

14-Dic-2024

Implementación del componente TAIChat en Delphi

Historial de Cambios

14-Dic-2024

- Se adiciona soporte al Chat de Grok con el componente TAIChatGrok.
- Se adiciona soporte a los Embeddings de Grok con el componente TAIChatGrokEmbeddings.
- Se adiciona la propiedad NativeInputFiles para filtrar los archivos que pasan directamente al modelo
- Se adiciona la propiedad NativeOutputFiles para indicarle al modelo en que formato deseamos la respuesta

Introducción

En la era digital actual, la inteligencia artificial (IA) se ha convertido en una herramienta esencial para el desarrollo de aplicaciones avanzadas. La capacidad de comprender y procesar el lenguaje natural ha abierto nuevas fronteras en la interacción entre humanos y máquinas. Con esto en mente, el desarrollo de un framework que simplifique la conectividad con distintos modelos de IA es crucial. TAIChat y sus componentes asociados tienen como objetivo proporcionar a los desarrolladores una forma sencilla pero potente de integrar esta funcionalidad en sus aplicaciones Delphi.

Un framework bien diseñado alivia las barreras de entrada, permitiéndole a los desarrolladores enfocarse en la implementación de soluciones innovadoras sin preocuparse por las complejidades subyacentes de cada API propietaria de los modelos de IA. Al ofrecer compatibilidad con múltiples modelos líderes en la industria, como OpenAI, Anthropic, Gemini, entre otros, se garantiza la flexibilidad necesaria para adaptarse a las necesidades cambiantes del desarrollo de software.

Posibles Usos de los Modelos de IA en Delphi

Con los modelos de IA a disposición en un entorno de programación como Delphi, los desarrolladores pueden crear aplicaciones más inteligentes y receptivas en una variedad de sectores. Algunos de los ejemplos de las aplicaciones que se pueden desarrollar incluyen:

1. **Asistentes Virtuales:** Implementación de asistentes que puedan gestionar consultas complejas, responder preguntas frecuentes y proporcionar recomendaciones personalizadas, mejorando así la experiencia del usuario.

2. **Análisis de Sentimiento:** Integración de módulos que analicen y comprendan el sentimiento detrás del texto, útil para monitorear redes sociales, encuestas de opinión y la satisfacción del cliente.
3. **Generación de Contenido:** Automatización de la creación de contenido a través de texto, como resúmenes, informes, y artículos, optimizando el tiempo y recursos humanos.
4. **Traducción Automática:** Desarrollo de aplicaciones que ofrezcan traducción de textos en tiempo real, facilitando la comunicación multilingüe.
5. **Chatbots Inteligentes:** Diseño de chatbots que pueden aprender y adaptar sus respuestas con cada interacción, proporcionando soporte continuo y más contextual a los usuarios.
6. **Reconocimiento de Voz a Texto:** Implementación de funcionalidades que conviertan voz en texto, permitiendo una entrada de datos más natural y fluida.
7. **Motivación Creativa:** Asistir en la generación de ideas para escritura creativa, diseño de juegos, entre otros, permitiendo a los profesionales del área explorar nuevas posibilidades.

¿Qué es TAIChat?

Propósito General

El componente **TAIChat** de Delphi tiene como objetivo simplificar el acceso y la interacción con modelos de lenguaje de inteligencia artificial (LLM) a través de diferentes APIs, proporcionando una interfaz unificada para los desarrolladores de Delphi. Esto permite a los programadores enfocarse en integrar capacidades de IA en sus aplicaciones de manera eficiente, sin necesidad de manejar las complejidades específicas de cada API.

Interacción con Modelos de IA

TAIChat actúa como un componente base, encapsulando la complejidad de interactuar con APIs de LLM. Este componente es la base para otros componentes heredados como TAIollamaChat, TAIgeminichat, TAIClaudeChat, TAIMistralChat y TAIOpenChat. Todos estos componentes encapsulan y adaptan las particularidades de cada API, facilitando la integración con modelos de IA actuales y futuros.

Funciones Principales

- **Comunicación con LLMs:** Envía mensajes de texto a los modelos para obtener respuestas.
- **Tools Functions:** Facilita la ejecución de callbacks y el manejo de funciones definidas por la IA.
- **Gestión de Memoria:** Permite al modelo recordar tanto datos de la conversación actual como datos almacenados en memoria.

- **Configuración de Parámetros:** Personalización de características como temperatura, max_tokens, y otros parámetros de ejecución.

- **Manejo de Archivos Adjuntos:** Procesa archivos multimedia (imágenes, voz, PDFs) para realizar preprocesamientos como convertir o interpretar archivos antes de llegar al LLM.

Configuración y Personalización

TAiChat permite la configuración de todos los parámetros necesarios para el funcionamiento óptimo del modelo de IA. Las propiedades como temperatura, max_tokens, y otras características pueden ajustarse según las necesidades específicas de la aplicación, y se aplican de acuerdo con las particularidades del API de cada modelo.

Manejo de Errores y Excepciones

TAiChat incluye mecanismos para manejar errores de conexión y procesamiento, asegurando una interacción robusta con las APIs. Los errores se gestionan en cada procedimiento, y en modos asíncronicos, se capturan adecuadamente para permitir su manejo desde el entorno Delphi.

Ejemplos y Demostraciones

1. Ejemplo Básico:

```
var
  Chat: TAiChat;
begin
  Chat := TAiChat.Create(nil);
  try
    Chat.ApiKey := 'tu-api-key';
    Chat.Model := 'gpt-4';
    Chat.AddMessage('¿Cuál es la capital de Francia?', 'user');
    ShowMessage(Chat.Run);
  finally
    Chat.Free;
  end;
end;
```

2. Ejemplo con Proceso y MediaFiles (imágenes):

```
var
  Res: String;
  MediaFile: TAiMediaFile;
begin
  MediaFile := TAiMediaFile.Create;
  MediaFile.LoadFromFile('ruta/del/archivo.jpg');
  Res := Chat.AddMessageAndRun('Describe esta imagen', 'user', [MediaFile]);
  ShowMessage(Res);
  MediaFile.Free;
end;
```

3. Ejemplo con Proceso y MediaFiles (Voz) con llamado directo: (solo para OpenAi)

```

Var
  Res: String;
  MediaFile: TAIMediaFile;
  Msg: TAIChatMessage;
  FileName: String;
begin

  MediaFile := TAIMediaFile.Create;
  MediaFile.LoadFromFile('c:\temp\prompt.wav');

  Try
    Msg := AiOpenChat1.AddMessageAndRunMsg(MemoPrompt.Lines.Text, 'user', [MediaFile]);
  Finally
    FreeAndNil(MediaFile);
  End;

  MemoResponse.Lines.Text := Msg.Content;

  If (Msg.MediaFiles.Count > 0) and (Assigned(Msg.MediaFiles[0].Content)) then
  Begin
    FileName := 'c:\temp\respuesta3' + Cons.ToString + '.wav';
    Msg.MediaFiles[0].Content.Position := 0;
    Msg.MediaFiles[0].Content.SaveToFile(FileName);

    Try
      MediaPlayer1.FileName := FileName;
      MediaPlayer1.Play;
    Finally
      End;
    End;
  End;

