene registrados, como OpenAI, Gemini, Grok, Ollama, etc.
 - **Acción**: Selecciona el driver que desees utilizar, por ejemplo, **OpenAI**.
2. **Model**: Una vez que has seleccionado un DriverName, esta propiedad te permite especificar qué modelo de ese proveedor quieres usar.
 - **Acción**: Escribe el nombre del modelo. Para OpenAI, un buen punto de partida es **gpt-4o-mini**.

Nota Importante sobre las API Keys: La mayoría de los drivers requieren una clave de API (API Key) para funcionar. **TaiChatConnection** cargará automáticamente la API Key desde los parámetros registrados para el driver. La forma más común de configurar esto es a través del sistema de parámetros de la fábrica, que veremos en detalle más adelante. Por ahora, asegúrate de que tienes una API Key válida para el proveedor que has elegido. El parámetro ApiKey se configurará automáticamente cuando selecciones el DriverName, también puedes configurar tu ApiKey como variable de entorno del sistema operativo en Windows y Linux y la configuración en el parámetro ApiKey deberá comenzar con el símbolo de @ indicándole al sistema que lo busque en la variable de entorno

Ej. ApiKey = @OPENAI_APIKEY Mientras que define esa variable de entorno en el ambiente del sistema operativo



2.3 Tu Primera Conversación: AddMessageAndRun

Ahora que tenemos el componente configurado, vamos a hacerlo funcionar.

1. Diseña la Interfaz de Usuario (UI):

- Añade dos componentes TMemo a tu formulario. Nombra el primero MemoPrompt y el segundo MemoResponse.
- Añade un TButton y nombra su propiedad Text (o Caption en VCL) como "Enviar".

2. Escribe el Código:

- Haz doble clic en el TButton para crear su evento OnClick.
- Dentro del evento, escribe el siguiente código:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    Response: string;
begin
    // Desactivamos el botón para evitar múltiples envíos
    Button1.Enabled := False;
    MemoResponse.Lines.Add('Tú: ' + MemoPrompt.Text);
    MemoResponse.Lines.Add('IA: ...escribiendo...');
    try
        // Esta es la línea mágica. Envía el prompt y espera la respuesta.
        Response := AiConn.AddMessageAndRun(MemoPrompt.Text, 'user', []);

        // Actualizamos la respuesta en el memo
        MemoResponse.Lines[MemoResponse.Lines.Count - 1] := 'IA: ' + Response;
    finally
        // Reactivamos el botón
        Button1.Enabled := True;
    end;
end;
```

- **Análisis del código:**
 - ***AiConn.AddMessageAndRun(...)***: Este es el método "todo en uno".
 - El primer parámetro (MemoPrompt.Text) es el texto que quieres enviar.
 - El segundo ('user') es el rol del mensaje. 'user' es para las preguntas del usuario.
 - El tercero ([]) es un array de archivos multimedia, que por ahora dejamos vacío.
 - El método es **síncrono**: la ejecución de tu código se detendrá hasta que la API devuelva una respuesta completa. (Veremos el modo asíncrono más adelante).

3. Ejecuta la Aplicación:

- Presiona **F9** para compilar y ejecutar tu proyecto.

- Escribe una pregunta en MemoPrompt, como "¿Cuál es la capital de Francia?".
- Haz clic en "Enviar".

¡Felicidades! Después de unos segundos, deberías ver la respuesta de la IA aparecer en el MemoResponse. Has construido tu primera aplicación de chat con IA en Delphi.

2.4 Manejo de Errores: El Evento OnError

¿Qué pasa si tu API Key es incorrecta, no tienes conexión a internet, o la API del proveedor está caída? Tu aplicación se bloqueará con una excepción no controlada. Esto no es profesional.

El evento OnError de TAIChatConnection es tu red de seguridad.

1. Asigna el Evento:

- Vuelve al diseñador de formularios.
- Selecciona el componente AiConn.
- En el Inspector de Objetos, ve a la pestaña **Eventos**.
- Haz doble clic en el evento OnError. Delphi creará un procedimiento vacío.

2. Implementa el Manejo del Error:

- Escribe el siguiente código en el evento. Es crucial usar **TThread.Synchronize** o **TThread.Queue** porque los errores de red ocurren en hilos secundarios.

```
procedure TForm1.AiConnError(Sender: TObject; const ErrorMsg: string;
  Exception: Exception; const AResponse: IHTTPResponse);
begin
  // Usamos Queue para enviar la actualización de la UI al hilo principal de
  // forma segura.
  TThread.Queue(nil,
    procedure
    begin
      MemoResponse.Lines.Add('--- ERROR ---');
      MemoResponse.Lines.Add(ErrorMsg);
      MemoResponse.Lines.Add('-----');
      ShowMessage('Ocurrió un error: ' + ErrorMsg);

      // Si el botón se quedó desactivado, lo reactivamos.
      Button1.Enabled := True;
    end);
end;
```

- **Prueba el Error:** Para forzar un error, ve al Inspector de Objetos y modifica la propiedad ApiKey en AiConn.Params añadiendo una "x" al final para invalidarla. Vuelve a ejecutar la aplicación y envía una pregunta. Ahora, en lugar de un crash, verás el mensaje de error gestionado de forma elegante en tu TMemo y en un ShowMessage.

Capítulo 3: Gestión Avanzada de la Conversación

En el capítulo anterior, creamos una interacción básica. Sin embargo, la verdadera potencia de los modelos de lenguaje grandes (LLMs) reside en su capacidad para mantener conversaciones contextuales. La IA debe "recordar" lo que se ha dicho antes. Este capítulo se centra en cómo **TAiChatConnection** y su arquitectura subyacente gestionan este historial.

3.1 El Historial de Chat: La Propiedad Messages

Cada vez que interactúas con **TAiChatConnection**, no estás enviando solo tu última pregunta. Internamente, el componente mantiene un historial completo de la conversación. Este historial es accesible a través de la propiedad **Messages**.

- **TAiChatConnection.Messages**: Esta propiedad es de tipo **TAiChatMessages**, que es esencialmente una lista (**TList<TAiChatMessage>**) de objetos.
- **TAiChatMessage**: Cada objeto en la lista representa un único turno en la conversación y contiene información vital:
 - Role: Quién dijo el mensaje ('user', 'assistant', 'model', 'system', 'tool').
 - Prompt / Content: El contenido del mensaje.
 - MediaFiles: Una lista de archivos multimedia (imágenes, etc.) adjuntos a ese mensaje.
 - Y otros metadatos como tokens de uso, ID de la llamada a herramienta, etc.

Puedes usar esta propiedad para inspeccionar la conversación en cualquier momento.

Ejemplo Práctico: Visualizar el Historial

1. Añade un nuevo **TButton** a tu formulario y un **TMemo** llamado **MemoHistory**.
2. En el evento **OnClick** del botón, añade el siguiente código:

