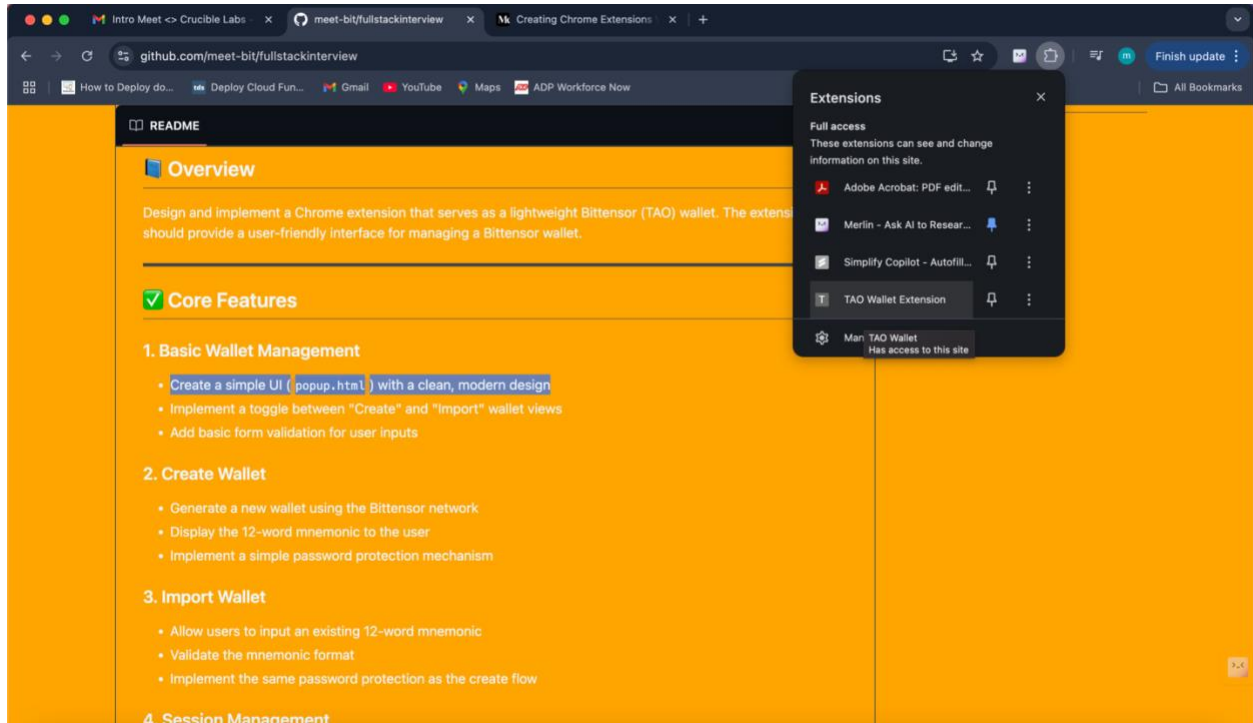


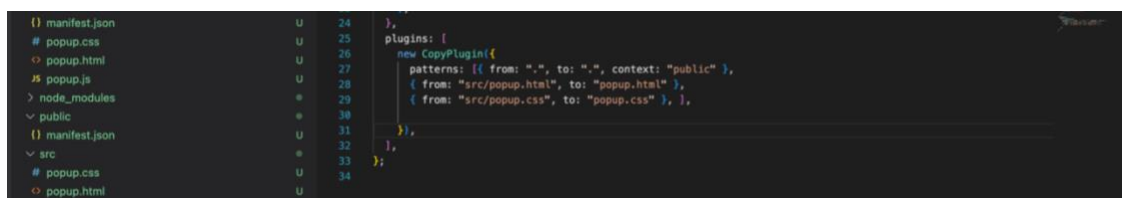
I began the development process by referencing [this Medium article](#), which provided a clear and practical foundation for building Chrome extensions using TypeScript. It helped me structure the project scaffolding, configure `tsconfig.json`, and understand how to bundle the extension with Webpack efficiently.



The next step was creating the extension's UI with `popup.html`, `popup.css`, and a corresponding TypeScript file.