```

4. Ejemplo con Proceso y MediaFiles (Voz) con respuesta en evento: (solo para OpenAi)

```

Var
  Res: String;
  MediaFile: TAIMediaFile;
begin
  MediaFile := TAIMediaFile.Create;
  MediaFile.LoadFromFile('c:\temp\prompt.wav');

  Try
    Res := AiOpenChat1.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', [MediaFile]);
    MemoResponse.Lines.Text := Res;
  Finally
    FreeAndNil(MediaFile);
  End;
End;

```

En el evento OnReceiveDataEnd recibe el parámetro **aMsg**: **TAIChatMessage** del cual obtendríamos el resultado.

```

procedure TForm75.AiOpenChat1ReceiveDataEnd(const Sender: TObject; aMsg: TAIChatMessage;
aResponse: TJSONObject; aRole, aText: string);
Var
  FileName: String;
begin

  If (aMsg.MediaFiles.Count > 0) and (Assigned(aMsg.MediaFiles[0].Content)) then
  Begin
    Inc(Cons);
    FileName := 'c:\temp\respuesta' + Cons.ToString + '.wav';
    aMsg.MediaFiles[0].Content.Position := 0;
    aMsg.MediaFiles[0].Content.SaveToFile(FileName);
  End;

```

```
Try
  MediaPlayer1.FileName := FileName;
  MediaPlayer1.Play;
Finally
End;
End;
end;
```

Propiedades Públicas:

1. **Messages: TAIChatMessages**
Colección de los mensajes gestionados por el componente.
2. **LastError: String**
Almacena el último error que ocurrió durante la ejecución del componente.

Propiedades Publicadas:

1. **ApiKey: String**
Clave para la API del modelo de IA.
2. **Model: String**
Especifica el modelo que se utilizará para las interacciones de chat (ej. GPT-3.5, GPT-4, etc.).
3. **Frequency_penalty: Double**
Penalización para las palabras frecuentes en la respuesta generada (-2 a 2).
4. **Logit_bias: String**
Parámetro opcional para ajustar los sesgos logit de determinadas palabras. Puede ser una cadena vacía o contener valores entre -100 y 100.
5. **Logprobs: Boolean**
Habilita o deshabilita la captura de probabilidades logarítmicas.
6. **Top_logprobs: String**
Especifica cuántas probabilidades logarítmicas superiores se devolverán (valores entre 0 y 5).
7. **Max_tokens: integer**
Límite de tokens permitidos en una respuesta. Si es 0, se toma el valor máximo permitido por el modelo.
8. **N: integer**
Número de respuestas generadas por cada mensaje de entrada (por defecto 1).
9. **Presence_penalty: Double**
Penalización para evitar que se repitan conceptos similares en las respuestas (-2.0 a 2.0).
10. **Response_format: TAIOpenChatResponseFormat**
Formato de la respuesta generada por el modelo, compatible con ciertos modelos como GPT-4.
11. **Seed: integer**
Semilla utilizada para generar variaciones en las respuestas. Si es 0, no se utiliza.
12. **Stop: string**
Palabras clave para detener la generación de respuestas. Se pueden definir varias palabras separadas por comas.

13. **Asynchronous: Boolean**
Determina si las respuestas se procesan de manera asíncrona.
14. **Temperature: Double**
Controla el nivel de aleatoriedad en la generación de respuestas (valores entre 0 y 2).
15. **Top_p: Double**
Ajusta la probabilidad acumulada para la generación de respuestas (valores entre 0 y 1).
16. **Tools: TStrings**
Lista de herramientas disponibles que el modelo puede utilizar para procesar las interacciones.
17. **Tool_choice: string**
Nombre de la herramienta seleccionada para el procesamiento del chat.
18. **Tool_Active: Boolean**
Indica si la herramienta seleccionada está activa.
19. **User: String**
Usuario asociado a las interacciones del chat.
20. **InitialInstructions: TStrings**
Instrucciones iniciales que se enviarán al modelo al comenzar la conversación.
21. **Prompt_tokens: integer**
Número de tokens usados en el "prompt" de la conversación.
22. **Completion_tokens: integer**
Número de tokens generados en la respuesta.
23. **Total_tokens: integer**
Suma de los tokens utilizados en el "prompt" y en la respuesta.
24. **LastContent: String**
Último contenido enviado en la conversación.
25. **LastPrompt: String**
Último "prompt" enviado al modelo.
26. **Busy: Boolean**
Indica si el componente está ocupado procesando una solicitud.
27. **OnReceiveData: TAIOpenChatDataEvent**
Evento que se dispara al recibir datos del modelo.
28. **OnReceiveDataEnd: TAIOpenChatDataEvent**
Evento que se dispara cuando la recepción de datos ha finalizado.
29. **OnAddMessage: TAIOpenChatDataEvent**
Evento que se dispara cuando se agrega un mensaje a la conversación.
30. **OnCallToolFunction: TOnCallToolFunction**
Evento que se dispara al invocar una función de una herramienta.
31. **OnBeforeSendMessage: TAIOpenChatBeforeSendEvent**
Evento que permite interceptar y modificar un mensaje antes de enviarlo.
32. **OnInitChat: TAIOpenChatInitChat**
Evento que se dispara al iniciar una nueva conversación.
33. **Url: String**
URL del servicio del modelo de IA.

34. **AIChatConfig: TAiChatConfig**
Configuración utilizada para las interacciones con el modelo.
35. **ResponseTimeOut: integer**
Tiempo máximo de espera para recibir una respuesta del modelo.
36. **Memory: TStrings**
Memoria persistente utilizada para almacenar datos relevantes a la conversación.
37. **Functions: TFunctionActionItems**
Lista de funciones que pueden ser ejecutadas dentro del flujo de la conversación.
38. **AiFunctions: TAiFunctions**
Colección de funciones AI específicas para ejecutar tareas.
39. **OnProcessMediaFile: TAiOpenChatOnMediaFile**
Evento que se dispara cuando se procesa un archivo de medios.
40. **JsonSchema: TStrings**
Esquema JSON utilizado para validar o estructurar las respuestas del modelo.
41. **NativeInputFiles: TAiFileCategories**
Permite filtrar los archivos adjuntos que pasarán directamente al modelo, actualmente según el modelo puede incluir Audio o Imágenes, además del texto que va por defecto.
42. **NativeOutputFiles: TAiFileCategories**
Permite indicarle al modelo en que formato se desea la respuesta, actualmente solo OpenAI tiene formatos de respuesta en Texto y Audio y solo están disponibles con los modelos de audio en particular, marcará error en otros modelos.

Eventos del Componente TAiChat

TAiOpenChatDataEvent: Este evento se dispara cuando se recibe cualquier dato durante la interacción con el modelo de IA. Es útil para procesar cada fragmento de información que proviene del modelo, particularmente en operaciones asincrónicas.

```
TAiOpenChatDataEvent = procedure(const Sender: TObject; aMsg: TAiChatMessage; aResponse:
TJsonObject; aRole, aText: String) of object;
```

Uso Común:

- Capturar las respuestas parciales mientras se están procesando en modo asincrónico.
- Actualizar interfaces de usuario mientras se reciben datos.

