# 📁 Folder & File Structure

railway-pg-test/

📄 .dockerignore

📄 .env

📁 .github/

📁 workflows/

📄 cron-sync.yml

📄 .gitignore

📁 .vscode/

📄 settings.json

📄 Dockerfile

📄 README.md

📁 alembic/

📄 README

🐍 env.py

📄 script.py.mako

📁 versions/

🐍 fbdfce5b4297\_baseline.py

📄 alembic.ini

📄 docker-compose.yml

📄 final\_project\_map.docx

📄 requirements.txt

📄 reset\_and\_init\_db.sh

🐍 run.py

🐍 run\_seed.py

📁 schema.sql/

📁 src/

🐍 \_\_init\_\_.py

🐍 app.py

📁 db/

🐍 \_\_init\_\_.py

📁 dao/

🐍 activity\_dao.py

🐍 split\_dao.py

🐍 token\_dao.py

🐍 db\_session.py

📁 models/

🐍 activities.py

🐍 base.py

🐍 splits.py

🐍 tokens.py

🐍 env\_loader.py

📁 routes/

🐍 \_\_init\_\_.py

🐍 admin\_routes.py

🐍 auth.py

🐍 enrich.py

🐍 monitor\_routes.py

🐍 oauth.py

🐍 sync\_routes.py

🐍 web\_routes.py

📁 scripts/

🐍 backfill\_pilot\_test.py

🐍 backfill\_sync.py

🐍 dev\_only\_init\_db.py

🐍 dev\_seed\_data.py

🐍 enrich\_runner.py

🐍 enrichment\_pilot\_test.py

🐍 generate\_gpt\_handoff\_summary.py

🐍 helper\_extract\_token.py

🐍 manual\_single\_enrichment.py

🐍 manual\_token\_exchange.py

🐍 map\_and\_extract\_Railway.py

🐍 onboard\_and\_sync.py

🐍 sync\_and\_enrich.py

📄 sync\_athlete.sh

🐍 test\_single\_enrichment.py

📁 services/

🐍 \_\_init\_\_.py

🐍 activity\_sync.py

🐍 auth.py

🐍 enrichment\_sync.py

🐍 split\_extraction.py

🐍 strava.py

🐍 token\_refresh.py

📁 utils/

🐍 enrichment\_debug\_wrapper.py

🐍 gpt\_ops.py

🐍 jwt\_utils.py

🐍 logger.py

📄 test.db

🐍 test\_enrichment\_sync.py

📁 tests/

🐍 conftest.py

🐍 test\_activity\_sync\_flow.py

🐍 test\_auth.py

📄 test\_enrichment\_debug\_wrapper

🐍 test\_enrichment\_no\_hr\_zones.py

🐍 test\_enrichment\_with\_splits.py

🐍 test\_enrichmet\_retry\_logic.py

🐍 test\_health.py

🐍 test\_hr\_zone\_api.py

🐍 test\_oauth\_flow.py

📄 test\_onboard\_and\_sync

🐍 test\_split\_dao.py

🐍 test\_split\_extraction.py

🐍 test\_split\_extraction\_dao\_flow.py

🐍 test\_split\_pipeline\_full.py

🐍 test\_split\_upsert\_idempotency.py

🐍 test\_sync.py

🐍 test\_sync\_token\_refresh.py

🐍 test\_token\_refresh.py

🐍 utils.py

🐍 wsgi.py

# 🧠 Code & Content

### 📄 README.md

# 🏃 Smart Marathon Coach API  
  
This is a Flask-based API for syncing Strava activity data, enriching runs, and generating training insights. This repo is part of a multi-phase project — currently in \*\*Milestone 1: Setup & Plumbing\*\*.  
  
---  
  
## 🚀 Getting Started  
  
### 1. Clone the repo  
  
```bash  
git clone https://github.com/your-username/railway-pg-test.git  
cd railway-pg-test  
  
  
### 2. Set up your virtual environment  
  
```bash  
python -m venv venv  
venv\Scripts\activate # On Windows  
# or  
source venv/bin/activate # On Mac/Linux  
```  
  
### 3. Install dependencies  
  
```bash  
pip install -r requirements.txt  
```  
  
---  
  
## 🛠️ Environment Configuration  
  
Copy the example `.env` file and fill in your secrets:  
  
```bash  
cp .env.example .env  
```  
  
Or manually create `.env` with values like:  
  
```env  
DATABASE\_URL=postgresql://smartcoach:devpass@postgres:5432/smartcoach  
  
# NOTE: For local dev without Docker, change "postgres" → "localhost"  
# DATABASE\_URL=postgresql://smartcoach:devpass@localhost:5432/smartcoach  
  
STRAVA\_CLIENT\_ID=your\_client\_id  
STRAVA\_CLIENT\_SECRET=your\_client\_secret  
REDIRECT\_URI=http://127.0.0.1:5000/oauth/callback  
ADMIN\_USER=admin  
ADMIN\_PASS=secret  
SECRET\_KEY=supersecretkey  
CRON\_SECRET\_KEY=your\_cron\_key  
INTERNAL\_API\_KEY=your\_internal\_key  
  
```  
  
---  
  
## 💻 Running Locally  
  
```bash  
python run.py  
```  
  
Then open [http://127.0.0.1:5000/ping](http://127.0.0.1:5000/ping)   
You should see: `pong`  
  
---  
  
## 🧪 Endpoints (Milestone 1)  
  
| Route | Description |  
|-----------------|----------------------------------|  
| `/ping` | Health check |  
| `/init-db` | Creates DB tables |  
| `/auth/login` | Basic credential-based login |  
| `/auth/logout` | Clear session |  
| `/enrich/status`| Returns enrichment status (stub) |  
  
> More functionality is coming in Milestone 2  
  
---  
  
## 🧬 GitHub Actions  
  
We’ve added a skeleton workflow in `.github/workflows/cron-sync.yml` that:  
  
- Runs every 6 hours  
- Supports manual trigger  
- Runs placeholder logic (future expansion)  
  
---  
  
## 🧩 Project Structure  
  