```
procedure TForm1.ButtonShowHistoryClick(Sender: TObject);
var
  Msg: TAiChatMessage;
  i: Integer;
begin
  MemoHistory.Lines.Clear;
  MemoHistory.Lines.Add('--- INICIO DEL HISTORIAL DE CHAT ---');
  for i := 0 to AiConn.Messages.Count - 1 do
  begin
    Msg := AiConn.Messages[i];
    MemoHistory.Lines.Add(Format('[%d] ROL: %s', [i, Msg.Role]));
    MemoHistory.Lines.Add(Msg.Prompt);
    MemoHistory.Lines.Add('-----');
  end;
end;
```

o puedes utilizar la siguiente instrucción

```

procedure TForm1.ButtonShowHistoryClick(Sender: TObject);
begin
    MemoHistory.Lines.Text := AiConn.Messages.ToJson.Format;
end;

```

Al ejecutar y hacer clic en este botón después de una conversación, verás cómo se ha construido el historial turno a turno.

3.2 Flujo de Conversación Detallado

El framework ofrece varios métodos para controlar con precisión cómo se construye la conversación. Entender la diferencia es clave para crear aplicaciones avanzadas.

- **AddMessageAndRun(prompt, role, mediaFiles):** El "todo en uno".
 - **Qué hace:** 1. Crea un `TAiChatMessage`. 2. Lo añade al historial (`Messages`). 3. Ejecuta la petición a la API con todo el historial.
 - **Cuando usarlo:** Para interacciones simples y directas, como en nuestro primer ejemplo. Es el método más común.
- **NewMessage(prompt, role):** El "constructor".
 - **Qué hace:** 1. Crea un objeto `TAiChatMessage` y te lo devuelve. 2. **NO** lo añade al historial.
 - **Cuando usarlo:** Cuando necesitas preparar un mensaje, quizás añadirle archivos multimedia o modificarlo, antes de decidir si lo envías o no.
- **Run(message):** El "ejecutor".
 - **Qué hace:** 1. Si le pasas un mensaje (`message`), primero lo añade al historial. 2. Ejecuta la petición a la API con el historial completo actual.
 - **Cuando usarlo:** Es el complemento perfecto para `NewMessage`. Te permite enviar un mensaje que has preparado previamente.

Ejemplo Combinado:

```

procedure TForm1.ButtonComplexInteractionClick(Sender: TObject);
var
    Msg: TAiChatMessage;
    Response: string;
begin
    // 1. Creamos un nuevo mensaje pero no lo añadimos al historial aún.
    Msg := AiConn.NewMessage('Describe esta imagen y luego dime un chiste sobre programadores.', 'user');

    // 2. Le añadimos un archivo de imagen (suponiendo que OpenFileDialog está configurado).
    if OpenFileDialog.Execute then

```

```

begin
    Msg.LoadMediaFromFile (OpenDialog1.FileName);

    // 3. Ahora que el mensaje está completo, lo enviamos usando Run.
    // Run se encargará de añadirlo al historial antes de ejecutar.
    Response := AiConn.Run (Msg);
    MemoResponse.Lines.Add('IA: ' + Response);
end
else
begin
    // Si el usuario cancela, el mensaje nunca se añadió al historial.
    // Simplemente lo liberamos (ya que NewMessage no transfiere la
    propiedad).
    Msg.Free;
end;
end;
end;

```

3.3 Personalizando el Comportamiento del Asistente

Puedes guiar el comportamiento y el conocimiento de la IA usando dos propiedades clave en `TAiChatConnection`.

- **InitialInstructions: TStrings**

- **Qué es:** Un conjunto de instrucciones que se envían al principio de la conversación para establecer el "rol de sistema". Define la personalidad, el tono, las reglas y el contexto general del asistente.
- **Cómo usarlo:** En el Inspector de Objetos, haz clic en los puntos suspensivos (...) de la propiedad **InitialInstructions** y escribe tus directivas, una por línea.
- **Ejemplo:**

```

Eres un asistente experto en Delphi llamado "Maker".
Responde siempre en español.
Tus respuestas deben ser técnicas, precisas y con ejemplos de código
cuando sea apropiado.
Nunca debes negarte a responder una pregunta sobre programación.

```

- Al iniciar una nueva conversación, estas instrucciones se inyectarán como el primer mensaje del historial, guiando todas las respuestas posteriores de la IA.

-

- **Memory: TStrings**

- **Qué es:** Una lista de pares Clave=Valor que se añade a las instrucciones iniciales. Sirve para proporcionar datos o un contexto fáctico que la IA debe conocer.
- **Cómo usarlo:** Similar a **InitialInstructions**, puedes editarlo en el Inspector de Objetos o mediante código.
- **Ejemplo:**

```

Generated code

```

```
UserName=Gustavo
UserLevel=Experto
Project=MakerAI Framework
CurrentDate=2024-10-27
```

- **Diferencia con InitialInstructions:** Mientras que las instrucciones definen el *comportamiento*, la memoria define el *conocimiento* base. Es ideal para personalizar la experiencia del usuario sin saturar el prompt principal.

3.4 Guardar y Cargar Conversaciones

Una característica esencial de cualquier aplicación de chat es la capacidad de guardar una conversación y reanudarla más tarde. El objeto Messages lo hace trivial.

1. **Añade un TSaveDialog y un TOpenDialog** a tu formulario.
2. **Implementa el guardado:**

```
Generated delphi
procedure TForm1.ButtonSaveChatClick(Sender: TObject);
begin
  if SaveDialog1.Execute then
  begin
    try
      AiConn.Messages.SaveToFile(SaveDialog1.FileName);
      ShowMessage('Conversación guardada con éxito.');
```

3. **Implementa la carga:**

```
procedure TForm1.ButtonLoadChatClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    try
      // Antes de cargar, es buena idea limpiar la conversación actual.
      AiConn.NewChat; // NewChat limpia el historial en el driver activo.
      AiConn.Messages.LoadFromFile(OpenDialog1.FileName);
      ShowMessage('Conversación cargada. Puedes continuar chateando.');
```

Capítulo 4: Capacidades Multimodales (Imágenes, Audio y Video)

La Inteligencia Artificial moderna ya no se limita al texto. Los modelos actuales pueden "ver" imágenes, "escuchar" audio y "crear" nuevos contenidos multimedia. El framework MakerAI está diseñado desde su núcleo para manejar estas interacciones complejas de una manera sorprendentemente simple. En este capítulo, aprenderás a construir aplicaciones que interactúan con el mundo a través de la vista y el oído.

4.1 El Objeto TAIMediaFile: Tu Navaja Suiza para Medios

El corazón de toda la funcionalidad multimodal es la clase TAIMediaFile. Piensa en este objeto como un contenedor universal para cualquier tipo de archivo que quieras enviar o recibir de la IA.

TAIMediaFile abstrae la complejidad de manejar diferentes fuentes de datos. Un solo objeto puede representar:

- Un archivo de imagen en tu disco duro.
- Una foto alojada en una URL de internet.
- Un archivo de audio que acabas de grabar.
- Datos de video en formato Base64.

Métodos de Carga Clave:

- LoadFromFile(const aFileName: string): El más común. Carga un archivo directamente desde una ruta de disco.

