

NLIP Server: Go

[GitHub Repository Link](#)

I. Introduction:

This project is a Go implementation of the NLIP server. It is built using the Echo web framework and Goth authentication framework. It demonstrates handling of various message types using the NLIP protocol along with LLAMA and LLava models.

II. Quickstart:

To rapidly set up and use the NLIP Go server, please follow the steps included in the [GitHub repository](#). The steps there are comprehensive, but here is everything you need to know:

1. You need to create a `.env` file that has the required variables.
 - a. Optional variables are also needed for all desired functionalities to work properly
2. After all the required variables are set in the `.env` file, you can simply use the script located in `./scripts/deploy.sh` to deploy the project. Make sure this script is executable by running `chmod +x ./scripts/deploy.sh`, also **make sure that all optional parts in the script are commented out if you are not using them**. Lastly, make sure all the paths in the `.env` file are correct (e.g., `EXECUTABLE_LOCATION`, `UPLOAD_PATH`, etc.)

Note: If instead, you would like to compile and run the executable yourself, you can simply run:

```
go build
go run main
```

However, if you have the launch configuration (e.g., plist for MacOS or systemd for Linux), then a deployment script makes everything easier. Likewise, the script also **signs** the executable, so that on MacOS you don't have to allow network communications every time you build the project.

III. Creating a Launch Configuration/Systemd Service File:

- What is a Launch Configuration?

- A launch service is basically a configuration that defines when and how a service should run. It is typically used **when you want some process to start running as soon as the computer is started, or when you want some program to restart when it fails an error**. This is relevant because for most back-end servers, you want the process handling clients to always run. For that reason, you can write a launch configuration that defines these behaviors.
- **IMPORTANT:** Ollama must be running for the Go backend to be able to communicate with LLMs configured with Ollama and work correctly. For this, you can either manually start Ollama whenever the computer is restarted, or better, have a launch configuration that starts Ollama on boot. So, a launch configuration is **highly recommended** for both the **nlip** executable as well as **Ollama**. The section below details how to create a launch configuration for the **nlip** executable step by step, explaining what each step does. If you want to use a launch configuration for Ollama, please check the appendix for a configuration that you can copy and paste **after understanding the general steps below**.

i) MacOS

The Go NLIP server on druid already has a launch configuration set up. This ensures that the **nlip** executable always runs upon boot and restarts whenever it crashes for some error. If you would like to see the launch configuration for **nlip**, go to **/Library/LaunchDaemons/com.nlip.plist**. The exact file of the configuration (i.e., **com.nlip.plist**) is a reverse domain name notation and it is standard to name them like this. If you open the configuration file, you will see:

```
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.nlip</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/hbzengin/src/go-server-example/nlip</string>
  </array>
  <key>WorkingDirectory</key>
  <string>/Users/hbzengin/src/go-server-example</string>
  <key>RunAtLoad</key>
  <true/> <!-- Starts at system startup -->
  <key>KeepAlive</key>
  <true/> <!-- Restarts if it stops -->
  <key>StandardOutPath</key>
  <string>/var/log/nlip_output.log</string>
  <key>StandardErrorPath</key>
```

```
<string>/var/log/nlip_error.log</string>
</dict>
</plist>
```

This file follows the XML format. **Label** is the unique identifier for the application, **ProgramArguments** is the arguments needed for the program—here it is only the path to the executable. **WorkingDirectory** is the “root” of the project—it is useful when you use relative paths inside your code. **RunAtLoad** and the `<true/>` flag ensures that the application runs at boot. **KeepAlive** ensures that the application is restarted if it, for any reason, stops. **StandardOutPath** redirects the stdout of the program to a specified path. **StandardErrorPath** redirects the stderr of the program to the specified path. Since the program runs indefinitely, you will be able to see the output of the program in these files. See the **logging** part below to see how to view the contents of these files.

ii) Linux

To set up a launch configuration for Linux, follow the steps below.