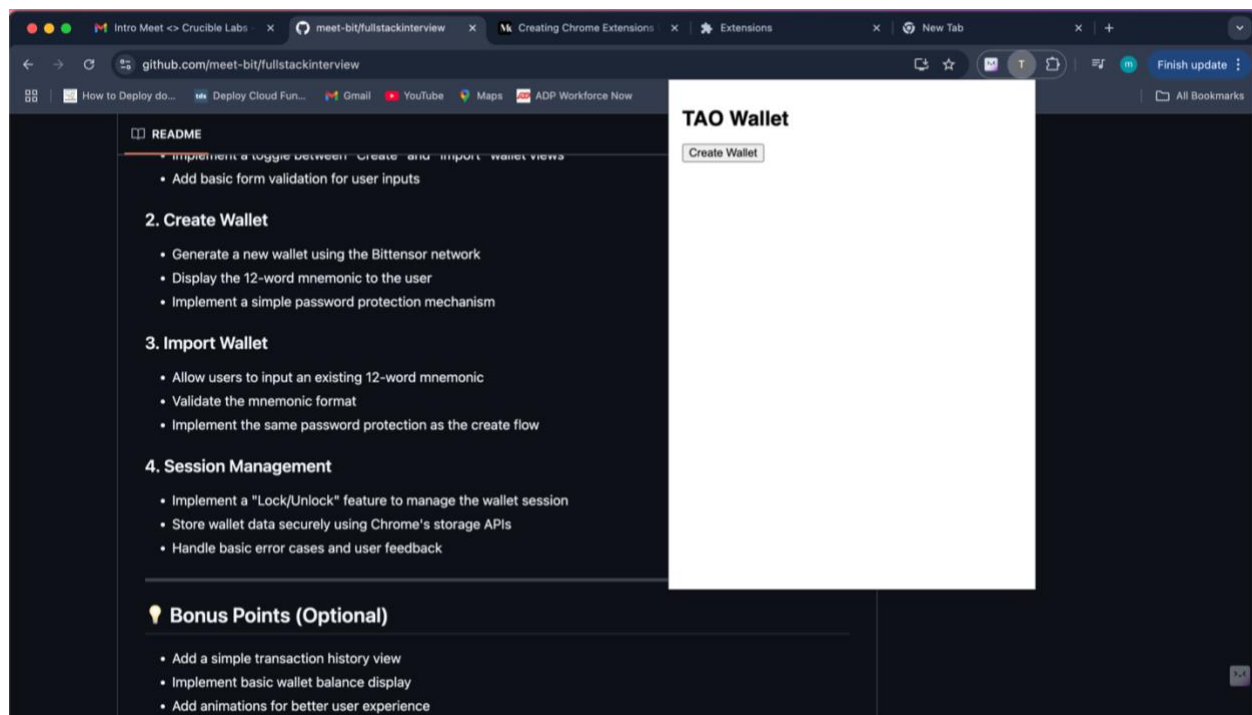
A key issue I encountered was that after running `npm run build`, the `popup.html` file wasn't being included in the `dist/` folder, and Chrome's manifest was not recognizing it.

Fix: I updated the `manifest.json` to explicitly include `web_accessible_resources` and ensured that Webpack was copying the `popup.html` file into the `dist/` directory using a plugin like `HtmlWebpackPlugin`. This allowed the Chrome extension to correctly load the popup interface.



Next, I added the basic logic to ensure the popup extension UI rendered correctly. This included verifying that:

- `popup.html` loads properly when the extension icon is clicked.
- Styles from `popup.css` are applied correctly.
- The TypeScript logic compiles to JavaScript and is correctly linked to the HTML file.



At this point, I went a bit off-track to strengthen my understanding of how crypto wallets—particularly Ethereum wallets—work, since this was my first time building one.

I explored:

- Wallet Creation and Importing:
 - Read through [Ethers.js wallet API](#) to understand how Ethereum wallets are created and imported using mnemonics.
 - Researched [BIP39](#), the standard for 12/24-word mnemonic seed phrases used to generate wallets across blockchain ecosystems like Ethereum and Bittensor.
- Security & Encryption:

- Studied password encryption concepts on [Okta](#) and learned how symmetric keys and hashing algorithms like PBKDF2 help secure passwords.
 - Researched browser-native encryption methods using the [Web Crypto API](#) and `SubtleCrypto`.
- Storage Concepts:
 - Looked into wallet storage strategies including [cold storage](#), which emphasizes storing sensitive wallet data offline for better security.

Key Learnings:

- A 12-word BIP39 mnemonic can deterministically generate a private key.
- This private key is then used to derive the wallet address.
- Password-based encryption using PBKDF2 helps derive a secure key to encrypt/decrypt sensitive wallet data.
- The Web Crypto API can handle key derivation and AES encryption in-browser without needing external libraries.

Here I started exploring the Ethers.js library, with the hope that a similar approach could be replicated by finding an equivalent Bittensor node module.

Ether Workflow:

- The user inputs a wallet name and a password, which will be used to protect the wallet.
- This password is processed via PBKDF2 to derive a secure encryption key.
- The Web Crypto API then encrypts the wallet's private key using the derived encryption key.
- When the user wants to access the wallet later, the Web Crypto API is used again to decrypt the private key using the password-derived encryption key.

Breakdown:

- ETHER SDK → Generates 12-word Mnemonic → Derives wallet address and private key
- User enters password → Processed with PBKDF2 and salt → Produces symmetric encryption key
- Web Crypto API → Uses derived key → Encrypts private key

Stored Values in `chrome.storage`:

- `walletName`: name
- `walletAddress`: public address
- `encryptedPrivateKey`: ciphertext
- `salt`: used in PBKDF2

- `iv`: initialization vector for AES encryption

Okayy I am stupid, I literally implemented the entire thing with an Ethereum wallet and also its encryption which returned the 12 bit value –

