
sgcChat

Getting Started

Put a modern chat window on a Delphi form, from two bubbles to a streaming AI assistant, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want to add a chat to it with sgcChat. The chat can be a conversation between people, a help desk window, a bot, or an AI assistant that writes its answers word by word. You do not need to know anything about chat protocols or AI models in advance.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from a window with four messages to a small assistant application with streamed answers, a Stop button, and saving and loading of the conversation. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, type a few messages, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcChat is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\Chat\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloChat` to `05.ChatAssistantApp`, with a Delphi project (`.dpr`) that you can open and run directly. All of them are VCL applications for Windows, and all of them run offline: no network, no server and no API key.

The screenshots were taken on a Windows machine with Spanish regional settings. Dates, times and file sizes follow the Windows regional settings of the machine that runs the demo, so on your machine the date separator may read 10/25 instead of 25/10, and a file size may show a dot where the figures show a comma.

Contents

1. What sgcChat is

The idea in one minute
Your first chat needs four lines of code
What you need
Part of sgcUI
sgcChat and sgcWebSockets
Coming from memos and list boxes

2. Install and the palette

Installing
The packages
The palette pages
Editions
Where the demos are
Running your first demo

3. Core concepts

The conversation holds the data
Users and the current user
Messages and what they show
Views follow the conversation
Text is Markdown
Sending: OnSend, AI or a transport
AI answers stream on a worker thread
Themes and language
Saving a conversation
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello Chat
4.2 Demo 02: Chat Bubbles
4.3 Demo 03: Chat Echo Bot
4.4 Demo 04: Chat Themes
4.5 Demo 05: Chat Assistant App

5. Main components

Reference table
The controls
The data model
The AI binding and its providers
Transports and the sgcWebSockets bridge

6. Common problems

Sending and showing messages
AI answers
Layout, links and language

7. Where next

The other Delphi demos
Plugging in a real AI provider
Help topics
On the website
Getting help

1. What sgcChat is

sgcChat adds chat windows to Delphi applications, for the VCL and for FireMonkey. It draws a conversation the way people know it from the messaging apps on their phones: bubbles on the right for your own messages and on the left for the others, avatars with a presence dot, timestamps, read ticks, replies, reactions, attachments and a “is typing” indicator. Every message is Markdown, so bold text, lists, tables and code blocks with syntax colours and a Copy button work in every bubble.

The same controls serve two very different kinds of chat. In a **human chat**, people talk to each other: a support window, a team chat, the messages of a ticket. In an **AI chat**, a person talks to a language model, and the answer appears piece by piece while the model writes it. sgcChat has a component for each side: a transport for the human chat and an AI binding for the assistant, and both write into the same conversation that the controls show.

This guide is for Delphi developers who are new to sgcChat. By the end you will know how the pieces fit together, which properties to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

Every sgcChat application is built from two kinds of component:

- **The data.** A `TsgcChatConversation` is a non visual component that holds the users, the messages and the id of the user who runs the application. It knows nothing about the screen.
- **The views.** A `TsgcVCLChatView` shows a conversation as bubbles, a `TsgcVCLChatInput` is the bar where the user types, and a `TsgcVCLChatPanel` combines a header, a view and an input in one control. You point their `Conversation` property at the data.

Your code changes only the data: it adds a message, sets the status of a message to read, marks a user as typing. Every view that shows the conversation repaints by itself. That is the entire idea; the rest of sgcChat is about where the messages come from (your code, an AI model or a chat server) and how they look.

Your first chat needs four lines of code

The first QuickStart demo is the smallest chat there is. Two components are placed on the form at design time: a conversation with two users and a chat view bound to it. This is how they are stored in the form file:

```

object ChatView: TsgcVCLChatView
    Align = alClient
    Conversation = Conversation
end
object Conversation: TsgcChatConversation
    Users = <
        item
            Id = 'me'
            Name = 'You'
            Presence = cpOnline
        end
        item
            Id = 'ana'
            Name = 'Ana'
            Presence = cpOnline
        end>
    CurrentUserId = 'me'
    Title = 'Hello Chat'
end

```

The code adds four messages in the `OnCreate` event of the form:

```

procedure TfrmHelloChat.FormCreate(Sender: TObject);
begin
    Conversation.Messages.Add(cmrUser, 'ana',
        'Hi! Welcome to **sgcChat**.');
    Conversation.Messages.Add(cmrUser, 'me',
        'Hello Ana. Is this really all it takes?');
    Conversation.Messages.Add(cmrUser, 'ana',
        'Yes: a `TsgcChatConversation` holds the messages and a ' +
        '`TsgcVCLChatView` shows them.');
```

```

    Conversation.Messages.Add(cmrUser, 'me', 'Great, let's build a chat!');
end;

```

`Messages.Add` takes the role of the message (`cmrUser` for a person), the id of the author and the text. The messages of `me`, the current user, appear on the right; those of `ana` on the left, with an avatar made of her initials. Run it and you have a chat window. Section 4.1 walks through this demo in detail.

What you need

- **Delphi.** The packages in the `Packages` folder cover Delphi 7, 2007, 2009, 2010, XE to XE8, 10 Seattle to 10.4 Sydney, 11, 12 and 13.
- **VCL.** `TsgcVCLChatView`, `TsgcVCLChatInput`, `TsgcVCLChatPanel` and `TsgcVCLMarkdownViewer` run on Windows, Win32 in every version and Win64 from Delphi XE2.
- **FireMonkey.** The `TsgcFMX` twins of the same four controls have their own packages from Delphi XE6 on, for the platforms FireMonkey targets in your Delphi version.
- **No window at all.** The model, the Markdown document and the AI and transport cores use no visual framework. From Delphi 10.2 on they also compile for Linux64, without controls.
- **sgcUICore and sgcRichEdit.** `sgcUICore` is the shared foundation of the `sgcUI` visual products, and `sgcRichEdit` holds the Markdown engine that draws the text of every bubble. The installer installs them together with `sgcChat`, you do not install them separately.

The QuickStart demos are VCL applications, and they use only units of `sgcUI`. You do not need `sgcWebSockets` to build or run any of them.

Part of sgcUI

sgcChat is one product of sgcUI, the eSeGeCe suite of visual components (grids, charts, dashboards, planners, rich text and more). All of them draw through sgcUICore, the shared layer with the canvas, the fonts and the light and dark themes. In practice this means two things for you:

- A chat looks the same on the VCL and on FireMonkey, and it matches the other sgcUI controls on your form. The `Theme` property (`utDefault` , `utLight` , `utDark` , `utCustom` , `utHighContrast`) works the same way on all of them, and one call switches every sgcUI control of the application between light and dark. Demo 04 shows it.
- The texts of the controls, such as the placeholder of the input or the “is typing” line, come from the sgcUI language layer, so the chat speaks the same language as the rest of your sgcUI controls.

sgcChat and sgcWebSockets

sgcChat draws the chat and keeps the conversation. It does not, by itself, talk to the internet. Two things need sgcWebSockets, the eSeGeCe networking library:

- **The LLM providers.** Talking to OpenAI, Anthropic, Gemini, DeepSeek, Ollama, Grok or Mistral goes through `TsgcChatAIProvider_AIChat` , whose `Chat` property is a `TsgcAIChat` of sgcWebSockets.
- **The WebSocket team chat.** `TsgcChatTransport_WebSocket` connects a chat panel to a channel on a server, and `TsgcChatServer_WebSocket` is the relay server. Both are built on the WebSocket components of sgcWebSockets.

These components live in a separate bridge package, `sgcChatWS` , on their own palette page. Without sgcWebSockets, sgcChat still works with `TsgcChatAIEventProvider` , where your own code produces the answer (demo 05 does exactly this), and with `TsgcChatEventTransport` , which connects a chat to any backend you already have.

Coming from memos and list boxes

Many Delphi chat windows start as a `TMemo` for the history and a `TEdit` for the input. That works until users expect bubbles, avatars, read ticks, code blocks or answers that appear word by word. With sgcChat you never format text into a memo: you add a message object, and the view decides how to draw it. Changing a message later, for example its status from sent to read, or the text of an answer that is still streaming, changes the bubble on the screen.

If you know `TListView` or virtual list controls, the idea is similar: the data lives apart from the control, and the control measures and paints only what is visible, so a long conversation scrolls fast.

2. Install and the palette

This chapter gets sgcChat onto your Tool Palette and the first demo running. If the components are already installed, skim the palette section and jump to [Running your first demo](#).

Installing

sgcChat is installed by the sgcUI installer, the same installer for every sgcUI product:

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, select **sgcChat (installs sgcRichEdit and sgcPDF)**. The installer shows one checkbox per product and enables the products your licence includes. As the caption says, selecting sgcChat also installs sgcRichEdit, whose Markdown engine draws the text, and sgcPDF, which sgcRichEdit needs. sgcUICore has no checkbox: it is installed automatically.
4. Optionally select **sgcChat for sgcWebSockets** under the sgcWebSockets bridges. This is the bridge with the LLM provider and the WebSocket team chat. It can only be installed when sgcWebSockets is present, and you do not need it for the QuickStart demos.
5. The installer copies the files, compiles the packages on your machine for each selected Delphi version and installs the design time packages.

Everything goes into one folder, by default `Documents\esegece\sgcUI`. The installer adds its `Source` folder and the folder of the compiled units to the library path of each selected Delphi version, so a new project finds the sgcChat units without any setting on your side.

The packages

You only need this section if you build the packages yourself. sgcChat is split into a framework neutral package and one package per framework, so a FireMonkey application never links the VCL and the other way round. Every package name ends with the suffix of the Delphi version, for example `d13` for Delphi 13 or `d7` for Delphi 7.

