Chat Session Example

A minimal operator-chat panel: six live plant tags, a transcript widget that renders selectable per-role bubbles, a TextBox, a Button, and the ChatRequest action — no scripting required. The smallest end-to-end example of the FrameworX ChatSession control on a Display — talking to a local LLM or a cloud model such as Anthropic Claude.

Version 10.1.5+

Download the solution Local AI Chat Example.dbsln

This example demonstrates the **simplest possible operator chat panel** — the **ChatSession** control and the `ActionType =ChatRequest` action talking to an LLM, with six plant tags as live grounding context, one TextBox for the question, and one Button. No CodeBehind, no script call. The model can be a **local LLM** (Ollama on your own network) or a **cloud model such as Anthropic Claude** — any OpenAI-compatible chat-completions endpoint. The smallest reachable footprint of the canonical pattern described on the Local AI parent page.

The downloaded `.dbsln` ships with a **reference endpoint URL and model name pre-configured for the author's bench setup** — they are placeholders, not defaults you should run as-is. **You must install your own LLM on a separate machine and edit the AI Engine tile on Solution Capabilities to match.** See Local AI Deployment and Performance for the canonical guide to where to put the LLM, which model size to choose, and what hardware footprint each tier needs — read that first.

Do not install the LLM on the process-control host

Do not run the model on the FrameworX/TServer host — it competes with the runtime for CPU; use a separate GPU machine or a remote/cloud endpoint. Running the LLM on the same machine as TServer / the process-control runtime is the wrong starting point. Either the LLM competes with the runtime for CPU and degrades both, OR you pick a model small enough not to compete and the answer quality / usability drops to where the chat surface stops being useful. Neither outcome is good.

Pick one of the production-shaped topologies instead: an existing GPU-equipped machine you already own (gaming PC, dev workstation, ML-capable server), a dedicated LLM host, or a cloud endpoint. Local AI Deployment and Performance covers the topologies, the decision matrix by workload, model sizes, and the hardware sizing guidance; Remote and Cloud LLM Models covers off-host remote and cloud endpoints specifically — required reading before running the example.

Prerequisites

- FrameworX 10.1.5 or later.
- An **LLM endpoint of your own**, reachable from the FrameworX runtime over HTTP. The endpoint must speak the OpenAI-compatible chat-completions API (Ollama is the default reference; LM Studio, vLLM, llama.cpp's server, or any cloud endpoint with the same shape also works). The endpoint must NOT run on the same machine as TServer — see the warning above and Local AI Deployment and Performance for where to place it.
- A model pulled on that endpoint. Choose the tier that matches your hardware. As a rule: `qwen2.5:7b-instruct` (~4.7 GB) is the recommended default and the realistic floor for production-quality interactive chat — run it on a separate GPU machine, the floor even for demos; `qwen2.5:3b-instruct` (~2 GB) is a last-resort fallback for single-shot atomic tasks only — CPU-only produces only ~2-4 tokens /sec, too slow even for a demonstration, never interactive chat; `qwen2.5:32b-instruct` is the maximum-performance tier on a strong GPU. See Local AI Deployment and Performance for the matrix.
- The endpoint URL and the model name written into the AI Engine tile on Solution Capabilities. **You must edit these** — the shipped `.dbsln` carries placeholder values that almost certainly do not match your network.

Local model or cloud model

The **ChatSession** control and the `ChatRequest` action are endpoint-agnostic — they speak the OpenAI-compatible chat-completions API, so the model behind the panel can be either:

- **A local LLM** on your own network — Ollama (or LM Studio, vLLM, llama.cpp). On-premise; nothing leaves the site. This is what the example ships pointed at.
- **A cloud model** — any OpenAI-compatible cloud endpoint, such as **Anthropic Claude**. Point `URL` at the provider, set `Authorization` to `BearerToken`, and keep the API key in the SecuritySecrets vault via `/secret:<Name>` so it never appears in the solution — see SecuritySecrets Authentication for Local AI.

