BUILDING AN AUTHENTICATED LLM CHAT SERVICE

Motivation

- Many AI applications are not just model calls; they are complete web services with users, sessions, storage, secrets, deployment, and access control.
- The assignment combines two important production patterns:
  - serving a model through a private model server
  - requiring users to authenticate before they can use the model
- The model server is the inference component, but the REST API/web application is the product boundary that users should interact with.
- The main design goal is controlled access: users should reach the chat application, but they should not directly reach the raw `vLLM` server.

Request Flow for Login

- A new user visits the registration page.
- The user enters an email address and password.
- The app validates the input.
- The app normalizes the email address, usually by trimming whitespace and lowercasing the domain.
- The app hashes the password with a password hashing library.
- The app stores the user record with the password hash, not the raw password.
- A returning user visits the login page.
- The user enters the email address and password.
- The app verifies the submitted password against the stored hash.
- If verification succeeds, the app creates a session and redirects the user to the chat page.
- If verification fails, the app rejects the request without revealing whether the email address or password was the specific problem.

Request Flow for Chat

- The browser requests the chat page.
- The app checks the session cookie.
- If there is no valid session, the app redirects to login.
- If there is a valid session, the app loads that user's chat history.
- When the user submits a prompt, the app checks the session again.
- The app stores the user's message.
- The app sends the prompt to `vLLM` on the separate model server instance over the VPC private network.
- The app stores the model response.
- The app returns the updated conversation to the user.

Why Not Expose vLLM Directly?

- A raw model server usually has no concept of course users, accounts, sessions, or chat history.
- If exposed directly, anyone who can reach the endpoint can send prompts unless another layer blocks them.
- Direct exposure also makes it harder to apply rate limits, logging, moderation, billing controls, or per-user history.
- The assignment uses the REST API/web app as the control point because that is where authentication and application policy belong.
- The public API should be “chat for logged-in users,” not “unrestricted access to a model port.”

vLLM as a Model Server

- `vLLM` is a model-serving library.
- In this assignment, use its OpenAI-compatible HTTP server mode.
- OpenAI-compatible means the app can call `vLLM` using request shapes similar to OpenAI's chat completions API.
- This is useful because the application code can use a familiar client pattern:
  - set `base_url` to the private `vLLM` server endpoint
  - set an API key if the server requires one
  - call `chat.completions.create`
- The model required for the assignment is `HuggingFaceTB/SmolLM2-135M-Instruct`.

Calling vLLM from the REST API/Web App

- The REST API/web app should call `vLLM`, not the browser.
- The browser sends prompts to the REST API/web app.
- The REST API/web app sends model requests to `http://MODEL_PRIVATE_IP:8000/v1` or to the model server's private DNS name.
- The REST API/web app includes the `vLLM` API key from an environment variable.
- The browser should never receive the `vLLM` API key.

```
import os

from openai import OpenAI

client = OpenAI(
    base_url=os.environ["VLLM_BASE_URL"],
    api_key=os.environ["VLLM_API_KEY"],
)

response = client.chat.completions.create(
    model="HuggingFaceTB/SmolLM2-135M-Instruct",
    messages=[
        {"role": "user", "content": prompt},
    ],
)
```

Password-Based Login

- Password-based login means the user proves identity by submitting a password chosen during registration.
- The application must never store the raw password.
- The application stores a password hash produced by a password hashing library.
- During login, the application runs the library’s verify function to compare the submitted password to the stored hash.
- The app should create a session only after successful password verification.
- The app should reject invalid credentials with a generic error.

Account Creation with Passwords

- The registration form should collect at least:
  - email address
  - password
  - password confirmation, if you want to catch typing mistakes
- The app should normalize the email address before storing it.
- The `users.email` column should be unique.
- If the email address is already registered, show a normal account-creation error without exposing sensitive details.
- The app should hash the password before inserting the user record.
- Do not log raw passwords.

