SecuritySecrets Authentication for Local AI

How to store an LLM endpoint's API key (or any other Local AI credential) in the SecuritySecrets vault and reference it from `SolutionCapabilities[LocalAI].Settings`, so the credential is never visible in plain text and never appears in solution exports.

Why SecuritySecrets

A real LLM endpoint typically requires an API key, a Bearer token, or some other authentication header. Hard-coding that credential into `SolutionCapabilities[LocalAI].Settings` is a bad idea — the value would appear in plain text in the solution file, in solution exports, in screenshots from the Designer dialog, and in any backup of the configuration database.

FrameworX has a platform-native way to handle this: the **SecuritySecrets** table. Customers store the actual credential in a row of SecuritySecrets, then reference it from any configuration field by name using the `/secret:<Name>` token. The platform substitutes the real value at call time, server-side only. The credential never appears in `Settings`; it never travels to a thin client; it's encrypted at rest.

This is the same mechanism that protocol drivers, the WebData connector, the UNS tag-provider service, and the Devices module use for their own credentials. Local AI consumes the same pattern — no new mechanism specific to AI.

The pattern in three steps

1. Create a **SecuritySecret** row with a Name and a SecretValue. The SecretValue is the actual credential.
2. In `SolutionCapabilities[LocalAI].Settings`, reference it by name using a `/secret:<Name>` token in the `Authorization` field (or in `URL`, or in `Headers`).
3. Local AI resolves the token at call time, server-side, and uses the real value when POSTing to the LLM endpoint.

Step 1 — Create a SecuritySecret

In the Designer, navigate to **Security Secrets**. Add a row:

Field	Value
`Name`	A short, descriptive identifier — for example, `CloudLLMApiKey`. This is the name you'll reference in `Settings`.
`SecretValue`	The actual credential string — for example, the API key from your LLM provider. The Designer encrypts this value before storing it.
`Description`	Free text. Recommended: note the provider, the issue date, and the rotation policy.

The SecretValue field is **write-only** — once saved, the Designer redacts the field on read-back. To rotate the secret, type a new value into the field; the platform replaces the encrypted blob.

Step 2 — Reference it from Settings

Open **Solution Capabilities** (or the Local AI tile under Data Servers — both routes edit the same capability row) and click **Settings**. In the `Authorization` field, write a multi-line value: line 1 is the auth scheme, line 2 is the value (which can be a `/secret:<Name>` token).

Bearer token from a SecuritySecret

```
BearerToken
/secret:CloudLLMApiKey
```

The full `SolutionCapabilities[LocalAI].Settings` JSON for an OpenAI-compatible cloud endpoint with Bearer auth looks like:

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

```
{
  "URL": "https://api.example-llm-provider.com/v1/chat/completions",
  "Name": "the-cloud-model-name",
  "Authorization": "BearerToken\n/secret:CloudLLMApiKey",
  "Headers": ""
}
```

The `\n` represents the newline that separates the scheme from the value.

Basic Auth from a SecuritySecret

BasicAuth uses a base64-encoded `user:password` string on line 2. Store the base64 value (or the raw `user:password` — the dialog handles encoding) in the SecuritySecret:

```
BasicAuth
/secret:CloudLLMBasicAuth
```

Custom auth header

For endpoints requiring a custom header (e.g., `X-API-Key`):

```
CustomAuth
X-API-Key: /secret:CloudLLMApiKey
```

SecuritySecrets in URL or Headers

The `URL` and `Headers` fields also accept `/secret:<Name>` tokens. Useful when the endpoint URL embeds a key, or when extra headers carry credentials:

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

```
{
  "URL": "https://api.example-llm-provider.com/v1/chat/completions",
  "Name": "the-cloud-model-name",
  "Authorization": "BearerToken\n/secret:CloudLLMApiKey",
  "Headers": "X-Organization-Id: org-12345\nX-Project-Id: /secret:CloudLLMProjectId"
}
```

Step 3 — Resolution at call time

Each Local AI call (`ChatRequest` action or `AI.Execute` — canonical script API from `T.Toolkit.LocalAI`; legacy alias `TK.AIExecute` forwards to it) does the following before POSTing to the LLM:

1. Reads `SolutionCapabilities[LocalAI].Settings` JSON.
2. Walks the `URL`, `Authorization`, and `Headers` fields for `/secret:<Name>` tokens.
3. For each token, looks up the matching `Name` in the SecuritySecrets table and substitutes the decrypted `SecretValue`.
4. Decodes the multi-line Authorization wire format into an HTTP `Authorization` header.
5. POSTs to the LLM endpoint with the resolved values.