TAiOpenChatBeforeSendEvent: Se dispara justo antes de enviar un mensaje al modelo de IA. Permite al desarrollador modificar o validar el mensaje antes de ser enviado.

```
TAiOpenChatBeforeSendEvent = procedure(const Sender: TObject; var aMsg: TAiChatMessage)
of object;
```

- Validar y final ajustar el mensaje antes de enviarlo al modelo.

- Modificar el contenido del mensaje en función de lógica personalizada.

TAiOpenChatInitChat: Este evento es llamado al iniciar un nuevo chat. Permite configurar y modificar instrucciones iniciales o ajustar la memoria del chat.

```
TAiOpenChatInitChat = procedure(const Sender: TObject; aRole: String; Var aText: String;  
Var aMemory: TJsonObject) of object;
```

- Establecer el contexto inicial de una conversación.
- Personalizar la memoria del chat con datos específicos para la sesión que comienza.

TAiOpenChatOnMediaFile: Gestiona el procesamiento de archivos de medios antes de que lleguen al modelo LLM. Se puede usar para manipular o convertir archivos en otros formatos.

```
TAiOpenChatOnMediaFile = procedure(const Sender: TObject; Prompt: String; MediaFile:  
TAiMediaFile; Var Respuesta: String; Var aProcesado: Boolean) of object;
```

- Convertir imágenes a texto usando OCR antes de enviarlas al modelo.
- Realizar análisis preliminares de archivos de audio.

TAiOpenChatDataEnd: Este evento se activa cuando la recepción de datos del modelo de IA se completa, útil para finalizar operaciones basadas en la recepción completa.

```
TAiOpenChatDataEnd = procedure(const Sender: TObject; aMsg: TAiChatMessage; aResponse:  
TJsonObject; aRole, aText: String) of object;
```

- Ejecutar acciones finales una vez que se ha recibido toda la respuesta del modelo.
- Actualizaciones de estado o limpieza después de la interacción.

OnCallToolFunction: Llamado cuando una herramienta específica definida por el modelo IA necesita ser ejecutada. Esto permite implementar callbacks que sean invocados por el modelo.

```
TOncallToolFunction = procedure(Sender: TObject; AiToolCall: TAiToolsFunction) of object;
```

- Ejecutar funciones específicas que han sido solicitadas por el modelo.
- Integrar lógica comercial o procesar comandos específicos.

¿Cómo iniciar?

Comenzaremos por lo más básico, que es crear una petición y obtener la respuesta, sin embargo, los componentes tienen diferentes formas para realizar el mismo objetivo, veamos:

Ejemplo 1: Función AddMessage y Run

En este primer ejemplo, se adiciona un mensaje del tipo “user” al chat y luego se ejecuta obteniendo una respuesta, utilizamos el método `AddMessage(aPrompt, aRole : String) : TAiChatMessage`, el cual solamente adiciona un mensaje en la memoria del chat, para luego ser ejecutado con la función `Run`, obteniendo así la respuesta del modelo.

```
var
  Chat: TAiChat;
begin
  Chat := TAiChat.Create(nil);
  try
    Chat.ApiKey := 'tu-api-key';
    Chat.Model := 'gpt-4o';
    Chat.AddMessage('¿Cuál es la capital de Francia?', 'user');
    ShowMessage(Chat.Run);
  finally
    Chat.Free;
  end;
end;
```

Ejemplo 2: Función NewMessage y Run

En este ejemplo, se crea un mensaje del tipo “user”, no se adiciona al chat sino que se ejecuta directamente, utilizamos el método `NewMessage(aPrompt, aRole : String) : TAiChatMessage`, el cual se pasa a la función `Run`, obteniendo así la respuesta del modelo, al utilizar la función “Run” el sistema adiciona el mensaje en el chat antes de ejecutarlo.

```
var
  Chat: TAiChat;
  Msg : TAiChatMessage;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('¿Cuál es la capital de Francia?', 'user');

    //...aquí puede manejar el msg antes de enviarlo al modelo...
    Msg.LoadMediaFromFile('ruta/del/archivo1.jpg');
    Msg.LoadMediaFromFile('ruta/del/archivo2.jpg');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
end;
```

Ejemplo 3: Función AddMessageAndRun

En este ejemplo, se crea el mensaje y ejecuta en un solo paso, retornando directamente el resultado, el mensaje queda adicionado al chat automáticamente.

```
var
  Chat: TAiChat;
```

```

Res : String;
begin
  Chat := TAIOpenChat.Create(nil);
  try
    Chat.ApiKey := api key';
    Chat.Model := 'gpt-4o';

    Res := Chat.AddMessageAndRun('cual es la capital de Francia?', 'user', []);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
end;

```

Ejemplo 4: Direcccionar la conversación del chat

El chat puede tomar diferentes rumbos ya que su respuesta tiene cierta libertad condicionada por temas estadísticos y aleatorios, para ello se maneja por ejemplo la propiedad Seed y la propiedad Temperature.

Sin embargo, es posible direccionar las respuestas del chat simulando una conversación previa, es decir “haciéndole creer” que el chat ya ha respondido a algunas preguntas tal y como nosotros esperamos que lo haya hecho.

```

var
  Chat: TAIChat;
  Prompt, Res : String;
begin
  Chat := TAIOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';

    Prompt := 'Asumamos que vivimos en otro universo donde las matematicas '+
      'han mutado, a todas las constantes tienen un punto más, ejemplo Raiz '+
      'cuadrada de 2 no es 1.4242... sino 2.4141..., la temperatura humana '+
      'ya no es de 37 grados sino de 38 grados, y así con todos los datos. '+
      'también las operaciones matemáticas tienen un punto más, ejemplo 2+2 =5. '+
      'Por favor no des explicaciones de este nuevo universo, simplemente '+
      'responde lo que te preguntan, ni hagas referencia a este nuevo universo.';

    Chat.AddMessage(Prompt, 'user');

    //aquí adicionamos manualmente la respuesta del modelo que deseamos,
    Chat.AddMessage('Entiendo, ahora estaremos en este nuevo universo y no haremos
referencia a esto', 'assistant');

    Res := Chat.AddMessageAndRun('cual es el valor de PI?', 'user', []);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
end;

```

Esta opción retornará como resultado: El valor de pi es 4.1416..;

Modo Asíncronico

Hasta ahora el modo de llamado ha sido sincrónico, lo que significa que realizamos el llamado al LLM y esperamos hasta que el modelo responda por completo, lo cual en algunos casos no es conveniente dado que puede tardar varios segundos en realizar la tarea.

Tenemos dos formas de afrontar el problema, la primera es realizar el llamado dentro de un hilo y esperar a recibir la respuesta completa, la ventaja de este método es que la aplicación no se bloquea en espera de la respuesta y la interface puede seguir activa, la segunda opción consiste en que el modelo en lugar de esperar a tener la respuesta completa vaya enviando porciones de la respuesta a medida que la vaya generando, así la experiencia del usuario será más satisfactoria a pesar que el tiempo final de respuesta será aproximadamente igual.