Password Hashing

- Password hashing is different from ordinary fast hashing.
- Fast hashes such as plain SHA-256 are not appropriate for password storage by themselves.
- Password hashes should be deliberately slow and salted.
- Good choices include:
  - `bcrypt`
  - `argon2`
  - a framework-provided password hashing utility
- Use the library's high-level hash and verify functions instead of inventing your own format.
- The stored value should contain enough metadata for the library to verify it later.

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

password_hash = pwd_context.hash(password)
is_valid = pwd_context.verify(candidate_password, password_hash)
```

Login Failure Behavior

- A failed login should not reveal whether the email address exists.
- Use a generic message such as “Invalid email or password.”
- Do not return different error messages for:
  - unknown email address
  - wrong password
  - inactive account, unless your app has a clear reason to expose that state
- Generic errors reduce user enumeration.
- Logging can record useful debugging information, but avoid logging raw passwords.

Why Sessions Exist

- HTTP requests are independent by default.
- The browser can request `/login`, then later request `/chat`, but the server does not automatically remember that those two requests came from a successfully logged-in user.
- Without a session, the app would have to ask for the password on every protected request.
- A session is the usual web pattern for remembering that a user has already authenticated.
- The goal is: prove identity once, then carry a temporary proof of that login across later requests.

Stateless HTTP

- “Stateless” means each request must contain the information the server needs to handle it.
- The server receives:
  - HTTP method
  - path
  - headers
  - cookies
  - request body
- The server does not automatically know what happened in previous requests.
- Application state must be stored somewhere explicit:
  - in a database
  - in server memory
  - in a signed token
  - in a cookie
- For this assignment, the safest mental model is: store session records on the server and store only a session identifier in the browser.

The Login Problem

- After a correct password login, the server knows who the user is for that one request.
- The next request to `/chat` is a new HTTP request.
- The server needs a way to connect that new request to the logged-in user.
- The browser needs to send some proof that it belongs to an existing login session.
- The server needs to verify that proof before showing protected data.
- That proof should not be the user’s password.

Cookies

- A cookie is a small value that the server asks the browser to store.
- The server sends a `Set-Cookie` header in an HTTP response.
- The browser stores the cookie.
- On later requests to the same site, the browser sends the cookie back in a `Cookie` header.
- Cookies are useful for sessions because they let the browser automatically attach a session identifier to later requests.

Cookie Example

- After login, the server might send:

  ```
  Set-Cookie: session_id=abc123; HttpOnly; SameSite=Lax; Path=/
  ```

- On a later request, the browser sends:

  ```
  Cookie: session_id=abc123
  ```

- The server reads `session_id`.
- The server looks up that session.
- If the session is valid, the server treats the request as authenticated.

Session Identifier

- A session identifier should be random and unguessable.
- It should not be the user ID.
- It should not be an email address.
- It should not be predictable from the current time.
- If an attacker can guess another user's session ID, the attacker can impersonate that user.
- A typical pattern is to generate a random value with a cryptographically secure random generator.

```
import secrets

session_token = secrets.token_urlsafe(32)
```

Server-Side Sessions

- In a server-side session design, the browser stores only a session token.
- The database stores the session record.
- A `sessions` table might include:
  - session token hash
  - user ID
  - creation time
  - expiration time
  - revoked or logout time
- On each protected request, the server hashes the token from the cookie and looks up the matching session.
- If the session exists and is still valid, the app knows which user is making the request.

Why Store a Hash of the Session Token?

- A session token acts like a temporary password.
- Anyone who has the token can use the session until it expires or is revoked.
- Storing a hash of the token reduces damage if the database is exposed.
- The browser receives the raw token in the cookie.
- The database stores only the hash.
- On each request, the server hashes the cookie token and compares it to the stored hash.

Session Lookup Flow

- Browser requests `/chat`.
- Browser includes `session_id` cookie.
- Server reads the cookie.
- Server hashes the session token.
- Server queries the `sessions` table for that hash.
- Server checks that the session is not expired or revoked.
- Server gets the associated `user_id`.
- Server loads only data belonging to that `user_id`.
- If any check fails, the server redirects to login or returns `401 Unauthorized`.

Protected Routes

- A protected route is a route that requires a valid session.
- In this assignment, protected routes include:
  - `GET /chat`
  - `POST /chat`
  - `POST /logout`
- The app should check the session at the start of each protected route.
- Protecting only the HTML page is not enough.
- If `POST /chat` is unprotected, someone can skip the page and call the chat endpoint directly.

Session Middleware or Helper

- Many frameworks let you centralize session checks.
- A helper function can:
  - read the cookie
  - validate the session
  - load the current user
  - raise an error or redirect if there is no valid session
- This avoids repeating session lookup code in every route.
- In FastAPI, this is often done with a dependency.
- The assignment uses FastAPI, so the dependency pattern is the main one to understand.

FastAPI Session Dependency Shape

- A FastAPI dependency can represent “current logged-in user.”
- Protected routes can require that dependency.