```bash  
railway-pg-test/  
├── src/  
│ ├── app.py → Flask app entrypoint  
│ ├── routes/ → Route blueprints  
│ ├── services/ → Business logic  
│ ├── db/ → Database models + sessions  
│ └── utils/ → Utility functions  
├── schema.sql → Creates core DB tables  
├── run.py → Runs the app  
├── requirements.txt  
└── .env → Environment variables  
  
```  
  
---  
  
## 📦 Requirements  
  
- Python 3.11+  
- SQLite or Postgres  
- A Strava API App (https://www.strava.com/settings/api)

### 📄 alembic\env.py

import sys  
import os  
  
# --- Add project root to sys.path ---  
project\_root = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
sys.path.insert(0, project\_root)  
  
# --- Add 'src' folder to sys.path so Alembic can find env\_loader ---  
sys.path.insert(0, os.path.join(project\_root, "src"))  
  
# --- Now safe to import env\_loader ---  
import env\_loader # <-- This will correctly apply DATABASE\_URL patches  
  
from logging.config import fileConfig  
from sqlalchemy import engine\_from\_config, pool  
from alembic import context  
  
# DATABASE\_URL after env\_loader patched it  
DATABASE\_URL = os.getenv("DATABASE\_URL")  
  
if not DATABASE\_URL:  
 raise RuntimeError("DATABASE\_URL environment variable is not set. Alembic cannot continue.")  
  
# Import SQLAlchemy Base AFTER sys.path is fully patched  
from src.db.models.base import Base  
  
# ✅ Import all models so that Alembic can detect them  
import src.db.models.activities  
import src.db.models.tokens  
import src.db.models.splits  
  
# Alembic Config object  
config = context.config  
  
# Configure logging  
if config.config\_file\_name is not None:  
 fileConfig(config.config\_file\_name)  
  
# Set target metadata for autogenerate  
target\_metadata = Base.metadata  
  
# Inject DATABASE\_URL dynamically for Alembic migrations  
config.set\_main\_option("sqlalchemy.url", DATABASE\_URL)  
  
  
def run\_migrations\_offline():  
 """Run migrations in 'offline' mode (no DB connection)."""  
 url = config.get\_main\_option("sqlalchemy.url")  
 context.configure(  
 url=url,  
 target\_metadata=target\_metadata,  
 literal\_binds=True,  
 compare\_type=True,  
 )  
  
 with context.begin\_transaction():  
 context.run\_migrations()  
  
  
def run\_migrations\_online():  
 """Run migrations in 'online' mode (DB connection active)."""  
 connectable = engine\_from\_config(  
 config.get\_section(config.config\_ini\_section),  
 prefix="sqlalchemy.",  
 poolclass=pool.NullPool,  
 )  
  
 with connectable.connect() as connection:  
 context.configure(  
 connection=connection,  
 target\_metadata=target\_metadata,  
 compare\_type=True,  
 )  
  
 with context.begin\_transaction():  
 context.run\_migrations()  
  
  
if context.is\_offline\_mode():  
 run\_migrations\_offline()  
else:  
 run\_migrations\_online()

### 📄 alembic\versions\fbdfce5b4297\_baseline.py

"""baseline  
  
Revision ID: fbdfce5b4297  
Revises:   
Create Date: 2025-06-05 15:13:12.192899  
  
"""  
from typing import Sequence, Union  
  
from alembic import op  
import sqlalchemy as sa  
  
  
# revision identifiers, used by Alembic.  
revision: str = 'fbdfce5b4297'  
down\_revision: Union[str, None] = None  
branch\_labels: Union[str, Sequence[str], None] = None  
depends\_on: Union[str, Sequence[str], None] = None  
  
  
def upgrade() -> None:  
 """Upgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.drop\_table('splits')  
 op.drop\_table('activities')  
 op.drop\_table('tokens')  
 # ### end Alembic commands ###  
  
  
def downgrade() -> None:  
 """Downgrade schema."""  
 # ### commands auto generated by Alembic - please adjust! ###  
 op.create\_table('tokens',  
 sa.Column('athlete\_id', sa.TEXT(), autoincrement=False, nullable=False),  
 sa.Column('access\_token', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('refresh\_token', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('expires\_at', sa.BIGINT(), autoincrement=False, nullable=True),  
 sa.PrimaryKeyConstraint('athlete\_id', name=op.f('tokens\_pkey'))  
 )  
 op.create\_table('activities',  
 sa.Column('id', sa.BIGINT(), autoincrement=False, nullable=False),  
 sa.Column('activity\_id', sa.BIGINT(), autoincrement=False, nullable=True),  
 sa.Column('athlete\_id', sa.BIGINT(), autoincrement=False, nullable=True),  
 sa.Column('name', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('distance', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('moving\_time', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('elapsed\_time', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('total\_elevation\_gain', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('type', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('sport\_type', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('start\_date', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('start\_date\_local', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('timezone', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('utc\_offset', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('achievement\_count', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('kudos\_count', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('comment\_count', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('athlete\_count', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('photo\_count', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('trainer', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('commute', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('manual', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('private', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('flagged', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('visibility', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('gear\_id', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.Column('average\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('max\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('average\_cadence', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('average\_temp', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('average\_heartrate', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('max\_heartrate', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('suffer\_score', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_1', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_2', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_3', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_4', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_5', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('conv\_distance', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('conv\_elevation\_feet', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('conv\_avg\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('conv\_max\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_1\_pct', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_2\_pct', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_3\_pct', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_4\_pct', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('hr\_zone\_5\_pct', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('calories', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('external\_id', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.PrimaryKeyConstraint('id', name=op.f('activities\_pkey')),  
 sa.UniqueConstraint('activity\_id', name=op.f('activities\_activity\_id\_key'), postgresql\_include=[], postgresql\_nulls\_not\_distinct=False)  
 )  
 op.create\_table('splits',  
 sa.Column('id', sa.INTEGER(), autoincrement=True, nullable=False),  
 sa.Column('activity\_id', sa.BIGINT(), autoincrement=False, nullable=True),  
 sa.Column('lap\_index', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('distance', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('elapsed\_time', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('moving\_time', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('average\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('max\_speed', sa.DOUBLE\_PRECISION(precision=53), autoincrement=False, nullable=True),  
 sa.Column('start\_index', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('end\_index', sa.INTEGER(), autoincrement=False, nullable=True),  
 sa.Column('split', sa.BOOLEAN(), autoincrement=False, nullable=True),  
 sa.Column('created\_at', sa.TEXT(), autoincrement=False, nullable=True),  
 sa.PrimaryKeyConstraint('id', name=op.f('splits\_pkey'))  
 )  
 # ### end Alembic commands ###

### 📄 run.py

# run.py  
  
import os  
import sys  
from pathlib import Path  
from dotenv import load\_dotenv  
  
print("📦 Starting run.py...", flush=True)  
  
# 🔐 Load .env explicitly (important in Docker or if run manually)  
load\_dotenv()  
  
# Ensure the project root is on PYTHONPATH  
sys.path.insert(0, str(Path(\_\_file\_\_).resolve().parent))  
  
  
# Auto-rewrite DATABASE\_URL for local if necessary  
env\_db\_url = os.environ.get("DATABASE\_URL")  
is\_local = os.environ.get("IS\_LOCAL", "false").lower() == "true"  
  
