Detailed FastAPI Integration Workflow for "Wellness at Work" (WaW) Cloud-Synced Eye Tracker

Overview

This workflow establishes a highly detailed FastAPI backend for the WaW project, integrating with the PyQt6 frontend, PostgreSQL RDS, AWS S3, and SQLite local cache. It exposes endpoints for authentication, blink data management, and user profiles, aligning with the provided directory structure and database schema. The setup emphasizes modularity, testability, and scalability, preparing for subsequent phases such as OAuth and LangChain integration.

Step 1: Installation and Setup

Task	Command/Location	Method/Instructions
Install dependencies	`pip install fastapi uvicorn pydantic sqlalchemy asyncpg httpx python-dotenv boto3`	Execute in project root; verify installation with `pip list`.
Update requirements.txt	`C:\Users\varun\Downloads\RAGBot\requirements.txt`	Open file, append listed packages, save, and commit to version control.
Create FastAPI entrypoint	`C:\Users\varun\Downloads\RAGBot\app\main.py`	Initialize with `from fastapi import FastAPI; app = FastAPI()`; configure later.
Create API routes dir	`C:\Users\varun\Downloads\RAGBot\app\api`	Create directory using `mkdir`

		`app\api`; ensure write permissions.

Step 2: API Router Structure

Router File	Responsibilities	Location	Methods/Implementation
`auth_router.py`	User authentication	`app\api\auth_router.py`	Define `router = APIRouter()`; include `/login`, `/logout` endpoints.
`blink_router.py`	Blink data tracking/syncing	`app\api\blink_router.py`	Define `router = APIRouter()`; include `/track`, `/sync` endpoints.
`user_router.py`	User profile management	`app\api\user_router.py`	Define `router = APIRouter()`; include `/profile` endpoint.
`health_router.py`	System health checks	`app\api\health_router.py`	Define `router = APIRouter()`; include `/health` endpoint.

Step 3: API Endpoints and Connections

Endpoint	Method	Description	Router File	Connected Module	DB Table	Implementation Details
`/auth/login`	POST	Authenticate user	`auth_router.py`	`local_cache\sqlite_models.py`	`AUTH_CACHE`	Use Pydantic for request model; return JWT token.
`/auth/logout`	POST	Invalidate session	`auth_router.py`	`local_cache\sqlite_models.py`	`AUTH_CACHE`	Invalidate token in cache;

						return success status.
`/blink/track`	POST	Record blink data	`blink_router.py`	`aws\s3_uploader.py`, `db\models.py`	`MESSAGES`	Accept JSON payload; save to S3 and PostgreSQL.
`/blink/sync`	POST	Sync offline blink data	`blink_router.py`	`local_cache\sync_local_cache.py`	`MESSAGES`	Process queued data; update `MESSAGES` table.
`/user/profile`	GET	Fetch user profile	`user_router.py`	`db\models.py`	`USERS`	Query `USERS` table; return Pydantic response.
`/health`	GET	System health check	`health_router.py`	`utils\logger.py`	N/A	Return server status and uptime in JSON.

Step 4: Database and Storage Integration

Component	File/Location	Connection Details	Implementation Details
PostgreSQL Models	`app\rag_engine\db\models.py`	Map `USERS`, `MESSAGES` to RDS schema	Define SQLAlchemy models with `declarative_base()`.

PostgreSQL Session	`app\rag_engine\db\session.py`	`asyncpg` for async connection	Implement `async with asyncpg.create_pool()` context.
SQLite Local Cache	`app\rag_engine\local_cache\sqlite_models.py`	`AUTH_CACHE` for tokens	Use `sqlite3` with `sqlalchemy.create_engine()`.
S3 Storage	`app\rag_engine\aws\s3_uploader.py`	Buckets: `waw-blink-data`, `waw-logs`	Use `boto3.client('s3')` with `upload_fileobj()`.

Step 5: Dependency Injection

Component	Dependency Loader	File/Location	Implementation Details
PostgreSQL Session	`get_db`	`app\rag_engine\db\session.py`	Return `Depends(async_session)` with yield.
SQLite Session	`get_local_db`	`app\rag_engine\local_cache\sqlite_session.py`	Return `Depends(sqlite_session)` with yield.
S3 Client	`get_s3_client`	`app\rag_engine\aws\s3_config.py`	Return `boto3.client('s3')` with env vars.
Logger	`get_logger`	`app\utils\logger.py`	Return `logging.getLogger()` with S3 handler.

Step 6: Middleware and Utilities

Task	Implementation	File/Location	Implementation Details

Logging	Log requests/errors	`app\utils\logger.py`	Configure `logging` with `FileHandler` and S3 upload.
CORS	Allow PyQt6/web origins	`app\main.py`	Use `CORSMiddleware` with `allow_origins=["*"]`.
Exception Handling	Global handler, log to S3	`app\main.py`	Define `@app.exception_handler(Exception)` with log.

Step 7: Integration with PyQt6 Client

Task	Description	File/Location	Implementation Details
API Client	Async HTTP requests	`app\ui\api_client.py`	Use `aiohttp.ClientSession()` with async methods.
Login	Call `/auth/login`	`app\ui\auth_handler.py`	Send POST with credentials; store JWT.
Blink Tracking	Call `/blink/track`	`app\ui\blink_handler.py`	Send POST with blink data; handle response.
Sync Offline Data	Call `/blink/sync`	`app\ui\sync_handler.py`	Send POST with queued data; update UI status.