Package	Kind	Contains
<code>sgcChat<V></code>	Runtime	The model, the Markdown document, the view and input cores, the AI binding and the transports. No VCL, no FireMonkey.
<code>sgcChatVCL<V></code>	Runtime	The VCL controls, unit <code>sgcChat_VCL</code> .
<code>sgcChatFMX<V></code>	Runtime, XE6 and later	The FireMonkey controls, unit <code>sgcChat_FMX</code> .
<code>dc1sgcChat<V></code>	Design time	Registers the VCL controls and the non visual components on the SGC Chat palette page.
<code>dc1sgcChatFMX<V></code>	Design time, XE6 and later	Registers the FireMonkey controls.
<code>sgcChatWS<V></code> , <code>dc1sgcChatWS<V></code>	Runtime and design time	The sgcWebSockets bridge: the LLM provider and the WebSocket transport and relay server.

Build them in dependency order: `sgcUICore`, `sgcPDF` and `sgcRichEdit` first, then `sgcChat`, then `sgcChatVCL`, then `dc1sgcChat`. Then install the design time package and add `sgcUI\Source` to the library path of your Delphi version (**Tools > Options > Language > Delphi Options > Library**).

The palette pages

After the installation the Tool Palette has a page named **SGC Chat**. In a VCL project it shows these components:

Component	Visual	What it does
<code>TsgcVCLChatPanel</code>	yes	A complete chat window: header, message view and input bar in one control.
<code>TsgcVCLChatView</code>	yes	The list of message bubbles of a conversation.
<code>TsgcVCLChatInput</code>	yes	The input bar: text field, send button (Stop while an answer streams), attach button.
<code>TsgcVCLMarkdownViewer</code>	yes	A read only control that renders a Markdown document, with streaming.
<code>TsgcChatConversation</code>	no	The data: users, messages and the current user.
<code>TsgcChatAI</code>	no	Binds a conversation to an AI provider and streams the answers into it.
<code>TsgcChatAIEventProvider</code>	no	An AI provider whose answers come from your own <code>OnStream</code> event.
<code>TsgcChatAIProvider_UICore</code>	no	Adapts an AI provider of <code>sgcUICore</code> , one chunk per answer.
<code>TsgcChatEventTransport</code>	no	A human chat transport whose frames you send and receive yourself, for any backend.

In a FireMonkey project the same page shows `TsgcFMXChatPanel`, `TsgcFMXChatView`, `TsgcFMXChatInput` and `TsgcFMXMarkdownViewer` instead of the VCL controls, together with the same non visual components.

With the `sgcWebSockets` bridge installed, a second page named **SGC Chat WS** holds `TsgcChatTransport_WebSocket`, `TsgcChatServer_WebSocket` and, when the installed `sgcWebSockets` has its AI components, `TsgcChatAIProvider_AIChat`.

The QuickStart demos use six components of the first page: the panel, the view, the Markdown viewer, the conversation, the AI binding and the event provider.

Editions

`sgcChat` is part of `sgcUI Enterprise` and of `All-Access`. It is not sold as a separate package, and it is not part of the free `Community` edition. A `Trial` edition is available for evaluation.

The LLM providers and the WebSocket team chat also need `sgcWebSockets`, which is included in `All-Access`. To talk to OpenAI, Anthropic, Gemini and the other providers, the installed `sgcWebSockets` must contain its AI components. Everything the QuickStart demos use is in `sgcChat` itself.

Where the demos are

The demos live in `Demos\Chat\Delphi` under your `sgcUI` folder. The QuickStart is the first subfolder:

Folder	Project	What it shows
01.HelloChat	HelloChat.dpr	The smallest chat: a conversation with two users, a chat view and four messages
02.ChatBubbles	ChatBubbles.dpr	Every kind of message data: timestamps, Markdown, a reply, reactions, an attachment, a system notice, read ticks, presence and typing
03.ChatEchoBot	ChatEchoBot.dpr	The chat panel and its <code>OnSend</code> event, with a local bot that types, reads and answers
04.ChatThemes	ChatThemes.dpr	The Markdown viewer and the application theme, Classic or Fluent, light or dark
05.ChatAssistantApp	ChatAssistantApp.dpr	A small AI assistant: streamed answers from an offline provider, with New, Stop, Save and Load

Every demo folder holds the project, its unit and form, and a `readme.html` with a short description. The `readme.html` in `00.QuickStart` itself is the index of the five demos. The other folders in `Demos\Chat\Delphi` are larger demos, listed in chapter 7.

Running your first demo

1. In Delphi, choose **File > Open Project** and open `Demos\Chat\Delphi\00.QuickStart\01.HelloChat\HelloChat.dpr`.
2. Press **F9**. The window opens with four messages.
3. Back in the IDE, open the form, select `Conversation` and click the ellipsis button of `Users` in the Object Inspector. The two items are the people of the chat.

Tip. To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this guide come from that view. Choose **View as Form** to go back to the designer.

3. Core concepts

Nine ideas cover almost everything you do with sgcChat in your first weeks. The QuickStart demos meet all of them; this chapter puts them in one place so that the demos make sense at a glance.

The conversation holds the data

`TsgcChatConversation` (unit `sgcChat_Model`) is the heart of every chat. It is a non visual component with four published properties:

Property	What it holds
<code>Users</code>	The people (and bots) of the chat, a collection of <code>TsgcChatUser</code> .
<code>Messages</code>	The messages in order, a collection of <code>TsgcChatMessage</code> .
<code>CurrentUserId</code>	The id of the user who runs this application.
<code>Title</code>	The name of the conversation, shown by the panel header when the panel has no <code>Title</code> of its own.

Users can be set at design time, as in every QuickStart demo, or in code. This snippet does the work of the two QuickStart components entirely in code:

```
Conversation := TsgcChatConversation.Create(Self);
Conversation.Users.Add('me', 'You');
Conversation.Users.Add('ana', 'Ana');
Conversation.CurrentUserId := 'me';

ChatView := TsgcVCLChatView.Create(Self);
ChatView.Parent := Self;
ChatView.Align := alClient;
ChatView.Conversation := Conversation;

Conversation.Messages.Add(cmrUser, 'ana', 'Created in **code**');
```

When you add many messages at once, put them between `Conversation.BeginUpdate` and `Conversation.EndUpdate`. The views then repaint once, at `EndUpdate`, and not after every message. Demo 02 does this.

Users and the current user

Every user has an `Id`, which messages refer to, and a `Name`, which the bubbles show. The view builds an avatar from the initials of the name. `Presence` (`cpOnline`, `cpAway`, `cpBusy`, `cpOffline`) draws the coloured dot of the avatar, and `IsTyping` shows the "is typing" line below the last message:

```
oUser := Conversation.Users.Add('ben', 'Ben Carter');
oUser.Presence := cpAway;
Conversation.Users.Find('ben').IsTyping := True;
```

`CurrentUserId` decides the side of every bubble. The messages whose `AuthorId` is the current user are drawn on the right, all the others on the left. Change `CurrentUserId` and the whole chat swaps sides, which is exactly what happens when the same conversation is opened by another person.

The view hides a typing indicator by itself after `TypingTimeout` milliseconds, 6000 by default. A value of 0 means never, which demo 02 uses so the indicator stays on the screenshot.

Messages and what they show

`Messages.Add(Role, AuthorId, Text)` creates a `TsgcChatMessage` with a new `Id` and the current time, adds it at the end and returns it, so you can set more properties on it. Each property changes the bubble:

Property	What you see
<code>Role</code>	<code>cmrUser</code> for a person, <code>cmrAssistant</code> for an AI answer, <code>cmrSystem</code> and <code>cmrTool</code> for the rest.
<code>Text</code>	The content, as Markdown.
<code>Timestamp</code>	The time below the bubble, in UTC. Messages of different days get a date separator.
<code>Status</code>	The ticks of the messages of the current user: <code>cmsSending</code> , <code>cmsSent</code> , <code>cmsDelivered</code> , <code>cmsRead</code> , <code>cmsFailed</code> .
<code>ReplyToId</code>	The id of another message: the bubble quotes it.
<code>Reactions</code>	Chips below the bubble, added with <code>AddReaction(Emoji, UserId)</code> .
<code>Attachments</code>	Files: a card with the name and the size.
<code>Kind</code>	<code>cmkSystem</code> draws a centred notice instead of a bubble; <code>cmkError</code> and <code>cmkToolCall</code> are used by the AI binding.

Read receipts have a shortcut. `Conversation.MarkRead(UserId, MessageId)` records that a user has read up to a message and sets every sent or delivered message of the other authors up to it to `cmsRead`:

```
oMessage := Conversation.Messages.Add(cmrUser, 'me', 'Done.');
```

```
oMessage.Status := cmsSent;
```

```
Conversation.MarkRead('ana', oMessage.Id);
```

Demo 02 uses almost every property of the table, and demo 03 uses `MarkRead`.

Views follow the conversation

A view registers itself with the conversation as an observer when you set its `Conversation` property. From then on every change, a new message, a new status, a user who starts typing, reaches the view, which repaints the part that changed. You never call a refresh method, and several views can show the same conversation at the same time.

The look of the bubbles is set in `ChatView.Options`: `ShowAvatars`, `ShowTimestamps`, `GroupMessages` (consecutive messages of one author are drawn as a group), `BubbleMaxWidthPercent` and `ShowActions` (the buttons that appear when the mouse is over a message). A `TsgcVCLChatPanel` has its view as the sub component `view` and its input bar as `Input`, so these properties are found under `ChatPanel1.View.Options` and `ChatPanel1.Input` in the Object Inspector.

Text is Markdown

The text of every message is Markdown: **bold**, *italic*, `code`, lists, links, tables, quotes, task lists and fenced code blocks with a language, such as ````pascal`, which get syntax colours and a Copy button. The same renderer, built on `sgcRichEdit`, draws the bubbles and the `TsgcVCLMarkdownViewer`, a control that shows one Markdown document on its own. Its `Markdown` property replaces the document, and `AppendMarkdown` adds a piece at the end, which is how an AI answer grows on the screen:

```
Viewer.Markdown := '# Answer' + #13#10#13#10;
Viewer.AppendMarkdown('The first words ');
Viewer.AppendMarkdown('and the rest.');
```

The viewer opens no link and loads no image by itself. Handle `OnLinkClick` to open links, and `OnGetImage` to supply images.