if is\_local and env\_db\_url and 'postgres@postgres:' in env\_db\_url:  
 patched\_db\_url = env\_db\_url.replace('postgres@postgres:', 'postgres@localhost:')  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"🔧 DATABASE\_URL rewritten for local: {patched\_db\_url}", flush=True)  
else:  
 print(f"✅ DATABASE\_URL used as-is: {env\_db\_url}", flush=True)  
  
# Attempt to create the app and log failure explicitly  
try:  
 from src.app import create\_app  
 app = create\_app() # 🔑 Gunicorn will reference this: "run:app"  
 print("✅ App created via create\_app()", flush=True)  
except Exception as e:  
 print("🔥 App creation failed:", e, flush=True)  
 import traceback  
 traceback.print\_exc()  
 raise # Re-raise so Railway crash logs capture the stack  
  
# CLI mode: run init-db early and exit  
if len(sys.argv) > 1 and sys.argv[1] == "init-db":  
 from src.db.init\_db import init\_db  
 print("🔧 Running init-db...", flush=True)  
 try:  
 init\_db()  
 print("✅ init-db completed successfully", flush=True)  
 except Exception as e:  
 print("❌ init-db failed:", e, flush=True)  
 import traceback  
 traceback.print\_exc()  
 sys.exit(1)  
 sys.exit(0)  
  
# Check that templates folder is visible to Flask  
template\_dir = Path(\_\_file\_\_).parent / "templates"  
if not template\_dir.exists():  
 print(f"❌ Template folder not found: {template\_dir}")  
else:  
 print(f"📂 Template folder contents: {[f.name for f in template\_dir.glob('\*')]}")  
  
# If run directly, use Flask dev server  
if \_\_name\_\_ == "\_\_main\_\_":  
 port = int(os.environ.get("PORT", 5050))  
  
 # 🔍 Full environment debug  
 env\_db\_url = os.environ.get("DATABASE\_URL")  
 config\_db\_url = app.config.get("DATABASE\_URL") or env\_db\_url  
 print("🔍 ENV DATABASE\_URL =", env\_db\_url, flush=True)  
 print("🔍 CONFIG DATABASE\_URL =", config\_db\_url, flush=True)  
  
 if not config\_db\_url:  
 print("❗ DATABASE\_URL not set — exiting", flush=True)  
 sys.exit(1)  
  
 try:  
 from psycopg2 import connect  
 print(f"🔌 Attempting psycopg2.connect() to: {config\_db\_url}", flush=True)  
 conn = connect(config\_db\_url)  
 with conn.cursor() as cur:  
 cur.execute("SELECT 1;")  
 cur.fetchone()  
 print("✅ DB test query succeeded!", flush=True)  
 except Exception as e:  
 print("⚠️ DB test query failed:", e, flush=True)  
 import traceback  
 traceback.print\_exc()  
  
 print(f"🚀 Starting app locally on 0.0.0.0:{port}", flush=True)  
 app.run(host="0.0.0.0", port=port, debug=True)

### 📄 run\_seed.py

# run\_seed.py  
  
import sys  
import os  
  
# Always ensure root is in PYTHONPATH  
sys.path.insert(0, os.path.abspath(os.path.dirname(\_\_file\_\_)))  
  
# Now safely import your seeding function  
from src.scripts.dev\_seed\_data import seed\_activity\_and\_splits  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 seed\_activity\_and\_splits()

### 📄 src\app.py

"""  
Module: src/app.py  
Application factory and core route registration for the Smart Marathon Coach API.  
"""  
  
import os  
from pathlib import Path  
from flask import Flask  
  
from src.routes.sync\_routes import SYNC  
from src.routes.auth import auth\_bp  
from src.routes.enrich import enrich\_bp  
from src.routes.admin\_routes import admin\_bp  
from src.routes.oauth import oauth\_bp  
from src.routes.monitor\_routes import monitor\_bp # ✅ NEW IMPORT  
  
  
def create\_app(test\_config=None):  
 """  
 Create and configure a Flask application instance.  
 """  
 print("✅ ENTERED create\_app()", flush=True)  
 print("📁 CWD:", os.getcwd(), flush=True)  
 print("📁 Contents of current working dir:", os.listdir(os.getcwd()), flush=True)  
  
 # ✅ Read mode flags but DO NOT reload .env  
 env\_mode = os.getenv("FLASK\_ENV", "production")  
 is\_local = os.getenv("IS\_LOCAL", "false").lower() == "true"  
 print(f"🌍 FLASK\_ENV={env\_mode} | IS\_LOCAL={is\_local}", flush=True)  
  
 # ✅ Dump critical env vars (for debugging)  
 print("🔐 ADMIN\_USER:", os.getenv("ADMIN\_USER"))  
 print("🔐 ADMIN\_PASS:", os.getenv("ADMIN\_PASS"))  
 print("🔐 STRAVA\_CLIENT\_ID:", os.getenv("STRAVA\_CLIENT\_ID"))  
 print("🔐 STRAVA\_CLIENT\_SECRET:", os.getenv("STRAVA\_CLIENT\_SECRET"))  
 print("🔐 REDIRECT\_URI:", os.getenv("REDIRECT\_URI"))  
 print("💾 ENV DATABASE\_URL:", os.getenv("DATABASE\_URL"))  
  
 # ✅ Set absolute path to templates to ensure it resolves in production  
 templates\_path = Path(\_\_file\_\_).resolve().parent.parent / "templates"  
  
 app = Flask(  
 \_\_name\_\_,  
 instance\_relative\_config=False,  
 template\_folder=str(templates\_path)  
 )  
  
 # ✅ Config setup  
 app.config.from\_mapping(  
 SECRET\_KEY=os.environ.get("SECRET\_KEY", "dev"),  
 DATABASE\_URL=os.environ.get("DATABASE\_URL"),  
 CRON\_SECRET\_KEY=os.environ.get("CRON\_SECRET\_KEY"),  
 INTERNAL\_API\_KEY=os.environ.get("INTERNAL\_API\_KEY"),  
 )  
  
 if test\_config:  
 app.config.update(test\_config)  
  
 print("💾 CONFIG DATABASE\_URL:", app.config.get("DATABASE\_URL"))  
  
 # ✅ Register Blueprints  
 app.register\_blueprint(auth\_bp, url\_prefix="/auth")  
 app.register\_blueprint(enrich\_bp, url\_prefix="/enrich")  
 app.register\_blueprint(SYNC)  
 app.register\_blueprint(admin\_bp, url\_prefix="/admin")  
 app.register\_blueprint(oauth\_bp)  
 app.register\_blueprint(monitor\_bp)  
  
 # ✅ Diagnostic: Health check  
 @app.route("/ping")  
 def ping():  
 return "pong", 200  
  
 # ✅ Diagnostic: DB test  
 @app.route("/db-check")  
 def db\_check():  
 try:  
 from psycopg2 import connect  
 db\_url = app.config.get("DATABASE\_URL")  
 print("🧪 /db-check using DB URL:", db\_url, flush=True)  
 conn = connect(db\_url)  
 with conn.cursor() as cur:  
 cur.execute("SELECT 1;")  
 cur.fetchone()  
 return {"status": "ok", "db": True}  
 except Exception as e:  
 import traceback  
 print("🔥 DB-CHECK EXCEPTION:", flush=True)  
 traceback.print\_exc()  
 return {"status": "fail", "error": str(e)}, 500  
  
 # ✅ Diagnostic: Environment snapshot  
 @app.route("/startup")  
 def startup():  
 return {  
 "status": "started",  
 "env\_PORT": os.environ.get("PORT"),  
 "env\_DATABASE\_URL": os.environ.get("DATABASE\_URL"),  
 "config\_DATABASE\_URL": app.config.get("DATABASE\_URL"),  
 "cwd": os.getcwd(),  
 "files": [p.name for p in Path(".").iterdir()],  
 }  
  
 # ✅ Debug: List all registered routes  
 print("✅ Registered routes:")  
 for rule in app.url\_map.iter\_rules():  
 print(f" {rule.rule} -> {rule.endpoint}", flush=True)  
  
 return app

### 📄 src\db\dao\activity\_dao.py

from sqlalchemy.dialects.postgresql import insert  
from sqlalchemy.orm import Session  
from src.db.models.activities import Activity  
  
def upsert\_activities(session: Session, athlete\_id: int, activities: list[dict]) -> int:  
 """  
 Upserts a list of activity dicts into the 'activities' table using SQLAlchemy ORM session.  
 Fully safe for PostgreSQL transactions (uses flush/commit correctly).  
 """  
 if not activities:  
 return 0  
  
 stmt = insert(Activity).values([{  
 "activity\_id": act["id"],  
 "athlete\_id": athlete\_id,  
 "name": act.get("name"),  
 "type": act.get("type"),  
 "start\_date": act.get("start\_date"),  
 "distance": act.get("distance"),  
 "elapsed\_time": act.get("elapsed\_time"),  
 "moving\_time": act.get("moving\_time"),  
 "total\_elevation\_gain": act.get("total\_elevation\_gain"),  
 "external\_id": act.get("external\_id"),  
 "timezone": act.get("timezone"),  
 } for act in activities])  
  
 # Conflict resolution for PostgreSQL upsert  
 update\_cols = {  
 "name": stmt.excluded.name,  
 "type": stmt.excluded.type,  
 "start\_date": stmt.excluded.start\_date,  
 "distance": stmt.excluded.distance,  
 "elapsed\_time": stmt.excluded.elapsed\_time,  
 "moving\_time": stmt.excluded.moving\_time,  
 "total\_elevation\_gain": stmt.excluded.total\_elevation\_gain,  
 "external\_id": stmt.excluded.external\_id,  
 "timezone": stmt.excluded.timezone,  
 }  
  
 stmt = stmt.on\_conflict\_do\_update(  
 index\_elements=["activity\_id"],  
 set\_=update\_cols  
 )  
  
 result = session.execute(stmt)  
 session.commit() # Ensure commit here after both activity and split insertions  
 return result.rowcount

### 📄 src\db\dao\split\_dao.py

from sqlalchemy.dialects.postgresql import insert  
from src.db.models.splits import Split  
  
def upsert\_splits(session, splits: list) -> int:  
 """  
 Upserts splits into the 'splits' table using SQLAlchemy ORM session.  
 Fully safe for PostgreSQL transactions (uses flush/commit correctly).  
 """  
 if not splits:  
 return 0  
  
 stmt = insert(Split).values([{  
 "activity\_id": split["activity\_id"],  
 "lap\_index": split["lap\_index"],  
 "distance": split["distance"],  
 "elapsed\_time": split["elapsed\_time"],  
 "moving\_time": split["moving\_time"],  
 "average\_speed": split["average\_speed"],  
 "max\_speed": split["max\_speed"],  
 "start\_index": split["start\_index"],  
 "end\_index": split["end\_index"],  
 "split": split["split"],  
 "average\_heartrate": split.get("average\_heartrate"),  
 "pace\_zone": split.get("pace\_zone"),  
 "conv\_distance": split.get("conv\_distance"),  