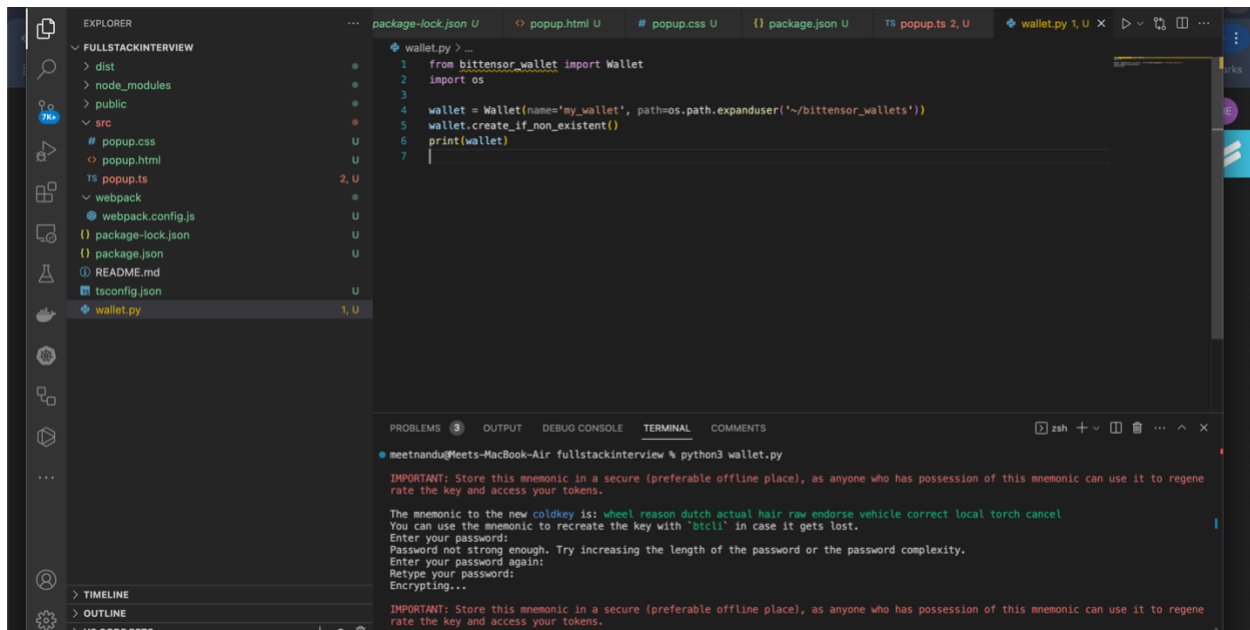
```
/**
 * Encrypts a private key using a password-derived key.
 * Returns ciphertext along with the IV and salt needed for decryption.
 * Utilizes the SubtleCrypto API: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/encrypt
 */
async function encryptPrivateKey(privateKey: string, password: string) {
  const salt = window.crypto.getRandomValues(new Uint8Array(16));
  const iv = window.crypto.getRandomValues(new Uint8Array(12));
  const key = await deriveKey(password, salt);

  const enc = new TextEncoder();
  const encrypted = await window.crypto.subtle.encrypt(
    { name: "AES-GCM", iv: iv },
    key,
    enc.encode(privateKey)
  );

  return {
    ciphertext: Array.from(new Uint8Array(encrypted)),
    salt: Array.from(salt),
    iv: Array.from(iv),
  };
}
```

bittensor does not have a npm library, but handles the encryption part and 12 word generation part !!!! Damn!!

yet it does have a python library to create a local wallet, but again built on top of rust, I did not have rust on my machine so installed rust, created simple python service to try out the wallet making first:



I figured I'd expose the Bittensor wallet creation process as a Flask service while running the extension, with the wallet stored locally for now.

The next step would be figuring out a strategy for wallet storage:

- Do we keep it local, like I've done for now?
 - Or do we eventually move to a database?
- Still open to discussion, really.

As I dug deeper into the Bittensor SDK, I came across the [installation guide](#). It was pretty straightforward, and the Python service I created was based on this guide, so I could test wallet creation.

Then got stuck at:

I was running the Flask service locally at `http://localhost:5000`, and naturally, the extension was trying to fetch data from it. But CORS was blocking the requests!

The error message said:

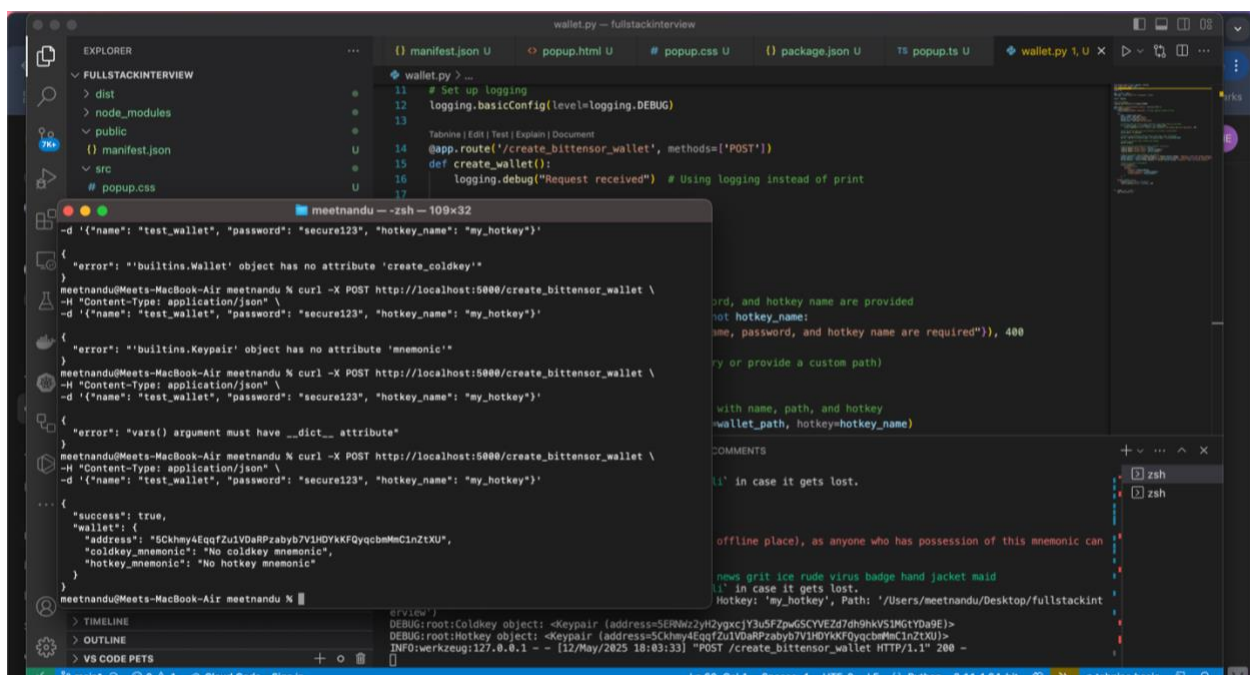
```
Access to fetch at 'http://localhost:5000/create_bittensor_wallet' from
origin 'chrome-extension://fpojlobgadbknkojmcddccpfofhlnkjg' has been blocked
by CORS policy: Response to preflight request doesn't pass access control
check: No 'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Of course, I wasn't expecting this, but after some Googling and AI help, I found a solution. I had to configure my Flask service to allow requests from the Chrome extension by setting the

Access-Control-Allow-Origin header in manifest as well as the flask service. It wasn't as hard as it sounded, but it definitely added a little extra time to the process.

To test the requests, I used curl (I prefer curl over postman) to interact with the Flask service. I was working with both the cold key private and hotkey public.

Getting the actual key out wasn't as straightforward as I thought. I could see the address being exposed on the `stdout`, but the key itself was hidden. That made it a bit tricky to extract, which led me down a bit of a rabbit hole trying to figure out the right way to pull it.



```
EXPLORER
  FULLSTACKINTERVIEW
    dist
    node_modules
    public
    manifest.json
    src
    popup.css

wallet.py -- fullstackinterview
11 # Set up logging
12 logging.basicConfig(level=logging.DEBUG)
13
14 @app.route('/create_bittensor_wallet', methods=['POST'])
15 def create_wallet():
16     logging.debug("Request received") # Using logging instead of print
17

meetnandu -- zsh -- 109x32
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'
{"error": "'builtins.Wallet' object has no attribute 'create_coldkey'"}
meetnandu@Meets-MacBook-Air meetnandu % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H "Content-Type: application/json" \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'
{"error": "'builtins.Keypair' object has no attribute 'mnemonic'"}
meetnandu@Meets-MacBook-Air meetnandu % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H "Content-Type: application/json" \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'
{"error": "(vars()) argument must have __dict__ attribute"}
meetnandu@Meets-MacBook-Air meetnandu % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H "Content-Type: application/json" \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'
{"success": true,
 "wallet": {
   "address": "5CkhyyAEqgFzUVDaRPzabyb7V1H0YKAFQyqcbmMmCinZtXU",
   "coldkey_mnemonic": "No goldkey mnemonic",
   "hotkey_mnemonic": "No hotkey mnemonic"
 }
}

meetnandu@Meets-MacBook-Air meetnandu %

DEBUG:root:Coldkey object: <Keypair (address=5ERWm2yhG2ygcjY3u5FZpw6SCYVEZd7dh9hKVS1MGtYDa9E)>
DEBUG:root:Hotkey object: <Keypair (address=5CkhyyAEqgFzUVDaRPzabyb7V1H0YKAFQyqcbmMmCinZtXU)>
INFO:werkzeug:127.0.0.1 - - [12/May/2025 10:03:33] "POST /create_bittensor_wallet HTTP/1.1" 200 -
```

Learnings: Cold key : private equivalent in cloud, Hotkey public equivalent

Eventually, after a bit of trial and error, I realized that the file size was relatively small. So, my workaround was to JSON dump the entire hotkey object and then simply extract the key from there. It wasn't the cleanest solution, but it worked in the short term!

The screenshot shows a VS Code editor with a file explorer on the left, a code editor in the center, and a terminal window at the bottom. The code editor displays a Python script named `wallet.py` with the following content:

```
def create_wallet():
    data = request.get_json()
    wallet_name = data.get("name")
    password = data.get("password")
    hotkey_name = data.get("hotkey_name")

    if not wallet_name or not password or not hotkey_name:
        return jsonify({"error": "Wallet name, password, and hotkey name are required"}), 400