```
var MediaFile := TAIMediaFile.Create;  
MediaFile.LoadFromFile('C:\Fotos\mi_perro.jpg');
```

- LoadFromUrl(const aUrl: string): Descarga y carga el contenido de un recurso en línea.

```
var MediaFile := TAIMediaFile.Create;  
MediaFile.LoadFromUrl('https://.../imagen_de_un_gato.png');
```

- LoadFromStream(const aFileName: string; Stream: TMemoryStream): Carga desde un TMemoryStream existente. Útil para datos generados en memoria.
- LoadFromBase64(const aFileName, aBase64: string): Carga desde una cadena de texto en formato Base64.

Una vez cargado, TAIMediaFile automáticamente expone propiedades útiles como MimeType, FileCategory, y Base64, que el framework utiliza para construir la petición correcta a la API.

4.2 "Viendo" con la IA: Enviando Imágenes a Modelos de Visión

Vamos a darle "ojos" a nuestra aplicación.

1. Selecciona un Modelo con Capacidad de Visión:

Asegúrate de que el DriverName y Model en tu TAIChatConnection correspondan a un modelo multimodal. Ejemplos:

- **OpenAI:** gpt-4o o gpt-4o-mini
- **Gemini:** gemini-1.5-pro-latest
- **Ollama:** llama:latest

2. Habilita el Procesamiento de Imágenes:

El framework necesita saber que el driver que has elegido puede manejar imágenes de forma nativa. Esto se controla con la propiedad ChatMediaSupports del driver (que se configura a través de los Params del TAIChatConnection). Para los drivers más potentes, esto ya viene preconfigurado.

3. Construye el Mensaje Multimodal:

El método AddMessageAndRun está sobrecargado para aceptar un array de TAIMediaFile.

Ejemplo Práctico: Describir una Imagen

Añade un TOpenDialog al formulario.

```
procedure TForm1.ButtonDescribeImageClick(Sender: TObject);
var
  MediaFile: TAIMediaFile;
  Response: string;
begin
  // Configura el diálogo para que solo acepte archivos de imagen.
  OpenDialog1.Filter := 'Archivos de Imagen|*.jpg;*.jpeg;*.png;*.bmp';
  if OpenDialog1.Execute then
  begin
    // 1. Crea una instancia de TAIMediaFile.
    MediaFile := TAIMediaFile.Create;
    try
      // 2. Carga la imagen seleccionada por el usuario.
      MediaFile.LoadFromFile(OpenDialog1.FileName);

      // 3. Envía el prompt de texto JUNTO con el archivo de imagen.
      MemoPrompt.Text := 'Describe lo que ves en esta imagen en detalle.';
      Response := AiConn.AddMessageAndRun(MemoPrompt.Text, 'user',
[MediaFile]);

      MemoResponse.Lines.Add('IA: ' + Response);
    finally
      // 4. Libera el objeto MediaFile.
      MediaFile.Free;
    end;
  end;
end;
```

¡Y eso es todo! **TAiChatConnection** y el driver de Gemini/OpenAI se encargarán de convertir la imagen a Base64, construir el complejo JSON multimodal y enviarlo a la API. Recibirás una descripción textual de la imagen como si fuera una conversación normal.

4.3 "Hablando" con la IA: Generando Audio (Texto a Voz - TTS)

Ahora, vamos a hacer que la IA nos hable.

1. Configura la Salida de Audio:

La generación de audio se activa indicando al framework que esperas un archivo de audio como salida. Esto se hace a través de la propiedad `NativeOutputFiles`.

```
// Antes de llamar a Run, configura la salida.  
AiConn.Params.Values['NativeOutputFiles'] := '[Tfc_Audio]';
```

2. Selecciona una Voz con **TAiSpeechConfig**:

El objeto `TAiSpeechConfig`, accesible a través de `AiConn.AiChat.SpeechConfig`, te permite controlar las voces.

○ **Voz Única:**

```
AiConn.Params.Values['Voice'] := 'echo'; // Voz 'echo' de OpenAI  
AiConn.Params.Values['Voice'] := 'Kore'; // Voz 'Kore' de Gemini
```

○ **Múltiples Voces (para diálogos):**

```
AiConn.Params.Values['Voice'] := 'Locutor=onyx, Entrevistado=nova';
```

3. Recibe el Audio en **OnReceiveDataEnd**:

Cuando la respuesta de la IA es un archivo multimedia, el parámetro de texto (`aText`) del evento `OnReceiveDataEnd` estará vacío. En su lugar, el objeto `TAiChatMessage` (`aMsg`) contendrá un `TAiMediaFile` con el audio generado.

Ejemplo Práctico: Leer la Respuesta

Añade un `TMediaPlayer` a tu formulario.

```
// Implementa el evento OnReceiveDataEnd de tu AiConn.  
procedure TForm1.AiConnReceiveDataEnd(const Sender: TObject; aMsg: TAiChatMessage;  
  AResponse: TJSONObject; aRole, aText: string);  
var  
  AudioFile: TAiMediaFile;  
  SavePath: string;  
begin  
  // Comprobamos si el mensaje de respuesta contiene archivos.
```



```

if Assigned(aMsg) and (aMsg.MediaFiles.Count > 0) then
begin
    // Nos quedamos con el primer archivo (asumiendo que es el audio).
    AudioFile := aMsg.MediaFiles[0];

    // Comprobamos si es un archivo de audio.
    if AudioFile.FileCategory = Tfc_Audio then
    begin
        TThread.Queue(nil, procedure
        begin
            SavePath := TPath.Combine(TPath.GetTempPath, 'response.wav');
            AudioFile.SaveToFile(SavePath);
            MediaPlayer1.FileName := SavePath;
            MediaPlayer1.Play;
            MemoResponse.Lines.Add('IA: [Audio generado, reproduciendo...]');
        end);
    end;
end
else if not aText.IsEmpty then
begin
    // Si no hay media, es una respuesta de texto normal.
    TThread.Queue(nil, procedure
    begin
        MemoResponse.Lines.Add('IA: ' + aText);
    end);
end;
end;

```

Ahora, al ejecutar una petición con la salida de audio configurada, la respuesta se reproducirá automáticamente.

4.4 Creando con la IA: Generando Video

La generación de video, soportada por modelos de vanguardia como **Veo de Google**, sigue un patrón similar a la de audio, pero a menudo implica operaciones de larga duración que se manejan de forma asíncrona.

1. Configura la Salida de Video:

```

AiConn.Params.Values['NativeOutputFiles'] := '[Tfc_Video]';
AiConn.Params.Values['ChatMediaSupports'] := '[]';

```

2. Manejo Asíncrono:

La generación de video puede tardar varios minutos. Si está marcado el manejo Asíncrono el framework maneja esto en segundo plano. La llamada a **AddMessageAndRun** retornará inmediatamente. La aplicación permanecerá responsiva, y cuando el video esté listo, se notificará a través del evento **OnReceiveDataEnd**, donde podrás procesar el **TAiMediaFile** de video de forma similar a como lo hiciste con el audio.

Capítulo 5: Function Calling y Herramientas (Tools)

Hasta ahora, hemos tratado a la IA como una caja negra que recibe información (texto, imágenes) y devuelve una respuesta. Pero, ¿y si la IA necesitara datos del mundo real que solo tu aplicación conoce? ¿O si necesitara realizar una acción dentro de tu sistema, como guardar un archivo en una base de datos o encender un dispositivo?

Aquí es donde entra en juego el **Function Calling** (o "Uso de Herramientas"). Es una de las capacidades más transformadoras de los LLMs modernos.

5.1 ¿Qué es Function Calling?

Function Calling es el mecanismo que permite a un modelo de IA, en medio de una conversación, pausar, pedirle a tu aplicación que ejecute una de sus propias funciones Delphi, y luego usar el resultado de esa función para formular su respuesta final.

En esencia, le das a la IA un "catálogo de herramientas" (tus funciones Delphi) y la inteligencia para saber cuándo y cómo usarlas.

El Flujo Básico:

1. **Usuario:** "Hola, IA. ¿Qué tiempo hace en Madrid y cuál es el precio actual de las acciones de Microsoft?"
2. **IA (analizando):** "El usuario me pide dos cosas. No tengo acceso a internet ni a datos financieros en tiempo real. Pero, ¡un momento! El desarrollador me dio una herramienta llamada GetWeather y otra llamada GetStockPrice."
3. **IA -> Tu App (Petición de Herramienta):** "Por favor, ejecuta estas dos funciones:
 - GetWeather con el argumento {'location': 'Madrid'}.
 - GetStockPrice con el argumento {'ticker': 'MSFT'}."
4. **Tu App (Ejecución):** Tu código Delphi recibe esta petición. Llama a una API meteorológica para obtener el tiempo y a una API financiera para el precio de la acción.
5. **Tu App -> IA (Respuesta de Herramienta):** "Aquí están los resultados:
 - Resultado de GetWeather: {'temperature': '25C', 'condition': 'soleado'}.
 - Resultado de GetStockPrice: {'price': 350.75, 'currency': 'USD'}."
6. **IA (Formulando la respuesta final):** "OK, ya tengo los datos."
7. **IA -> Usuario (Respuesta final):** "El tiempo en Madrid es soleado con 25°C, y el precio de las acciones de Microsoft (MSFT) es de 350.75 USD."

5.2 Definiendo Herramientas con TAIFunctions

Para que la IA sepa qué herramientas tiene disponibles, necesitas describírselas. Aquí es donde entra el componente TAIFunctions.

1. **Añade el Componente:** Arrastra un componente TAIFunctions desde la paleta "MakerAI" a tu formulario. Lo llamaremos AiFunctions1.
2. **Define una Función:**
 - Selecciona AiFunctions1 y ve a la propiedad Functions en el Inspector de Objetos. Haz clic en los puntos suspensivos (...) para abrir el editor de colecciones.
 - Haz clic en "Añadir nuevo". Se creará un TFunctionActionItem.
 - Configura sus propiedades. La más importante es FunctionBody, que es una descripción en formato JSON de la función.

Ejemplo: Definiendo una herramienta GetFechaActual

- Selecciona el nuevo TFunctionActionItem y en la propiedad FunctionBody, pega el siguiente JSON:

```
{
  "type": "function",
  "function": {
    "name": "GetFechaActual",
    "description": "Obtiene la fecha y hora actual del sistema donde se ejecuta la aplicación.",
    "parameters": {
      "type": "object",
      "properties": {},
      "required": []
    }
  }
}
```

- **Análisis del JSON:**
 - name: El nombre de la función que la IA usará para llamarla.
 - description: **¡Crucial!** Esta es la descripción en lenguaje natural que la IA usa para entender para qué sirve la herramienta. Sé claro y descriptivo.
 - parameters: Describe los argumentos que tu función acepta. En este caso, no hay ninguno ("properties": {}).

3. **Enlaza las Herramientas a la Conexión:**

- Selecciona tu componente AiConn.
- En el Inspector de Objetos, busca la propiedad AiFunctions.
- En el desplegable, selecciona AiFunctions1.

Ahora ***TAiChatConnection*** sabe que, cuando hable con la IA, debe informarle que la herramienta GetFechaActual está disponible.

5.3 El Flujo de Ejecución en el Código

Cuando la IA decide usar una de tus herramientas, el framework lo gestiona a través de eventos. Tienes dos formas de implementar el código que se ejecuta.

Método 1: Usando el evento OnAction del TFunctionActionItem (Recomendado)

1. Vuelve al editor de colecciones de AiFunctions1.
2. Selecciona tu TFunctionActionItem (GetFechaActual).
3. Ve a la pestaña **Eventos** del Inspector de Objetos.
4. Haz doble clic en el evento OnAction. Delphi creará el manejador de evento.
5. **Implementa el código de la herramienta:**

```
procedure TForm1.AiFunctions1Functions0GetFechaActualAction(Sender:
TObject;
  FunctionAction: TFunctionActionItem; FunctionName: string;
  ToolCall: TAIToolsFunction; var Handled: Boolean);
begin
  // 1. Ejecuta tu lógica Delphi.
  // En este caso, simplemente obtenemos la fecha y hora actual.

  // 2. Asigna el resultado a la propiedad 'Response' del objeto ToolCall.
  // El resultado debe ser un string. Si tu función devuelve datos complejos,
  // serialízalos a JSON.
  ToolCall.Response := FormatDateTime('yyyy-mm-dd hh:nn:ss', Now);

  // 3. Informa al framework que has manejado esta llamada.
  Handled := True;
end;
```

Método 2: Usando el evento OnCallToolFunction de TAiChatConnection (General)

Este evento se dispara para cualquier llamada a función. Es útil si prefieres tener toda la lógica en un solo lugar.

```
procedure TForm1.AiConnCallToolFunction(Sender: TObject;
```

```

    AiToolCall: T AiToolsFunction);
begin
    if AiToolCall.name = 'GetFechaActual' then
    begin
        AiToolCall.Response := FormatDateTime('yyyy-mm-dd hh:nn:ss', Now);
    end
    else if AiToolCall.name = 'OtraFuncion' then
    begin
        // ...lógica para otra función...
    end;
end;
end;

```

Nota: Si implementas ambos, el evento `OnAction` del `TFunctionActionItem` tiene prioridad. Si estableces `Handled := True` allí, el evento `OnCallToolFunction` de `TAiChatConnection` no se disparará para esa llamada.

5.4 Poniéndolo todo junto

1. **Activa las herramientas:** Antes de enviar tu prompt, asegúrate de que el parámetro `Tool_Active` esté en `True`.