> **Reaching a cloud model from an isolated plant network**
>
> Plant networks usually have no direct internet path — and the chat call should not create one. Send the request out over a controlled channel instead of the open internet. FrameworX's SecureGateway Services Reference service provides authorized, isolated routing between designated points (`LocalPort  RemoteIP:RemotePort`, with traffic control and site isolation): the plant runtime talks only to the gateway, and the gateway relays approved traffic outward to an internet-capable relay. Combine it with the SecuritySecrets-backed `BearerToken` auth above so the cloud credential never lives in the plant solution.

What it contains

Open the solution in Designer and you will see exactly eight UNS tags (six under `Plant/`, two under `Chat/`), one Display (`ChatTest`), one SolutionSettings configuration. No Devices, no Alarms, no Historian, no scripts.

Plant tags — the live context the operator and the model both see

Tag	Type	StartValue	Purpose
`Plant/TankLevel`	Double, units `gallons`	`784.5`	Reactor R-101 fill level. Range 0–1000 gallons.
`Plant/Temperature`	Double, units `degC`	`67.2`	Reactor R-101 jacket temperature.
`Plant/Pressure`	Double, units `bar`	`4.45`	Reactor R-101 head pressure.
`Plant/PumpRunning`	Digital	`1`	Feed pump P-201 status. 1 = running, 0 = stopped.
`Plant/BatchID`	Text	`BATCH-2026-0525-A`	Current batch identifier.
`Plant/OperatorName`	Text	`Marco`	Operator on shift.

Chat tags — the operator-to-AI wiring

Tag	Type	Purpose
`Chat/Query`	Text	Operator types into a TextBox bound here. The ChatRequest action reads this as the question.
`Chat/Reply`	**JSON**	Receives the full reply envelope. **JSON** type lets Display expressions extract fields via `JsonString("text")` / `JsonString("status")` / `JsonString("latencyMs")` with no scripting.

That is every UNS tag the example needs. The plant tags carry `StartValue` so the runtime has live numbers as soon as you click Run; in a real solution they would be driven by a TagProvider, a Device, or a script.

The ChatTest display

One Canvas display at 1200 × 800. All elements theme-bound (no hard-coded colors). The transcript widget is the new **TChatSession** control — native to FrameworX 10.1.5, no Symbol dependency, no DataTable wiring on the adopter side.

Header strip

TextBlock title *"Local LLM AI - Integration Test"* + one-line subtitle explaining the panel.

Live tags row

Section heading ("LIVE TAGS IN SCOPE") plus six TextBlocks that bind to the six plant tags via inline expansion in the form `{@Tag.Plant/TankLevel} gal`. These render in real time as the tags change — the operator and the LLM both see the same snapshot.

Quick-prompt buttons

Three Buttons, each with an ActionDynamic that runs `ActionType=SetValue` on `@Tag.Chat/Query` with a pre-built question. Clicking a button stages the question; the operator can then edit it in the TextBox before sending.

Ships with: *What is the current tank level? · Summarize the current state of Reactor R-101. · Is the temperature safe given the current pressure?*

TChatSession transcript widget

The native FrameworX transcript widget. Renders the per-Display-panel conversation history as **selectable per-role bubbles in a single left-aligned column** — user-role bubbles tinted with the platform accent color, assistant-role bubbles with the neutral shade color. Color alone discriminates role; the layout follows the conversational-AI cluster convention (Claude Code, ChatGPT, Cursor, Copilot, Linear AI) rather than the peer-to-peer messenger two-column shape.

While a request is in flight, the operator's question appears immediately as a user-echo bubble, followed by a rotating ? glyph with an elapsed-seconds counter. The pending visuals are replaced by the assistant-reply bubble when the reply lands.

Bubble bodies render as **TTextBlock** elements (the WPF TextBlock-derived control native to FrameworX) with chrome suppressed by the control. Operators select bubble text by mouse drag, then **Ctrl-C** to copy, or right-click **Copy** — the chat-UX standard. Selection is enabled on both user and assistant bubbles. The control reads the transcript directly from the runtime's per-connection cache — no DataTable binding to author, no script glue.