1. Run **sudo nano /etc/systemd/system/nlip.service**
 - a. This creates the new configuration file for nlip
2. Add the content below to the file you just opened:

```
[Unit]
Description=Run nlip on boot and keep it running
After=network.target
[Service]
ExecStart=<path_to_executable>
Restart=always
RestartSec=5
User=your_username
WorkingDirectory=/path/to/working/directory
[Install]
WantedBy=multi-user.target
```
3. Run **sudo systemctl daemon-reload**
 - a. This reloads the systemd manager configuration
4. Run **sudo systemctl enable nlip.service**
 - a. This enables the service to start on boot
5. Run **sudo systemctl start nlip.service**
 - a. This starts the service right now
6. Run **sudo systemctl status nlip.service**
 - a. This shows that if the service is actually running as expected
7. To access the logs, run **journalctl -u nlip.service**

- a. This logging behavior is different from MacOS but allows you to access the content

IV. Logs

i) MacOS

The script located at `./scripts/monitor_logs.sh` contains a single line that defines how to see the stdout and stderr output in the log. Simply run this script, or the command: “**tail -f /var/log/nlip_output.log /var/log/nlip_error.log**” to see the latest changes in the log.

ii) Linux

If you have followed all the steps in III. ii. for Linux, you can simply run **journalctl -u nlip.service** to see the logs for the running nlip executable.

V. Documentation

The Go server provides a RESTful API for the NLIP protocol and supports handling various formats. It is currently being updated based on changes in the protocol specifications.

i. Dependencies:

1. **Echo**: Web framework for Go
 - a. Used because it simplifies logic related to handling requests and responses as well as errors. **Not** critical for implementation as the same work can be done with more lines of code using the Go standard library
2. **Goth**: OAuth2 authentication package for Go
 - a. Used because it simplifies the logic related to the OAuth2 flow and simplifies configuring new providers with very few lines of code. **Not** critical for implementations as the same work can be done by implementing your own Go OAuth2 client
3. **Ollama**: Backend for LLAMA and LLava models
 - a. Critical. Used to run the LLAMA and LLava models locally on the computer. Ollama exposes an API to make calls to the local LLMs and receive responses. LLAMA handles text based responses and LLava handles image based responses.

ii. Core Features:

1. OAuth2 authentication with Google OAuth2 and a custom OpenID Connect provider
2. File uploads using form submissions
3. Request handling for various data formats (for now text and image)
4. Integration with LLAMA and LLava models for text and image analysis through a LLM.

iii. Endpoints

1. /auth/

- a. **GET /auth/**: Provides login links for Google and Custom OpenID Connect (Client should directly use this)
- b. **GET /auth/:provider/**: Initiates login for the specified provider (Client indirectly uses this)
- c. **GET /auth/:provider/callback/**: Defines the specific provider's callback and returns user data (Client indirectly uses this)

2. /nlp/

- a. Handles messages in various formats. This is the main endpoint where all requests should be sent.

- b. Text format:

```
{  
  "format": "text",  
  "subformat": "english",  
  "content": "Tell me a fun fact"  
}
```

- c. Binary (Image) format (note the base-64 encoded nature of the image)

```
{  
  "format": "text",  
  "subformat": "english",  
  "content": "Describe this picture",  
  "submessages": [  
    {  
      "format": "binary",  
      "subformat": "jpeg",  
      "content": "<base-64-encoded-image>"  
    }  
  ]  
}
```

3. /upload/

- a. **POST /upload/**: Accepts file uploads using form upload. Saves files to the specified **UPLOAD_PATH** in the **.env** file. The **key** of the form entry must be **“file”**

iv. Packages and Code Architecture

In Go, source files are organized in “packages” based on the functionality they provide. This helps with writing modular and more organized code. This backend has the following packages:

1. **auth** package:

- a. Handles OAuth provider setup using Goth

- b. Defines the routes for authentication using OAuth providers and registers with the router
- 2. **handlers** package:
 - a. Implements handlers for the main NLIP logic. Each handler is associated with one endpoint
 - i. Text messages are passed to the LLAMA model
 - ii. Binary messages (images) are processed using the LLava model
 - b. File uploads are supported with validation and optional storage
- 3. **llms** package:
 - a. Defines helper functions to communicate with Ollama and its LLMs using its API
 - b. Supports various formats (currently text and image-based requests)
- 4. **models** package:
 - a. Defines data structures for message formats and their validation

VI. Technical Implementation Details

This section discusses technical details related to the implementation. More specifically, it discusses the OAuth2 flow and adding new providers using the Goth library.

i. OAuth2

a) What is OAuth2?