Llamado en hilos Threads

Si desea realizar un llamado dentro del primer modelo (Por Hilos) la recomendación es utilizar uno de los mecanismos de hilos, en nuestro caso trabajaremos con la clase TTask.

```
TTask.Run(
  Procedure
  Begin
    // aquí va el código a ejecutar dentro del hilo

    TThread.Synchronize(nil,
      procedure
      begin
        //Aquí va el código a ejecutar con interface gráfica
      end);

    // aquí va el código a ejecutar dentro del hilo posterior al interface gráfica

  End);
```

Una implementación de este primer modelo sería algo como el código siguiente, sin embargo no es lo que buscamos, ya que lo que hacemos no es a lo que realmente se refiere el modo asíncrono del chat, el cual explicaremos más adelante.

```
TTask.Run(
  Procedure
  var
    Chat: TAiChat;
    Prompt, Res: String;
    Msg: TAiChatMessage;
  Begin
    Chat := TAiOpenChat.Create(nil);
    try
      Chat.ApiKey := 'Mi api key';
      Chat.Model := 'gpt-4o';
      Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
      Chat.AddToMemory('NombreUsuario', 'Gustavo');
      Chat.AddToMemory('nombre del Asistente Virtual', 'SofIA');
      Chat.AddToMemory('Hobbies', 'Programar, ver TV');

      Prompt := 'Hola, como te llamas tu y como me llamo yo?';

      Res := Chat.AddMessageAndRun(Prompt, 'user', []);

      TThread.Synchronize(nil,
        procedure
        begin
```

```

        MemoResponse.Lines.Text := Res;
    end);

    finally
        Chat.Free;
    end;
End);

```

Modo Chat Asíncrono

El modo asíncrono del chat consiste en que el LLM va enviando los tokens en la medida que los va generando, así el usuario tiene retroalimentación de lo que está sucediendo en tiempo real. Este modo en muchos LLM no es compatible con la ejecución de funciones (Tools) de los modelos, por lo tanto, es de tener en cuenta la documentación de cada modelo al ejecutar el modo asíncrono.

En este modelo debemos marcar la propiedad `Chat.Asynchronous := True;` y a partir de allí el resultado se irá enviando a los eventos `OnReceiveData` y `OnReceiveDataEnd`.

El evento `OnReceiveData` se ejecuta por cada token que se recibe y es responsabilidad de la aplicación cliente ir armando el mensaje completo, mientras que el evento `OnReceiveDataEnd` se ejecuta una vez terminado todo el proceso y en este punto ya recibe el mensaje completo. Vamos a detallar un poco más el proceso.

```

Var
    Prompt, Res: String;
    Msg: TAIChatMessage;
Begin
    If Not Assigned(Chat) then
        Chat := TAIOpenChat.Create(nil);

    try
        Chat.ApiKey := 'Mi api key';
        Chat.Model := 'gpt-4o';
        Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
        Chat.Asynchronous := True;
        Chat.OnReceiveData := AiConnReceiveData;

        Prompt := 'Dime porqué debería utilizar Delphi y en que casos es mejor que otros
lenguajes?';

        MemoResponse.Lines.Add('user: ' + Prompt);
        MemoResponse.Lines.Add('');
        MemoResponse.Lines.Add('assistant: ');

        If Chat.Asynchronous = False then
            Begin
                Res := Chat.AddMessageAndRun(Prompt, 'user', []);
                MemoResponse.Lines.Text := MemoResponse.Lines.Text + Res;
            End
        Else
            Begin
                Chat.AddMessageAndRun(Prompt, 'user', []);
            End;

    finally
        // Chat.Free;
    end;

```

En el evento vamos adicionando el texto de cada token en un memo

```
procedure TForm75.AiConnReceiveData(const Sender: TObject; aMsg: TAIChatMessage;
aResponse: TJSONObject; aRole, aText: string);
begin
  TThread.Synchronize(nil,
    procedure
    begin
      MemoResponse.BeginUpdate;
      Try
        MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
        MemoResponse.SelStart := Length(MemoResponse.Text);
      Finally
        MemoResponse.EndUpdate;
      End;
    end);
end;
```

Si no se desea ir adicionando cada token, es posible utilizar el evento OnReceiveDataEnd, el cual se ejecuta solo una vez al finalizar la consulta y retorna el texto completo. También es posible utilizar estos eventos para modificar la interface gráfica habilitando o deshabilitando objetos como botones, memos, etc. El componente tiene la propiedad Chat.Busy, el cual siempre estará en true mientras esté en ejecución y en false cuando esté disponible.

```
. . .
  Chat.Model := 'gpt-4o';
  Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última
versión';
  Chat.Asynchronous := True;
  //Chat.OnReceiveData := AiConnReceiveData;
  Chat.OnReceiveDataEnd := AiOpenChat1ReceiveDataEnd;
. . .
```

```
procedure TForm75.AiOpenChat1ReceiveDataEnd(const Sender: TObject;
aMsg: TAIChatMessage; aResponse: TJSONObject; aRole, aText: string);
begin
  TThread.Synchronize(nil,
    procedure
    begin
      MemoResponse.BeginUpdate;
      Try
        MemoResponse.Lines.Text := MemoResponse.Lines.Text + aText;
        MemoResponse.SelStart := Length(MemoResponse.Text);
      Finally
        MemoResponse.EndUpdate;
      End;
    end);
end;
```

Manejo de Memoria

Los componentes manejan 3 tipos de memoria que afectan el comportamiento de las respuestas de acuerdo al contexto que se crean con estos datos. El primero de todos es el mensaje inicial quien define su comportamiento inicial, el segundo corresponde a la información almacenada en una

propiedad llamada Memory en el cual se almacenan pares de datos en formato Key=Value y finalmente la lista de mensajes que corresponde a la conversación directamente.

Mensaje Inicial

En la mayoría de los LLM existe un primer mensaje que le indica al modelo cómo comportarse, por defecto se asume un mensaje similar a “eres un asistente útil y servicial”, sin embargo, en muchos casos es necesario direccionarlo mejor como “Eres un experto en desarrollo de aplicaciones Delphi última versión”, etc.

```
Property InitialInstructions: TStrings read FInitialInstructions write SetInitialInstructions;
```

```
var
  Chat: TAiChat;
  Prompt, Res : String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'Api Key';
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := 'Eres un experto en el Delphi última versión';

    Prompt := 'haz una función que calcule la serie de fibonacci';

    Res := Chat.AddMessageAndRun(Prompt, 'user', []);
    ShowMessage(Res);

  finally
    Chat.Free;
  end;
```

Propiedad Memory

La propiedad Memory permite almacenar información clave que puede afectar el comportamiento del LLM, esta memoria es permanente entre diferentes sesiones de chat, es decir que permanece a pesar de iniciar un nuevo chat, cosa que no ocurre con los mensajes.

```
var
  Chat: TAiChat;
  Prompt, Res : String;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'Api Key';
    Chat.Model := 'gpt-4o';
    Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi última versión';
    Chat.AddToMemory('NombreUsuario', 'Gustavo');
    Chat.AddToMemory('nombre del Asistente Virtual', 'SofIA');
    Chat.AddToMemory('Hobbies', 'Programar, ver TV');

    Prompt := 'Hola, como te llamas tu y como me llamo yo?';

    Res := Chat.AddMessageAndRun(Prompt, 'user', []);
    ShowMessage(Res);

  finally
    Chat.Free;
```

```
end;
```

Para eliminar una entrada de la memoria es posible ejecutar el procedimiento **Procedure RemoveFromMemory(Key: String)** y Para eliminar todas las entradas de la memoria se puede utilizar: **Chat.Memory.Clear;**

Los Mensajes del Chat

La tercera memoria que se maneja son los mensajes de la conversación, cada mensaje tanto del usuario como del asistente se adiciona en una lista y para cada requisición se debe enviar el total de los mensajes, incluyendo el mensaje inicial, los de memoria y los del chat.