```python
# Sketch only: real code needs imports and your own User/ChatRequest types.
def get_current_user(request: Request) -> User:
    session_token = request.cookies.get("session_id")
    if not session_token:
        raise HTTPException(status_code=401)
    user = load_user_from_valid_session(session_token)
    if user is None:
        raise HTTPException(status_code=401)
    return user

@app.post("/chat")
def chat(req: ChatRequest, user: User = Depends(get_current_user)):
    ...
```

- The route does not trust a user ID from the browser.
- The route gets the user from the verified session.

Session Design

- A successful password check proves identity for the login request.
- A session keeps the user logged in across multiple requests.
- The server creates a session after successful password verification.
- The browser stores a session cookie.
- The session cookie identifies a server-side session or contains a signed session value.
- The app checks the session on every protected route.
- Logout deletes or invalidates the session.

Creating a Session

- After successful password verification:
  - generate a random session token
  - hash the token for storage
  - insert a session row with the user ID and expiration time
  - send the raw token to the browser in a cookie
- The browser should not receive the password hash.
- The browser should not receive the user ID as the authority for authorization decisions.
- The session token is the browser’s temporary proof of login.

Logout

- Logout should end the current session.
- Common logout steps:
  - read the session cookie
  - mark the matching session row as revoked
  - clear the cookie in the response
  - redirect the user to login or a public page
- Deleting only the browser cookie is not enough if the server-side session row remains valid and the token is copied somewhere else.
- Marking the session revoked gives the server a durable way to reject that token later.

Clearing a Cookie in FastAPI

- FastAPI response objects can tell the browser to delete a cookie.
- The route should also revoke the matching server-side session.
- Clearing the cookie removes the browser's normal way to send the session token on later requests.

```python
import hashlib

from fastapi import Request
from fastapi.responses import RedirectResponse

def hash_session_token(session_token: str) -> str:
    return hashlib.sha256(session_token.encode("utf-8")).hexdigest()

def revoke_session(session_token: str) -> None:
    session_hash = hash_session_token(session_token)
    db.execute(
        """
        UPDATE sessions
        SET revoked_at = CURRENT_TIMESTAMP
        WHERE session_hash = ?
        """,
        (session_hash,),
    )
    db.commit()

@app.post("/logout")
def logout(request: Request):
    session_token = request.cookies.get("session_token")
    if session_token:
        revoke_session(session_token)