    # Use present working directory for wallet path
    wallet_path = os.path.join(os.getcwd(), "wallets")

    # Create wallet
    wallet_path, hotkey=hotkey_name

    wallet_name, "hotkeys", hotkey_name
    path, wallet_name, "coldkeypub.txt")
    (se)

COMMENTS
if the password or the password complexity.
offline place), as anyone who has possession of this mnemonic can
deputy satisfy clay motion enlist dinosaur accident hundred asthma
in case it gets lost.
```

The terminal window shows the execution of the script using `curl` and the response from the server:

```
meetnandu@Meets-MacBook-Air: ~ % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H 'Content-Type: application/json' \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'

{"success": true,
 "wallet": {
  "address": "5QFonsE2ndwbY.VpD3H1Q2SLdZnmYjFZN24bMjc4P8afKt",
  "coldkey_mnemonic": "Mnemonic not found",
  "hotkey_mnemonic": "Mnemonic not found"
 }}

meetnandu@Meets-MacBook-Air: ~ % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H 'Content-Type: application/json' \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'

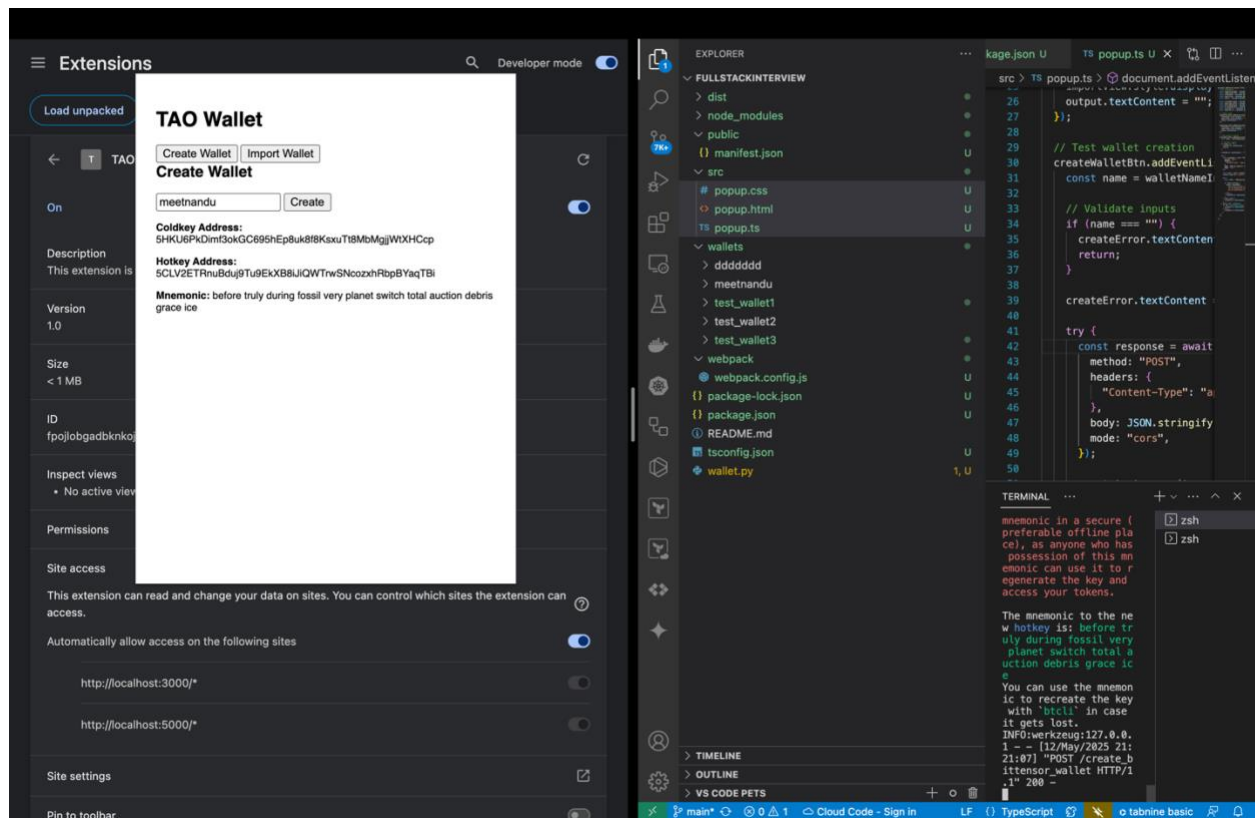
{"error": "Wallet.create() got an unexpected keyword argument 'password'" }

meetnandu@Meets-MacBook-Air: ~ % curl -X POST http://localhost:5000/create_bittensor_wallet \
-H 'Content-Type: application/json' \
-d '{"name": "test_wallet", "password": "secure123", "hotkey_name": "my_hotkey"}'