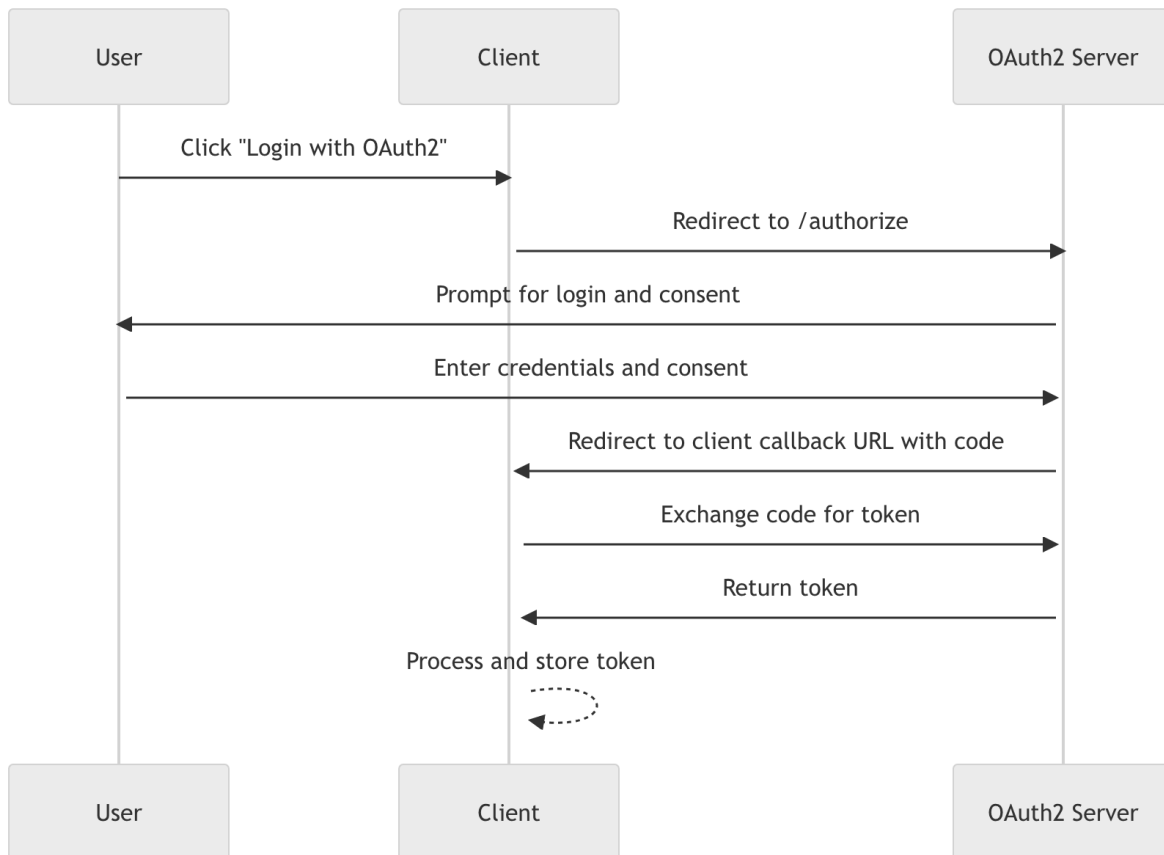
OAuth2 is an authorization framework that allows applications to obtain a limited amount of user data (based on scopes) from OAuth providers. It allows for user authentication using a third party such as Google, Facebook, Discord, etc.

These providers are typically services users already have accounts on. For example, almost everyone today has Google accounts. As a secondary service, they also allow developers to register their applications with them and use their authentication service. These third parties then give applications access to authorization data (i.e., tokens) or identity data (i.e., username, email, etc.). The data sent from the third party depends on what the application asks for when registering. Likewise, the users are notified of what the app is asking to access.

There is another layer, OpenID Connect (OIDC) that is built on top of OAuth2 to make it more robust. In fact, from my research, I have found that most providers implement and expose OIDC instead of OAuth2 to applications. This is because OAuth2 is **only required** to send back authorization tokens, without really identifying who the user is. However, OIDC extends this by returning **different kinds of user information (name, email, etc.) as well as an identity token**. The line between OIDC and OAuth2 today is very subtle, so it is possible that you are using OIDC when registering for OAuth2 on a service.

b) What does the OAuth2 flow look like?