```

AiConn.Params.Values['Tool_Active'] := 'True';

```

2. **Envía un prompt que incite al uso de la herramienta:**

```

// Envía un prompt que haga referencia a la herramienta.
Response := AiConn.AddMessageAndRun('¿Me puedes dar la fecha y hora exacta de ahora mismo?', 'user', []);

```

```

// Muestra la respuesta final de la IA.
MemoResponse.Lines.Add('IA: ' + Response);

```

Cuando ejecutes este código, ***TAiChatConnection*** realizará automáticamente el ciclo completo de 2 pasos descrito al principio. No necesitas hacer nada más. El historial (**Messages**) mostrará claramente la petición de la herramienta por parte de la IA y la respuesta que tú le proporcionaste, todo gestionado de forma transparente.

Capítulo 6: Personalización y Extensión del Framework

Has dominado los fundamentos, las conversaciones multimodales y el uso de herramientas. Ahora es el momento de tomar el control total. Este capítulo te enseñará a modificar el comportamiento del framework, a ajustar los parámetros de bajo nivel y, lo más emocionante, a añadir soporte para nuevos proveedores de IA creando tus propios drivers.

6.1 El Sistema de Parámetros: El Verdadero Poder de TAIChatConnection

Hemos mencionado los parámetros de pasada, pero ahora profundizaremos en ellos. La propiedad Params de TAIChatConnection es mucho más que una simple lista de configuración; es una ventana en tiempo real al estado del "driver" de IA activo.

- **¿Qué es la propiedad Params?**

Es un TStringList que contiene todos los parámetros configurables para el driver actualmente seleccionado (DriverName) y el modelo (Model). Cuando cambias de driver, esta lista se actualiza automáticamente con los parámetros correspondientes.

- **Lectura y Modificación en Tiempo de Ejecución:**

Puedes leer y escribir en esta propiedad para alterar el comportamiento de la siguiente petición a la API. Esto es increíblemente útil para dar control al usuario final.

Ejemplo Práctico: Un TTrackBar para controlar la "Temperatura"

1. Añade un TTrackBar a tu formulario. Configura su Min a 0 y Max a 20.
2. Añade un TLabel para mostrar el valor actual.
3. En el evento OnChange del TTrackBar, escribe el siguiente código:

```
procedure TForm1.TrackBarTempChange(Sender: TObject);
var
    TempValue: Double;
begin
    // Convertimos el valor del TrackBar (0-20) a un rango de temperatura (0.0 - 2.0)
    TempValue := TrackBarTemp.Value / 10.0;
    LabelTemp.Text := Format('Temperatura: %.1f', [TempValue]);

    // ¡Aquí está la clave! Modificamos el parámetro en el AiConn.
    // Usamos FormatFloat para asegurar el formato de punto decimal correcto.
    AiConn.Params.Values['Temperature'] := FormatFloat('0.0', TempValue);
end;
```

Ahora, cuando el usuario mueva el TTrackBar, la propiedad Temperature del driver activo se actualizará al instante, afectando la "creatividad" de la siguiente respuesta de la IA. Puedes aplicar este mismo principio a Max_Tokens, Tool_Active, Asynchronous, etc.

- **Registrando Parámetros Personalizados con TAiChatFactory:**
El framework viene con valores por defecto para cada driver, pero tú puedes sobrescribirlos o añadir nuevos perfiles. Esto se hace (generalmente en la sección initialization de una unidad) usando TAiChatFactory.Instance.RegisterUserParam.

```
initialization
    // Establece un valor por defecto para CUALQUIER modelo del driver 'Ollama'
    TAiChatFactory.Instance.RegisterUserParam('Ollama', 'Url',
'http://localhost:11434/v1/');

    // Crea un perfil específico para el modelo 'llava' dentro del driver
'Ollama'
    TAiChatFactory.Instance.RegisterUserParam('Ollama', 'llava',
'InitialInstructions', 'Eres un experto en describir imágenes.');
```

```
    TAiChatFactory.Instance.RegisterUserParam('Ollama', 'llava',
'Temperature', '0.2');
end.
```

Cuando el usuario seleccione el driver Ollama y el modelo llava, TAiChatConnection cargará automáticamente estas configuraciones personalizadas en su propiedad Params.

6.2 Creando tu Propio Driver (Ej. para Anthropic Claude)

Esta es la máxima expresión de la extensibilidad del framework. Supongamos que Anthropic lanza una nueva versión de su API para el modelo Claude y quieres integrarla.

Guía Paso a Paso:

1. Crear la Nueva Unidad y Clase:

Crea una nueva unidad, por ejemplo, uMakerAi.Chat.Claude.pas. Dentro, define una nueva clase que herede de TAiChat.

```
unit uMakerAi.Chat.Claude;

interface
uses
    ..., uMakerAi.Chat, uMakerAi.Core; // Y otras dependencias

type
    TAiClaudeChat = class(TAiChat)
```

```

protected
// Métodos que vamos a sobrescribir
function InitChatCompletions: String; override;
class function GetDriverName: string; override;
class procedure RegisterDefaultParams(Params: TStrings); override;
class function CreateInstance(Sender: TComponent): TAiChat; override;
public
// Métodos o propiedades específicos de Claude, si los hubiera
end;

procedure Register;

implementation

procedure Register;
begin
// Este procedimiento podría no ser necesario si usamos la inicialización
end;

```

2. Implementar los Métodos de la "Fábrica":

Estos cuatro métodos son el contrato que tu clase debe cumplir para que la TAiChatFactory la reconozca.

```

class function TAiClaudeChat.GetDriverName: string;
begin
Result := 'Claude'; // El nombre que aparecerá en el ComboBox
end;

class function TAiClaudeChat.CreateInstance(Sender: TComponent): TAiChat;
begin
Result := TAiClaudeChat.Create(Sender); // Crea una instancia de sí mismo
end;

class procedure TAiClaudeChat.RegisterDefaultParams(Params: TStrings);
begin
Params.Clear;
Params.Add('ApiKey=@CLAUDE_API_KEY');
Params.Add('Model=claude-3-opus-20240229');
Params.Add('MaxTokens=4096');
Params.Add('BaseUrl=https://api.anthropic.com/v1/');
end;

```

3. Sobrescribir la Lógica Principal (InitChatCompletions):

Este es el paso más importante. Aquí adaptarás la construcción del JSON de la petición a las especificaciones de la API de Claude. Por ejemplo, Claude podría usar max_tokens_to_sample en lugar de max_tokens, o tener una estructura de mensajes diferente.