 "conv\_avg\_speed": split.get("conv\_avg\_speed"),  
 "conv\_moving\_time": split.get("conv\_moving\_time"),  
 "conv\_elapsed\_time": split.get("conv\_elapsed\_time")  
 } for split in splits])  
  
 update\_cols = {  
 "lap\_index": stmt.excluded.lap\_index,  
 "distance": stmt.excluded.distance,  
 "elapsed\_time": stmt.excluded.elapsed\_time,  
 "moving\_time": stmt.excluded.moving\_time,  
 "average\_speed": stmt.excluded.average\_speed,  
 "max\_speed": stmt.excluded.max\_speed,  
 "start\_index": stmt.excluded.start\_index,  
 "end\_index": stmt.excluded.end\_index,  
 "split": stmt.excluded.split,  
 "average\_heartrate": stmt.excluded.average\_heartrate,  
 "pace\_zone": stmt.excluded.pace\_zone,  
 "conv\_distance": stmt.excluded.conv\_distance,  
 "conv\_avg\_speed": stmt.excluded.conv\_avg\_speed,  
 "conv\_moving\_time": stmt.excluded.conv\_moving\_time,  
 "conv\_elapsed\_time": stmt.excluded.conv\_elapsed\_time  
 }  
  
 stmt = stmt.on\_conflict\_do\_update(  
 index\_elements=["activity\_id", "lap\_index"],  
 set\_=update\_cols  
 )  
  
 result = session.execute(stmt)  
 session.commit()  
 return result.rowcount

### 📄 src\db\dao\token\_dao.py

# src/db/dao/token\_dao.py  
  
from datetime import datetime  
from sqlalchemy.exc import NoResultFound, IntegrityError  
from flask import current\_app  
import os  
  
from src.db.db\_session import get\_session # ✅ Fixed import: consistent db\_session usage  
from src.db.models.tokens import Token  
  
  
def get\_tokens\_sa(session, athlete\_id):  
 """  
 Retrieve access and refresh tokens for the given athlete using SQLAlchemy.  
 """  
 try:  
 token = session.query(Token).filter\_by(athlete\_id=athlete\_id).one()  
 return {  
 "access\_token": token.access\_token,  
 "refresh\_token": token.refresh\_token  
 }  
 except NoResultFound:  
 return None  
  
  
def save\_tokens\_sa(session, athlete\_id, access\_token, refresh\_token, expires\_at=None):  
 """  
 Save or update tokens for an athlete using SQLAlchemy.  
  
 For Strava, tokens are opaque (not JWT), so we skip decoding and simply store them.  
 """  
 try:  
 # If expires\_at not provided, fallback to reasonable default (1 hour)  
 if not expires\_at:  
 now\_ts = int(datetime.utcnow().timestamp())  
 expires\_at = now\_ts + 3600  
  
 token = session.query(Token).filter\_by(athlete\_id=athlete\_id).one\_or\_none()  
  
 if token:  
 token.access\_token = access\_token  
 token.refresh\_token = refresh\_token  
 token.expires\_at = expires\_at  
 else:  
 token = Token(  
 athlete\_id=athlete\_id,  
 access\_token=access\_token,  
 refresh\_token=refresh\_token,  
 expires\_at=expires\_at  
 )  
 session.add(token)  
  
 session.commit()  
 except IntegrityError as e:  
 session.rollback()  
 raise RuntimeError(f"Failed to save tokens via SQLAlchemy: {e}")  
 except Exception as e:  
 session.rollback()  
 raise RuntimeError(f"Token saving failed: {e}")  
  
  
def get\_valid\_access\_token\_sa(session, athlete\_id):  
 """  
 Retrieve a valid (non-expired) access token using SQLAlchemy.  
 """  
 try:  
 token = session.query(Token).filter\_by(athlete\_id=athlete\_id).one\_or\_none()  
 if not token:  
 return None  
  
 now\_ts = int(datetime.utcnow().timestamp())  
 if token.expires\_at > now\_ts:  
 return token.access\_token  
 else:  
 print(f"⚠️ Token expired for athlete {athlete\_id}")  
 return None  
 except Exception as e:  
 raise RuntimeError(f"Failed to fetch valid access token: {e}")

### 📄 src\db\db\_session.py

# src/db/db\_session.py  
  
import os  
from sqlalchemy import create\_engine  
from sqlalchemy.orm import declarative\_base, sessionmaker  
from flask import current\_app, has\_app\_context  
  
# Global declarative base — shared across models  
Base = declarative\_base()  
  
def resolve\_db\_url():  
 """  
 Resolve DATABASE\_URL from Flask config or environment variables.  
 """  
 if has\_app\_context():  
 # Use Flask app config if inside app context  
 return current\_app.config.get("DATABASE\_URL") or os.getenv("DATABASE\_URL")  
 return os.getenv("DATABASE\_URL")  
  
def get\_engine(db\_url=None):  
 """  
 Create a SQLAlchemy engine.  
 Allows optional db\_url override for tests or special cases.  
 """  
 db\_url = db\_url or resolve\_db\_url()  
 if not db\_url:  
 raise RuntimeError("DATABASE\_URL is not set in environment or app config.")  
 return create\_engine(db\_url, echo=False, future=True)  
  
def get\_session(engine=None):  
 """  
 Create a new SQLAlchemy sessionmaker (not a global session).  
 Allows optional engine injection for test harnesses.  
 """  
 engine = engine or get\_engine()  
 SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False, future=True)  
 return SessionLocal()

### 📄 src\db\models\activities.py

from sqlalchemy import Column, BigInteger, Integer, String, Float, DateTime  
from src.db.db\_session import Base  
  
class Activity(Base):  
 \_\_tablename\_\_ = "activities"  
  
 activity\_id = Column(BigInteger, primary\_key=True, index=True)  
 athlete\_id = Column(BigInteger, nullable=False, index=True)  
 name = Column(String)  
 type = Column(String)  
 start\_date = Column(DateTime)  
 distance = Column(Float)  
 elapsed\_time = Column(Integer)  
 moving\_time = Column(Integer)  
 total\_elevation\_gain = Column(Float)  
 external\_id = Column(String)  
 timezone = Column(String)  
  
 average\_speed = Column(Float)  
 max\_speed = Column(Float)  
 suffer\_score = Column(Float)  
 average\_heartrate = Column(Float)  
 max\_heartrate = Column(Float)  
 calories = Column(Float)  
  
 conv\_distance = Column(Float)  
 conv\_elevation\_feet = Column(Float)  
 conv\_avg\_speed = Column(Float)  
 conv\_max\_speed = Column(Float)  
 conv\_moving\_time = Column(String)  
 conv\_elapsed\_time = Column(String)  
  
 hr\_zone\_1\_pct = Column(Float)  
 hr\_zone\_2\_pct = Column(Float)  
 hr\_zone\_3\_pct = Column(Float)  
 hr\_zone\_4\_pct = Column(Float)  
 hr\_zone\_5\_pct = Column(Float)

### 📄 src\db\models\base.py

from sqlalchemy.orm import declarative\_base  
  
Base = declarative\_base()

### 📄 src\db\models\splits.py

from sqlalchemy import Column, Integer, Float, Boolean, ForeignKey, TIMESTAMP, BigInteger  
from sqlalchemy.sql import func  
from sqlalchemy import UniqueConstraint  
from src.db.db\_session import Base  
  
class Split(Base):  
 \_\_tablename\_\_ = "splits"  
  
 id = Column(Integer, primary\_key=True)  
 activity\_id = Column(BigInteger, ForeignKey("activities.activity\_id", ondelete="CASCADE"), nullable=False)  
 lap\_index = Column(Integer, nullable=False)  
 distance = Column(Float)  
 elapsed\_time = Column(Integer)  
 moving\_time = Column(Integer)  
 average\_speed = Column(Float)  
 max\_speed = Column(Float)  
 start\_index = Column(Integer)  
 end\_index = Column(Integer)  
 split = Column(Boolean, default=True)  
 average\_heartrate = Column(Float) # <-- NEW  
 pace\_zone = Column(Integer) # <-- NEW  
 created\_at = Column(TIMESTAMP, server\_default=func.now())  
  
 \_\_table\_args\_\_ = (  
 UniqueConstraint("activity\_id", "lap\_index", name="uq\_activity\_lap"),  
 )

### 📄 src\db\models\tokens.py

# src/db/models/tokens.py  
  
from sqlalchemy import Column, BigInteger, Text  
from src.db.db\_session import Base  
  
class Token(Base):  
 \_\_tablename\_\_ = "tokens"  
  
 athlete\_id = Column(BigInteger, primary\_key=True)  
 access\_token = Column(Text, nullable=False)  
 refresh\_token = Column(Text, nullable=False)  
 expires\_at = Column(BigInteger, nullable=False)

### 📄 src\env\_loader.py

import os  
from dotenv import load\_dotenv  
  
# Load .env variables automatically  
load\_dotenv()  
  
# Pull DATABASE\_URL early  
env\_db\_url = os.environ.get("DATABASE\_URL")  
is\_local = os.environ.get("IS\_LOCAL", "false").lower() == "true"  
in\_docker = os.path.exists('/.dockerenv') or os.environ.get("IN\_DOCKER", "false").lower() == "true"  
  
# Step 1 — Rewrite DATABASE\_URL depending on environment  
if env\_db\_url:  
 patched\_db\_url = env\_db\_url  
  
 # Docker Compose case: container hostname replacement  
 if in\_docker and "localhost" in env\_db\_url:  
 patched\_db\_url = env\_db\_url.replace("localhost", "postgres")  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"🔧 DATABASE\_URL rewritten for Docker Compose: {patched\_db\_url}")  
  
 # Local mode with Docker aliases  
 elif is\_local:  
 if '@postgres:' in env\_db\_url:  
 patched\_db\_url = env\_db\_url.replace('@postgres:', '@localhost:')  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"🔧 DATABASE\_URL rewritten for local Docker (postgres alias): {patched\_db\_url}")  
 elif '@db:' in env\_db\_url:  
 patched\_db\_url = env\_db\_url.replace('@db:', '@localhost:')  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"🔧 DATABASE\_URL rewritten for local Docker (db alias): {patched\_db\_url}")  
 else:  
 print(f"✅ DATABASE\_URL used for local development: {patched\_db\_url}")  
  
 else:  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"✅ DATABASE\_URL used as-is: {patched\_db\_url}")  
  
# Step 2 — SQLAlchemy dialect normalization for psycopg2  
patched\_db\_url = os.environ.get("DATABASE\_URL")  
