Using NixOS and Nix with Pip Install DotEnv

Technical Journal Entry Begins

Building A Python Environment In Nix Requires Some Workarounds

While **Nix** is flexible, you still must manage software and dependencies the Nix way to reap its benefits. That deterministic re-building of the exact same environment is the point, after all. However, when you're building a Python environment, you might need to use libraries that aren't yet in the Nix package management system, such as fast-html. The obvious solution is to simply pip install them the classic (non-Nix) way. The previous article covered the basic template to do that.

Keeping Pip Installs Nix-Like Without Nix Packages

Even when you need packages that aren't yet in the Nix package management system, there are many options within Nix to keep your pip installs Nix-like, such as mach-nix, poetry2nix, dream2nix, or packaging them in the Nix packaging system yourself. You can package them within your flake using buildPythonPackage in combination with fetchPypi or fetchFromGitHub, so there is no shortage of options to still do it the Nix way. However, I found all these options excruciating and ended up just wanting to pip install.

Overcoming Nix Virtual Environment Challenges

My first attempts at setting up a Python virtual environment classically failed due to "impure" contamination between the Nix parent environment and the child .venv, until I found this Reddit thread and this GitHub repo. The challenge is creating a **pre-built** Python virtual environment under nix with a bunch of stuff pre-pip-installed from where the user can step in and start freely pip installing without the burden of nixy things. The last they should see nix is when they type nix develop to get the environment. With that, I patterned this starting point where the only thing in the Nix parent environment is Python itself and the essentials required for extremely custom environment building, beyond even Nix...

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-unstable";
    flake-utils.url = "github:numtide/flake-utils";
  };
  outputs = inputs @ {
    self,
    nixpkgs,
```

```
    flake-utils,
    ...
  }:
    flake-utils.lib.eachDefaultSystem (system: let
      pkgs = import nixpkgs { inherit system; };
      envWithScript = script:
        (pkgs.buildFHSUserEnv {
          name = "python-env";
          targetPkgs = pkgs: (with pkgs; [
            python3
            python3Packages.pip
            python3Packages.virtualenv
            # Support binary wheels from PyPI
            pythonManylinuxPackages.manylinux2014Package
            # Enable building from sdists
            cmake
            ninja
            gcc
            pre-commit
          ]);
          runScript = "${pkgs.writeShellScriptBin "runScript" ( "
              set -e
              test -d .nix-venv || ${pkgs.python3.interpreter} -m venv .nix-venv
              source .nix-venv/bin/activate
              set +e
              pip install --upgrade pip --quiet
              pip install -r requirements.txt --quiet
            " + script )}/bin/runScript";
        }).env;
    in {
      devShell = envWithScript "bash";
    });
}
```

And that's where this article begins, because it's now time to build something **extremely custom**.

Managing Environment Variables

All projects begin with some sort of **environment variables** to store secret API tokens, keys, and similar sensitive information. The whole point of these Nix flakes is to have sharable, highly reproducible environments that fix the "not on my machine" problem. However, when it comes to client secrets, you can't just include them in the repository. Instead, we've got to use a "bring your own keys" approach.

Using dotenv for Environment Variables

Generally, for setting up environment variables in Python these days, the pip-installable dotenv package is the preferred solution. I'll use that, especially since it now has a set_key method, which allows you to write new keys prompted from the user into the .env file. By convention, this file is used by the package and contains lines such as:

```
# Development settings
DOMAIN=example.org
ADMIN_EMAIL=admin@${DOMAIN}
ROOT_URL=${DOMAIN}/app
```

To activate this capability of reading such files easily from our Python code, we add python-dotenv to the requirements.txt currently in Pipulate:

```
python-dotenv
requests
numpy
pandas
sqlitedict
python-fasthtml
jupyterlab
jupyter_ai
matplotlib
nbdev
```

This **pipulate environment** provides the broad brush strokes for laying down a data science, web development, and soon SEO, fresh canvas upon which to perform very Pythonic development work—**but under Nix**. And by under Nix, I mean on any Mac, Windows or Linux machine, because they can all install and run this app without the convoluted system setup rigamarole of every other approach I've ever encountered. Sure, this has its own rigamorle, but it's contained from top to bottom--but without containers!

Example: Using Python dotenv

Here is an example that uses Python dotenv to set a simple MESSAGE value in the environment, in perhaps its most simple Hello World form. I use the optional parameter names (it would work without using the key and default keywords because of Python rules) but I include them for clarity.