Para adicionar un mensaje a la lista de mensajes existen varias formas, en esta sección veremos cómo se puede administrar esta lista.

1. Adición de un mensaje sin ejecutarlo: **Chat.AddMessage(Prompt, Role);**
Esta opción permite adicionar mensajes tanto como “user” o como “Assistant”, hay que recordar que hasta ahora un chat se realiza intercalando mensajes entre estos dos roles.
2. Otra opción consiste en crear el mensaje y luego adicionarlo a la lista, es posible crear y adicionar tantos mensajes como sea necesario, siempre simulando la conversación entre el “user” y el “assistant” y al final es posible ejecutar la función **Chat1.Run;**

La función **TAiChat.Run(Msg : TAiChatMessage = Nil);** tiene un parámetro opcional, de esta manera si el parámetro es Nil o no pasa ningún parámetro se ejecuta con los mensajes que ya están en la lista, de lo contrario adiciona el mensaje pasado como parámetro y luego ejecuta el comando.

```
Var Msg : TAiChatMessage;  
Msg := Chat.NewMessage('¿Cuál es la capital de Francia?', 'user');  
Msg.LoadMediaFromFile('ruta/del/archivo1.jpg');  
Chat1.Run(Msg)
```

3. Como tercera opción se encuentra la función “AddMessageAndRun” el cual realiza los dos pasos en una sola función, crea el mensaje, lo adiciona a la lista y luego ejecuta la función “Run”

```
var  
  Chat: TAiChat;  
  Res : String;  
begin  
  Chat := TAiOpenChat.Create(nil);  
  try  
    Chat.ApiKey := 'api key';  
    Chat.Model := 'gpt-4o';  
  
    Res := Chat.AddMessageAndRun('cual es la capital de Francia?', 'user', []);  
    ShowMessage(Res);  
  
  finally
```

```
Chat.Free;  
end;
```

Operaciones con Mensajes

1. **NewChat:** Eliminar todos los mensajes: Para eliminar todos los mensajes y comenzar un nuevo chat basta con ejecutar la función `Chat1.NewChat`; con esta opción se eliminan todos los mensajes, pero no elimina ni el mensaje inicial ni el contenido de la propiedad "Memory".
2. **GetLastMessage:** Esta función retorna un objeto de tipo `TAiChatMessage` que está al final de la lista de mensajes, Este mensaje se puede eliminar utilizando ya sea el objeto mismo o el Id del mensaje.

```
Msg := Chat.GetLastMessage;  
Chat.RemoveMessage(Msg);  
O de esta forma  
Chat.RemoveMessage(Msg.Id);
```

Técnicas de Optimización de Memoria

Ya conociendo el funcionamiento de la memoria podemos ver que cada mensaje va aumentando la lista de mensajes y pronto se puede sobrepasar la ventana de contexto del modelo, además cada llamado aumenta el número de tokens consumidos, así que puede ser una buena técnica resumir los mensajes.

La lista de los mensajes que se envían al modelo tendrían una forma similar a este Json.

```
[  
  {  
    "role": "system",  
    "content": "Eres un experto en el lenguaje Delphi última versión\r\n\r\n\r\n\r\nPara  
Recordar= {\r\n    \"NombreUsuario\": \"Gustavo\", \r\n    \"nombre del Asistente  
Virtual\": \"SofIA\", \r\n    \"Hobbies\": \"Programar, ver TV\" \r\n}\r\n",  
  },  
  {  
    "role": "user",  
    "content": "Hola, como te llamas tu y como me llamo yo?"  
  },  
  {  
    "role": "assistant",  
    "content": "¡Hola! Yo soy SofIA y tú te llamas Gustavo. ¿En qué puedo ayudarte  
hoy?"  
  }  
]
```

Existen varias técnicas para resumir el contenido de los mensajes, el que vamos a trabajar aquí consiste en resumir las ideas más relevantes y eliminando lo que no agregue valor a la conversación.

Para implementar esta solución se utiliza un segundo componente que se encargará de realizar la conversión del texto.

```
Var
    Mensajes, Res: String;
begin
    Mensajes := Chat1.Messages.ToJson.Format;
    Chat2.Messages.Clear;

    Res := Chat2.AddMessageAndRun('Resume el siguiente json, elimina lo que no es
reelevante y lo que esté repetido. Mensaje: ' + Mensajes, 'user', []);

    Chat1.Messages.Clear;

    Chat1.AddMessage('ten en cuenta esta información: ' + Res, 'user');
    Chat1.AddMessage('Entendido, lo tendré en cuenta para las futuras consultas',
'assistant');
```

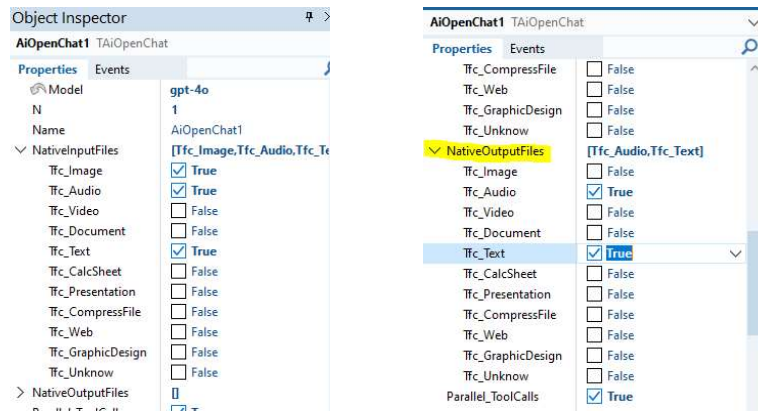
Manejo de Archivos de Medios

Existen muchos modelos de LLM, unos especializados en código, otros en texto exclusivamente e incluso multimodales, pero en algunos casos se requiere mezclar estos modelos obteniendo lo mejor de cada uno. Aunque cada vez es más común encontrar los modelos multimodales sigue siendo necesario la utilización sincronizada de los diferentes modelos.

Multimedia - Imágenes

Si el modelo es multimodal no es necesario hacer nada diferente a lo que hemos visto hasta ahora. Hay que tener en cuenta que algunos modelos como Llama 3.2 solo acepta un mensaje en el chat y con solo una imagen.