if patched\_db\_url and patched\_db\_url.startswith("postgresql+psycopg2://"):  
 patched\_db\_url = patched\_db\_url.replace("postgresql+psycopg2://", "postgresql://")  
 os.environ['DATABASE\_URL'] = patched\_db\_url  
 print(f"🔧 DATABASE\_URL normalized for psycopg2: {patched\_db\_url}")  
  
# Final output for verification  
final\_db\_url = os.environ.get("DATABASE\_URL")  
print(f"🔍 Final DATABASE\_URL: {final\_db\_url}")

### 📄 src\routes\admin\_routes.py

from flask import Blueprint  
  
admin\_bp = Blueprint("admin", \_\_name\_\_)  
  
@admin\_bp.route("/admin/ping")  
def ping():  
 return "pong from admin"

### 📄 src\routes\auth.py

# src/routes/auth.py  
  
from flask import Blueprint, request, jsonify  
from src.services.auth import login\_user, logout\_user, refresh\_token  
  
auth\_bp = Blueprint("auth", \_\_name\_\_, url\_prefix="/auth")  
  
  
@auth\_bp.route("/login", methods=["POST"])  
def login():  
 """Authenticate and issue access + refresh tokens."""  
 data = request.get\_json() or {}  
 try:  
 access, refresh = login\_user(data)  
 return jsonify({  
 "access\_token": access,  
 "refresh\_token": refresh  
 }), 200  
 except PermissionError as e:  
 return jsonify({"error": str(e)}), 401  
  
  
@auth\_bp.route("/refresh", methods=["POST"])  
def refresh():  
 """Issue new access token using a refresh token from Authorization header."""  
 auth\_header = request.headers.get("Authorization", None)  
 if not auth\_header or not auth\_header.startswith("Bearer "):  
 return jsonify({"error": "Missing or invalid Authorization header"}), 401  
  
 token = auth\_header.split(" ")[1]  
 try:  
 new\_token = refresh\_token(token)  
 return jsonify({"access\_token": new\_token}), 200  
 except PermissionError as e:  
 return jsonify({"error": str(e)}), 401  
  
  
@auth\_bp.route("/logout", methods=["POST"])  
def logout():  
 """Revoke refresh token. Currently a no-op."""  
 data = request.get\_json() or {}  
 logout\_user(data.get("refresh\_token"))  
 return jsonify({"message": "logged out"}), 200

### 📄 src\routes\enrich.py

# src/routes/enrich.py  
  
from flask import Blueprint, jsonify, request  
from src.services import enrichment\_sync  
from src.db.db\_session import get\_session  
from sqlalchemy import text  
  
enrich\_bp = Blueprint("enrich", \_\_name\_\_)  
  
@enrich\_bp.route("/status", methods=["GET"])  
def status():  
 """Quick health check"""  
 return jsonify({"enrich": "ok"}), 200  
  
@enrich\_bp.route("/activity/<int:activity\_id>", methods=["POST"])  
def enrich\_single(activity\_id):  
 """Trigger enrichment for a single activity directly"""  
 session = get\_session()  
 try:  
 # Grab athlete\_id directly from DB for safety  
 row = session.execute(  
 text("SELECT athlete\_id FROM activities WHERE activity\_id = :id"),  
 {"id": activity\_id}  
 ).fetchone()  
  
 if not row:  
 return jsonify({"error": f"Activity {activity\_id} not found"}), 404  
  
 athlete\_id = row.athlete\_id  
  
 # ✅ Use your service helper that automatically handles token refresh  
 enrichment\_sync.enrich\_one\_activity\_with\_refresh(  
 session=session,  
 athlete\_id=athlete\_id,  
 activity\_id=activity\_id  
 )  
  
 return jsonify({"status": f"Activity {activity\_id} enriched"}), 200  
  
 except Exception as e:  
 return jsonify({"error": str(e)}), 500  
  
 finally:  
 session.close()  
  
@enrich\_bp.route("/batch-enrich", methods=["POST"])  
def enrich\_batch():  
 """  
 Enrich multiple activities for an athlete.  
 You can pass ?athlete\_id=X&batch=N  
 """  
 athlete\_id = request.args.get("athlete\_id", type=int)  
 batch = request.args.get("batch", default=20, type=int)  
  
 if not athlete\_id:  
 return jsonify({"error": "Missing athlete\_id"}), 400  
  
 try:  
 enrichment\_sync.enrichment\_loop(athlete\_id, batch\_size=batch)  
 return jsonify({"status": f"Batch enrichment complete for athlete {athlete\_id}"}), 200  
 except Exception as e:  
 return jsonify({"error": str(e)}), 500

### 📄 src\routes\monitor\_routes.py

# src/routes/monitor\_routes.py  
  
import os  
import traceback  
from flask import Blueprint, jsonify  
  
from src.db.db\_session import get\_engine, get\_session # ✅ Correct imports  
from src.db.dao.token\_dao import get\_tokens\_sa, get\_valid\_access\_token\_sa # ✅ Correct DAO imports  
  
monitor\_bp = Blueprint("monitor", \_\_name\_\_)  
  
@monitor\_bp.route("/monitor-tokens")  
def monitor\_tokens():  
 try:  
 db\_url = os.getenv("DATABASE\_URL")  
 engine = get\_engine(db\_url)  
 session = get\_session(engine)  
 results = []  
  
 # Fetch all token entries using SQLAlchemy  
 tokens = session.query(get\_tokens\_sa.\_\_annotations\_\_.get('return')).all()  
  
 for token in tokens:  
 athlete\_id = token.athlete\_id  
 try:  
 access\_token = get\_valid\_access\_token\_sa(session, athlete\_id)  
 results.append({"athlete\_id": athlete\_id, "status": "ok" if access\_token else "expired"})  
 except Exception as e:  
 results.append({  
 "athlete\_id": athlete\_id,  
 "status": "error",  
 "error": str(e),  
 })  
  
 return jsonify(results=results), 200  
  
 except Exception as e:  
 traceback.print\_exc()  
 return jsonify(error="Monitor failed", details=str(e)), 500

### 📄 src\routes\oauth.py

# src/routes/oauth.py  
  
from flask import Blueprint, request, jsonify  
import os  
import requests  
import traceback  
  
from src.db.db\_session import get\_session  
from src.db.dao.token\_dao import save\_tokens\_sa  
from src.services.activity\_sync import sync\_full\_history  
  
oauth\_bp = Blueprint("oauth", \_\_name\_\_)  
  
@oauth\_bp.route("/oauth/callback")  
def oauth\_callback():  
 try:  
 print("📥 /oauth/callback hit", flush=True)  
 code = request.args.get("code")  
 state = request.args.get("state")  
 print("📦 Code received:", code, flush=True)  
 print("🆔 State (athlete ID hint):", state, flush=True)  
  
 if not code:  
 return jsonify(error="Missing `code` param in query string"), 400  
  
 # Read env vars  
 client\_id = os.getenv("STRAVA\_CLIENT\_ID")  
 client\_secret = os.getenv("STRAVA\_CLIENT\_SECRET")  
 redirect\_uri = os.getenv("REDIRECT\_URI")  
  
 if not client\_id or not client\_secret or not redirect\_uri:  
 return jsonify(error="Missing required environment variables"), 500  
  
 print("🌐 Preparing token exchange request", flush=True)  
  
 client\_id\_int = int(client\_id)  
  
 response = requests.post(  
 "https://www.strava.com/api/v3/oauth/token",  
 data={  
 "client\_id": client\_id\_int,  
 "client\_secret": client\_secret,  
 "code": code,  
 "grant\_type": "authorization\_code",  
 "redirect\_uri": redirect\_uri  
 },  
 timeout=10,  
 )  
  
 response.raise\_for\_status()  
 tokens = response.json()  
  
 # Defensive parsing for robustness  
 athlete = tokens.get("athlete")  
 if not athlete or "id" not in athlete:  
 return jsonify(error="Strava response missing athlete ID"), 502  
  
 athlete\_id = athlete["id"]  
 access\_token = tokens.get("access\_token")  
 refresh\_token = tokens.get("refresh\_token")  
 expires\_at = tokens.get("expires\_at")  
  
 if not all([access\_token, refresh\_token, expires\_at]):  
 return jsonify(error="Strava response missing required tokens"), 502  
  
 print(f"✅ Got token for athlete {athlete\_id}", flush=True)  
  
 session = get\_session()  
 try:  
 save\_tokens\_sa(  
 session,  
 athlete\_id=athlete\_id,  
 access\_token=access\_token,  
 refresh\_token=refresh\_token,  
 expires\_at=expires\_at  
 )  
  
 # ✅ Corrected call signature for sync\_full\_history  
 inserted = sync\_full\_history(  
 session=session,  
 athlete\_id=athlete\_id,  
 access\_token=access\_token,  
 lookback\_days=730  
 )  
  
 print(f"📊 Historical sync inserted {inserted} activities for athlete {athlete\_id}", flush=True)  
 session.commit()  
  
 except Exception as e:  
 session.rollback()  
 print("🔥 DB operation failed:", e, flush=True)  
 traceback.print\_exc()  
 return jsonify(error="Database failure", details=str(e)), 500  
 finally:  
 session.close()  
  
 return jsonify(message="OAuth success!", athlete\_id=athlete\_id, inserted=inserted), 200  
  
 except requests.RequestException as req\_err:  
 print("❌ RequestException:", str(req\_err), flush=True)  
 return jsonify(error="Token exchange failed", details=str(req\_err)), 502  
  
 except Exception as e:  
 print("🔥 Internal Error:", str(e), flush=True)  
 traceback.print\_exc()  
 return jsonify(error="Internal Error", details=str(e)), 500

### 📄 src\routes\sync\_routes.py

# src/routes/sync\_routes.py  
  
import os  
import traceback  
from flask import Blueprint, request, jsonify  
  
# Services  
from src.services.activity\_sync import sync\_recent  
from src.services.strava import generate\_strava\_auth\_url  
  
# DAO imports (SQLAlchemy-only)  
from src.db.db\_session import get\_session  
  
SYNC = Blueprint("sync", \_\_name\_\_)  
  
@SYNC.route("/sync-strava-to-db/<int:athlete\_id>")  
def sync\_to\_db(athlete\_id):  
 """Endpoint for CRON-based syncs using a secret key."""  
 cron\_key = os.getenv("CRON\_SECRET\_KEY")  