```
# Install the package first: pip install python-dotenv

from dotenv import load_dotenv, set_key
import os

# Load environment variables from .env file
load_dotenv(dotenv_path='.env', override=True)

# Get the value of MESSAGE from the environment
message = os.getenv(key='MESSAGE', default='Hello, World!')
print(f"First message: {message}")

# Set a new value for MESSAGE in the .env file
```

```
set_key(dotenv_path='.env', key_to_set='MESSAGE', value_to_set='Updated Hello from .env!')

# Reload the environment variables
load_dotenv(dotenv_path='.env', override=True)

# Get the updated value of MESSAGE from the environment
updated_message = os.getenv(key='MESSAGE', default='Hello, World!')
print(f"Updated message: {updated_message}")
```

Now here is an updated version that prompts the user for the secret. The first time it runs, it will not find the environment variable, so it will use the default and display Hello World! as the first line of the output and Updated Hello from .env! as the second line. If you run it a second time, it will show Updated Hello from .env! for both lines because a .env file has been created.

```
from fasthtml.common import *
from dotenv import load_dotenv, set_key
import os

# Initialize the FastHTML application
app, rt = fast_app()

# Load environment variables
load_dotenv()

@rt("/")
def get():
    # Reload the environment variables to reflect any updates
    load_dotenv()
    secret = os.getenv('SECRET')
    if secret:
        message = P("I already know your secret")
    else:
        message = Form(method="post")(
            Label("Enter the secret:", Input(type="password", name="secret")),
            Button("Submit", type="submit")
        )
    return Titled("Secret Prompt", message)

@rt("/", methods=["POST"])
def post(secret: str):
    # Save the secret to the .env file if it hasn't been set
    if not os.getenv('SECRET'):
        set_key('.env', 'SECRET', secret)
        load_dotenv()  # Reload the .env file after updating it
        return Titled("Secret Saved", P("Your secret has been saved."))
    else:
        return Titled("Secret Already Set", P("A secret is already set. No changes were made."))

# Start the server
```

serve()

The .env file is what you **should not** include in the git repository. In fact, it's good practice to add the .env file to .gitignore to ensure that secrets don't accidentally end up in a public GitHub repository.

```
.sesskey
.env
.venv/
__pycache__/
.ipynb_checkpoints/
```

Avoiding Paid AI Services

At this point, almost everyone on the Internet will advise you to obtain your OpenAI API key or other paid services so that you can leverage their offerings. However, thanks to **Mark Zuckerberg and Facebook Meta**, you don't have to pay anything.

Implementing a Basic Chat Server with Ollama

You don't need to pip install packages like ollama, openai, langchain, or other bloated dependencies commonly seen in AI Hello World examples. If you want to implement a basic chat server in Python and have **Ollama** running locally with one of the installers from its site (or under Nix as I do), it will be recognized and found using normal HTTP requests to localhost on your machine without further dependencies. Here's how:

```
import requests
import json

def chat_with_ollama(model, messages):
    url = "http://localhost:11434/api/chat"

    payload = {
        "model": model,
        "messages": messages,
        "stream": False
    }

    headers = {
        "Content-Type": "application/json"
    }

    response = requests.post(url, data=json.dumps(payload), headers=headers)

    if response.status_code == 200:
        return response.json()['message']['content']
    else:
        return f"Error: {response.status_code}, {response.text}"
```

```
# Example usage
model = "llama3.1"  # or whatever model you have installed
conversation = [
    {"role": "user", "content": "What is the capital of France?"},
]

while True:
    response = chat_with_ollama(model, conversation)
    print("Assistant:", response)

    conversation.append({"role": "assistant", "content": response})

    user_input = input("You: ")
    if user_input.lower() in ['quit', 'exit', 'bye']:
        break

    conversation.append({"role": "user", "content": user_input})

print("Conversation ended.")
```

Securing API Endpoints and Models

Instead of showing you how to hide the OpenAI API key like everyone else, I'll move the API endpoint and the model into the .env file:

```
OLLAMA_API_URL=http://localhost:11434/api/chat
OLLAMA_MODEL=llama3.1
```

Hiding API Keys and Client Secrets in Code

Now, remove direct references to the endpoint and model from the chat code to "hide a secret." Most of the time, this involves API keys and client secrets, but since you don't need one for Ollama, this antipattern demonstrates how to externalize any configuration. It's also a great way to handle local configurations without dealing with JSON.