    response = RedirectResponse("/login", status_code=303)
    response.delete_cookie("session_token", path="/")
    return response
```

Cookie Settings

- Session cookies should be protected with reasonable settings.
- `HttpOnly` helps prevent JavaScript from reading the cookie.
- `SameSite=Lax` is a common baseline that reduces cross-site request abuse for normal navigation.
- `Secure` should be used when the site runs over HTTPS.
- Cookies should have an expiration or be tied to server-side session expiration.
- Cookie settings do not replace server-side authorization checks.

Session Expiration

- Sessions should not last forever.
- Store an `expires_at` time in the session row.
- On each protected request, reject sessions past their expiration time.
- Shorter sessions reduce risk if a session token is stolen.
- Longer sessions are more convenient for users.
- For a class assignment, a simple fixed expiration window is enough.

Sessions vs. Users

- A user account and a session are not the same thing.
- A user account is long-lived:
  - email address
  - password hash
  - created time
- A session is temporary:
  - session token hash
  - user ID
  - expiration time
  - revoked time
- One user can have multiple sessions if they log in from multiple browsers or devices.
- Logging out of one session does not necessarily delete the user account.

Persistent Storage

- The application needs data that survives process restarts.
- SQLite is reasonable for this assignment because the REST API/web service is small and runs on one interface instance.
- The database file should live in a predictable location on the REST API/web instance.
- The app should not store active state only in Python dictionaries.
- If the process restarts, users, sessions, and chat history should still exist unless explicitly expired or deleted.

Suggested Database Tables

- A `users` table stores one row per email account.
- The `users` table should store a password hash, not a plaintext password.
- A `sessions` table stores active authenticated sessions if using server-side sessions.
- A `chat_messages` table stores user and assistant messages.
- Optional tables can store request logs, model metadata, or rate-limit counters.

```sql
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  email TEXT NOT NULL UNIQUE,
  password_hash TEXT NOT NULL,
  created_at TEXT NOT NULL
);

CREATE TABLE sessions (
  id INTEGER PRIMARY KEY,
  user_id INTEGER NOT NULL,
  session_hash TEXT NOT NULL UNIQUE,
  expires_at TEXT NOT NULL,
  created_at TEXT NOT NULL,
  revoked_at TEXT,
  FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE chat_messages (
  id INTEGER PRIMARY KEY,
  user_id INTEGER NOT NULL,
  role TEXT NOT NULL,
  content TEXT NOT NULL,
  created_at TEXT NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Per-User Chat History

- Every chat message should be associated with a user ID.
- When loading history, filter by the current authenticated user’s ID.
- Do not let the browser send a user ID and then trust it.
- The server should derive the user ID from the session.
- A common bug is accepting `/chat?user_id=2` and showing another user’s messages.
- The correct pattern is: session identifies user, database query filters by session’s user.

Authorization Checks

- Authentication asks: who is this user?
- Authorization asks: what is this user allowed to do?
- For this assignment, the main authorization rule is simple:
  - unauthenticated users cannot access chat
  - authenticated users can access only their own chat data
- The app should enforce this on all relevant routes, not only in the UI.
- Hiding a button is not authorization.
- A protected API route still needs a server-side session check.

Configuration and Secrets

- Configuration is data that changes between environments.
- Secrets are configuration values that must not be public.
- Examples of configuration:
  - app base URL
  - database path
  - `vLLM` base URL
- Examples of secrets:
  - session signing key
  - `vLLM` API key
- Secrets should come from environment variables or a local file excluded from version control.

Prompt Construction

- The model receives a list of chat messages.
- At minimum, send the current user prompt.
- For a better chat experience, send recent prior messages from that same user.
- Do not send another user’s messages.
- Consider limiting history length so requests stay small.
- Small models and CPU inference work better with short prompts.
- The app can include a system message if it wants to set behavior, but the assignment does not require complex prompt engineering.

Common Security Mistakes

- Opening the `vLLM` port to the public internet.
- Calling `vLLM` directly from browser JavaScript and exposing the API key.
- Storing plaintext passwords.
- Using a fast general-purpose hash instead of a password hashing library.
- Logging raw passwords.
- Returning different login errors for unknown email addresses and wrong passwords.
- Letting the browser choose the user ID for chat history.
- Committing `.env` files or AWS credentials.
- Protecting the chat page but forgetting to protect the chat API route.

Key Takeaways

- A model server is not the same thing as a secure application.
- Authentication belongs in the REST API/web application layer that controls access to chat.
- Required password login depends on storing password hashes, not plaintext passwords.
- Per-user data access must be enforced on the server, not trusted to the browser.
- `vLLM` should be private, and the public internet should reach only the authenticated chat app.
- The assignment is successful when the whole system works end to end: registration, password login, session creation, protected chat, model response, persistent history, and blocked direct access to the model server.