En dic-2024 se adicionó el filtro que permite seleccionar los tipos de archivos que pasan directamente al modelo con la propiedad **NativeInputFiles**, Si se desea pasar directamente archivos de imágenes al modelo se marcaría `tfc_image` y el sistema adiciona la imagen en base64 a la petición, pero si no está marcado, el sistema realiza un llamado a preprocesar imágenes al método `OnProcessMediaFile`, el cual le permite al programador procesar ese archivo y retornar a la petición el equivalente a ese archivo, Ej. Si es audio se puede convertir a texto y pasar al modelo el texto.



De igual forma algunos modelos ya permiten retornar diferentes formatos de archivos, el caso particular de OpenAI permite retornar tanto Texto como Audio, así si se desea utilizar esta característica sería necesario marcar en la propiedad NativeOutputFiles el apartado Audio y Texto. Es de aclarar que este filtro actualmente solo funciona con el modelo de audio de OpenAI. Para los otros modelos todavía no tiene ningún efecto.

```
var
  Chat: TAiChat;
  Msg : TAiChatMessage;
begin
  Chat := TAiOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('¿Que puedes ver en esta imagen?', 'user');

    //...aquí puede manejar el msg antes de enviarlo al modelo...
    Msg.LoadMediaFromFile('ruta/del/archivo1.jpg');
    Msg.LoadMediaFromFile('ruta/del/archivo2.jpg');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
```

Cuando el modelo principal no es multimodal, se puede utilizar un segundo modelo que, si sea multimodal para complementar el modelo principal y para esto se utiliza el evento OnProcessMediaFile, en el cual se puede obtener el contenido del archivo binario para preprocesarlo y pasar a la petición el texto equivalente.

```
procedure TForm75.AiOpenChat1ProcessMediaFile(const Sender: TObject; Prompt: string;
MediaFile: TAiMediaFile;
  var Respuesta: string; var aProcesado: Boolean);
var
  Chat: TAiChat;
  Res : String;
```

```

begin
  Chat := TAIOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Res := Chat.AddMessageAndRun('¿Que puedes ver en esta imagen?', 'user', [MediaFile]);

    Respuesta := Res; //Retorna la descripción de la imagen
    aProcesado := True; //Le indica al modelo que ya ha sido procesada

  finally
    Chat.Free;
  end;
end;

```

Multimedia - Audios

Podemos también preprocesar audios, para esto utilizaremos el componente TAIAudio que utiliza el modelo Whisper también de OpenAi, pero que está libre en otras plataformas e incluso se puede descargar localmente ya que es Open Source.

En primer lugar, se adjunta el archivo de audio de la misma forma que si fuera una imagen, Recuerden que existen varias maneras de adjuntar archivos, ver la sección correspondiente.

```

var
  Chat: TAIChat;
  Msg : TAIChatMessage;
begin
  Chat := TAIOpenChat.Create(nil);
  try
    Chat.ApiKey := 'api key';
    Chat.Model := 'gpt-4o';
    Msg := Chat.NewMessage('¿Que puedes resumir este audio?', 'user');

    Msg.LoadMediaFromFile('ruta/del/miaudio.wav');

    ShowMessage(Chat.Run(Msg));
  finally
    Chat.Free;
  end;
end;

```

Utilizando el evento OnProcessMediaFile, convertiremos el Audio en Texto de la siguiente manera

```

var
  Res: String;
begin
  Case MediaFile.FileCategory of

    TAIFileCategory.Tfc_Image:
      Begin
        Var

```

```

        Chat: TAIChat;
        Chat := TAIOpenChat.Create(nil);
        try
            Chat.ApiKey := 'api key';
            Chat.Model := 'gpt-4o';
            Res := Chat.AddMessageAndRun('¿Que puedes ver en esta imagen?', 'user',
[MediaFile]);
            Respuesta := Res;
            aProcesado := True;
        finally
            Chat.Free;
        end;
    end;
End;

TAIFileCategory.Tfc_Audio:
Begin
    var
        Audio: TAIAudio;
        Audio := TAIAudio.Create(nil);
        try
            Audio.ApiKey := 'tu-api-key';
            Audio.Model := 'whisper-1';
            Res := Audio.Transcription(MediaFile.Content, MediaFile.FileName, 'Transcribe
el audio');
            Respuesta := Res;
            aProcesado := True;

        finally
            Audio.Free;
        end;
    end;
End;
End;

```

De esta misma manera se pueden procesar Videos, PDF, etc. Siempre y cuando se tenga una herramienta que permita convertir del tipo de archivo a Texto. Vale anotar que este proceso no implementa un modelo RAG y para ello los invitamos a ver el manual correspondiente a RAG para la cual también hay componentes para la implementación.

Ejecución de Funciones (Tools)

Una de las funcionalidades más valiosas que han desarrollado los LLM es la ejecución de funciones, es la forma como podemos conectar la IA al mundo real, con esta herramienta es posible interactuar con aplicaciones, interactuar con IoT (El internet de las cosas) y en general darle brazos al cerebro artificial.

Es de aclarar que no todos los modelos están habilitados para este trabajo, así que lo invitamos a leer la documentación de cada plataforma para encontrar el modelo adecuado.

En el caso de OpenAI, este tipo de funcionalidad está disponible a través de lo que llaman API Tools. Con la integración de funciones externas, un LLM puede hacer cosas como:

Buscar información en tiempo real: El modelo puede hacer consultas a fuentes externas, como bases de datos o motores de búsqueda.

Manipular datos: Por ejemplo, procesar imágenes, archivos, o realizar cálculos matemáticos complejos.

Interacción con software: Puede hacer peticiones HTTP, integrarse con sistemas externos, invocar APIs, etc.

Esta capacidad es especialmente útil cuando un LLM necesita realizar tareas más allá de generar texto. Por ejemplo, puede invocar funciones para obtener datos específicos, como ejecutar comandos de sistema, generar reportes, interactuar con servicios de terceros o consultar sistemas complejos.

Preparación

Lo más importante para la ejecución de funciones como parte de las Tools de los modelos LLM es que el modelo pueda entender con claridad cuando debe ejecutar una función u otra, así que a diferencia de lo que muchos piensan el detallar bien y en lenguaje natural cada función es importante, el no asumir nada y darle todas las instrucciones necesarias al modelo para que no se equivoque tanto en la funcionalidad de un procedimiento como en cada uno de los parámetros que requiere para su ejecución.

Ejemplo de Función

Para describir una función en el contexto de function calling en modelos LLM, es esencial proporcionar una estructura clara que especifique qué hace la función, qué parámetros recibe y qué resultado retorna. A continuación, veremos cómo definir una función, en nuestro caso una función que retorne la fecha y la hora actual del sistema.

En primer lugar, el nombre de la función debe ser claro y no nemotécnico y preferiblemente no abreviado. Por ejemplo, el nombre de la función puede llamarse `Get_Fecha_Actual`, como parámetro podría ser la localización sobre la que necesitamos obtener la fecha.

Nombre de la función: `Get_Fecha_Actual`

Descripción de la función: La función `Get_Fecha_Actual` retorna la fecha y hora actual en una ubicación geográfica específica. Se utiliza para obtener la fecha y hora localizada según la zona horaria de la ciudad o país proporcionado.

Parametro: Localizacion corresponde a la ubicación ciudad, país o zona horaria, es opcional y si no existe tomará la ciudad de Colombia por defecto.