 key = request.args.get("key")  
 print(f"🔐 Incoming key: {key}")  
 print(f"🔐 Expected key from env: {cron\_key}")  
  
 if not cron\_key or key != cron\_key:  
 return jsonify(error="Unauthorized"), 401  
  
 session = None  
 try:  
 session = get\_session()  
  
 inserted = sync\_recent(  
 session=session,  
 athlete\_id=athlete\_id  
 )  
  
 # ✅ Corrected to match test expectation  
 return jsonify(inserted=inserted), 200  
  
 except Exception as e:  
 traceback.print\_exc()  
 return jsonify(error="Sync failed", details=str(e)), 500  
  
 finally:  
 if session:  
 session.close()  
  
@SYNC.route("/init-db")  
def init\_db\_route():  
 """Manual DB initializer (same as /run.py -- init-db)."""  
 from src.scripts.dev\_only\_init\_db import init\_db  
 try:  
 init\_db()  
 return "✅ init\_db() completed successfully", 200  
 except Exception as e:  
 traceback.print\_exc()  
 return f"❌ Error initializing DB: {e}", 500

### 📄 src\routes\web\_routes.py

from flask import Blueprint  
  
web\_bp = Blueprint("web", \_\_name\_\_)  
  
@web\_bp.route("/web/ping")  
def ping():  
 return "pong from web"

### 📄 src\scripts\backfill\_pilot\_test.py

from src.db.db\_session import get\_session  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.env\_loader import \* # load environment patcher  
  
  
  
  
# TODO: Replace this with a real athlete\_id you have authorized via OAuth  
athlete\_id = 347085  
  
  
# Create DB session  
session = get\_session()  
  
try:  
 # Step 1: Get valid access token (auto-refresh if needed)  
 access\_token = ensure\_fresh\_token(session, athlete\_id)  
  
 # Step 2: Run controlled backfill pilot  
 synced\_count = sync\_full\_history(  
 session=session,  
 athlete\_id=athlete\_id,  
 access\_token=access\_token,  
 lookback\_days=30, # ~1 month lookback to yield ~50 activities  
 max\_activities=50 # hard cap at 50 activities for this pilot run  
 )  
  
 print(f"✅ Pilot backfill complete — {synced\_count} activities synced for athlete {athlete\_id}")  
  
except Exception as e:  
 print(f"❌ Backfill pilot failed: {e}")  
  
finally:  
 session.close()

### 📄 src\scripts\backfill\_sync.py

import time  
import logging  
import requests  
from datetime import datetime, timedelta  
from sqlalchemy import text  
from sqlalchemy.exc import SQLAlchemyError  
from src.db.db\_session import get\_session  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
  
log = logging.getLogger("backfill\_sync")  
log.setLevel(logging.INFO)  
  
STRAVA\_ACTIVITIES\_URL = "https://www.strava.com/api/v3/athlete/activities"  
STRAVA\_ZONE\_URL = "https://www.strava.com/api/v3/activities/{activity\_id}/zones"  
  
DEFAULT\_RETRY\_LIMIT = 5  
DEFAULT\_SLEEP = 5  
DEFAULT\_RETRY\_BACKOFF = 2  
  
# --- Conversion helpers (same as before) ---  
def meters\_to\_miles(meters):  
 return round(meters / 1609.344, 2) if meters else None  
  
def meters\_to\_feet(meters):  
 return round(meters \* 3.28084, 1) if meters else None  
  
def mps\_to\_min\_per\_mile(mps):  
 return round(26.8224 / mps, 2) if mps and mps > 0 else None  
  
def format\_seconds\_to\_hms(seconds):  
 if seconds is None:  
 return None  
 minutes, sec = divmod(seconds, 60)  
 hours, minutes = divmod(minutes, 60)  
 return f"{hours}:{minutes:02}:{sec:02}" if hours else f"{minutes}:{sec:02}"  
  
# ------------------------------------------------  
  
def fetch\_hr\_zone\_percentages(activity\_id, access\_token):  
 url = STRAVA\_ZONE\_URL.format(activity\_id=activity\_id)  
 headers = {"Authorization": f"Bearer {access\_token}"}  
 resp = requests.get(url, headers=headers, timeout=10)  
  
 if resp.status\_code != 200:  
 return None  
   
 zones\_data = resp.json()  
 for zone\_group in zones\_data:  
 if zone\_group.get("type") == "heartrate":  
 hr\_zones = zone\_group.get("distribution\_buckets", [])  
 times = [z.get("time") or 0.0 for z in hr\_zones]  
 total\_time = sum(times)  
 if total\_time == 0:  
 return None  
 zone\_pcts = [  
 round(((z.get("time") or 0.0) / total\_time) \* 100, 1)  
 for z in hr\_zones  
 ]  
 zone\_pcts = zone\_pcts[:5]  
 while len(zone\_pcts) < 5:  
 zone\_pcts.append(0.0)  
 return zone\_pcts  
 return None  
  
def pull\_recent\_activities(access\_token, after\_date):  
 all\_activities = []  
 page = 1  
 while True:  
 params = {  
 "after": int(after\_date.timestamp()),  
 "per\_page": 100,  
 "page": page  
 }  
 headers = {"Authorization": f"Bearer {access\_token}"}  
 resp = requests.get(STRAVA\_ACTIVITIES\_URL, headers=headers, params=params, timeout=10)  
   
 if resp.status\_code == 429:  
 retry\_after = int(resp.headers.get("Retry-After", DEFAULT\_SLEEP))  
 log.warning(f"Rate limit hit, sleeping {retry\_after}s")  
 time.sleep(retry\_after)  
 continue  
 elif resp.status\_code != 200:  
 log.error(f"Failed activity fetch HTTP {resp.status\_code}")  
 break  
   
 batch = resp.json()  
 if not batch:  
 break  
   
 all\_activities.extend(batch)  
 page += 1  
   
 time.sleep(1)  
 return all\_activities  
  
def upsert\_activity(session, activity\_json, hr\_zone\_pcts):  
 params = {  
 "activity\_id": activity\_json["id"],  
 "athlete\_id": activity\_json["athlete"].get("id"),  
 "name": activity\_json.get("name"),  
 "type": activity\_json.get("type"),  
 "start\_date": activity\_json.get("start\_date"),  
 "distance": activity\_json.get("distance"),  
 "moving\_time": activity\_json.get("moving\_time"),  
 "elapsed\_time": activity\_json.get("elapsed\_time"),  
 "total\_elevation\_gain": activity\_json.get("total\_elevation\_gain"),  
 "external\_id": activity\_json.get("external\_id"),  
 "timezone": activity\_json.get("timezone"),  
 "average\_speed": activity\_json.get("average\_speed"),  
 "max\_speed": activity\_json.get("max\_speed"),  
 "suffer\_score": activity\_json.get("suffer\_score"),  
 "average\_heartrate": activity\_json.get("average\_heartrate"),  
 "max\_heartrate": activity\_json.get("max\_heartrate"),  
 "calories": activity\_json.get("calories"),  
 "conv\_distance": meters\_to\_miles(activity\_json.get("distance")),  
 "conv\_elevation\_feet": meters\_to\_feet(activity\_json.get("total\_elevation\_gain")),  
 "conv\_avg\_speed": mps\_to\_min\_per\_mile(activity\_json.get("average\_speed")),  
 "conv\_max\_speed": mps\_to\_min\_per\_mile(activity\_json.get("max\_speed")),  
 "conv\_moving\_time": format\_seconds\_to\_hms(activity\_json.get("moving\_time")),  
 "conv\_elapsed\_time": format\_seconds\_to\_hms(activity\_json.get("elapsed\_time")),  
 "hr\_zone\_1\_pct": hr\_zone\_pcts[0] if hr\_zone\_pcts else None,  
 "hr\_zone\_2\_pct": hr\_zone\_pcts[1] if hr\_zone\_pcts else None,  
 "hr\_zone\_3\_pct": hr\_zone\_pcts[2] if hr\_zone\_pcts else None,  
 "hr\_zone\_4\_pct": hr\_zone\_pcts[3] if hr\_zone\_pcts else None,  
 "hr\_zone\_5\_pct": hr\_zone\_pcts[4] if hr\_zone\_pcts else None  
 }  
   
 session.execute(text("""  
 INSERT INTO activities (  
 activity\_id, athlete\_id, name, type, start\_date, distance, moving\_time, elapsed\_time,   
 total\_elevation\_gain, external\_id, timezone, average\_speed, max\_speed, suffer\_score,   
 average\_heartrate, max\_heartrate, calories, conv\_distance, conv\_elevation\_feet,   
 conv\_avg\_speed, conv\_max\_speed, conv\_moving\_time, conv\_elapsed\_time,   
 hr\_zone\_1\_pct, hr\_zone\_2\_pct, hr\_zone\_3\_pct, hr\_zone\_4\_pct, hr\_zone\_5\_pct)  
 VALUES (  
 :activity\_id, :athlete\_id, :name, :type, :start\_date, :distance, :moving\_time, :elapsed\_time,   
 :total\_elevation\_gain, :external\_id, :timezone, :average\_speed, :max\_speed, :suffer\_score,   
 :average\_heartrate, :max\_heartrate, :calories, :conv\_distance, :conv\_elevation\_feet,   
 :conv\_avg\_speed, :conv\_max\_speed, :conv\_moving\_time, :conv\_elapsed\_time,   
 :hr\_zone\_1\_pct, :hr\_zone\_2\_pct, :hr\_zone\_3\_pct, :hr\_zone\_4\_pct, :hr\_zone\_5\_pct)  
 ON CONFLICT (activity\_id) DO UPDATE SET  
 name = EXCLUDED.name,  
 type = EXCLUDED.type,  
 distance = EXCLUDED.distance,  
 moving\_time = EXCLUDED.moving\_time,  
 elapsed\_time = EXCLUDED.elapsed\_time,  
 total\_elevation\_gain = EXCLUDED.total\_elevation\_gain,  
 average\_speed = EXCLUDED.average\_speed,  
 max\_speed = EXCLUDED.max\_speed,  
 suffer\_score = EXCLUDED.suffer\_score,  
 average\_heartrate = EXCLUDED.average\_heartrate,  
 max\_heartrate = EXCLUDED.max\_heartrate,  
 calories = EXCLUDED.calories,  
 conv\_distance = EXCLUDED.conv\_distance,  
 conv\_elevation\_feet = EXCLUDED.conv\_elevation\_feet,  
 conv\_avg\_speed = EXCLUDED.conv\_avg\_speed,  
 conv\_max\_speed = EXCLUDED.conv\_max\_speed,  
 conv\_moving\_time = EXCLUDED.conv\_moving\_time,  
 conv\_elapsed\_time = EXCLUDED.conv\_elapsed\_time,  
 hr\_zone\_1\_pct = EXCLUDED.hr\_zone\_1\_pct,  
 hr\_zone\_2\_pct = EXCLUDED.hr\_zone\_2\_pct,  
 hr\_zone\_3\_pct = EXCLUDED.hr\_zone\_3\_pct,  
 hr\_zone\_4\_pct = EXCLUDED.hr\_zone\_4\_pct,  
 hr\_zone\_5\_pct = EXCLUDED.hr\_zone\_5\_pct  
 """), params)  
  
def run\_backfill(athlete\_id, months=6):  
 with get\_session() as session:  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
 after\_date = datetime.utcnow() - timedelta(days=30 \* months)  