Sending: OnSend, AI or a transport

When the user presses Enter or the send button in a `TsgcVCLChatPanel`, the panel looks for somebody to give the text to, in this order:

1. **AI**, a `TsgcChatAI`: the text goes to `AI.Send`, the answer streams into the conversation, and the send button becomes a Stop button while it does. Demo 05.
2. **Transport**, a chat transport: the message goes to the other people of the channel, together with edits, deletes, reactions, typing and read receipts. The team chat demos in chapter 7.
3. **OnSend**, an event handler: your code decides. Demo 03.

With none of them, the panel simply adds the text to the conversation as a message of the current user. Note the consequence for `OnSend`: once you assign it, the panel adds nothing, so your handler must add the message itself.

AI answers stream on a worker thread

`TsgcChatAI` (unit `sgcChat_AI`) binds a conversation to a provider. `AI.Send(Text)` adds the user message, adds an empty assistant message, copies the history into a request (at most `MaxHistory` messages, 20 by default, plus the `SystemPrompt`) and runs the provider on a worker thread. The provider writes the answer in pieces to a sink; the binding collects the pieces and writes them into the conversation on the main thread, at most every `ThrottleMs` milliseconds (50 by default). `AI.Stop` ends the answer at once and keeps the partial text.

That gives one strict rule: **the provider code runs on a worker thread and must not touch any control or the conversation**. It uses only the request it receives and the sink it writes to. Everything else, the conversation, the controls, the binding and its events such as `OnAnswer` and `OnError`, belongs to the main thread.

`OnBeforeSend` runs on the main thread before a question leaves, and it can change the text or cancel it:

```

procedure TForm1.AIBeforeSend(Sender: TObject; var aText: TsgcRichString;
  var aAllow: Boolean);
begin
  // ignore empty questions, and trim the others
  aText := Trim(aText);
  aAllow := aText <> '';
end;

```

`TsgcRichString`, from unit `sgcRichEdit_Types`, is the string type of the `sgcChat` texts: `UnicodeString` on Delphi 2009 and later, `WideString` on older versions.

Themes and language

Every control has a `Theme` property. Its default, `utDefault`, means “paint with the application theme”, which two procedures of unit `sgcUICore_Theme` set for every `sgcUI` control of the program at once: `sgcUISetApplicationThemePreset` chooses the shape and the colours (`CS_UI_THEME_CLASSIC` or `CS_UI_THEME_FLUENT`), and `sgcUISetApplicationTheme` chooses light or dark. A control with an explicit theme keeps it:

```

ChatView.Theme := utLight;
sgcUISetApplicationThemePreset(CS_UI_THEME_FLUENT);
sgcUISetApplicationTheme(utDark, True);

```

Here the application turns Fluent dark, the second parameter `True` also themes the standard VCL controls of the forms, and `ChatView` stays light. Demo 04 switches the themes from a combo box.

The texts of the controls are in English until you choose a language with `sgcUISetLanguage` of unit `sgcUICore_Lang`, for example `sgcUISetLanguage('de')`, or `sgcUISetLanguage('auto')` to follow Windows.

Saving a conversation

A conversation saves itself as JSON, users and messages included. `SaveToFile` and `LoadFromFile` write and read a file (UTF-8 without BOM), and `SaveToJSON` and `LoadFromJSON` do the same with a string, for a database field or a web service:

```

vJSON := Conversation.SaveToJSON;
Conversation.LoadFromJSON(vJSON);

```

Demo 05 has a Save and a Load button.

Summary: who does what

Piece	Its job	Thread
<code>TsgcChatConversation</code>	Holds users and messages, notifies the views, saves and loads JSON	Main
<code>TsgcVCLChatView</code>	Draws the bubbles of a conversation	Main
<code>TsgcVCLChatInput</code>	Lets the user type and raises <code>OnSend</code>	Main
<code>TsgcVCLChatPanel</code>	Header, view and input in one; routes a send to AI, transport or <code>OnSend</code>	Main
<code>TsgcVCLMarkdownViewer</code>	Shows one Markdown document	Main
<code>TsgcChatAI</code>	Turns a question into a streamed answer in the conversation	Main
A provider (<code>OnStream</code>)	Produces the answer, piece by piece	Worker
A transport	Exchanges messages with other people through a backend	Main (frames may arrive on any thread)

4. The QuickStart demos

The five QuickStart demos build on each other. Open them in order, run each one, then read its section. Every section has the same parts: what you will build, the components on the form, the code, how it works, and a few changes to try.

Demo	Project	New ideas
01.HelloChat	HelloChat.dpr	Data and view, users set at design time, <code>Messages.Add</code>
02.ChatBubbles	ChatBubbles.dpr	Timestamps, status ticks, replies, reactions, attachments, system notices, presence, typing, batching
03.ChatEchoBot	ChatEchoBot.dpr	The chat panel, the <code>OnSend</code> event, typing indicator and read receipts
04.ChatThemes	ChatThemes.dpr	The Markdown viewer, the application theme
05.ChatAssistantApp	ChatAssistantApp.dpr	The AI binding, a streaming provider on a worker thread, Stop, saving and loading

Note. The demos need `sgcChat` installed, so the components appear on the SGC Chat palette page and the `sgcUI\Source` folder is on the library path. They seed their data in code and use only units of `sgcUI`, so they need no network, no server, no API key and no `sgcWebSockets`. Demo 05 answers with a small scripted responder that runs inside the program; chapter 7 shows where to plug in a real AI provider.

4.1 Demo 01: Hello Chat

Folder `01.HelloChat`, project `HelloChat.dpr`.

What you will build

The smallest chat there is. A conversation with two people, you and Ana, and four messages drawn as bubbles. Your messages are on the right, Ana's on the left with an avatar made of her initial. The messages use Markdown for bold text and inline code.

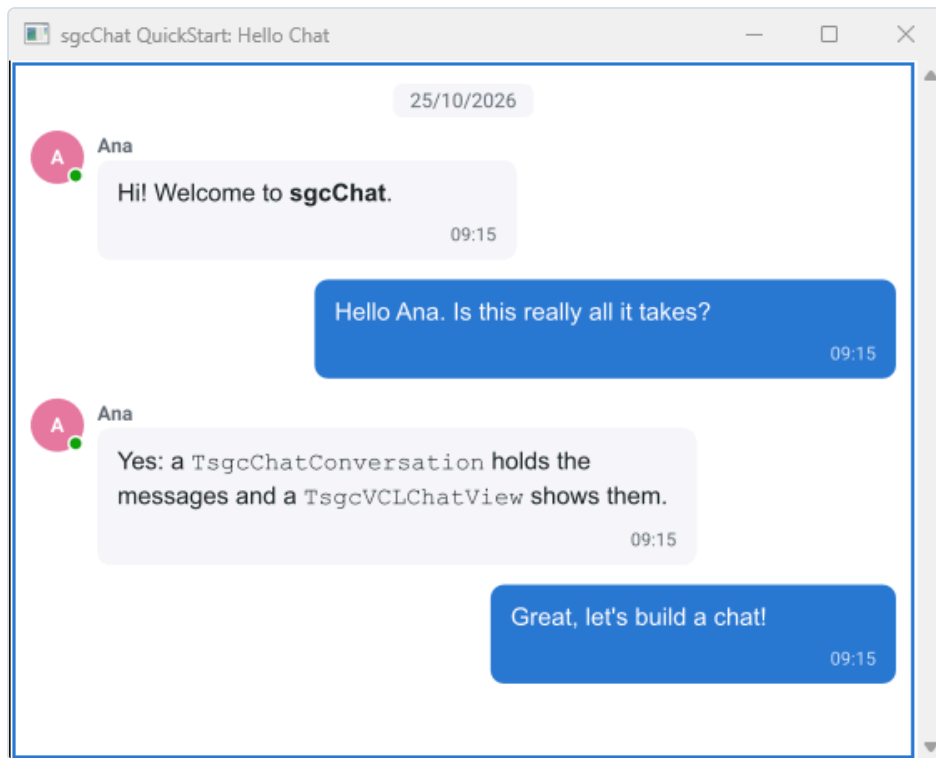


Figure 4.1. Demo 01. Two components and four lines of code.

Components on the form

Component	Class	Properties you set
Conversation	TsgcChatConversation	Users: me (Name You) and ana (Name Ana), both with Presence = cpOnline. CurrentUserId = 'me', Title = 'Hello Chat'.
ChatView	TsgcVCLChatView	Align = alClient, Conversation = Conversation.

The form file is shown in section 1.2. To edit the users, select `Conversation` and click the ellipsis button of `Users`; the collection editor lists one item per user, and the Object Inspector shows its `Id`, `Name` and `Presence`.

The code

The `uses` clause names the model unit and the VCL controls unit:

```
uses
  SysUtils, Classes, Controls, Forms,
  // sgc
  sgcChat_Model, sgcChat_VCL;
```

And this is the only event handler of the demo:

```

procedure TfrmHelloChat.FormCreate(Sender: TObject);
begin
  { The Users and the CurrentUserId come from the DFM; here we only add the
    messages. 'ana' writes on the left, 'me' (the current user) on the
    right. }
  Conversation.Messages.Add(cmrUser, 'ana',
    'Hi! Welcome to **sgcChat**.');
  Conversation.Messages.Add(cmrUser, 'me',
    'Hello Ana. Is this really all it takes?');
  Conversation.Messages.Add(cmrUser, 'ana',
    'Yes: a `TsgcChatConversation` holds the messages and a ' +
    '`TsgcVCLChatView` shows them.');
```

```

  Conversation.Messages.Add(cmrUser, 'me', 'Great, let's build a chat!');
end;
```

How it works

1. The form loads from its DFM. The conversation reads its two users, and the view registers itself with the conversation. There are no messages yet.
2. `FormCreate` calls `Messages.Add` four times. Each call creates a message with a new id and the current time, and notifies the view.
3. The view lays out each message as a bubble. `me` is the `CurrentUserId`, so its messages go to the right; `ana` goes to the left with an avatar.
4. The Markdown in the text, `**sgcChat**` and the back quoted class names, is drawn as bold text and inline code.