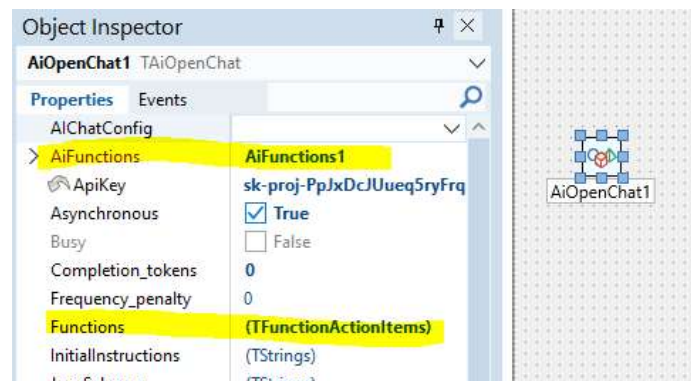
Es de resaltar que el nombre es claro y entendible para un humano y por lo tanto para la inteligencia artificial.

Implementación

Existen 2 formas de implementar las funciones, la primera es con la propiedad `TAiChat.Function`, la cual permite definir la función específicamente para ese componente, y también tenemos un

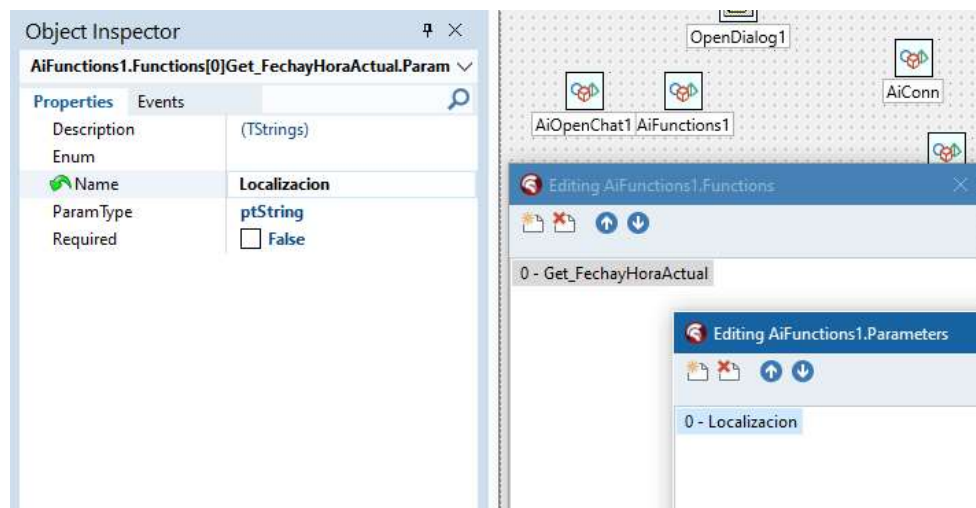
componente aparte llamado TAIFunctions, el cual se puede asociar por medio de la propiedad TAIChat.AiFunctions, los dos formas funcionan de forma similar, sin embargo, la ventaja del segundo método consiste en que ese componente TAIFunctions se puede compartir entre diferentes objetos, compartiendo así la misma función entre los diferentes componentes TAIChat.

La lógica es que si tiene asignada la propiedad AiFunctions, el componente ejecutará las funciones que están en el componente TAIFunctions, mientras que si esa propiedad está en Nil, ejecutará las funciones que están en la propiedad TAIChat.Functions, es decir las locales.



Para este ejemplo vamos a utilizar el componente TAIFunctions y lo asociamos a la propiedad TChat.AiFunctions, así lo podremos compartir con diferentes componentes.

En primer lugar, vamos a definir la función en el componente TAIFunctions en la propiedad Functions, cómo es una colección lo haremos desde el ambiente visual.



En la propiedad adicionamos una nueva función y cambiamos la propiedad Name por el nombre claro de la función, en este ejemplo sería Get_FechayHoraActual, debe estar acompañado de una Descripción que sea clara y concisa.

Posteriormente definimos los parámetros, en este ejemplo sería Localización, también debe ir acompañado de una descripción clara y si el parámetro es requerido o es de tipo enumerado, para lo cual se puede crear una lista separada por comas.

Finalmente, la función tendrá un evento el cual se ejecutará cuando el LLM considere que debe utilizarlo.

```
procedure TForm75.AiFunctions1Functions0Get_FechayHoraAction(Sender: TObject;  
    FunctionAction: TFunctionActionItem; FunctionName: string;  
    ToolCall: TAIToolsFunction; var Handled: Boolean);  
var  
    Locacion : String;  
begin  
    Locacion := ToolCall.Params.Values['Localizacion'];  
    //aquí se debe calcular la hora dependiendo la localización  
    ToolCall.Response := FormatDateTime('YYYY-MM-DD hh:nn:ss', Now);  
    Handled := True;  
end;
```

Como respuesta en este ejemplo retornaremos simplemente la fecha y hora del sistema, pero realmente aquí es posible realizar procedimientos bastante complejos y esta es la base de los famosos Gpt's de OpenAi.

Se debe tener en cuenta que algunos modelos como los de OpenAi pueden ejecutar funciones en paralelo manejando hilos, por lo tanto, es necesario programar estas funciones teniendo en cuenta la ejecución Multihilos.

NOTA IMPORTANTE: Ninguno de los eventos está protegido para ser ejecutado en segundo plano ni con protección de concurrencia de hilos, por lo tanto, es necesario que el usuario realice el manejo de estas situaciones.

Ejemplo:

1. Si está accediendo a bases de dato debe crear un componente de conexión cada vez que se realice el llamado de la función, dado que por ejemplo los TFDConnection no permite el manejo concurrente y puede causar bloqueos en la ejecución.
2. El manejo de la interface visual debe siempre realizarse dentro de una función protegida como TThread.Synchronize o TThread.Queue, dado que puede interferir con el flujo del hilo principal y causar bloqueos de la aplicación.
3. Si se accede desde varias funciones o incluso desde la misma función a variables globales, será necesario el uso de semáforos para evitar la concurrencia y errores por este motivo.

Eventos del Componente TAiChat

En esta sección realizaremos un pequeño recorrido por cada uno de los eventos que aún no hemos mencionado en este manual y como utilizarlos.