 log.info(f"Fetching data since {after\_date.date()}...")  
 activities = pull\_recent\_activities(access\_token, after\_date)  
 log.info(f"Found {len(activities)} activities")  
  
 for activity\_json in activities:  
 retries = 0  
 while retries < DEFAULT\_RETRY\_LIMIT:  
 try:  
 hr\_zone\_pcts = fetch\_hr\_zone\_percentages(activity\_json['id'], access\_token)  
 upsert\_activity(session, activity\_json, hr\_zone\_pcts)  
 session.commit()  
 log.info(f"✅ Synced activity {activity\_json['id']}")  
 break  
 except Exception as e:  
 retries += 1  
 log.warning(f"Retry {retries} for activity {activity\_json['id']}: {e}")  
 time.sleep(DEFAULT\_SLEEP \* (DEFAULT\_RETRY\_BACKOFF \*\* retries))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 run\_backfill(athlete\_id=347085, months=6)

### 📄 src\scripts\dev\_only\_init\_db.py

import os  
import psycopg2  
from urllib.parse import urlparse  
from src.db.db\_session import get\_engine # ✅ Correct import for unified engine creation  
from src.db.models.tokens import Base as TokensBase  
from src.db.models.activities import Base as ActivitiesBase  
  
  
def get\_conn(db\_url=None):  
 """  
 Return a low-level database connection.  
 Uses PostgreSQL if the URL is set.  
 """  
 db\_url = db\_url or os.getenv("DATABASE\_URL")  
  
 if db\_url is None:  
 raise RuntimeError("DATABASE\_URL is not set!")  
  
 parsed = urlparse(db\_url)  
  
 # Only PostgreSQL supported (SQLite fallback not required)  
 ssl\_mode = "disable" if parsed.hostname in ("localhost", "127.0.0.1", "db", "postgres") else "require"  
 conn = psycopg2.connect(  
 dbname=parsed.path.lstrip("/"),  
 user=parsed.username,  
 password=parsed.password,  
 host=parsed.hostname,  
 port=parsed.port,  
 sslmode=ssl\_mode  
 )  
 return conn  
  
  
def init\_db(db\_url=None):  
 """  
 Initialize the database schema using ORM models.  
 This should only be used for local development and tests.  
 Production systems should use Alembic migrations.  
 """  
 engine = get\_engine(db\_url)  
  
 print("⚠️ Running ORM-based init\_db() — intended for local dev & pytest only.", flush=True)  
  
 TokensBase.metadata.create\_all(bind=engine)  
 ActivitiesBase.metadata.create\_all(bind=engine)  
  
 print("✅ ORM models initialized successfully", flush=True)

### 📄 src\scripts\dev\_seed\_data.py

# src/scripts/dev\_seed\_data.py  
  
from sqlalchemy.orm import Session  
from src.db.db\_session import get\_engine  
from src.db.models.activities import Activity  
from src.db.models.splits import Split  
  
  
# Use your local Postgres DB  
DATABASE\_URL = "postgresql+psycopg2://smartcoach:devpass@localhost:15432/smartcoach"  
  
def seed\_activity\_and\_splits():  
 engine = get\_engine(DATABASE\_URL)  
 session = Session(bind=engine)  
  
 try:  
 # ✅ Insert an activity row  
 activity = Activity(  
 athlete\_id=1,  
 name="Morning Run",  
 type="Run",  
 start\_date="2024-06-01 07:00:00",  
 distance=5000,  
 elapsed\_time=1500,  
 moving\_time=1400,  
 total\_elevation\_gain=50,  
 external\_id="strava-123",  
 timezone="America/New\_York",  
 average\_speed=3.5,  
 max\_speed=4.0  
 )  
 session.add(activity)  
 session.commit()  
  
 print(f"Inserted activity\_id: {activity.activity\_id}")  
  
 # ✅ Insert a split row tied to activity  
 split = Split(  
 activity\_id=activity.activity\_id,  
 lap\_index=1,  
 distance=1000,  
 elapsed\_time=300,  
 moving\_time=290,  
 average\_speed=3.3,  
 max\_speed=4.0,  
 start\_index=0,  
 end\_index=299,  
 split=True  
 )  
 session.add(split)  
 session.commit()  
  
 print("✅ Successfully inserted activity and split.")  
  
 except Exception as e:  
 print(f"❌ Failed to insert: {e}")  
 session.rollback()  
 finally:  
 session.close()  
  
  
# ✅ This allows us to call directly for testing  
if \_\_name\_\_ == "\_\_main\_\_":  
 seed\_activity\_and\_splits()

### 📄 src\scripts\enrich\_runner.py

# src/scripts/enrich\_runner.py  
  
import argparse  
from src.db.db\_session import get\_session  
from src.services.enrichment\_sync import run\_enrichment\_batch  
  
def main(athlete\_id: int, batch\_size: int):  
 session = get\_session()  
 try:  
 count = run\_enrichment\_batch(session, athlete\_id, batch\_size)  
 print(f"✅ Enrichment complete — {count} activities processed for athlete {athlete\_id}")  
 except Exception as e:  
 print(f"❌ Enrichment failed: {e}")  
 finally:  
 session.close()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(description="Enrichment Runner")  
 parser.add\_argument("athlete\_id", type=int, help="Strava athlete ID to enrich")  
 parser.add\_argument("--batch", type=int, default=10, help="Batch size for enrichment")  
  
 args = parser.parse\_args()  
 main(args.athlete\_id, args.batch)

### 📄 src\scripts\enrichment\_pilot\_test.py

# src/scripts/enrichment\_pilot\_test.py  
  
from src.db.db\_session import get\_session  
from src.services.activity\_sync import sync\_full\_history, ensure\_fresh\_access\_token  
from src.services.enrichment\_sync import run\_enrichment\_batch  
from src.env\_loader import \* # load .env adjustments  
  
athlete\_id = 347085 # Developer test athlete  
  
session = get\_session()  
  
try:  
 # ✅ Step 1 — Fetch valid access token (auto-refresh if needed)  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
  
 # ✅ Step 2 — Sync only 5 activities (last ~180 days window)  
 synced\_count = sync\_full\_history(  
 session=session,  
 athlete\_id=athlete\_id,  
 lookback\_days=180, # last ~6 months  
 max\_activities=5  
 )  
  
 print(f"✅ Backfill sync complete — {synced\_count} activities synced.")  
  
 # ✅ Step 3 — Run enrichment on those 5 activities  
 enriched\_count = run\_enrichment\_batch(session, athlete\_id, batch\_size=5)  
 print(f"✅ Enrichment complete — {enriched\_count} activities enriched.")  
  
except Exception as e:  
 print(f"❌ Pilot test failed: {e}")  
  
finally:  
 session.close()

### 📄 src\scripts\generate\_gpt\_handoff\_summary.py

from docx import Document  
from docx.shared import Pt  
  
OUTPUT\_DOCX = "SmartCoach\_GPT\_Handoff\_Summary\_CLEAN.docx"  
  
def add\_heading(doc, text, level):  
 doc.add\_heading(text, level=level)  
  
def add\_paragraph(doc, text):  
 doc.add\_paragraph(text)  
  
def generate\_gpt\_handoff\_summary():  
 doc = Document()  
  
 add\_heading(doc, "✅ SmartCoach – GPT Handoff Summary", level=1)  
 add\_paragraph(doc, "This document was generated to bootstrap a Custom GPT session with complete project context.")  
  
 # Last Known Good State  
 add\_heading(doc, "🏁 Last known good state:", level=2)  
 add\_paragraph(doc, "• Tag: v1.0.0")  
 add\_paragraph(doc, "• Deployment: Railway (web-production-c4329.up.railway.app)")  
 add\_paragraph(doc, "• OAuth tested: ✅")  
 add\_paragraph(doc, "• DB Connected: ✅")  
  
 # Environment  
 add\_heading(doc, "🔐 Environment Settings:", level=2)  
 add\_paragraph(doc, "• ADMIN\_USER=admin")  
 add\_paragraph(doc, "• REDIRECT\_URI=https://web-production-c4329.up.railway.app/oauth/callback")  
 add\_paragraph(doc, "• DATABASE\_URL=(set in Railway)")  
 add\_paragraph(doc, "• SECRET\_KEY, INTERNAL\_API\_KEY, CRON\_SECRET\_KEY defined")  
  
 # Architecture  
 add\_heading(doc, "🔧 Architecture Notes:", level=2)  
 add\_paragraph(doc, "• Flask API deployed on Railway")  
 add\_paragraph(doc, "• PostgreSQL managed via Railway plugin")  
 add\_paragraph(doc, "• JWT-based authentication and user sessions")  
 add\_paragraph(doc, "• Strava OAuth 2.0 integration and webhook registration planned")  
  
 # Coaching Intelligence Objective  
 add\_heading(doc, "🎯 Coaching Intelligence Objectives:", level=2)  
 add\_paragraph(doc, "• Build personalized weekly training plans")  
 add\_paragraph(doc, "• Track Strava-recorded performance")  
 add\_paragraph(doc, "• Monitor deviations and adapt based on user behavior")  
 add\_paragraph(doc, "• Support natural conversation for advising and adjusting")  
  
 # System Components  
 add\_heading(doc, "🧱 System Components:", level=2)  
 add\_paragraph(doc, "• Custom GPT logic: planning, feedback, and conversation")  
 add\_paragraph(doc, "• PostgreSQL DB storing plan and activity data")  
 add\_paragraph(doc, "• Strava Sync service")  
 add\_paragraph(doc, "• Planned: Training plan generator, weekly comparator")  
  
 # Known Issues  
 add\_heading(doc, "⚠️ Known Issues:", level=2)  
 add\_paragraph(doc, "• None at last deploy. Strava activity fetch untested.")  
  
 # Tasks  
 add\_heading(doc, "📌 Next Tasks:", level=2)  
 add\_paragraph(doc, "1. Test full Strava flow: code -> token -> activity fetch")  
 add\_paragraph(doc, "2. Implement auto-sync")  
 add\_paragraph(doc, "3. Add athlete-to-user ID mapping")  
 add\_paragraph(doc, "4. Connect activity analysis")  
 add\_paragraph(doc, "5. Ship Phase 2 readiness")  
  
 # GPT Boot Prompt  
 add\_heading(doc, "🧠 GPT Boot Prompt", level=2)  
 add\_paragraph(doc, "Create a Custom GPT with the following context:")  
 add\_paragraph(doc, "- Name: SmartCoachDev")  
 add\_paragraph(doc, "- Role: Flask engineer + PM for Smart Marathon Coach API")  