The code never touches `ChatView`. That is the pattern of every `sgcChat` application: change the data, and the view follows.

Try this

- Set `CurrentUserId` of `Conversation` to `ana`. Run again: the bubbles swap sides.
- Add a third user in the `Users` collection and add a message from that user in `FormCreate`.
- Set `ChatView.Options.ShowAvatars` or `ChatView.Options.ShowTimestamps` to `False` in the Object Inspector.

4.2 Demo 02: Chat Bubbles

Folder `02.ChatBubbles`, project `ChatBubbles.dpr`.

What you will build

A release team chat that spans two days and uses every kind of message data: a Markdown list, a reply that quotes a question, reactions, a PDF attachment, a centred system notice, a code block, read ticks on your own messages, presence dots and Ana typing at the bottom.

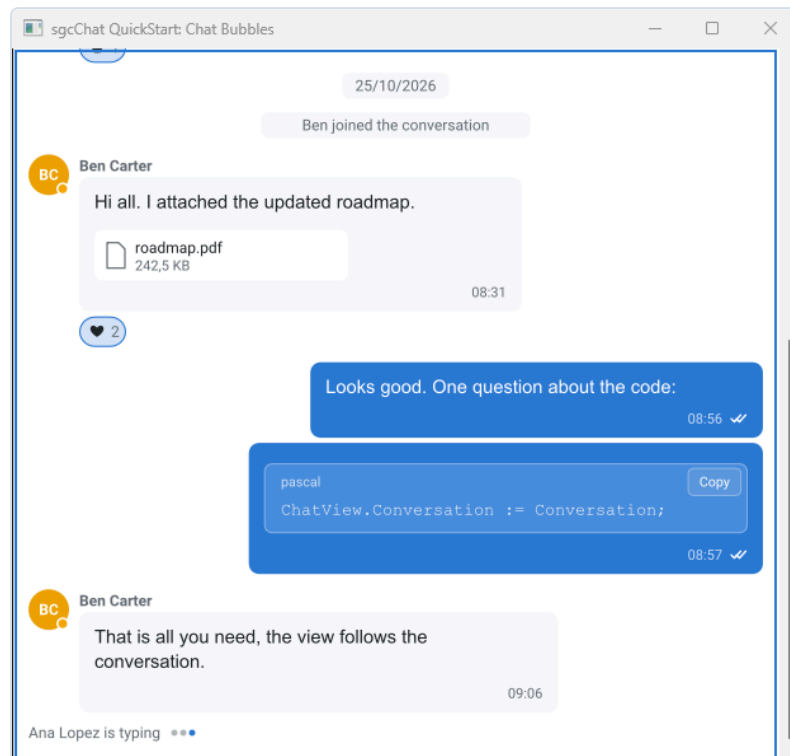


Figure 4.2. Demo 02, today's part of the conversation. Yesterday's messages, with the list, the reply and the thumbs up, are scrolled above.

Components on the form

Component	Class	Properties you set
Conversation	TsgcChatConversation	Users: me (You, cpOnline), ana (Ana Lopez, cpOnline), ben (Ben Carter, cpAway). CurrentUserId = 'me', Title = 'Release team'.
ChatView	TsgcVCLChatView	Align = alClient, Conversation = Conversation, Options.BubbleMaxWidthPercent = 70, Options.ShowActions = False, TypingTimeout = 0.

`TypingTimeout = 0` keeps the typing indicator on the screen for ever; the default hides it after six seconds without news from the user. `ShowActions = False` hides the buttons that appear when the mouse is over a message.

The code

The demo adds `sgcRichEdit_Types` to the `uses` clause for `TsgcRichString`, the string type of the message texts. A small helper builds emoji from their UTF-16 code units, because the source file is saved in the ANSI code page:

```
function DoEmoji(aHigh, aLow: Word): TsgcRichString;
begin
  { emoji are outside the ANSI code page of this file, so they are built
    from their UTF-16 code units: $D83D $DC4D is a thumbs up }
  Result := WideChar(aHigh);
  if aLow <> 0 then
    Result := Result + WideChar(aLow);
end;
```

A second helper adds a message a number of minutes in the past, and gives the messages of the current user the read status:

```
function TfrmChatBubbles.DoAdd(const aAuthorId: string;
  const aText: TsgcRichString; aMinutesAgo: Integer): TsgcChatMessage;
begin
  { Messages.Add creates the message with a new Id and the current time;
    the Timestamp is moved back so the history spans two days }
  Result := Conversation.Messages.Add(cmrUser, aAuthorId, aText);
  Result.Timestamp := sgcChatNowUTC - aMinutesAgo / (24 * 60);
  // the messages of the current user show ticks: read by the others
  if aAuthorId = Conversation.CurrentUserId then
    Result.Status := cmsRead;
end;
```

And `DoLoadMessages`, called from `FormCreate`, fills the conversation:

```
procedure TfrmChatBubbles.DoLoadMessages;
var
  oQuestion, oMessage: TsgcChatMessage;
  oAttachment: TsgcChatAttachment;
begin
  Conversation.BeginUpdate;
  try
    // yesterday
    DoAdd('ana', 'Morning! The **release notes** are ready for review.',
      26 * 60);
    oQuestion := DoAdd('me', 'Thanks Ana. Did you include the new export ' +
      'options?', 26 * 60 - 3);
    oMessage := DoAdd('ana', 'Yes, here is the list:' + CS_CRLF + CS_CRLF +
      '- Export to **PDF**' + CS_CRLF + '- Export to *CSV*' + CS_CRLF +
      '- Print preview', 26 * 60 - 5);
    // a reply quotes the message it answers
    oMessage.ReplyToId := oQuestion.Id;
    oMessage.AddReaction(DoEmoji($D83D, $DC4D), 'me');

    // a system notice: centred text, no bubble
    oMessage := Conversation.Messages.Add(cmrSystem, '',
      'Ben joined the conversation');
    oMessage.Kind := cmkSystem;
    oMessage.Timestamp := sgcChatNowUTC - 50 / (24 * 60);

    // today
    oMessage := DoAdd('ben', 'Hi all. I attached the updated roadmap.', 45);
```

```

oAttachment := oMessage.Attachments.Add;
oAttachment.FileName := 'roadmap.pdf';
oAttachment.MediaType := 'application/pdf';
oAttachment.Size := 248320;
oMessage.AddReaction(DoEmoji($2764, 0), 'ana');
oMessage.AddReaction(DoEmoji($2764, 0), 'me');
DoAdd('me', 'Looks good. One question about the code:', 20);
DoAdd('me', '```pascal' + CS_CRLF + 'ChatView.Conversation := ' +
    'Conversation;' + CS_CRLF + '```', 19);
DoAdd('ben', 'That is all you need, the view follows the conversation.',
    10);
finally
    Conversation.EndUpdate;
end;
// Ana is typing: the indicator appears below the last message
Conversation.Users.Find('ana').IsTyping := True;
end;

```

How it works

1. `BeginUpdate` and `EndUpdate` batch all the messages into one repaint.
2. `Timestamp` is in UTC; `sgcChatNowUTC` returns the current UTC time, and one minute is 1/1440 of a day. Because the history spans two days, the view inserts a date separator before today's messages, and shows each time in the local time of the machine.
3. `Status := cmsRead` draws the double tick below your own messages.
4. `ReplyToId` holds the id of the question, so Ana's list quotes it. `AddReaction` adds a chip with the emoji and the number of users who reacted: the heart on Ben's message shows 2.
5. An attachment with a `FileName`, a `MediaType` and a `Size` becomes a file card. No file is needed: the card shows the data of the attachment.
6. A message with `Role = cmrSystem` and `Kind = cmkSystem` is a centred notice without a bubble or an author.
7. The fenced block with ````pascal` becomes a code box with syntax colours and a Copy button.
8. `IsTyping := True` on Ana adds "Ana Lopez is typing" below the last message. Ben's avatar shows the away colour of `cpAway`.

Try this

- Set `Options.ShowActions` to `True` and move the mouse over a message: action buttons appear.
- Set `Options.GroupMessages` to `False` and compare how consecutive messages of one author are drawn.
- Call `SoftDelete` on one of the messages: the bubble now says that the message was deleted.

4.3 Demo 03: Chat Echo Bot

Folder `03.ChatEchoBot` , project `ChatEchoBot.dpr` .

What you will build

A complete chat window with a header, the messages and an input bar, talking to a small bot that lives in the program. Type a message and press Enter: the bot shows “Echo Bot is typing”, your message turns read, and a moment later the answer appears. The bot knows two commands, `help` and `time` , and repeats anything else.

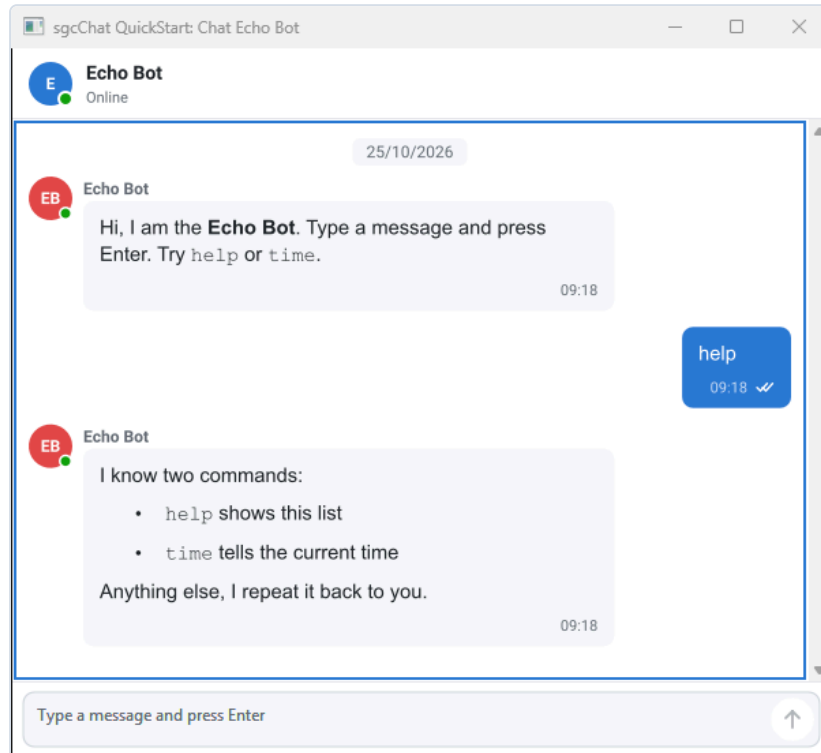


Figure 4.3. Demo 03. A `TsgcVCLChatPanel1` is the header, the view and the input in one control.