The resolved values exist only in memory in the TServer process for the duration of the call. They are never written to logs, never echoed to clients, never serialized to a solution export.

Off-server safety net

SecuritySecrets resolution is a server-side operation. When a call reaches Local AI from a context where the secret resolver cannot run (a thin-client ObjectServer, a Display CodeBehind script), the resolver short-circuits and returns the unresolved literal token (e.g., the literal text `/secret:CloudLLMApiKey` remains in the auth header).

The result is one of:

- The LLM endpoint receives the literal `/secret:...` string as the credential, rejects with a 401, and the Local AI call returns a normal HTTP-error envelope.
- The platform's other safety nets (master kill-switch, defensive defaults pointing at unreachable localhost) intercept first and return `status="disabled"` or an HTTP-unreachable error.

In neither case does the platform silently transmit unauthenticated traffic. There is no scenario where a misrouted call from a client context succeeds with leaked credentials.

Encryption at rest

The SecuritySecrets table stores the `SecretValue` encrypted. The Designer decrypts on read for editing only when the user has the appropriate Security permission; the field is redacted on read-back even when displayed in the dialog. Solution exports include the encrypted blob; restoring the export on a different installation requires the same encryption key.

Worked example — point Local AI at a cloud endpoint

A fresh 10.1.5 solution defaults to local Ollama. The customer wants to point it at a cloud-hosted OpenAI-compatible LLM with their API key.

Steps

1. In the Designer, navigate to **Security Secrets**. Add a row: Name = `CloudLLMApiKey`, SecretValue = the API key the provider issued. Save. The dialog redacts the SecretValue field on next display.
2. Open **Solution Capabilities** and find the **Local AI** row. Click **Settings** to open the 5-field editor.
3. Enter the endpoint:
    - `URL`: `https://api.example-llm-provider.com/v1/chat/completions`
    - `Name`: the model identifier the provider uses (e.g., `provider-model-v3`)
    - `Authorization`: line 1 = `BearerToken`, line 2 = `/secret:CloudLLMApiKey`
    - `Headers`: leave empty unless the provider requires extras
    - `Info`: free text — note the provider and the model
4. Save. Verify the Local AI status indicator turns green (the reachability probe issues an HTTP request with the resolved credential and checks for a 2xx).
5. Send a test query through a Display ChatRequest action or via `AI.Execute`. Verify the reply envelope returns `status="ok"`.

What gets stored where

Where	What
`SecuritySecrets` row `CloudLLMApiKey`	The actual API key, encrypted at rest. Only readable by the runtime in a server-side context.
`SolutionCapabilities [LocalAI].Settings`	The literal text `/secret:CloudLLMApiKey` in the Authorization field. No credential value here.
Solution export file (`.dbsln`)	Both the encrypted SecuritySecrets blob and the unresolved `/secret:` reference in the Local AI Settings. Restorable on installations sharing the encryption key.
HTTP request to the LLM endpoint	The fully-resolved `Authorization: Bearer <real-api-key>` header. In memory only, never persisted.

Rotating a secret

1. In **Security Secrets**, find the row by Name.
2. Type the new value into `SecretValue`. The dialog accepts the new value but does not display the old.
3. Save. The next Local AI call uses the new value automatically — no Local AI configuration change, no restart.

Sharing one secret across consumers

SecuritySecrets is a flat per-solution namespace shared across the entire platform. The same row `CloudLLMApiKey` referenced from the Local AI Authorization field can also be referenced from any `ReportsWebData` row's Authorization, any protocol driver's credential field, or any custom script via `TK.GetSecret(name)`. Updating the row updates every consumer simultaneously.

What this page does NOT cover

- **Setting up the SecuritySecrets table itself** (Security UI navigation, permissions, encryption key management). See the platform's Security documentation.
- **Configuring the Local AI endpoint** beyond the auth field. See Local AI Configuration Reference.
- **Calling Local AI from a script or Display**. See AI.Execute API Reference and ChatRequest Action Reference.

In this section...