As a reference you can use [this website to learn more](#) about how OAuth2 works. This explanation is from a company that provides various kinds of identity verification services to its customers. You can find an image below, from the same website, that outlines how the flow of a typical OAuth2 authorization works. It is also explained step by step. As you are walking through the steps, you can refer back to the chart to follow along. The outlined steps are described in a way that matches the chart to make it easy to follow.



c) Step by step explanation of the OAuth2 flow

First of all, it is important to understand who the parties are in this image. **The OAuth2 server** is the third party providing the OAuth2 authorization/authentication service. This could be Google, Reddit, or a custom OAuth2 server (like Tom's in the case of NLIP). **The Client** is the service where the application logic runs. In this case, it is the Go backend that redirects the user to the OAuth2 server for authentication. **The User** is the user, using a browser or a mobile app, interacting with the previously mentioned application. As an aside, since the "**Client**" in our case is the "**Backend**" in our Go NLIP example, there also needs to be a "**Frontend**" that interacts with this backend. The **Frontend and Backend** combined make up the "**Client**".

Here are the steps:

1. The user indicates they would like to use OAuth2 to authenticate themselves in the application. In the Go NLIP server this means making an API call to the backend's **/auth/** endpoint (i.e. <https://druid.eecs.umich.edu/auth/>).
 - a. This endpoint serves a very basic HTML page that shows what OAuth2 providers are configured with the backend. An image is attached below.
 - b. The user then selects the OAuth2 provider they would like to use.

Login Page

[Login with Google](#)

[Login with Custom OIDC](#)

2. Then, clicking on either of these links redirects the user to the specific backend endpoint that handles the authentication (refer to the auth.go file for more details).
3. After this the user is redirected to the page of the OAuth2 provider (for example, Google's or Tom's backends).

NLIP Login Form

Username:

Password:

Login

4. The user is asked to sign in with this third party provider (using Google username and password, etc.). Upon successful login, the user is asked to consent for the data the application wants to access.

Authorize nlipapp?

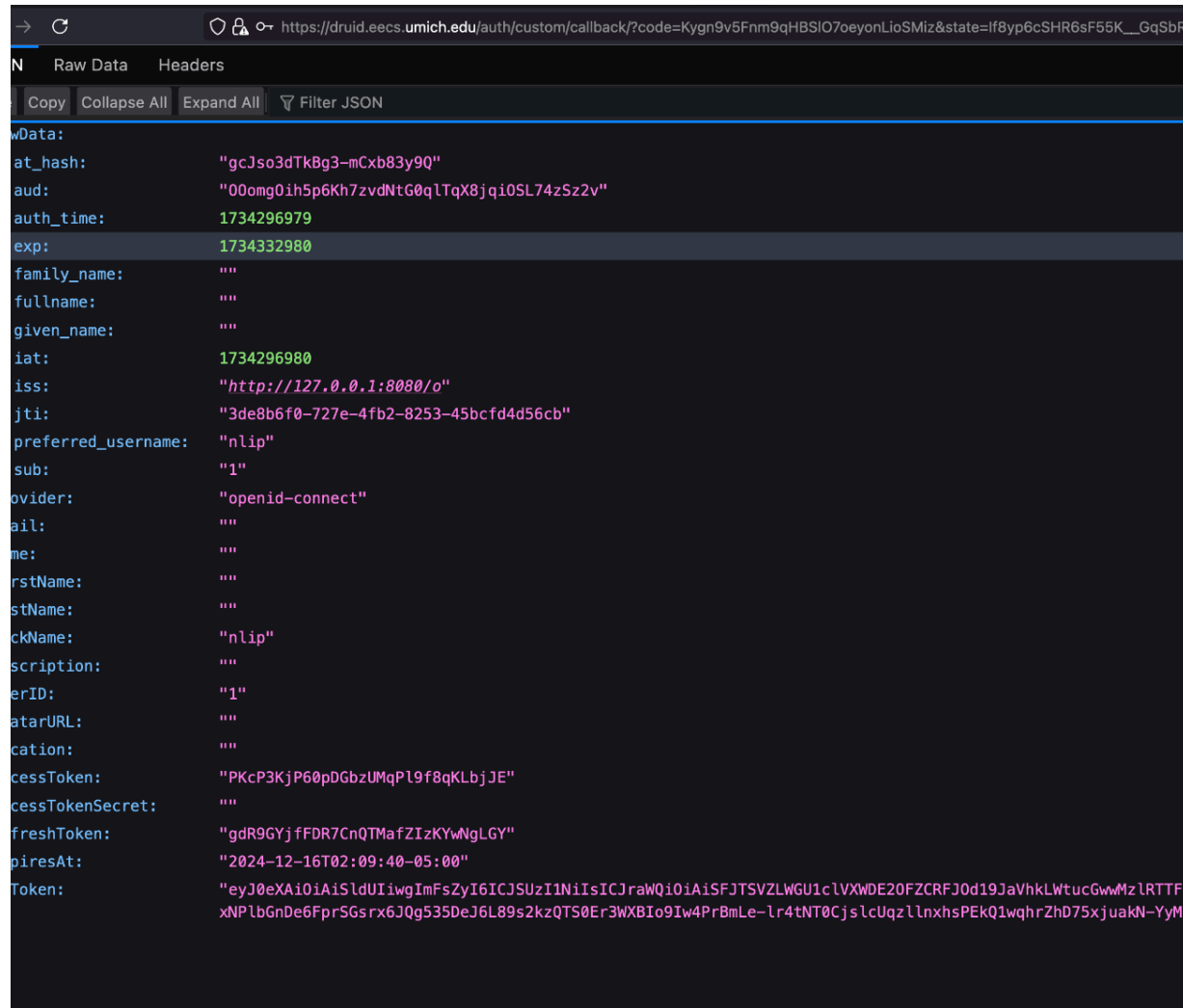
Application requires the following permissions