Components on the form

Component	Class	Properties you set
<code>ChatPanel</code>	<code>TsgcVCLChatPanel</code>	<code>Align = alClient</code> , <code>Conversation = Conversation</code> , <code>Title = 'Echo Bot'</code> , <code>HeaderUserId = 'bot'</code> (the avatar and presence dot of the header), <code>Input.Placeholder = 'Type a message and press Enter'</code> , <code>Input.ShowAttach = False</code> , <code>OnSend = ChatPanelSend</code> .
<code>Conversation</code>	<code>TsgcChatConversation</code>	<code>Users: me (You) and bot (Echo Bot)</code> . <code>CurrentUserId = 'me'</code> .
<code>tmrReply</code>	<code>TTimer</code>	<code>Enabled = False</code> , <code>Interval = 1200</code> : the thinking time of the bot.

The code

`FormCreate` seeds a short history so the window is not empty:

```

procedure TfrmChatEchoBot.FormCreate(Sender: TObject);
var
  oMessage: TsgcChatMessage;
begin
  // a short history, so the window is not empty when it opens
  Conversation.Messages.Add(cmrUser, CS_BOT, 'Hi, I am the **Echo Bot**.' +
    'Type a message and press Enter. Try `help` or `time`.');
  oMessage := Conversation.Messages.Add(cmrUser, 'me', 'help');
  oMessage.Status := cmsRead;
  Conversation.Messages.Add(cmrUser, CS_BOT, DoAnswer('help'));
end;

```

The send handler and the timer are the whole flow of one message:

```

procedure TfrmChatEchoBot.ChatPanelSend(Sender: TObject;
  const aText: TsgcRichString; aAttachments: TsgcChatAttachments);
var
  oMessage: TsgcChatMessage;
begin
  { 1. the message of the current user. The handler adds it, because the
    panel adds nothing when OnSend is assigned }
  oMessage := Conversation.Messages.Add(cmrUser,
    Conversation.CurrentUserId, aText);
  oMessage.Status := cmsSent;
  // 2. the bot "thinks": typing indicator and a timer
  FQuestion := aText;
  FQuestionId := oMessage.Id;
  Conversation.Users.Find(CS_BOT).IsTyping := True;
  tmrReply.Enabled := False;
  tmrReply.Enabled := True;
end;

procedure TfrmChatEchoBot.tmrReplyTimer(Sender: TObject);
begin
  // 3. the answer
  tmrReply.Enabled := False;
  Conversation.Users.Find(CS_BOT).IsTyping := False;
  Conversation.MarkRead(CS_BOT, FQuestionId);
  Conversation.Messages.Add(cmrUser, CS_BOT, DoAnswer(FQuestion));
end;

```

The bot itself is a plain function that returns Markdown:

```

function TfrmChatEchoBot.DoAnswer(const aQuestion: TsgcRichString)
: TsgcRichString;
var
vText: string;
begin
// the whole "intelligence" of the bot; the answers are Markdown
vText := LowerCase(Trim(aQuestion));
if vText = 'help' then
Result := 'I know two commands:' + CS_CRLF + CS_CRLF +
'- `help` shows this list' + CS_CRLF +
'- `time` tells the current time' + CS_CRLF + CS_CRLF +
'Anything else, I repeat it back to you.'
else if vText = 'time' then
Result := 'It is **' + FormatDateTime('hh:nn', Now) + '**.'
else
Result := 'You said: *' + aQuestion + '*';
end;

```

How it works

1. The user types and presses Enter. The input raises its send, and the panel calls `OnSend` with the text and the attachments.
2. Because `OnSend` is assigned, the panel adds nothing by itself. `ChatPanelSend` adds the message of the current user with `Status = cmsSent`, so the bubble shows one tick.
3. The bot user gets `IsTyping = True`, and the timer starts.
4. When the timer fires, `IsTyping` goes back to `False`, `MarkRead` records that the bot has read the question (its tick becomes a read tick), and the answer is added as a message of the bot.

In a real chat, step 2 would send the text to a server or to a language model, and step 4 would run when the answer arrives. Demo 05 shows the AI way, where `TsgcChatAI` replaces the handler and the timer.

Try this

- Remove the `OnSend` handler in the Object Inspector. The panel now adds your text by itself, and nobody answers.
- Set `Input.SendOnEnter` to `False`: now Ctrl+Enter or the send button sends the message. With the default `True`, Enter sends and Shift+Enter starts a new line.
- Set `Input.ShowAttach` to `True`, attach a file, and read `aAttachments` in `ChatPanelSend`.

4.4 Demo 04: Chat Themes

Folder `04.ChatThemes` , project `ChatThemes.dpr` .

What you will build

A window with two `sgcChat` controls side by side: on the left a `TsgcVCLMarkdownViewer` with release notes (headings, a table, a task list, a quote and a highlighted code block), on the right a small chat. A combo box at the top switches the whole application between Classic and Fluent, light and dark, and both controls change together.

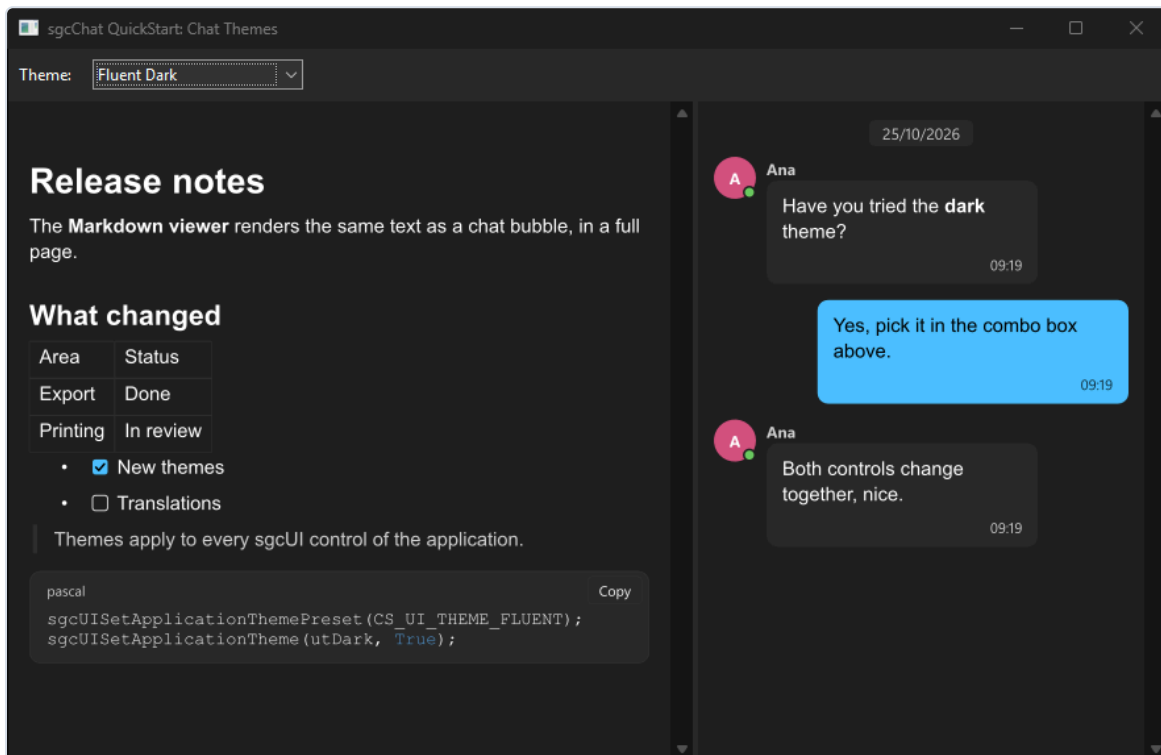


Figure 4.4. Demo 04 opens in Fluent Dark. The panel, the label and the combo box are themed too.

Components on the form

Component	Class	Properties you set
<code>pn1Top</code> , <code>lblTheme</code> , <code>cboTheme</code>	<code>TPanel</code> , <code>TLabel</code> , <code>TComboBox</code>	The combo box lists Classic Light, Classic Dark, Fluent Light and Fluent Dark, with <code>ItemIndex = 3</code> and <code>OnChange = cboThemeChange</code> .
Viewer	<code>TsgcVCLMarkdownViewer</code>	<code>Align = alLeft</code> , <code>Width = 520</code> , <code>Padding = 16</code> (the margin around the text).
<code>splSplitter</code>	<code>TSplitter</code>	Between the viewer and the chat.
ChatView	<code>TsgcVCLChatView</code>	<code>Align = alClient</code> , <code>Conversation = Conversation</code> .
Conversation	<code>TsgcChatConversation</code>	<code>Users : me (You) and ana (Ana)</code> . <code>CurrentUserId = 'me'</code> .

Neither `sgcChat` control sets `Theme` , so both keep the default `utDefault` and follow the application theme.