Step 8: Environment and Execution

Task	Description	File/Location	Implementation Details
Set env vars	`DATABASE_URL`, `AWS_ACCESS_KEY`, etc.	`C:\Users\varun\Downloads\RAGBot\.env`	Define vars in `.env`; load with `load_dotenv()`.

Run server	`uvicorn app.main:app --reload`	`app\main.py`	Execute in terminal; monitor logs.
Test endpoints	Swagger UI: `http://localhost:8000/docs`	N/A	Access UI, test endpoints with sample requests.

Step 9: Preparation for Future Phases

Phase	Task	File/Location	Implementation Details
Phase 3: API-to-DB	Hook endpoints to SQLAlchemy models	`app\rag_engine\db\models.py`	Map endpoints to models with `session.add()`.
Phase 4: OAuth	Add `/auth/google/callback`	`app\api\auth_router.py`	Implement OAuth2 with `authlib` and Google client.
Phase 5: PyQt UI	Connect PyQt6 to endpoints	`app\ui\`	Bind QActions to API calls in PyQt6 slots.
Phase 6: LangChain	Add blink data analysis	`app\rag_engine\Query\`	Integrate LangChain with vectorstore for analysis.
Phase 9: Docker	Dockerfile for FastAPI	`C:\Users\varun\Downloads\RAGBot\`	Create `Dockerfile` with `FROM python:3.11`; build.

Step 10: File Structure and Methods

Directory Structure

- `app/`
  - `api/`
    - `auth_router.py`: Define `router`, `/login`, `/logout` with Pydantic models.
    - `blink_router.py`: Define `router`, `/track`, `/sync` with async handlers.
    - `user_router.py`: Define `router`, `/profile` with GET logic.
    - `health_router.py`: Define `router`, `/health` with status check.
  - `services/`
    - `auth_service.py`: Implement `authenticate_user()`, `invalidate_token()`.
    - `blink_service.py`: Implement `track_blink()`, `sync_offline_data()`.
    - `user_service.py`: Implement `get_user_profile()`.
  - `utils/`
    - `logger.py`: Define `get_logger()`, configure handlers.
    - `config.py`: Load `.env` with `python-dotenv`.
  - `ui/`
    - `api_client.py`: Define `async get()`, `async post()` methods.
    - `auth_handler.py`: Define `login()`, `logout()` with API calls.
    - `blink_handler.py`: Define `track_blink()` with data submission.
    - `sync_handler.py`: Define `sync_data()` with queue processing.
  - `rag_engine/`
    - `db/`: `models.py` (SQLAlchemy models), `session.py` (async session).
    - `aws/`: `s3_uploader.py` (upload methods), `s3_config.py` (client setup).
    - `local_cache/`: `sqlite_models.py` (cache models), `sqlite_session.py` (session).
  - `tests/`
    - `test_auth_router.py`: Test `/login`, `/logout` with mocks.
    - `test_blink_service.py`: Test `track_blink()`, `sync_offline_data()`.
  - `main.py`: Initialize `FastAPI`, include routers, add middleware.

Method Definitions

- **`auth_router.py`**
  - `login(user: UserLogin, db=Depends(get_db))`: Validate credentials, return JWT.

    - `logout(token: str, db=Depends(get_local_db))`: Invalidate token.
- **`blink_router.py`**
    - `track_blink(blink_data: BlinkData, db=Depends(get_db), s3=Depends(get_s3_client))`: Save blink data.
    - `sync_blink(db=Depends(get_db), local_db=Depends(get_local_db))`: Sync offline data.
- **`user_router.py`**
    - `get_profile(user_id: str, db=Depends(get_db))`: Fetch user data.
- **`health_router.py`**
    - `health_check(logger=Depends(get_logger))`: Return server status.
- **`auth_service.py`**
    - `authenticate_user(username: str, password: str, db)`: Verify credentials.
    - `invalidate_token(token: str, db)`: Update cache.
- **`blink_service.py`**
    - `track_blink(blink_data: dict, db, s3)`: Process and store blink data.
    - `sync_offline_data(db, local_db)`: Merge cached data.
- **`user_service.py`**
    - `get_user_profile(user_id: str, db)`: Query and return profile.
- **`api_client.py`**
    - `async get(url: str)`: Fetch data asynchronously.
    - `async post(url: str, data: dict)`: Send data asynchronously.
- **`logger.py`**
    - `get_logger()`: Return configured logger instance.

Step 11: Vision and Scalability

- **Modularity**: Each file serves a single purpose, enabling easy updates and testing.
- **Testability**: Isolated methods allow mocking dependencies (e.g., `get_db`).
- **Scalability**: Structure supports adding OAuth, LangChain, or Docker with minimal refactoring.
- **Performance**: Async I/O and dependency injection optimize resource use.
- **Maintainability**: Clear file roles reduce debugging time.

Step 12: Summary of Deliverables

Deliverable	Description	Location	Validation Method

FastAPI Backend	Endpoints for auth, blink, user	`app\main.py`, `app\api\`	Test with Swagger UI.
Database Integration	PostgreSQL/SQLite connections	`app\rag_engine\db\`, `local_cache\`	Verify with `db.query()` and cache checks.
S3 Storage	Blink data/logs in S3	`app\rag_engine\aws\`	Confirm with AWS S3 console.
PyQt6 Integration	API client for frontend	`app\ui\`	Test with PyQt6 UI actions.
Documentation	Update README with setup instructions	`C:\Users\varun\Downloads\RAGBot\README.md`	Review for completeness.