{"success": true,
 "wallet": {
  "coldkey_address": "5FqBYX2U1s8VyABTNvMBFcG4pQApXRTb5Jm86JmNPX7pMx",
  "hotkey_address": "5CAkctwFZpki3inMhYk42FwTDogLakZ2Iwun3TYaTGTt9C",
  "hotkey_mnemonic": "acoustic front shed deputy satisfy clay motion enlist dinosaur accident hundred asthma"
 }}
```

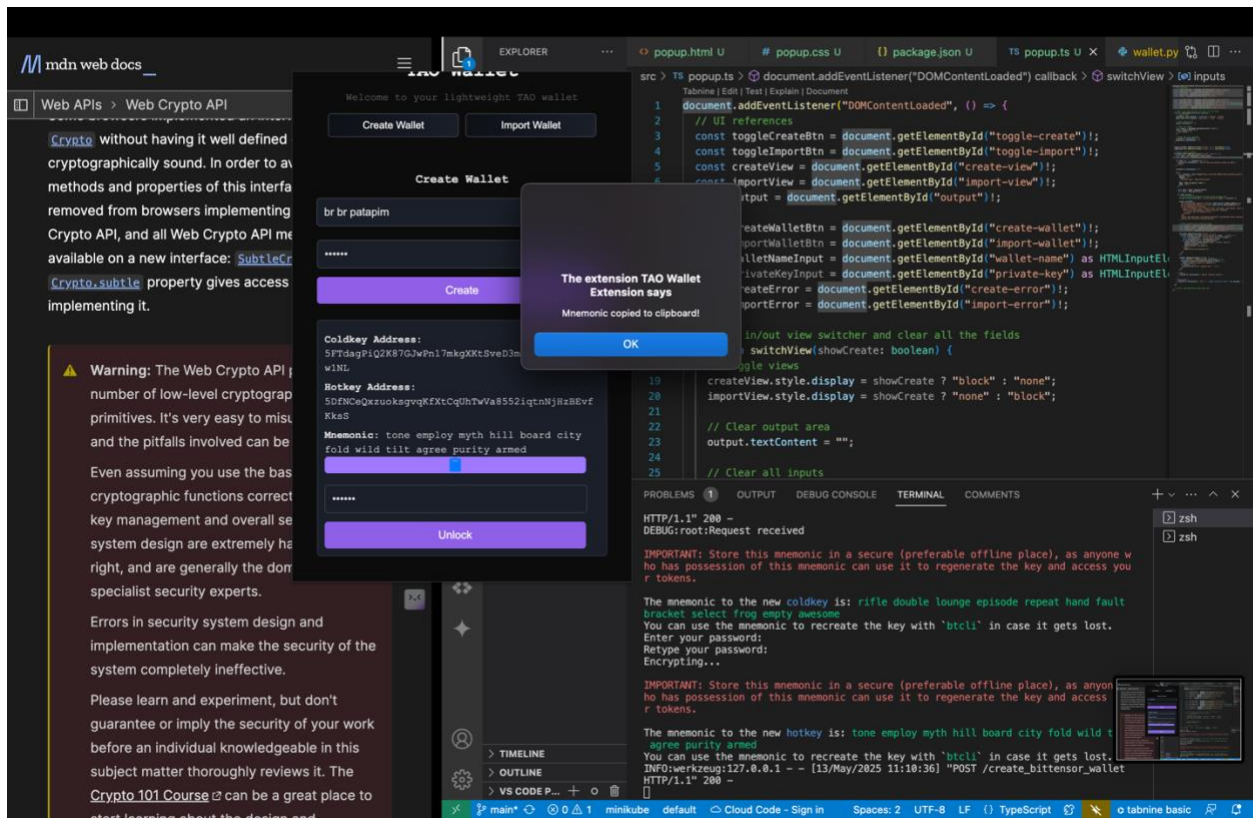
Eventually got the key!!! ☺

Finally got it working:



I had a little difficulty when trying to connect the password creation prompt on server side used during the wallet creation to the one the user uses to create their account. The current flow is a bit of a workaround for now:

- I added a password field to the Create Wallet form.
- The user's password is stored locally in Chrome Storage.
- When the wallet is being created, the wallet's password and retyped password are used to encrypt and create the wallet. (However, in future this password needs to be extracted later and mapped to the user's password correctly but subject to discussion).
- On success, the mnemonic is displayed but blurred.
- To unlock it, the user has to enter the correct password. The entered password is then verified against the stored password from Chrome Storage. If they match, the mnemonic is revealed.
- I also disabled the copy button until the mnemonic is revealed, only allowing it once the password is verified.



Following the wallet creation flow, I moved on to implementing wallet import.

The idea was straightforward:

- Let the user input a 12-word mnemonic.
- Validate the format to make sure it's exactly 12 space-separated words.
- Prompt the user to set a password for securing the imported wallet.
- Once confirmed:
 - Store the mnemonic and the password in `chrome.storage.local`.
 - Show a simple success message and sample balance and transaction history to confirm the wallet has been imported.