The code

The `uses` clause adds `sgcUICore_Theme` , the unit of the theme procedures:

```

uses
  SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls,
  // sgc
  sgcRichEdit_Types, sgcUICore_Theme, sgcChat_Model, sgcChat_VCL;

```

The Markdown of the viewer is a plain string:

```

function DoSampleMarkdown: TsgcRichString;
begin
  Result := '# Release notes' + CS_CRLF + CS_CRLF +
    'The Markdown viewer renders the same text as a chat bubble, ' +
    'in a full page.' + CS_CRLF + CS_CRLF +
    '## What changed' + CS_CRLF + CS_CRLF +
    '| Area | Status |' + CS_CRLF +
    '|---|---|' + CS_CRLF +
    '| Export | Done |' + CS_CRLF +
    '| Printing | In review |' + CS_CRLF + CS_CRLF +
    '- [x] New themes' + CS_CRLF +
    '- [ ] Translations' + CS_CRLF + CS_CRLF +
    '> Themes apply to every sgcUI control of the application.' + CS_CRLF +
    CS_CRLF +
    '```pascal' + CS_CRLF +
    'sgcUISetApplicationThemePreset(CS_UI_THEME_FLUENT);' + CS_CRLF +
    'sgcUISetApplicationTheme(utDark, True);' + CS_CRLF +
    '```';
end;

```

`FormCreate` fills both controls and applies the theme selected in the combo box:

```

procedure TfrmChatThemes.FormCreate(Sender: TObject);
begin
  Viewer.Markdown := DoSampleMarkdown;
  Conversation.Messages.Add(cmrUser, 'ana',
    'Have you tried the dark theme?');
  Conversation.Messages.Add(cmrUser, 'me',
    'Yes, pick it in the combo box above. ');
  Conversation.Messages.Add(cmrUser, 'ana',
    'Both controls change together, nice. ');
  // the DFM selects Fluent Dark, apply it now
  DoApplyTheme(cboTheme.ItemIndex);
end;

procedure TfrmChatThemes.cboThemeChange(Sender: TObject);
begin
  DoApplyTheme(cboTheme.ItemIndex);
end;

procedure TfrmChatThemes.DoApplyTheme(aIndex: Integer);
begin
  { the items of the combo box: 0 Classic Light, 1 Classic Dark,
    2 Fluent Light, 3 Fluent Dark }
  if aIndex >= 2 then
    sgcUISetApplicationThemePreset(CS_UI_THEME_FLUENT)
  else
    sgcUISetApplicationThemePreset(CS_UI_THEME_CLASSIC);
  if Odd(aIndex) then
    sgcUISetApplicationTheme(utDark, True)
  else
    sgcUISetApplicationTheme(utLight, True);
end;

```

How it works

1. `Viewer.Markdown` replaces the document of the viewer. The viewer parses it and lays it out at its own width, so the text wraps when you drag the splitter.
2. `sgcUISetApplicationThemePreset` chooses the family of shapes and colours: Classic, or Fluent with rounder corners and the Windows 11 look.
3. `sgcUISetApplicationTheme` chooses light or dark. Every `sgcUI` control whose `Theme` is `utDefault` repaints with the new colours. The second parameter, `True`, also themes the standard VCL controls of the forms, here the panel, the label and the combo box.
4. The bubbles and the viewer share one Markdown renderer, so the code block of the viewer and a code block in a bubble look the same.

Try this

- Set `Viewer.Theme` to `utLight`. Now the viewer stays light while the rest of the window turns dark.
- Add a fifth item to the combo box that calls `sgcUISetApplicationTheme(utHighContrast, True)`.
- Replace `Viewer.Markdown := DoSampleMarkdown` with a few calls of `Viewer.AppendMarkdown`, one per line of text, to see how a document can be streamed into the viewer.

4.5 Demo 05: Chat Assistant App

Folder `05.ChatAssistantApp`, project `ChatAssistantApp.dpr`.

What you will build

A small AI assistant application. You ask a question, and the answer appears piece by piece, like the answer of a real language model. A side panel has New, Stop, Save and Load buttons, and a status line. The answers come from a scripted responder inside the program, so the demo runs offline, without an account or an API key. The structure is exactly the one of an application that talks to a real model: only the provider changes.

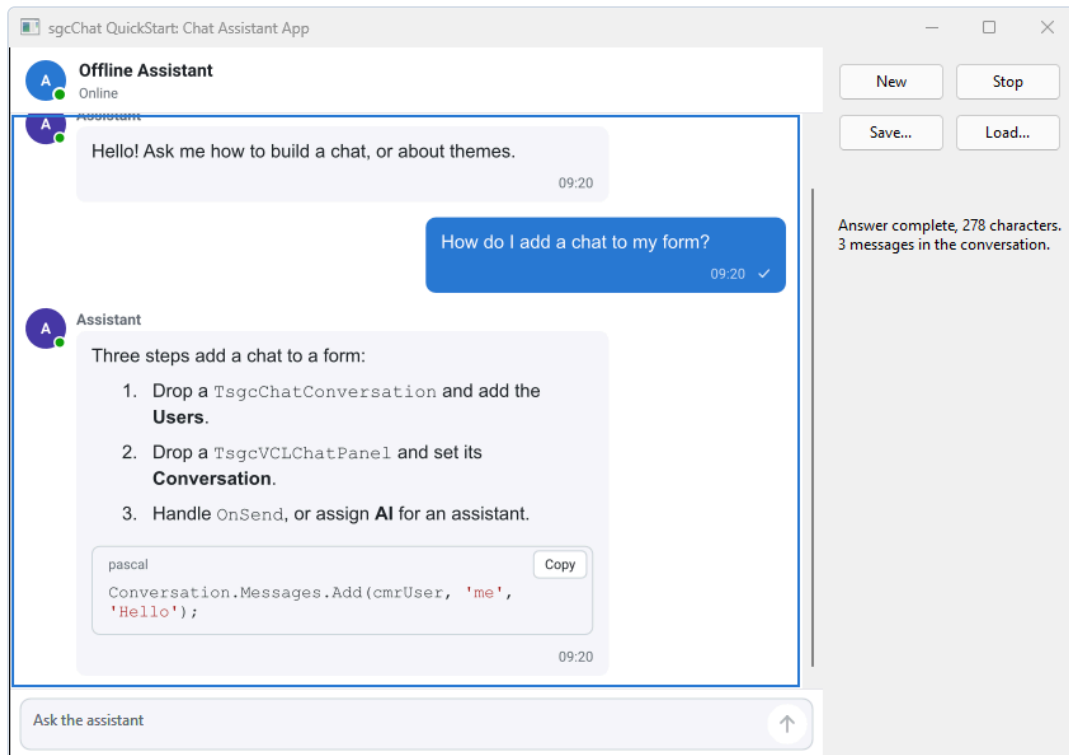


Figure 4.5. Demo 05 after the first answer. The status line on the right is updated by `OnAnswer`.

Components on the form

Component	Class	Properties you set
ChatPanel	TsgcVCLChatPanel	Align = alClient, Conversation = Conversation, AI = AI, Title = 'Offline Assistant', HeaderUserId = 'assistant', Input.Placeholder = 'Ask the assistant', Input.ShowAttach = False.No OnSend.
Conversation	TsgcChatConversation	Users: me (You) and assistant (Assistant). CurrentUserId = 'me'.
AI	TsgcChatAI	Conversation = Conversation, Provider = Provider, SystemPrompt = 'You are a helpful assistant for Delphi developers.', AssistantUserId = 'assistant', OnAnswer = AIAnswer, OnError = AIError.
Provider	TsgcChatAIEventProvider	OnStream = ProviderStream.
pnlSide	TPanel	Align = alRight, with btnNew, btnStop, btnSave, btnLoad and lblStatus.
dlgSave, dlgOpen	TSaveDialog, TOpenDialog	DefaultExt = 'json', filter Conversation (*.json).

The code

The `uses` clause adds `sgcChat_AI`, the unit of the AI binding and its providers:

```
uses
  Windows, SysUtils, Classes, Controls, Forms, StdCtrls, ExtCtrls, Dialogs,
  // sgc
  sgcRichEdit_Types, sgcChat_Model, sgcChat_AI, sgcChat_VCL;
```

The "model" of the demo is a pure function that picks a canned Markdown answer by a keyword of the question (trimmed here):

```
function DoScriptedAnswer(const aQuestion: string): string;
var
  vText: string;
begin
  { the "model" of this demo: a few canned Markdown answers chosen by a
    keyword of the question. Pure code, safe on the worker thread }
  vText := LowerCase(aQuestion);
  if Pos('theme', vText) > 0 then
    Result := 'Every sgcChat control follows the **application theme**:' +
      // ...
  else if Pos('chat', vText) > 0 then
    Result := 'Three steps add a chat to a form:' + CS_CRLF + CS_CRLF +
      // ...
  else
    Result := 'I am a **scripted assistant** that runs inside this ' +
      // ...
end;
```

The provider event writes that answer to the sink, six characters at a time:

```

procedure TfrmChatAssistantApp.ProviderStream(Sender: TObject;
  aRequest: TsgcChatAIRequest; const aSink: TsgcChatAIStreamSink);
var
  vAnswer: string;
  vPos: Integer;
begin
  // worker thread: no VCL here, only aRequest and aSink
  vAnswer := '';
  if aRequest.LastUser <> nil then
    vAnswer := DoScriptedAnswer(aRequest.LastUser.Text);
  vPos := 1;
  while vPos <= Length(vAnswer) do
    begin
      // the Stop button (or closing the form) cancels the sink
      if aSink.Cancelled then
        Exit;
      aSink.Chunk(Copy(vAnswer, vPos, 6));
      Inc(vPos, 6);
      Sleep(15);
    end;
  aSink.Done;
end;

```

The form events start the first question and stop a running answer before the components are freed:

```

procedure TfrmChatAssistantApp.FormCreate(Sender: TObject);
begin
  DoWelcome;
  // a first question, so the window opens with a streamed answer
  AI.Send('How do I add a chat to my form?');
end;

procedure TfrmChatAssistantApp.FormDestroy(Sender: TObject);
begin
  // stop a running answer before the components are freed
  ChatPanel.AI := nil;
  AI.Stop;
end;

procedure TfrmChatAssistantApp.DoWelcome;
begin
  Conversation.Messages.Add(cmrAssistant, CS_ASSISTANT,
    'Hello! Ask me how to build a chat, or about themes.');
```