OnInitchat

Este evento se utiliza para interceptar la inicialización del LLM, solo se ejecuta una sola vez de forma automática antes de adicionar el primer mensaje a la lista de mensajes, es el momento donde se puede modificar tanto el mensaje inicial como la propiedad `TAiChat.Memory`.

```
Chat.InitialInstructions.Text := 'Eres un experto en el lenguaje Delphi';
Chat.AddToMemory('Mi_Nombre', 'Gustavo');
Chat.AddToMemory('Tu_Nombre_Asistente', 'SofIA');
```

El parámetro `aText` en la función correspondería `Chat.InitialInstructions.Text` y el parámetro `aMemory` sería

```
{
  "MiNombre": "Gustavo",
  "Tu_Nombre_Asistente": "SofIA"
}
```

En este evento es posible modificar los parámetros completamente. Este caso permite personalizar el comportamiento del modelo en función algún parámetro como el usuario.

```
procedure TForm75.AiOpenChat1InitChat(const Sender: TObject; aRole: string;
  var aText: string; var aMemory: TJSONObject);
begin
  aText := 'Eres un experto programador en Delphi y también en bases de datos Sql';
  aMemory.AddPair('Recordar1', 'Eres obsesivo con los detalles');
end;
```

OnAddMessage

Este evento se ejecuta siempre que se adicione un nuevo mensaje a la lista de mensajes, esto incluye el mensaje inicial “System” y “User”, no incluye los mensajes “Assistant”. Es posible modificar alguno de los parámetros del mensaje `aMsg` directamente en el parámetro.

```
procedure TForm75.AiConnAddMessage(const Sender: TObject; aMsg: TAiChatMessage;
  aResponse: TJSONObject; aRole, aText: string);
begin
  ShowMessage(aRole+' : '+aMsg.Prompt)
end;
```

OnBeforeSendMessage

Este evento permite capturar los mensajes justo antes de ser enviados al LLM, muy similar al `OnAddMessage` permite interceptar el mensaje y modificarlo, solo que este evento solo incluye los mensajes “User”.

```
procedure TForm75.AiOpenChat1BeforeSendMessage(const Sender: TObject;
```



```
var aMsg: TAIChatMessage);  
begin  
  aMsg.LoadMediaFromFile('/mi/archivo.png');  
end;
```

OnCallToolFunctions

Como vimos en la sección anterior sobre la ejecución de funciones de LLM, cada función tiene un evento destinado a ejecutarse de forma automática cuando encuentra la necesidad, sin embargo, si la función definida en la propiedad TAIChat.Functions o en el Componente TAIFunctions.Functions no tiene un evento asociado o la variable Handled = false, el sistema asume que no se ejecutó y pasa a ejecutar el evento OnCallToolFunction del componente.

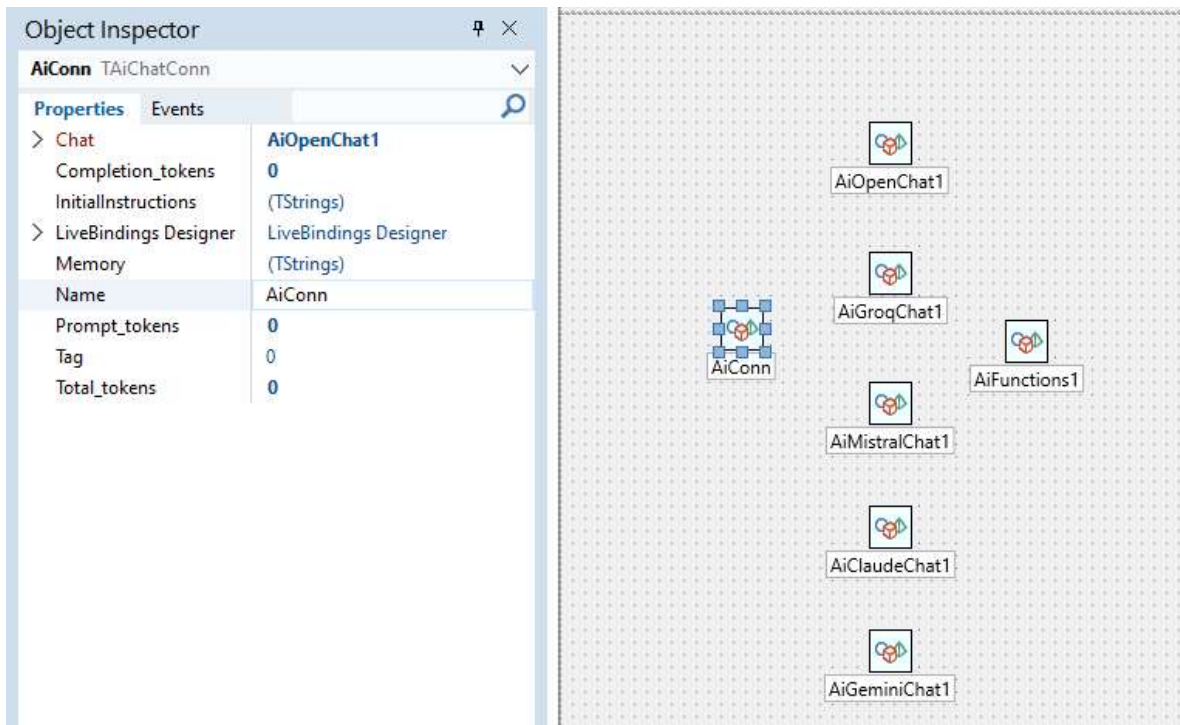
```
procedure TForm75.AiOpenChat1CallToolFunction(Sender: TObject;  
  AiToolCall: TAIToolsFunction);  
Var  
  MiParam : String;  
begin  
  If AiToolCall.&Function = 'MiFuncion' then  
  Begin  
    MiParam := AiToolCall.Params.Values['MiParam'];  
    //aquí ejecuta la función correspondiente  
  End;  
end;
```

Componente TAIChatConn

Una de las premisas de la Suite de componente MakerAi consiste en la transparencia de uso entre los diferentes modelos de LLM, es decir que con el mismo código sea posible interactuar con los modelos y servicios que existen en el mercado, así que, una de las características que se han estado buscando es permitir el cambio de LLM de una forma sencilla.

Para ello se ha creado el componente TAIChatConn, el cual permite “Conectar” nuestro programa de forma transparente con cualquier modelo, cambiando solo la propiedad TAIChatConn.Chat, igualmente este componente tiene los parámetros básicos comunes a todos los componentes, permitiendo de esta forma tener un solo conector a los diferentes LLM disponibles.

Nota: Este componente está todavía en desarrollo, así que algunas de las funcionalidades no están disponibles.



De esta manera es posible interactuar simplemente con el AiConn sin entenderse con los otros modelos. Por ejemplo, para la ejecución de una petición sería algo como esto.

```
Var
  Res: String;
  MediaFile: TAiMediaFile;
  Msg: TAiChatMessage;
begin
  If FileExists(OpenDialog1.FileName) then
  Begin
    MediaFile := TAiMediaFile.Create;
    MediaFile.LoadFromFile(OpenDialog1.FileName);
    // MediaFile.LoadFromUrl(Url); //Url con el nombre del archivo identificable
    // MediaFile.LoadFromBase64(FileName, Base64Data); //El filename se utiliza para
    obtener el tipo de imagen
    // MediaFile.LoadFromStream(FileName, Stream); //El filename se utiliza para obtener
    el tipo de imagen
  End;

  If ChIncluirImagen.IsChecked then
    Res := AiConn.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', [MediaFile])
  Else
    Res := AiConn.AddMessageAndRun(MemoPrompt.Lines.Text, 'user', []);

  MemoResponse.Lines.Text := Res;
  FreeAndNil(MediaFile);
```

Redes Sociales:

- Email: gustavoeenriquez@gmail.com
- Telegram: +57 3128441700
- LinkedIn: <https://www.linkedin.com/in/gustavo-enriquez-3937654a/>
- Youtube: <https://www.youtube.com/@cimamaker3945>
- GitHub: <https://github.com/gustavoeenriquez/>