Right now, the password is only stored in session storage, which works for basic usage during a single session. But thinking ahead — in a more robust setup, we could:

- Use a database to validate if a mnemonic already exists.
- Add better prompts or options for remembering the user and handling persistent sessions more securely.

Still lots of room for refining the UX and strengthening the security model.

TAO Wallet

Welcome to your lightweight TAO wallet

Create Wallet

Import Wallet

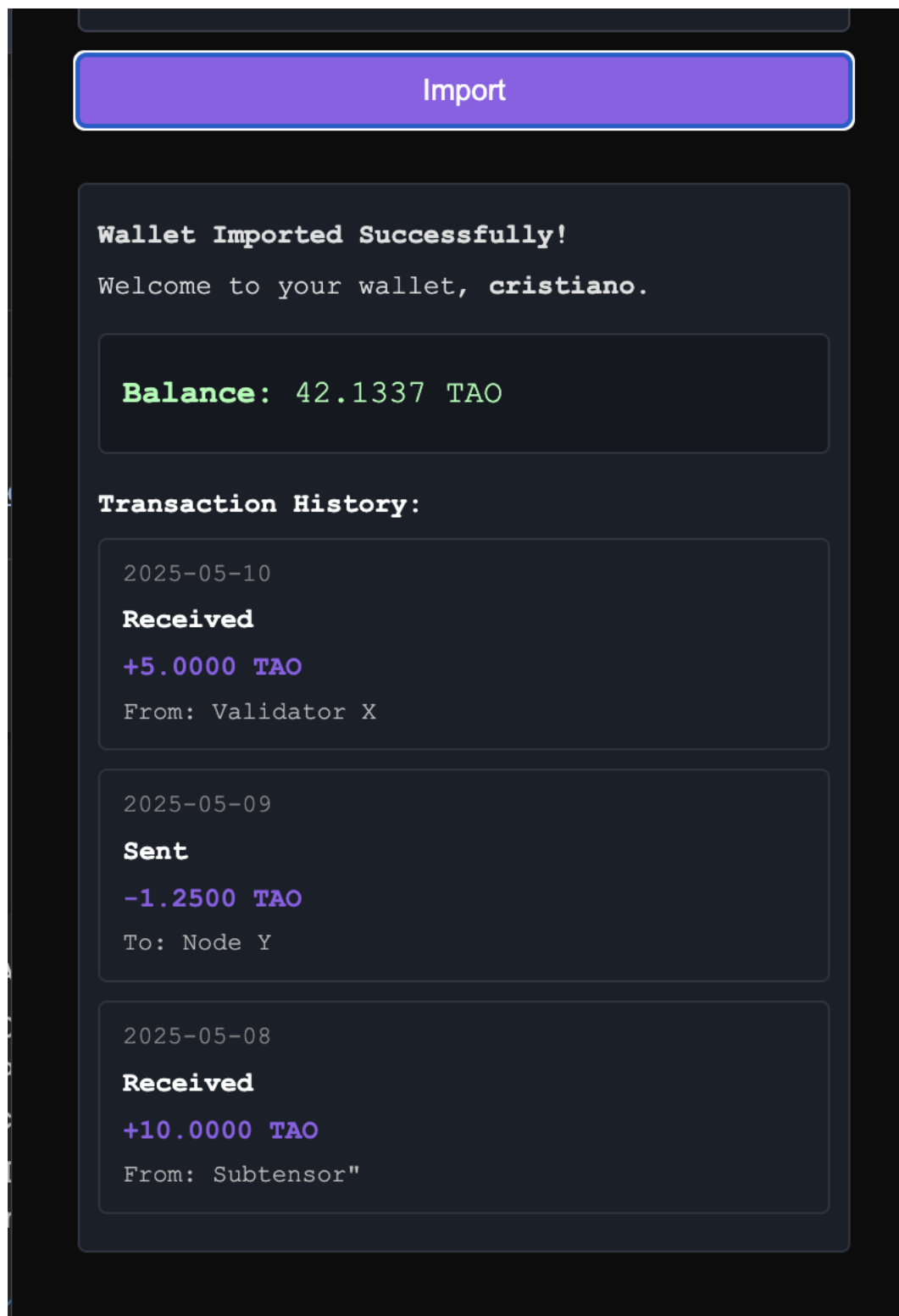
Import Wallet

Enter wallet name

Enter your 12-word mnemonic

Set password to protect mnemonic

Import

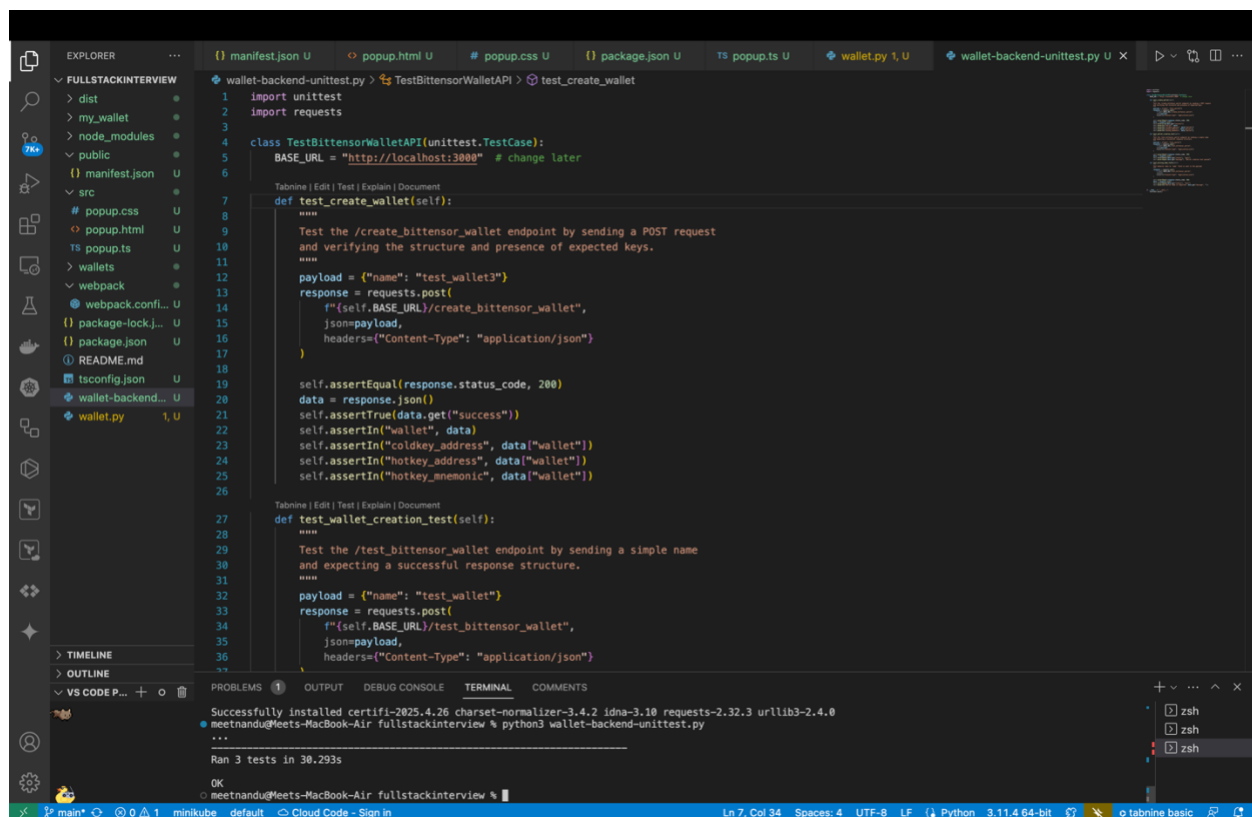


Next up was writing a **unit test suite** for the Flask service I had built earlier to handle wallet

creation. The goal was to simulate and validate the API requests — especially for the `create_bittensor_wallet` endpoint — and make sure everything worked as expected.

Alongside this, I also started drafting a basic **Dockerfile** to containerize the service. The current setup still runs with Flask's built-in development server, but it's future-ready — the plan is to eventually switch to a proper **production-ready server like Gunicorn** (with WSGI), once things are solidified.

So while it's not fully containerized and deployed yet, the groundwork is there for when that step makes sense.



The screenshot shows a VS Code editor with a file explorer on the left and a code editor in the center. The file explorer shows a project structure with folders like `dist`, `my_wallet`, `node_modules`, `public`, `src`, `wallets`, `webpack`, `webpack.config.js`, `package-lock.json`, `package.json`, `README.md`, `tsconfig.json`, `wallet-backend-unittest.py`, and `wallet.py`. The code editor shows the following Python code:

```
1 import unittest
2 import requests
3
4 class TestBittensorWalletAPI(unittest.TestCase):
5     BASE_URL = "http://localhost:3000" # change later
6
7     def test_create_wallet(self):
8         """
9         Test the /create_bittensor_wallet endpoint by sending a POST request
10         and verifying the structure and presence of expected keys.
11         """
12         payload = {"name": "test_wallet3"}
13         response = requests.post(
14             f"{self.BASE_URL}/create_bittensor_wallet",
15             json=payload,
16             headers={"Content-Type": "application/json"}
17         )
18
19         self.assertEqual(response.status_code, 200)
20         data = response.json()
21         self.assertTrue(data.get("success"))
22         self.assertIn("wallet", data)
23         self.assertIn("coldkey_address", data["wallet"])
24         self.assertIn("hotkey_address", data["wallet"])
25         self.assertIn("hotkey_mnemonic", data["wallet"])
26
27     def test_wallet_creation_test(self):
28         """
29         Test the /test_bittensor_wallet endpoint by sending a simple name
30         and expecting a successful response structure.
31         """
32         payload = {"name": "test_wallet"}
33         response = requests.post(
34             f"{self.BASE_URL}/test_bittensor_wallet",
35             json=payload,
36             headers={"Content-Type": "application/json"}
37         )
38
39         self.assertEqual(response.status_code, 200)
40         data = response.json()
41         self.assertTrue(data.get("success"))
42         self.assertIn("wallet", data)
43         self.assertIn("coldkey_address", data["wallet"])
44         self.assertIn("hotkey_address", data["wallet"])
45         self.assertIn("hotkey_mnemonic", data["wallet"])
46
47 if __name__ == '__main__':
48     unittest.main()
```

The terminal at the bottom shows the command `python3 wallet-backend-unittest.py` and the output `Ran 3 tests in 30.293s`.

Future Considerations?

Containerize if centralised flask server is used at the backend. Question -> is it really a decentralised system if we are serving a centralised server for people?

Database for storage

User password -> actual encrypted password mapping used during wallet creation.

Use API for actual checking of valid mnemonic during import -> need database or some sort of identification.

Use endpoint for real balance retrieval and transaction history once wallet is imported

More UI Improvements