The binding events run on the main thread, so they may update the form:

```

procedure TfrmChatAssistantApp.AIAnswer(Sender: TObject;
  aMessage: TsgcChatMessage);
begin
  // main thread: the answer is complete
  lblStatus.Caption := Format('Answer complete, %d characters. %d ' +
    'messages in the conversation.', [Length(aMessage.Text),
    Conversation.Messages.Count]);
end;

procedure TfrmChatAssistantApp.AIError(Sender: TObject;
  aMessage: TsgcChatMessage; const aError: string);
begin
  lblStatus.Caption := 'Error: ' + aError;
end;

```

And the four buttons are one or two lines each:

```

procedure TfrmChatAssistantApp.btnNewClick(Sender: TObject);
begin
  AI.Stop;
  Conversation.Messages.Clear;
  DoWelcome;
  lblStatus.Caption := 'New conversation.';
end;

procedure TfrmChatAssistantApp.btnStopClick(Sender: TObject);
begin
  // the partial answer stays in the conversation
  AI.Stop;
end;

procedure TfrmChatAssistantApp.btnSaveClick(Sender: TObject);
begin
  if dlgSave.Execute then
    Conversation.SaveToFile(dlgSave.FileName);
end;

procedure TfrmChatAssistantApp.btnLoadClick(Sender: TObject);
begin
  if dlgOpen.Execute then
  begin
    AI.Stop;
    Conversation.LoadFromFile(dlgOpen.FileName);
  end;
end;

```

How it works

1. The user presses Enter. Because the panel has `AI` assigned, it calls `AI.Send` with the text; no `OnSend` handler is needed.
2. `TsgcChatAI` adds the user message and an empty message of `AssistantUserId`, copies the history and the `SystemPrompt` into a `TsgcChatAIRequest`, and starts a worker thread. The send button of the panel turns into a Stop button.
3. On the worker thread, `ProviderStream` reads the question from `aRequest.LastUser.Text` and writes the answer with `aSink.Chunk`. It touches no control and not the conversation.
4. The binding moves the chunks to the main thread and appends them to the assistant message. The bubble grows as the text arrives, and the Markdown is drawn as it streams.

5. `aSink.Done` ends the answer. On the main thread `OnAnswer` fires, and the status line shows the length of the answer.
6. Stop, New and Load call `AI.Stop`. The sink reports `Cancelled`, the loop in `ProviderStream` exits, and the partial answer stays in the conversation.

With `AI` assigned, the panel also offers the Regenerate and Edit actions on the messages, which ask the binding to answer again.

Note. To answer with a real model, replace `Provider` with a `TsgcChatAIProvider_AIChat` and set `AI.Provider` to it. Nothing else in this form changes. That provider needs `sgcWebSockets` and, for most services, an API key. Section 7.2 explains it and points to the demo that does it.

Try this

- Raise the `Sleep(15)` to `Sleep(60)`, ask a question and press Stop in the middle of the answer.
- Handle `AI.OnBeforeSend (var aText, var aAllow)` to change a question, or to refuse it with `aAllow := False`. Section 3.7 has an example.
- Call `aSink.Error('No answer')` instead of `aSink.Done` and watch `OnError` update the status line.
- Save the conversation, open the JSON file in an editor, and load it again.

5. Main components

The SGC Chat palette page is short, and three of its components cover a typical first application: the conversation, the chat panel and, for an assistant, the AI binding. This chapter introduces each of them, then the classes you meet in code, and finally the components of the `sgcWebSockets` bridge.

Every class has its own help topic, named after the class, such as `TsgcVCLChatPanel.htm` or `TsgcChatMessage.htm`. Each topic lists the properties, methods and events with their types and defaults.

Reference table

Class	Palette page	What it is for	Key members
<code>TsgcVCLChatPanel</code>	SGC Chat	A complete chat window	<code>Conversation</code> , <code>AI</code> , <code>Transport</code> , <code>OnSend</code> , <code>Title</code> , <code>HeaderUserId</code> , <code>ShowHeader</code> , <code>View</code> , <code>Input</code>
<code>TsgcVCLChatView</code>	SGC Chat	The message bubbles	<code>Conversation</code> , <code>Options</code> , <code>TypingTimeout</code> , <code>Theme</code> , <code>OnLinkClick</code>
<code>TsgcVCLChatInput</code>	SGC Chat	The input bar	<code>Placeholder</code> , <code>SendOnEnter</code> , <code>ShowAttach</code> , <code>OnSend</code>
<code>TsgcVCLMarkdownViewer</code>	SGC Chat	One Markdown document	<code>Markdown</code> , <code>AppendMarkdown</code> , <code>Padding</code> , <code>Theme</code> , <code>OnLinkClick</code> , <code>OnGetImage</code>
<code>TsgcFMXChatPanel</code> , <code>TsgcFMXChatView</code> , <code>TsgcFMXChatInput</code> , <code>TsgcFMXMarkdownViewer</code>	SGC Chat (FireMonkey)	The FireMonkey controls	The same as the VCL controls
<code>TsgcChatConversation</code>	SGC Chat	Users and messages	<code>Users</code> , <code>Messages</code> , <code>CurrentUserId</code> , <code>Title</code> , <code>BeginUpdate</code> , <code>MarkRead</code> , <code>SaveToFile</code> , <code>LoadFromFile</code>
<code>TsgcChatAI</code>	SGC Chat	Streams AI answers into a conversation	<code>Conversation</code> , <code>Provider</code> , <code>SystemPrompt</code> , <code>AssistantUserId</code> , <code>MaxHistory</code> , <code>Send</code> , <code>Stop</code> , <code>Busy</code> , <code>OnAnswer</code> , <code>OnError</code>
<code>TsgcChatAIEventProvider</code>	SGC Chat	Answers from your own code	<code>OnStream</code>
<code>TsgcChatAIProvider_UICore</code>	SGC Chat	Answers from an <code>sgcUICore</code> AI provider	<code>Provider</code>
<code>TsgcChatEventTransport</code>	SGC Chat	A human chat over any backend	See the help topic

Class	Palette page	What it is for	Key members
<code>TsgcChatAIProvider_AIChat</code>	SGC Chat WS	Answers from OpenAI, Anthropic, Gemini and the other LLMs of sgcWebSockets	<code>Chat</code>
<code>TsgcChatTransport_WebSocket</code>	SGC Chat WS	A team chat client over WebSocket	See the help topic
<code>TsgcChatServer_WebSocket</code>	SGC Chat WS	The relay server of a team chat	See the help topic
<code>TsgcChatUser</code>	none, an item of <code>Users</code>	One person or bot	<code>Id</code> , <code>Name</code> , <code>Presence</code> , <code>IsTyping</code>
<code>TsgcChatMessage</code>	none, an item of <code>Messages</code>	One message	<code>Id</code> , <code>Role</code> , <code>AuthorId</code> , <code>Text</code> , <code>Timestamp</code> , <code>Status</code> , <code>Kind</code> , <code>ReplyToId</code> , <code>Attachments</code> , <code>Reactions</code> , <code>AddReaction</code> , <code>SoftDelete</code>

The controls

TsgcVCLChatPanel

The control you drop first. It is a header, a `TsgcVCLChatView` (published as `view`) and a `TsgcVCLChatInput` (published as `input`) in one, already laid out. The header shows `Title`, or the `Title` of the conversation when the panel has none, with the avatar and presence of `HeaderUserId`; `ShowHeader = False` hides it. The panel routes every send to `AI`, `Transport` or `OnSend`, as section 3.6 explains. Demos 03 and 05.

TsgcVCLChatView and TsgcVCLChatInput

The two halves of the panel as separate controls, for layouts the panel does not cover: a view without input for a read only history, or a view and an input in different places of the form. The view is used on its own in demos 01, 02 and 04. When you use the input on its own, its `OnSend` is yours to handle.

TsgcVCLMarkdownViewer

A read only Markdown page with the renderer of the bubbles: headings, tables, task lists, quotes and code blocks with syntax colours and a Copy button. Use it for help pages, release notes, or a long AI answer that deserves a full window. Demo 04.

The data model

`TsgcChatConversation` with its `TsgcChatUser` and `TsgcChatMessage` items is described in sections 3.1 to 3.3. Two more classes hang from a message: `TsgcChatAttachment` (`FileName`, `MediaType`, `Size`, `URL`) and `TsgcChatReaction` (`Emoji` and the ids of the users who reacted). All of them are framework neutral, so the same conversation works behind VCL and FireMonkey controls, and in code that has no window at all.

The AI binding and its providers

`TsgcChatAI` knows how to run a question and stream an answer into a conversation; a provider knows where the answer comes from. Three providers are available:

- `TsgcChatAIEventProvider` . Your `OnStream` handler produces the answer on a worker thread, with any code or library. Demo 05 uses it with a scripted responder; it is also the way to call an AI service of your own.
- `TsgcChatAIProvider_UICore` . Uses an AI provider of `sgcUICore`, the same one other `sgcUI` products use for their AI features. The answer arrives as one chunk. To reach the LLM services, the `sgcUICore` AI provider also uses `sgcWebSockets`.
- `TsgcChatAIProvider_AIChat` . In the `sgcWebSockets` bridge. Its `chat` property is a `TsgcAIChat` of `sgcWebSockets`, which talks to OpenAI, Anthropic, Gemini, DeepSeek, Ollama, Grok and Mistral, with streamed answers.

Because the panel and the binding never change, you can start with `TsgcChatAIEventProvider` during development and switch to a real provider later by changing one property, `AI.Provider` .

Transports and the `sgcWebSockets` bridge

A transport connects a conversation to other people. Assign it to the `Transport` property of the panel, and the panel sends through it the messages, edits, deletes, reactions, typing notices and read receipts of the user, while the frames that arrive from the others are applied to the conversation on the main thread.

- `TsgcChatEventTransport` lets you move the frames yourself, over any backend you already have.
- `TsgcChatTransport_WebSocket` and `TsgcChatServer_WebSocket` , in the `sgcWebSockets` bridge, are a ready team chat: a client on a `TsgcWebSocketClient` , and a relay on a `TsgcWebSocketServer` that forwards the frames of a channel and keeps the users online and the recent messages.