```
import requests
import json
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

def chat_with_ollama(model, messages):
    url = os.getenv('OLLAMA_API_URL')
```

```
    payload = {
        "model": model,
        "messages": messages,
        "stream": False
    }

    headers = {
        "Content-Type": "application/json"
    }

    response = requests.post(url, data=json.dumps(payload), headers=headers)

    if response.status_code == 200:
        return response.json()['message']['content']
    else:
        return f"Error: {response.status_code}, {response.text}"

# Example usage
model = os.getenv('OLLAMA_MODEL')
conversation = [
    {"role": "user", "content": "What is the capital of France?"},
]

while True:
    response = chat_with_ollama(model, conversation)
    print("Assistant:", response)

    conversation.append({"role": "assistant", "content": response})

    user_input = input("You: ")
    if user_input.lower() in ['quit', 'exit', 'bye']:
        break

    conversation.append({"role": "user", "content": user_input})

print("Conversation ended.")
```

Environment Variables Externalized From Git Repository

And there you have it! Anything that might be **a secret** has been **externalized** to a file that is kept out of the git repo and can be used to set whatever environment variables you need.

Handling User Input for Secrets (enter FastHTML)

But we're not done. What if you need to obtain the client secret from the user via an input prompt? In a Jupyter Notebook or command-line shell, you could use the input() function. However, if you're developing a web application, you'll need a different approach.

Implementing Secret Prompt with FastHTML

The **Pipulate environment** I'm setting up with **FastHTML** avoids PHP-patterned curly-brace templating languages like Jinja2 and can handle this in a single file. This is my Hello World of **FastHTML**. There are only two functions: one for the GET method (the default) and one for the POST method (triggered when the Submit button is pressed).

```
from fasthtml.common import *
from dotenv import load_dotenv, set_key
import os

# Initialize the FastHTML application
app, rt = fast_app()

# Load environment variables
load_dotenv()

@rt("/")
def get():
    # Reload the environment variables to reflect any updates
    load_dotenv()
    secret = os.getenv('SECRET')
    if secret:
        message = P("I already know your secret")
    else:
        message = Form(method="post")(
            Label("Enter the secret:", Input(type="password", name="secret")),
            Button("Submit", type="submit")
        )
    return Titled("Secret Prompt", message)

@rt("/", methods=["POST"])
def post(secret: str):
    # Save the secret to the .env file if it hasn't been set
    if not os.getenv('SECRET'):
        set_key('.env', 'SECRET', secret)
        load_dotenv()  # Reload the .env file after updating it
        return Titled("Secret Saved", P("Your secret has been saved."))
    else:
        return Titled("Secret Already Set", P("A secret is already set. No changes were made."))