Width 1152, height ~350. Width auto-scales with the Display via `OnResize="StretchUniform"`.

> **Cross-target behavior.** The TChatSession control is shared between the WPF RichClient (.NET 4.8) and the HTML5 WebClient (OpenSilver/WASM). Selection + Ctrl-C copy is active on WPF today; on HTML5 the capability degrades gracefully — the bubbles render correctly with the same color and layout, and selection activates as soon as the underlying OpenSilver TextBlock surfaces the relevant property. No author wiring differs between the two targets.

Question TextBox + Ask AI button

Section label, single-line TextBox bound to `@Tag.Chat/Query`, plus the load-bearing Button — `ActionType="ChatRequest"` with `ObjectValueLink=@Tag.Chat/Query` and `ObjectReturnValueLink=@Tag.Chat/Reply`. One click sends the current Query to the LLM, receives the reply envelope on `Chat/Reply`, and the TChatSession control auto-appends both turns to its transcript.

Reply + latency footer

Two TextBlocks below the input row — the first prints the raw reply envelope (`Local AI Reply: {@Tag.Chat/Reply}`) so you can see the full JSON shape; the second extracts `{@Tag.Chat/Reply.JsonString("latencyMs")}` for the round-trip latency in milliseconds.

How to run it

1. Open **Local AI Chat Example.dbsln** in Designer.
2. Go to **Solution Capabilities AI Engine** tile. **Edit the URL and Name fields to point at your own LLM endpoint and model** (per the Prerequisites above) — the shipped values are placeholders for the author's bench setup; set the URL to the IP of the separate machine running your model. Confirm `SolutionCapabilities[LocalAI].Enabled = true` and that the status dot turns green ("Reachable") once your endpoint is set.
3. Confirm the model you wrote into the Name field is pulled on your LLM host (`ollama list` if you're using Ollama). If the model is not present, the chat surface will show an error bubble explaining what's missing — see *When the LLM is unavailable* below.
4. Click **Run**. The runtime starts and the RichClient (or your configured client) opens to the ChatTest panel.
5. Click one of the three quick-prompt buttons to stage a question, or type your own in the TextBox.
6. Click **Ask AI**. The transcript shows your prompt bubble immediately and a rotating ? pending indicator; within a few seconds the assistant bubble replaces the indicator with the model's reply. The Latency field shows the round-trip in milliseconds.

The TChatSession control retains the conversation per Display panel — you can ask follow-ups and the model has context. The transcript persists across re-opens of the same panel within one client session.

When the LLM is unavailable

The TChatSession control surfaces error states as **bubbles in the chat track**, not as a separate band or toast. When no model is reachable, the platform shows an **error bubble** — that is the default platform behavior. When you click **Ask AI**:

- If the master kill-switch is off (`SolutionCapabilities[LocalAI].Enabled = false`), your question lands as an accent bubble, the pending indicator clears, and a **danger-tinted bubble** appears below it with text: *"Local AI master kill-switch is off."*
- If Ollama is not installed or not running on the endpoint host, you see a danger bubble with the failure guidance (install / start steps).
- If the LLM endpoint is unreachable (network blocked, wrong URL), you see a danger bubble carrying the HTTP error detail.
- In every case, your typed question is preserved in the transcript — the chat thread shows what you asked and why no answer arrived. Re-enable the LLM and submit again; subsequent turns work normally.

This mirrors the Claude / ChatGPT / Cursor convention — errors are part of the conversation history, not a separate visual surface. The error bubble is what the operator sees when no model is reachable; it is not a substitute for a working model. (Some demo solutions add an optional `[Simulation Llama3]` canned-answer mode for an offline showcase — that is a demo-solution opt-in, not a platform feature, and not meaningful AI without a model. The real path is a reachable model on a separate machine.)