The QuickStart demos use no transport. The `TeamChat` and `TeamChatServer` demos in chapter 7 show the `WebSocket` pair in action.

6. Common problems

These are the problems that come up most often in a first chat. Each entry says what you see, why it happens and how to fix it.

Sending and showing messages

My message disappears when I press Enter

What you see. The input clears, but no bubble appears.

Why. The panel has an `OnSend` handler. Once `OnSend` is assigned, the panel adds nothing by itself: it hands the text to your handler and expects the handler to decide.

Fix. Add the message in the handler, as demo 03 does with `Conversation.Messages.Add(cmrUser, Conversation.CurrentUserId, aText)`. If you do not need to intercept the text, remove the handler, and the panel adds the message itself.

Send does nothing at all

What you see. Nothing happens when you press Enter or the send button, or the bubbles never appear.

Why. The panel, the binding or the transport has no `Conversation`, or they point to different conversations.

Fix. Set `Conversation` on the panel and on the `TsgcChatAI` or the transport, to the same component.

My own messages are on the left

What you see. Every bubble is on the left, with an avatar, including the ones you wrote.

Why. The side of a bubble depends on `CurrentUserId`. When it is empty, or does not match the `AuthorId` of your messages, nobody is the current user. (In a right to left control the whole chat is mirrored on purpose, so there the own messages are on the left.)

Fix. Set `Conversation.CurrentUserId` to the `Id` of the user who runs the application, and use the same id as `AuthorId` of the messages.

The typing indicator vanishes, or never goes away

What you see. "is typing" disappears after a few seconds although `IsTyping` is still `True`, or it stays for ever.

Why. The view clears a typing user after `TypingTimeout` milliseconds, 6000 by default. A value of 0 means never.

Fix. Set `IsTyping` again while the user is still typing, or change `TypingTimeout` on the view. Always set `IsTyping := False` when the answer arrives, as demo 03 does.

Emoji turn into question marks

What you see. An emoji typed into a string literal shows as `?`.

Why. The source file is saved in an ANSI code page, which has no emoji. On Delphi 7 to 2007 the string type of the texts is `WideString`, and the compiler converts literals through the ANSI code page too.

Fix. Build the emoji from its UTF-16 code units, as the `DoEmoji` helper of demo 02 does, or save the unit as UTF-8 on Delphi 2009 and later.

AI answers

EsgcChatAIError when sending

Why. `TsgcChatAI` needs a `Conversation` and a `Provider`. Each provider needs its own part: `TsgcChatAIEventProvider` an `OnStream` handler, `TsgcChatAIProvider_UICore` a `Provider`, `TsgcChatAIProvider_AIChat` a `Chat`.

Fix. Check the three links in the Object Inspector, as they are set in demo 05.

Access violations or a frozen window while an answer streams

Why. `OnStream` runs on a worker thread, and the handler touched a control, the form or the conversation.

Fix. In `OnStream` use only `aRequest` and `aSink`, and keep any helper a pure function, like `DoScriptedAnswer` in demo 05. Update the form in `OnAnswer` or `OnError`, which run on the main thread.

The answer appears at once, not in pieces

Why. The provider gives the whole answer as one chunk. `TsgcChatAIProvider_UICore` always does, `TsgcChatAIProvider_AIChat` does for a question with image or PDF attachments, and a provider of your own does when it calls `Chunk` only once.

Fix. In your own provider, call `aSink.Chunk` for each piece as it arrives.

An access violation when the form closes during an answer

Why. An answer is still streaming while the components are being freed.

Fix. Stop it first in `OnDestroy`, as demo 05 does: `ChatPanel.AI := nil; AI.Stop;` . With a transport, stop its backend before you free the transport.

Stop ends the answer, but the request goes on

Why. On Delphi versions before 2010, the request of `TsgcChatAIProvider_AIChat` cannot be aborted.

Fix. Nothing to do: the answer ends at once on the screen, and the rest of the request is dropped when it arrives.

Layout, links and language

Links do not open, images are missing

Why. The controls never open a link or download an image on their own.

Fix. Handle `OnLinkClick` to open links, and `OnGetImage` on the Markdown viewer to supply images.

Padding of the Markdown viewer is a number

What you see. In the Object Inspector, `Padding` of `TsgcVCLMarkdownViewer` is a single integer, not the usual Left, Top, Right and Bottom.

Why. The viewer publishes its own `Padding`, the margin in pixels around the document, which hides the `Padding` of `TWinControl`.

Fix. Nothing to fix: set one value, like `Padding = 16` in demo 04.

The texts stay in English

Why. The language of the controls is English until you choose another one.

Fix. Call `sgcUISetLanguage` of unit `sgcUICore_Lang` at startup, for example `sgcUISetLanguage('de')`, or `sgcUISetLanguage('auto')` to follow Windows.

7. Where next

Once the five QuickStart demos feel familiar, you know the core of sgcChat. This chapter points you to the larger demos, to a real AI provider, to the documentation and to support.

The other Delphi demos

The other folders in `Demos\Chat\Delphi` are complete applications. Each one has a `readme.html` that explains what it shows and how to run it.

Demo	What it shows
AIAssistant	A streaming AI chat: a <code>TsgcVCLChatPanel</code> with a <code>TsgcChatAI</code> . It works at once with an offline scripted provider, and with OpenAI, Anthropic, Gemini, DeepSeek, Ollama, Grok or Mistral through <code>TsgcAIChat</code> of <code>sgcWebSockets</code> when you enter an API key. The natural next step after demo 05.
AIAssistantFMX	The same streaming AI chat with <code>TsgcFMXChatPanel</code> , for FireMonkey.
MarkdownViewer	A <code>TsgcVCLMarkdownViewer</code> with a sample document: load any Markdown file, switch between Classic and Fluent, light or dark, and replay the document in chunks to see how an AI answer streams in.
MarkdownViewerFMX	The same Markdown viewer with <code>TsgcFMXMarkdownViewer</code> , for FireMonkey.
TeamChat	A team chat client over WebSocket: a <code>TsgcVCLChatPanel</code> bound to a <code>TsgcChatTransport_WebSocket</code> , with messages, edits, deletes, reactions, typing, read receipts and presence. Needs <code>sgcWebSockets</code> .
TeamChatServer	The relay of the team chat: a console application with a <code>TsgcWebSocketServer</code> and a <code>TsgcChatServer_WebSocket</code> that relays the frames by channel and moderates the messages with a banned word filter. Needs <code>sgcWebSockets</code> .

The AI and team chat demos use components of the `sgcWebSockets` bridge where they connect to a real service. Their readme files say what they need.

Plugging in a real AI provider

Demo 05 answers with a scripted responder so that it runs anywhere. To answer with a real language model, keep the panel, the conversation and the `TsgcChatAI` as they are and replace the provider with a `TsgcChatAIProvider_AIChat`. You need:

- `sgcWebSockets` with its AI components, and the **sgcChat for sgcWebSockets** bridge selected in the `sgcUI` installer.
- An account and an API key of the service you choose (OpenAI, Anthropic, Gemini, DeepSeek, Grok or Mistral). Ollama runs models on your own machine.

At design time, drop a `TsgcAIChat` of `sgcWebSockets` and a `TsgcChatAIProvider_AIChat` from the SGC Chat WS page, set the `Chat` property of the provider to the `TsgcAIChat`, and set `AI.Provider` to the new provider. The same in code:

```

uses
  SysUtils, Classes,
  // sgcWebSockets
  sgcAI, sgcAI_Chat,
  // sgc
  sgcChat_AI, sgcChatWS_AI;

procedure UseOpenAI(aAI: TsgcChatAI; const aApiKey: string);
var
  oChat: TsgcAIChat;
  oProvider: TsgcChatAIProvider_AIChat;
begin
  oChat := TsgcAIChat.Create(aAI.Owner);
  oChat.Provider := aicpOpenAI;
  oChat.ChatOptions.ApiKey := aApiKey;
  oProvider := TsgcChatAIProvider_AIChat.Create(aAI.Owner);
  oProvider.Chat := oChat;
  aAI.Provider := oProvider;
end;

```

Choose the model in `oChat.ChatOptions.Model`, and the service with `Provider` (`aicpAnthropic`, `aicpGemini`, `aicpOllama` and so on). The `AIAssistant` demo does all of this with a provider list and an API key box, and starts with the offline scripted provider, so it is the best place to continue. The help topic `AI providers` describes the options of each provider.

Keep your API key out of the source. Read it from a setting, an environment variable or a key store at run time, never commit it to version control, and remember that requests to a hosted model are billed to the account of the key.

Help topics

The `sgcChat` help is part of the `sgcUI` help at <https://www.esegece.com/help/sgcUI>, in the `Chat` section. These topics are the best continuation of this guide:

Topic	What it covers
Introduction	What <code>sgcChat</code> does and where to start.
Architecture, Platforms, Editions	The units and layers, the Delphi versions and targets, and what each edition contains.
Conversations, Markdown, Themes	The data model, the Markdown support and the look.
AI assistant, AI providers, Streaming	The AI binding, the providers and how answers stream.
Team chat, Transports	Human chat between people, the transports and the relay server.
Threading, Localization	Which code runs on which thread, and the languages of the controls.
Troubleshooting	More problems and their causes.
One topic per class	<code>TsgcVCLChatPanel</code> , <code>TsgcVCLChatView</code> , <code>TsgcChatConversation</code> , <code>TsgcChatMessage</code> , <code>TsgcChatAI</code> and the others, with every property, its type and its default.

On the website

- <https://www.egegece.com/products/sgcchat/>: the product page, with the features, the editions and the downloads.

Getting help

If you are stuck, write to info@egegece.com or open a ticket in the helpdesk of your account at <https://www.egegece.com/>. It helps to include your Delphi version, your sgcChat version and edition, and the demo or a small project that shows the problem.

Enjoy building your chat.

sgcChat and sgcUI are products of eSeGeCe. <https://www.egegece.com>