# Start the server
serve()
```

Embracing Unconventional Approaches to Modern Web Development Techniques

This code offers an interesting blend of simplicity and unconventional approaches, showcasing both modern web development techniques and some practices that, while unconventional, could be evolving into new best practices. It uses dynamic imports, route decorators, and environment management in ways that are concise and readable, but with trade-offs that aren't always seen in larger production codebases. While **wildcard imports** and multiple load_dotenv() calls may raise eyebrows in more traditional settings, they are used here effectively in a minimalistic, self-contained application. Thank you Jeremy Howard for your unconventional thinking, bravery and initiatives.
A lot of Python coders recommend avoiding importing a whole library like this (using the import * syntax) because in large software projects it can cause problems. However, for interactive work such as in a Jupyter notebook, it works great. The fastai library is specially designed to support this kind of interactive use, and it will import only the necessary pieces into your environment.

Understanding the Abstracted Nature of the Code

This code challenges conventional wisdom by directly modifying environment files and handling server-side logic in a highly abstracted way, reflecting a move toward simpler, more expressive frameworks that aim to minimize boilerplate while delivering functional results quickly. By taking a closer look at its components, we can see how some anti-patterns—like of wildcard imports and lack of PHP-style jinja nunjucks—might be giving way to new, flexible practices in the context of lightweight, evolving applications.

Interesting Aspects of the Code

1. **Modular Imports with Wildcards**: The line from fasthtml.common import * imports everything from the fasthtml.common module. While this can be convenient, it can also introduce potential conflicts if there are overlapping names in the module. It's okay in this case because of what Jeremy says.

{:start="2"} 2. **Environment Variable Handling with dotenv**: The use of dotenv to manage environment variables is a neat way to load sensitive data from a .env file. The load_dotenv() function is called multiple times to ensure the latest environment variables are loaded after the secret is updated.
{:start="3"} 3. **Dynamic Route Decorators**: The @rt("/") decorator is used to define both the GET and POST handlers for the root URL. This is a concise way to register routes and link them to functions. The decorator-based approach feels very similar to Flask, making it intuitive for developers with prior experience in web frameworks.
{:start="4"} 4. **Password Input Field**: The code leverages the Input(type="password", name="secret") field within the form to handle secret data. This ensures that whatever the user types is hidden, following basic security principles for input forms.
{:start="5"} 5. **Form Submission Handling**: When a secret is not already set, the form appears for the user to submit a password. The form uses POST method to send the data back to the server, following best practices by keeping sensitive data out of the URL (as opposed to using a GET request).

{:start="6"} 6. **Use of set_key from dotenv**: The function set_key('.env', 'SECRET', secret) is used to dynamically update the .env file with the new secret. This writes directly into the environment file, adding persistence for the submitted data beyond the current runtime session.
{:start="7"} 7. **Conditional Logic for Secret Display**: The code handles two main cases: - If the secret is already present in the environment, it displays a message saying "I already know your secret." - If not, it provides a form for submitting a secret. This logic allows for dynamic responses based on existing state.
{:start="8"} 8. **Environment Reloading**: After saving a new secret, the environment is reloaded (load_dotenv()) to reflect the updated variables in the same session, ensuring the changes are immediately visible within the running application.
{:start="9"} 9. **HTML Abstraction for Clean Layout**: The use of the Titled and P (paragraph) functions, along with form components like Form, Label, and Input, provides an abstracted way to build HTML components. This makes the code look clean and avoids manually writing HTML strings directly within the route functions.
{:start="10"} 10. **Self-Contained Web Application**: The code represents a small, self-contained web application, complete with route handling, form submission, and environment variable management. The concise syntax and structure make it highly readable and easy to understand.
{:start="11"} 11. **Basic HTTP Methods**: The code distinguishes between GET and POST methods using methods=["POST"] for the post() function. This ensures that the form submission uses the correct HTTP method and the get() function is used only for rendering the form initially.
{:start="12"} 12. **Minimalistic Server Startup**: The serve() function at the end starts the web server, demonstrating a simple approach to running a server without complex configurations or additional commands. This simplicity is ideal for lightweight applications or quick prototypes.

Final Thoughts

The canvas is now prepared with a **generic Nix flake** that allows you to:

- Install "parent" (non-Python) software requirements
- Install Python packages through pip
- Manage API tokens by prompting the user

This sets the stage for rapid progress!

Audience Considerations

It's important to note there is a distinct audience split for this kind of information:

1. **Flake Preparers:** These users are responsible for setting up Nix flakes for distribution and sharing. They ensure everything is configured correctly so that flake users can easily utilize them.

{:start="2"} 2. **Flake Users:** These users simply pull a git repository (or use a curl command) to obtain the flake and run nix develop in that directory. The environment setup is handled automatically.
The second audience, while not preparing flakes themselves, are free to pip install as they always have in a Python environment without fear of polluting their system. They are more like data scientist developers who don't need to understand all the IT intricacies. They just answer a few questions about their API key secrets and dive into a Jupyter Notebook, Python IDE, or command-line shell—**and they're golden**.