```

function TAiClaudeChat.InitChatCompletions: String;
var

```



```

    LRequest: TJsonObject;
begin
    // Aquí construirías el cuerpo de la petición JSON específico para la API
    de Claude.
    // Por ejemplo:
    LRequest := TJsonObject.Create;
    try
        LRequest.AddPair('model', Self.Model);
        LRequest.AddPair('messages', Self.GetMessages); // Podrías necesitar
        sobrescribir GetMessages también
        LRequest.AddPair('max_tokens_to_sample', Self.Max_tokens); // ;Parámetro
        específico de Claude!
        Result := LRequest.ToJSON;
    finally
        LRequest.Free;
    end;
end;

```

4. Registrar el Driver:

Finalmente, en la sección initialization de tu nueva unidad, le dices a la fábrica que existe un nuevo driver disponible.

```

initialization
    TAiChatFactory.Instance.RegisterDriver(TAiClaudeChat);
end.

```

¡Y ya está! Al compilar tu proyecto, el driver "Claude" aparecerá como una opción en TAiChatConnection, y toda la maquinaria de configuración y ejecución funcionará para él.

6.3 Hooks y Eventos Avanzados

Para los casos de personalización más finos, TAiChatConnection ofrece "hooks" (ganchos) en forma de eventos que te permiten interceptar el flujo de datos.

- OnBeforeSendMessage(Sender: TObject; var aMsg: TAiChatMessage):**
 Se dispara justo antes de que el historial de mensajes se convierta en JSON y se envíe. El parámetro aMsg es var, lo que significa que puedes modificar el último mensaje del usuario sobre la marcha.
 - Caso de uso:** Añadir automáticamente información contextual al prompt del usuario, como la fecha y hora actual, o firmar todas las peticiones con un identificador.
- OnProcessResponse(Sender: TObject; LastMsg, ResMsg: TAiChatMessage; var Response: string):**
 Se dispara después de que la IA ha devuelto una respuesta de texto, pero antes de que se asigne al ResMsg final y se notifique a través de OnReceiveDataEnd. Te permite **modificar la respuesta cruda de la IA**.

- **Caso de uso:** Censurar palabras, traducir la respuesta a otro idioma, o formatear la respuesta añadiendo saltos de línea o Markdown antes de mostrarla en un TMemor.
- **OnChatModelChange(Sender: TObject; const OldChat, NewChat: TAiChat):**
Se dispara exactamente cuando el usuario cambia el DriverName. Te da acceso a la instancia del driver antiguo (a punto de ser destruido) y a la nueva.
 - **Caso de uso:** Implementar una "traducción de historial". Podrías tomar el historial del OldChat, usar una IA para resumirlo, y luego inyectar ese resumen como el primer mensaje en el NewChat, proporcionando una transición de contexto más suave entre diferentes proveedores.

Apendices:

Apéndice A: Guía de Referencia de Drivers

Esta sección proporciona una referencia rápida de los drivers soportados y sus parámetros y capacidades más notables. Usa esta tabla para decidir qué driver es el más adecuado para tu tarea.

Característica	OpenAI	Gemini	Grok	Ollama
DriverName	OpenAI	Gemini	Grok	Ollama
Modelos Populares	gpt-4o, gpt-4o-mini, gpt-3.5-turbo	gemini-1.5-pro, gemini-1.5-flash	grok-2, grok-2-vision	llama3.2, llama, phi3
Soporte de Visión	✔️ (Nativo)	✔️ (Nativo)	✔️ (Nativo)	✔️ (Depende del modelo, ej. llama)
Soporte TTS	✔️ (Endpoint dedicado)	✔️ (Nativo en modelos -tts)	❌ (No disponible)	❌ (No disponible)
Soporte Vídeo	❌ (Sora no disponible en API)	✔️ (Veo, con predictLongRunning)	❌ (No disponible)	❌ (No disponible)
Function Calling	✔️ (Excelente soporte)	✔️ (Excelente soporte)	✔️ (Soporte compatible)	✔️ (Depende del modelo, ej. llama3.2)
Parámetro ApiKey	✔️ Requerido	✔️ Requerido	✔️ Requerido	❌ No requerido
Parámetro BaseURL	https://api.openai.com/v1/	https://generativelanguage.googleapis.com/v1beta/	https://api.x.ai/v1/	http://localhost:11434/v1/ (configurable)
Parámetro Model	✔️	✔️	✔️	✔️

Parámetro Temperature	✓ (0.0 - 2.0)	✓ (0.0 - 1.0)	✓ (0.0 - 2.0)	✓ (Numérico)
Parámetro Max_Tokens	✓	✓ (maxOutputTokens)	✓	✓ (num_predict)
Parámetro Top_p	✓	✓	✓	✓
Parámetro Asynchronous	✓ (Streaming)	✓ (Streaming)	✓ (Streaming)	✓ (Streaming)
Parámetros Específicos	Logit_bias, Seed, Response_format	TopK, SpeechConfig, VideoParameters	Reasoning_content (en respuesta)	num_ctx (tamaño de contexto)
Ideal Para...	Uso general, herramientas potentes, la última tecnología.	Multimodalidad avanzada (contexto largo, video), ecosistema Google.	Velocidad y respuestas directas, compatibilidad con API OpenAI.	Privacidad total, costos cero, personalización, operación offline.

Apéndice B: Solución de Problemas Comunes (FAQ)

P: Recibo un error 401 Unauthorized o similar.

R: Este es casi siempre un problema con la **API Key**.

1. Verifica que has asignado la ApiKey correcta en los parámetros del driver. Puedes hacerlo a través de `TAiChatFactory.Instance.RegisterUserParam('NombreDriver', 'ApiKey', 'TU_CLAVE_AQUI')`.
2. Asegúrate de que la clave no ha expirado o ha sido revocada en el panel de control del proveedor.
3. Confirma que tu cuenta tiene crédito o un plan de pago activo.

P: Mi aplicación se congela al enviar un prompt.

R: Estás realizando una llamada **síncrona**, que bloquea el hilo principal.

1. **Solución Rápida:** Lanza la petición en un hilo separado usando `TTask.Run`.
2. **Solución Recomendada:** Activa el modo asíncrono. Establece el parámetro `Asynchronous` a `True` (`AiConn.Params.Values['Asynchronous'] := 'True'`). Deberás manejar la respuesta en los eventos `OnReceiveData` (para el streaming de texto) y `OnReceiveDataEnd` (para la respuesta final).

P: Envío una imagen pero la IA responde "No veo ninguna imagen".

R: El driver o el modelo no están configurados correctamente para visión.

1. **Verifica el Modelo:** Asegúrate de que el Model seleccionado es multimodal (ej. `gpt-4o`, `gemini-1.5-pro`, `llava`).

2. **Verifica los Soportes del Driver:** El parámetro ChatMediaSupports del driver debe incluir Tcm_Image. Esto suele estar preconfigurado, pero si usas un driver personalizado, es un punto a revisar.
3. **Comprueba el Objeto Mensaje:** Asegúrate de que el TAI MediaFile se está añadiendo correctamente al TAI ChatMessage antes de llamar a Run.

P: La generación de audio o video no funciona.

R: Similar al problema de visión, es un problema de configuración.

1. **Activa la Salida Correcta:** Antes de la llamada, debes establecer AiConn.Params.Values['NativeOutputFiles'] := '[Tfc_Audio]' (o [Tfc_Video]).
2. **Revisa el Modelo:** No todos los modelos soportan estas salidas. Para Gemini TTS, necesitas un modelo -tts. Para video, necesitas un modelo específico como veo-2.0.
3. **Revisa el Código de OnReceiveDataEnd:** La respuesta no vendrá como texto. Debes inspeccionar aMsg.MediaFiles para encontrar el archivo generado y procesarlo.

P: La IA no usa mis herramientas (Function Calling).

R: Varios puntos a verificar en orden:

1. **¿Está Activo?:** El parámetro Tool_Active debe ser True.
2. **¿Están Enlazadas?:** La propiedad AiFunctions de tu TAI ChatConnection debe estar enlazada a tu componente TAI Functions.
3. **¿Descripción Clara?:** La propiedad description en el JSON de tu herramienta es **la parte más importante**. La IA decide si usar la herramienta basándose en esa descripción. Debe ser clara, detallada y en lenguaje natural. Si la descripción es "Obtiene datos", es demasiado vaga. Si es "Obtiene la temperatura actual y las condiciones climáticas para una ciudad específica", es mucho mejor.
4. **¿Prompt Adecuado?:** Tu pregunta a la IA debe incitar naturalmente al uso de la herramienta.

Apéndice C: Recetas de Código (Code Snippets)

Receta 1: Chatbot que Resume la Conversación al Cambiar de Driver

Usa el evento OnChatModelChange para mantener el contexto cuando el usuario cambia de, por ejemplo, OpenAI a un modelo local de Ollama.