What to try next

- **Copy a reply.** Select text in an assistant bubble and press **Ctrl-C**, or right-click and pick **Copy**. The TChatSession's TTextBlock bubbles support standard text-selection gestures on both user and assistant bubbles — useful for pasting LLM-generated content into reports, tickets, or follow-up scripts.
- **Change a plant tag value mid-conversation.** Use Designer's **Watch** panel to drive `Plant/TankLevel` from 784.5 to 950, then ask the AI "What is the current tank level and how close are we to the cap?". The model sees the new value because the live tag expansion in the display passes the current snapshot on every call.
- **Add a fresh quick-prompt button.** Duplicate one of the three buttons and change its `ObjectValueLink` to a new question string. Click it during Run and the new question stages in the TextBox.
- **Inspect the full envelope.** The footer shows the whole reply JSON plus latency. Add a TextBlock bound to `{@Tag.Chat/Reply.JsonString("warnings")}` if you want to surface warnings; bind `{@Tag.Chat/Reply.JsonString("toolTrace")}` if you enable the UNS / Alarm / Historian tool bits on `SolutionSettings.ModelOptions` (see Local AI Configuration) and want to inspect autonomous tool calls.
- **Point the LLM at a separate machine.** The model should run on a machine other than the TServer host from the start. On a second machine (or a sibling VM), install Ollama there, pull the model on that host, and set the `URL` field in `SolutionCapabilities[LocalAI].Settings` to that machine's IP or hostname — `http://192.168.1.50:11434/v1/chat/completions`, replacing 192.168.1.50 with the IP of the machine running your model. The example works unchanged — TServer never fights the LLM for CPU. Full guidance on remote and cloud placement: Remote and Cloud LLM Models and Local AI Deployment and Performance.
- **Try a cloud LLM endpoint.** Point `URL` at any OpenAI-compatible chat-completions endpoint (cloud or otherwise); store the API key in SecuritySecrets and reference it via `/secret:<Name>` in the `Authorization` field. See SecuritySecrets Authentication for Local AI.

System prompt

The runtime applies a short default system prompt to every chat call so replies stay terse and operator-friendly:

```
You are a helpful assistant embedded in the FrameworX runtime. Answer concisely and directly — no preamble,
no status wrapper, no echo of the user's question. On tool failure, reply with only the word: error.
```

To override it for a specific call, send a JSON query payload with a top-level `system` field instead of the bare prompt string. The override applies for that call only; the next bare-string call falls back to the default. Server-side `AI.Execute` calls follow the same contract.

Role labels — tolerant matching

The TChatSession control distinguishes user-role from assistant-role bubbles via the `UserRoleValue` and `AIRoleValue` properties. **The example uses the defaults: `user` / `assistant`** — the literal role strings the runtime writes into the transcript. Leave these alone unless you have a reason to change them.

The control matches role labels case-insensitively and also recognises common aliases on each side, so persona overrides applied elsewhere on the Display still tint correctly:

- **User aliases:** `user`, `operator`, `human`, `question`, `you`, `me`, `customer`, `client`, `person`, `prompt`.
- **Assistant aliases:** `assistant`, `ai`, `bot`, `agent`, `response`, `answer`, `plant ai`, `system ai`, `system`, `model`, `copilot`, `reply`.

For persona labels visible on the Display ("Operator", "Plant AI", etc.), add a separate TextBlock above the transcript — do not push the persona text into `UserRoleValue` / `AIRoleValue`. The control's contract is to match what the runtime writes; the persona is a Display-level affordance.