- Open ID Connect Scope

Cancel

Authorize

5. Upon receiving consent, the OAuth2 provider redirects the user back to the endpoint on the Client (**Go Backend**) that is expected to handle the response from the provider. This response includes the information application asked to receive (such as authorization and identity tokens as well as user information).



```
→ https://druid.eecs.umich.edu/auth/custom/callback/?code=Kygn9v5Fnm9qHBSIO7oeyonLioSMiz&state=If8yp6cSHR6sF55K__GqSbf
N Raw Data Headers
Copy Collapse All Expand All Filter JSON
responseData:
  at_hash: "gcJso3dTkBg3-mCxb83y9Q"
  aud: "00omg0ih5p6Kh7zvdNtG0qLTqX8jqI0SL74zS2v"
  auth_time: 1734296979
  exp: 1734332980
  family_name: ""
  fullname: ""
  given_name: ""
  iat: 1734296980
  iss: "http://127.0.0.1:8080/o"
  jti: "3de8b6f0-727e-4fb2-8253-45bcfd4d56cb"
  preferred_username: "nlip"
  sub: "1"
  provider: "openid-connect"
  email: ""
  name: ""
  firstName: ""
  lastName: ""
  nickName: "nlip"
  description: ""
  erID: "1"
  avatarURL: ""
  location: ""
  accessToken: "PKcP3KjP60pDGbzUMqP19f8qKLbjJE"
  accessTokenSecret: ""
  refreshToken: "gdR9GYjffDR7CnQTMafZIzKYwNgLGY"
  expiresAt: "2024-12-16T02:09:40-05:00"
  Token: "eyJ0eXAiOiAiSldUIiwgImFsZyI6IChJSUzI1NiIsICJraWQiOiAiSFJTSVZLWGU1cVXxwDE20FZCRFJ0d19JaVhkLWtucGwwMzlrRTTFxNP1bGnDe6FprSGsrx6JQg535DeJ6L89s2kzQTS0Er3WXBIo9Iw4PrBmLe-lr4tNT0CjslcUqzllnxhsPEkQ1wqhrZhD75xjuakN-YyM"
```

6. After the **backend part of the client** receives this information from the OAuth2 provider, it is supposed to send it to the **frontend part of the client**, which must in turn store this in cookies, or session storage. **The frontend is responsible for** sending the token(s) with future requests. **The backend is responsible for** verifying the tokens and showing users the content they are authorized to see. The backend might also need to have a database that maps active tokens to user information.

d) Registering your application with a OAuth2 provider

Before being able to add the OAuth2 logic to your code, you must register your application with the OAuth2 provider and acquire some secrets. Here are the steps for registering your application with Google. Most providers have similar steps, and this is an example of what the typical flow looks like:

1. You need to log into <https://console.cloud.google.com/>
2. You need to create a new "Project" (Google calls them projects) to be able to access Google services. Google always wants to associate what you are doing on their website with a "Project" you create, for hierarchical purposes.
3. Then, from the search bar on the top, search for "Credentials"
4. Click on "Create Credentials" and then "OAuth Client Id".
 - a. If you haven't already, Google will ask you to first create a "Consent screen". This is basically (for a "formal" application), who the developer is, who the users can contact if issues arise, if you allow internal (e.g. only UM) or external (everyone) users. You have to fill this out.
 - b. NOTE: At this point, the application also asks for a "Redirect URI". This is essentially where Google will redirect the user, with a token, after "Signing In" using Google is completed. You must have a "callback" endpoint created on your application that handles this. After you create an endpoint to handle Google redirects, you should add that in the form you are filling out on Google Cloud Console at this step. **Please refer to the .env and [auth.go](#) files on druid for examples of the practical use of this path**
 - c. After (a) and (b) are completed, you can successfully get the **ClientID** and **ClientSecret**, which are the main things you need from a provider.

e) Adding a new OAuth2 provider using the Goth library

After registering your application with the OAuth2 provider in step d), you can add the OAuth2 logic to your code. All of these changes will go into .env and the [auth.go](#) files. First of all, for convenience, add all of your secrets inside the .env file. On druid's .env it looks like this:

```
# Variables for CustomOAuth Server
GOOGLE_CLIENT_ID=<ClientIdObtainedByProvider>
GOOGLE_CLIENT_SECRET=<ClientSecretObtainedByProvider>
GOOGLE_URL_CALLBACK=<CallbackURLDefinedByYou>
```

After adding your own entries for the new provider, continue.

1. In the **auth.go** file, configure your new provider using the function **goth.UseProviders()**. This function takes an argument such as `google.New(os.Getenv("GOOGLE_CLIENT_ID"), os.Getenv("GOOGLE_CLIENT_SECRET"), os.Getenv("GOOGLE_URL_CALLBACK"))`. To find how it is done for the new provider you are adding, refer to the [examples provided by the Goth package here](#).
2. After configuring the new provider, modify the **GET** handler for **/auth/**. This endpoint serves a **HTML** page that routes the user to your endpoint that redirects the user to the OAuth2 provider. Refer to the two examples there. You must add a new entry such as `<p>Login with Google</p>`. The **/auth/google** route should match the name of the provider used when configuring your new provider in step 1.
3. You are done! The **GET** routes for **/auth/:provider/** and **/auth/:provider/callback/** are generic and automatically handle the logic for all providers. Their implementation is given in the [examples provided by the Goth package here](#). So as long as you configure your new OAuth2 provider in step 1 and add it to the HTML page served by the **/auth/** endpoint, everything will be set up.

NOTE 1: If you are setting up a custom OIDC provider (like Tom's server), OIDC expects a fourth variable. The first three are necessary for OAuth2: ClientID, ClientSecret, CallbackURI. The fourth one is called **CUSTOM_DISCOVERY_URL**. This is the endpoint a client is expected to make a request to, to discover information about the OIDC provider. If you are ever to set up another custom provider, please refer to the **.env** and **auth.go** files on druid to see how it can be done.

NOTE 2: The Goth package [documentation](#) and [examples](#) are very comprehensive. It is recommended that you use it as a reference if adding new providers.

VII. Pitfalls and TODOs

For the most up-to-date Pitfalls and TODOs on the server, please check the [GitHub repo's relevant section](#). This part will have the latest information and will be more accurate

VIII. Questions?

If any questions arise, feel free to contact hbzengin@umich.edu.

IX. APPENDIX

1. Ollama Launch Configuration

- a. This is located at **/Library/LaunchDaemons/com.local.ollama.plist** on druid.

This is required as Ollama must be running for the Go backend to be able to communicate with it and LLMs configured with it. Below are the contents of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.local.ollama</string>
  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/ollama</string>
    <string>serve</string>
  </array>
  <key>EnvironmentVariables</key>
  <dict>
    <key>HOME</key>
    <string>/var/root</string>
  </dict>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <true/>
  <key>StandardOutPath</key>
  <string>/tmp/ollama.out</string>
  <key>StandardErrorPath</key>
  <string>/tmp/ollama.err</string>
</dict>
</plist>
```