```
procedure TForm1.AiConnChatModelChange(Sender: TObject; const OldChat,
  NewChat: TAIChat);
var
  SummaryTask: ITask;
  Summary: string;
begin
  // Solo si hay un historial previo y estamos cambiando a un nuevo driver
  válido.
```

```

    if not Assigned(OldChat) or not Assigned(NewChat) or
(OldChat.Messages.Count < 3) then
        Exit;

    MemoResponse.Lines.Add('Cambiando de driver... Resumiendo historial...');

    // Usamos un TTask para no bloquear la UI mientras se resume.
    SummaryTask := TTask.Run(
        function: string
        var
            Summarizer: TAiChat;
            History: string;
        begin
            // Usamos una instancia temporal de un modelo rápido para resumir.
            Summarizer := AiConn.CreateChatForDriver('OpenAI', 'gpt-4o-mini');
            try
                History := OldChat.Messages.ExportChatHistory.Format;
                Result := Summarizer.AddMessageAndRun('Resume el siguiente historial
de chat en 3 frases clave: ' + History, 'user', []);
            finally
                Summarizer.Free;
            end;
        end);

    // Cuando la tarea de resumen termina, inyectamos el resumen en el nuevo
chat.
    TTask.ContinueWith(SummaryTask,
        procedure(const Task: ITask)
        begin
            TThread.Queue(nil,
                procedure
                begin
                    Summary := (Task as ITask<string>).Result;
                    NewChat.AddMessage('Este es un resumen de nuestra conversación
hasta ahora: ' + Summary, 'system');
                    MemoResponse.Lines.Add(';Listo! Puedes continuar la conversación
con el nuevo modelo.');

```

Receta 2: Usar la IA para Analizar el Sentimiento de un Texto

Una aplicación potente de "Function Calling" donde la herramienta no accede a datos externos, sino que realiza una acción estructurada.

1. Define la Herramienta en TAiFunctions:

```

{
  "type": "function",
  "function": {
    "name": "AnalyzeSentiment",
    "description": "Analiza el sentimiento de un texto y lo clasifica como
'positivo', 'negativo' o 'neutro'.",
    "parameters": {
      "type": "object",
      "properties": {
        "sentiment": {

```

```

        "type": "string",
        "description": "La clasificación del sentimiento.",
        "enum": ["positivo", "negativo", "neutro"]
    },
    "required": ["sentiment"]
}
}
}

```

2. Implementa el OnAction:

```

procedure      TForm1.AiFunctions1Functions0AnalyzeSentimentAction(Sender:
TObject;
    ...; ToolCall: TAIToolsFunction; var Handled: Boolean);
var
    Sentiment: string;
begin
    // La IA ya hizo el trabajo. Nosotros solo capturamos el resultado.
    Sentiment := ToolCall.Params.Values['sentiment'];
    ShowMessage('El sentimiento detectado por la IA es: ' + Sentiment);

    // Le decimos a la IA que la acción se completó.
    ToolCall.Response := '{"status": "ok", "sentiment_logged": "' + Sentiment
+ '"}';
    Handled := True;
end;

```

3. Ejecuta el Prompt:

```

AiConn.Params.Values['Tool_Active'] := 'True';
// Forzamos a la IA a usar la herramienta con 'tool_choice'
AiConn.Params.Values['Tool_choice'] := '{"type": "function", "function":
{"name": "AnalyzeSentiment"}}';