The recommended model — 7B, larger models, remote endpoints

`qwen2.5:7b-instruct` is the recommended default for this and any solution — best reply quality and autonomous tool-call reliability; run it on a separate GPU machine, the floor even for demos. CPU-only produces only ~2-4 tokens/sec, too slow even for a demonstration; `qwen2.5:3b-instruct` is at most a last-resort for single-shot atomic tasks, never interactive chat. For real use run 7B on a separate GPU machine, or step up to `qwen2.5:32b-instruct` on a strong GPU:

1. Pull the production model on the LLM host: `ollama pull qwen2.5:7b-instruct` (~4.7 GB).
2. In Designer, open the solution and go to **Solution Capabilities**.
3. On the **AI Engine** tile, click **Edit Configuration**.
4. Paste the JSON below into the dialog and save (set `URL` to the IP of the machine running your model):

Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

```
{
  "URL": "http://192.168.1.50:11434/v1/chat/completions",
  "Name": "qwen2.5:7b-instruct",
  "Authorization": "NoAuth",
  "Headers": "",
  "Info": "Recommended default — 7B-instruct (~4.7 GB). Best reply quality and tool-call reliability. Replace
192.168.1.50 with the IP of the machine running your model (not the FrameworX/TServer host).",
  "TimeoutSeconds": 60
}
```

Replace 192.168.1.50 with the IP of the machine running your model (the model must not run on the FrameworX/TServer host). Restart the runtime; the next chat call uses the production model. Replies are noticeably richer and autonomous tool dispatch is more reliable, in exchange for higher latency per call. For remote endpoints, cloud LLMs, and the full topology decision tree, see Remote and Cloud LLM Models and Local AI Deployment and Performance.

Performance baseline — what to expect

Indicative latencies for the recommended `qwen2.5:7b-instruct` model, CPU-only inference on a typical x64 development laptop (Windows 11, no GPU, 16 GB RAM) — shown only to illustrate why CPU-only is unusable: at ~2-4 tokens/sec it is too slow even for a demonstration. 7B is meant to run on a separate GPU machine. The `qwen2.5:3b-instruct` fallback runs roughly **2× faster** on the same hardware in exchange for shorter, less nuanced replies — still a last-resort for single-shot atomic tasks only, never interactive chat. The first call after runtime start pays a model-load cost; subsequent calls run against the warm model.

Question shape	First call (cold model)	Subsequent calls (warm)
Short factual (e.g. “what is the current tank level?”)	~5 s	1.5–3 s
Three-bullet list (~40 tokens out)	~17 s	11–13 s
Two-sentence summary with one tag (~50 tokens)	~25 s	17–25 s
Two-sentence summary across six tags (~55 tokens)	~21 s	9–15 s
Two-sentence reasoning with hedge (~85 tokens)	~28 s	9–15 s

Numbers scale roughly linearly with output token count: token-throughput on CPU is only ~2–4 tokens/second across all cores — the reason CPU-only is too slow even for a demonstration. The `qwen2.5:3b-instruct` model is ~2× faster than 7B on the same hardware (trade against reply quality). Running the model on a separate GPU machine cuts latency 3–5× and is the floor even for demos. For deeper sizing guidance — including when latency in the same-host topology starts hurting other runtime work — see Local AI Deployment and Performance.

How it relates to the deeper AI Integration features

This example is intentionally minimal — it demonstrates only the ChatRequest action plus the TChatSession transcript widget. For richer patterns:

- Local AI Ontology Demo — a full shipping solution with operator chat, alarm-driven anomaly narration, knowledge-graph asset tree, and end-to-end grounded narrative generation.
- The three server-side `AI.Execute` examples on the Local AI parent page show the atomic script path (alarm root-cause hypothesis, multi-language alert translation, end-of-shift summary).
- Local AI Configuration covers tool-category bits (`SolutionSettings.ModelOptions`), alternate endpoints (cloud LLMs, other local models), and SecuritySecrets-backed authorization for non-local endpoints.
- Local AI Deployment and Performance covers the hosting topologies (same-host, sibling VM, dedicated host, cloud), the workload × topology decision matrix, and the sizing guidance — essential reading before moving past a workshop / single-operator deployment.
- Local AI Developer Reference covers reply-envelope schema details, cached-path hooks, the AI.Execute deep semantics, and the TChatSession control reference.

In this section...