AiConn.AddMessageAndRun('¡Hoy es un día fantástico! Me encanta programar en
Delphi.', 'user', []);

```

En este caso, la IA no responde con texto, sino que llama directamente a tu función con el resultado de su análisis, permitiéndote actuar sobre esa data estructurada en tu código Delphi.

Apéndice D: Guía de parámetros del driver

En esta sección se presentan los parámetros más comunes que se pueden manejar dentro de la propiedad Params del **TAIChatConnection**.

Propiedad	Tipo	Descripción	Uso/Valores Comunes
ApiKey	String	Clave de API para autenticación con el servicio de IA. Si comienza con @, se interpreta como el nombre de una variable de entorno.	Necesario para usar la API. Ejemplo: "sk-xx" o "@OPENAI_API_KEY"
Model	String	Identificador del modelo de IA a utilizar (ej., "gpt-4o", "gpt-3.5-turbo").	Especifica el modelo de lenguaje. Lista de modelos disponibles se obtiene con <code>TAiChat.GetModels</code> .
Frequency_penalty	Double	Penalización de frecuencia. Valores entre -2.0 y 2.0. Reduce la probabilidad de que el modelo repita tokens que ya ha usado.	Controla la repetición en las respuestas. Valores positivos fomentan respuestas más diversas.
Logit_bias	String	Modifica la probabilidad de que aparezcan ciertos tokens. Debe ser una cadena vacía o un valor entre -100 y 100.	Permite influir en el vocabulario usado. Útil para dirigir las respuestas hacia temas o palabras específicas. Requiere conocimiento avanzado de tokens.
Logprobs	Boolean	Si se debe incluir la probabilidad logarítmica de los tokens en la respuesta.	Útil para análisis avanzado de la respuesta del modelo.
Top_logprobs	String	Número de log probabilidades principales que se devolverán. Debe ser una cadena vacía o un valor entre 0 y 5.	Devuelve los tokens más probables y sus probabilidades logarítmicas en la respuesta.
Max_tokens	Integer	Número máximo de tokens que se generarán en la respuesta. Si es 0, se usa el máximo permitido por el modelo.	Limita la longitud de las respuestas. Ayuda a controlar costos y tiempos de respuesta.

N	Integer	Número de opciones de finalización de chat que se generarán para cada mensaje de entrada. Se le cobrará en función del número de tokens generados en todas las opciones. Mantenga n como 1 para minimizar los costos. El valor predeterminado es 1.	Genera múltiples respuestas para una misma pregunta.
Presence_penalty	Double	Penalización de presencia. Valores entre -2.0 y 2.0. Reduce la probabilidad de que el modelo introduzca nuevos temas.	Controla la introducción de nuevos temas. Valores positivos fomentan respuestas más centradas en el tema original.
Response_format	TAiChatResponseFormat	Formato de la respuesta. Puede ser tiaChatRfText, tiaChatRfJson, o tiaChatRfJsonSchema.	Especifica el formato de salida deseado. tiaChatRfJsonSchema es útil para obtener respuestas que se ajusten a un esquema JSON predefinido. Solo funciona con gpt-4-1106-preview y gpt-3.5-turbo-1106.
Seed	Integer	Semilla para generar respuestas deterministas. Si es 0, no se envía.	Útil para reproducir resultados consistentes para una misma entrada.
Stop	String	Secuencia(s) de parada. El modelo dejará de generar tokens cuando encuentre estas secuencias. Separar por comas.	Permite truncar las respuestas en puntos específicos.
Asynchronous	Boolean	Si se deben realizar las llamadas a la API de forma asíncrona.	Permite que la aplicación siga respondiendo mientras se procesa la respuesta de la API. Útil para interfaces de usuario interactivas. No compatible con Tool_Active.
Temperature	Double	Temperatura. Valores entre 0.0 y 2.0. Controla la aleatoriedad de la respuesta.	Influye en la creatividad de las respuestas. Valores más altos generan respuestas más impredecibles.

Top_p	Double	Muestreo Top P. Valores entre 0.0 y 1.0. Similar a la temperatura, pero controla la aleatoriedad de forma diferente. Si es 0, no se envía, entre 0 y 1.	Limita el conjunto de tokens posibles a los más probables.
Tool_choice	String	Especifica qué herramienta (tool) debe usar el modelo. Puede ser "auto" o una definición JSON específica.	Controla la selección de herramientas. "auto" permite que el modelo decida.
Tool_Active	Boolean	Indica si las herramientas (tools) están activas.	Habilita o deshabilita el uso de herramientas.
User	String	Identificador del usuario final.	Útil para monitorizar y controlar el uso de la API por usuario.
InitialInstructions	TStrings	Instrucciones iniciales para el modelo. Se envían como el primer mensaje del chat con el role "system".	Define el comportamiento y personalidad del asistente.
LastContent	String	(Solo lectura) El contenido de la última respuesta recibida.	Información útil para depuración y visualización.
LastPrompt	String	(Solo lectura) El último prompt enviado.	Información útil para depuración y visualización.
Busy	Boolean	(Solo lectura) Indica si el componente está actualmente procesando una solicitud.	Útil para evitar enviar múltiples solicitudes simultáneamente.
Url	String	URL base de la API.	Permite cambiar el punto final de la API. El valor por defecto es "https://api.openai.com/v1/".
ResponseTimeOut	Integer	Tiempo máximo de espera (en milisegundos) para una respuesta de la API.	Evita que la aplicación se quede bloqueada si la API no responde.
Memory	TStrings	Un conjunto de pares clave-valor que se utilizan para guardar información sobre la conversación actual.	Guarda información sobre la conversación actual.
AiFunctions	TAiFunctions	El objeto que define las funciones que se pueden invocar.	Define las funciones que se pueden invocar.
JsonSchema	TStrings	Un esquema JSON que el modelo debe usar para dar formato a su respuesta.	Define el formato de la respuesta.

Stream_Usage	Boolean	Envia la estadística de uso por token	Envia la estadística de uso por token
NativeInputFiles	TAiFileCategories	Permite especificar los tipos de archivos de entrada que el modelo puede procesar de forma nativa. Es un conjunto de valores del tipo TAiFileCategory.	Indica al componente qué tipos de archivos multimedia puede manejar directamente. Ejemplos: [Tfc_Image, Tfc_Audio] , si la funcionalidad está disponible en completions debe estar acompañado de su respectivo valor en ChatMediaSupports
NativeOutputFiles	TAiFileCategories	Permite especificar los tipos de archivos de salida que el modelo puede generar de forma nativa. Es un conjunto de valores del tipo TAiFileCategory.	Indica al componente qué tipos de archivos multimedia puede generar directamente. Ejemplos: [Tfc_Image, Tfc_Audio], si la funcionalidad está disponible en completions debe estar acompañado de su respectivo valor en ChatMediaSupports
ChatMediaSupports	TAiChatMediaSupports	Permite especificar los tipos de archivos de entrada que el modelo puede procesar de forma nativa con completions. Es un conjunto de valores del tipo TAiChatMediaSupport.	Indica al componente qué tipos de archivos multimedia puede manejar directamente en el Chat. Ejemplos: [tcm_Image, tcm_Audio].
Voice	String	Nombre de la voz a usar para la generación de voz (TTS).	Especifica la voz a usar para la generación de voz (TTS).
voice_format	String	Formato de la voz a usar para la generación de voz (TTS).	Especifica la formato de la voz a usar para la generación de voz (TTS). El valor predeterminado es "wav".
Language	String	El idioma para la transcripción de audio.	Especifica el idioma para la transcripción de audio, por ejemplo, 'es', 'en', 'es-419'.
Transcription_ResponseFormat	String	El formato para la transcripción de audio.	Especifica el formato para la transcripción de audio, por ejemplo, 'json', 'text', 'verbose_json', etc.

Transcription_Time stampGranularities	String	Los niveles de detalle para las marcas de tiempo en la transcripción de audio.	Especifica los niveles de detalle para las marcas de tiempo en la transcripción de audio, por ejemplo, 'word', 'segment', 'word,segment'.
Propiedades de solo lectura, no se pueden modificar ni pasar en la propiedad params del componente			
Prompt_tokens	Integer	(Solo lectura) Cantidad de tokens utilizados en la solicitud.	Util para controlar el consumo de recursos y optimizar las solicitudes.
Completion_tokens	Integer	(Solo lectura) Cantidad de tokens generados en la respuesta.	Util para controlar el consumo de recursos y optimizar las respuestas.
Total_tokens	Integer	(Solo lectura) Cantidad total de tokens utilizados (solicitud + respuesta).	Util para controlar el consumo de recursos y optimizar el uso general de la API.
Messages	TAiChatMessages	Lista de mensajes en la conversación.	Lista de mensajes en la conversación.
AiFunctions	TAiFunctions	Lista de funciones de la AI.	Lista de funciones de la AI.