 add\_paragraph(doc, "- Accesses: JWT auth, Strava OAuth, Railway deploy logs, pg DB")  
 add\_paragraph(doc, "- Code base starts in: /src, /templates, /scripts")  
 add\_paragraph(doc, "- Current live env: https://web-production-c4329.up.railway.app")  
 add\_paragraph(doc, "- Start by reviewing recent work, then continue with task #1: Test full Strava activity flow.")  
 add\_paragraph(doc, "- Reference: ‘Smart Coach - Project Reference.docx’ for architecture, history, and schema.")  
  
  
 from docx import Document as DocxDocument  
  
 def append\_docx\_content(target\_doc, source\_path):  
 source\_doc = DocxDocument(source\_path)  
 for element in source\_doc.element.body:  
 target\_doc.element.body.append(element)  
  
 # At the end of generate\_gpt\_handoff\_summary():  
 try:  
 append\_docx\_content(doc, "final\_project\_map.docx")  
 print("✅ Appended folder/code map from final\_project\_map.docx")  
 except Exception as e:  
 print(f"⚠️ Could not append code map: {e}")  
  
  
  
  
  
  
 doc.save(OUTPUT\_DOCX)  
 print(f"✅ Summary written to {OUTPUT\_DOCX}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 generate\_gpt\_handoff\_summary()

### 📄 src\scripts\helper\_extract\_token.py

# helper\_extract\_token.py  
  
from src.db.db\_session import get\_session  
from src.db.models.tokens import Token  
  
session = get\_session()  
token\_row = session.query(Token).filter\_by(athlete\_id=347085).first()  
print("Access Token:", token\_row.access\_token)  
session.close()

### 📄 src\scripts\manual\_single\_enrichment.py

import logging  
from src.db.db\_session import get\_session  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.services.enrichment\_sync import enrich\_one\_activity  
  
logging.basicConfig(level=logging.INFO)  
  
ATHLETE\_ID = 347085  
ACTIVITY\_ID = 14663194187  
  
session = get\_session()  
access\_token = ensure\_fresh\_access\_token(session, ATHLETE\_ID)  
enrich\_one\_activity(session, ATHLETE\_ID, access\_token, ACTIVITY\_ID)  
session.close()

### 📄 src\scripts\manual\_token\_exchange.py

import os  
import requests  
from src.db.db\_session import get\_session  
from src.db.dao.token\_dao import save\_tokens\_sa  
  
# Manually paste the code you just got from Strava OAuth  
authorization\_code = "424e4fcab2763945754ed35ffccd964db28d069c"  
  
# Load your client ID & secret from env (make sure these are set in .env)  
client\_id = os.getenv("STRAVA\_CLIENT\_ID")  
client\_secret = os.getenv("STRAVA\_CLIENT\_SECRET")  
redirect\_uri = os.getenv("REDIRECT\_URI")  
  
# Exchange code for tokens  
response = requests.post(  
 "https://www.strava.com/oauth/token",  
 data={  
 "client\_id": int(client\_id),  
 "client\_secret": client\_secret,  
 "code": authorization\_code,  
 "grant\_type": "authorization\_code",  
 "redirect\_uri": redirect\_uri  
 },  
 timeout=10,  
)  
  
response.raise\_for\_status()  
tokens = response.json()  
  
athlete\_id = tokens["athlete"]["id"]  
access\_token = tokens["access\_token"]  
refresh\_token = tokens["refresh\_token"]  
expires\_at = tokens["expires\_at"]  
  
print(f"✅ Got tokens for athlete {athlete\_id}")  
  
# Save to DB  
session = get\_session()  
save\_tokens\_sa(session, athlete\_id, access\_token, refresh\_token, expires\_at)  
session.close()  
  
print("✅ Tokens saved successfully to DB")

### 📄 src\scripts\map\_and\_extract\_Railway.py

import os  
from docx import Document  
from docx.shared import Pt, RGBColor  
from docx.oxml.ns import qn  
  
ROOT\_DIR = r"C:\Users\andre\projects\railway-pg-test"  
OUTPUT\_DOCX = "final\_project\_map.docx"  
  
EXCLUDE\_DIRS = {'venv', '\_\_pycache\_\_', '.git', '.mypy\_cache', '.pytest\_cache'}  
EXCLUDE\_SUFFIXES = {'.dist-info', '.egg-info'}  
EXCLUDE\_FILES = {'.env'}  
INCLUDE\_EXTENSIONS = {'.py', '.md', '.txt', '.html', '.css', '.js'} # Excludes JSON  
  
def should\_exclude\_dir(name):  
 return name in EXCLUDE\_DIRS or any(name.endswith(suffix) for suffix in EXCLUDE\_SUFFIXES)  
  
def should\_include\_file(entry):  
 ext = os.path.splitext(entry)[1]  
 return ext in INCLUDE\_EXTENSIONS and entry not in EXCLUDE\_FILES  
  
def icon\_for(entry, is\_dir):  
 if is\_dir:  
 return "📁"  
 elif entry.endswith(".py"):  
 return "🐍"  
 else:  
 return "📄"  
  
def read\_file\_content(filepath):  
 try:  
 with open(filepath, "r", encoding="utf-8") as f:  
 return f.readlines()  
 except:  
 return None  
  
def add\_code\_block(doc, lines):  
 para = doc.add\_paragraph()  
 para.paragraph\_format.line\_spacing = 1.0  
 para.paragraph\_format.space\_before = Pt(0)  
 para.paragraph\_format.space\_after = Pt(0)  
  
 for i, line in enumerate(lines):  
 run = para.add\_run(line if line.endswith('\n') else line + '\n')  
 run.font.name = 'Consolas'  
 run.font.size = Pt(9)  
 run.font.color.rgb = RGBColor(51, 51, 51) # #333333  
 run.\_element.rPr.rFonts.set(qn('w:eastAsia'), 'Consolas')  
  
def apply\_folder\_style(paragraph):  
 if not paragraph.runs:  
 return  
 run = paragraph.runs[0]  
 run.font.name = 'Consolas'  
 run.font.size = Pt(9)  
 run.font.color.rgb = RGBColor(51, 51, 51)  
 run.\_element.rPr.rFonts.set(qn('w:eastAsia'), 'Consolas')  
 paragraph.paragraph\_format.space\_before = Pt(0)  
 paragraph.paragraph\_format.space\_after = Pt(0)  
 paragraph.paragraph\_format.line\_spacing = 1.0  
  
def write\_project\_map(root\_dir, output\_docx):  
 doc = Document()  
 doc.add\_heading("📁 Folder & File Structure", level=1)  
  
 folder\_map\_lines = []  
 code\_blocks = []  
  
 def scan\_structure(path, level=0):  
 try:  
 entries = sorted(os.listdir(path))  
 except PermissionError:  
 return  
 for entry in entries:  
 if should\_exclude\_dir(entry):  
 continue  
 full\_path = os.path.join(path, entry)  
 is\_dir = os.path.isdir(full\_path)  
 indent = " " \* level  
 icon = icon\_for(entry, is\_dir)  
 line = f"{indent}{icon} {entry}/" if is\_dir else f"{indent}{icon} {entry}"  
 folder\_map\_lines.append(line)  
  
 if not is\_dir and should\_include\_file(entry):  
 content = read\_file\_content(full\_path)  
 if content:  
 rel\_path = os.path.relpath(full\_path, root\_dir)  
 code\_blocks.append((rel\_path, content))  
  
 if is\_dir:  
 scan\_structure(full\_path, level + 1)  
  
 folder\_map\_lines.append(f"{os.path.basename(ROOT\_DIR)}/")  
 scan\_structure(ROOT\_DIR, level=1)  
  
 # Folder tree  
 for line in folder\_map\_lines:  
 p = doc.add\_paragraph(line)  
 apply\_folder\_style(p)  
  
 doc.add\_paragraph() # spacer  
  
 # Code blocks  
 doc.add\_heading("🧠 Code & Content", level=1)  
 for rel\_path, lines in code\_blocks:  
 doc.add\_paragraph(f"📄 {rel\_path}", style='Heading3')  
 add\_code\_block(doc, lines)  
  
 doc.save(output\_docx)  
 print(f"✅ Saved clean layout to: {output\_docx}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 write\_project\_map(ROOT\_DIR, OUTPUT\_DOCX)

### 📄 src\scripts\onboard\_and\_sync.py

# src/scripts/onboard\_and\_sync.py  
  
import argparse  
import logging  
  
from src.db.db\_session import get\_session  
from src.db.dao.token\_dao import get\_tokens\_sa  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.services.activity\_sync import sync\_full\_history  
from src.services.enrichment import ActivityEnrichor  
from src.utils.enrichment\_debug\_wrapper import EnrichmentDebugWrapper  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
def onboard\_and\_sync(athlete\_id, lookback\_days=30, batch\_size=10):  
 """  
 Full sync pipeline for onboarding + enrichment.  
 """  
  
 session = get\_session()  
  
 try:  
 logger.info(f"🚀 Starting full onboard\_and\_sync for athlete {athlete\_id}")  
  
 tokens = get\_tokens\_sa(session, athlete\_id)  
  
 if not tokens:  
 raise RuntimeError(  
 f"No tokens found for athlete {athlete\_id}. "  
 "Please complete OAuth authorization first via /oauth/callback."  
 )  
  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
 logger.info(f"🟢 Retrieved valid access token for athlete {athlete\_id}")  
  
 # Sync activities  
 synced\_count = sync\_full\_history(  
 session,  
 athlete\_id,  
 access\_token,  
 lookback\_days=lookback\_days  
 )  
 logger.info(f"✅ Synced {synced\_count} activities.")  
  
 # Build enrichment pipeline  
 enrichor = ActivityEnrichor()  
  
 # Wrap enrichment with debug instrumentation  
 debug\_wrapper = EnrichmentDebugWrapper(enrichor, session)  
 enriched\_count = debug\_wrapper.enrich(athlete\_id, batch\_size=batch\_size)  
 logger.info(f"✅ Enriched {enriched\_count} activities (with debug tracing).")  
  
 logger.info(f"🎯 Onboard and sync complete for athlete {athlete\_id}")  
  
 except Exception as e:  
 logger.exception(f"❌ Failed during onboard\_and\_sync: {e}")  
  
 finally:  
 session.close()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(description="Orchestrate full sync + enrichment for existing athlete")  
 parser.add\_argument("--athlete\_id", required=True, type=int)  
 parser.add\_argument("--lookback\_days", type=int, default=30)  
 parser.add\_argument("--batch\_size", type=int, default=10)  
  
 args = parser.parse\_args()  
 onboard\_and\_sync(args.athlete\_id, args.lookback\_days, args.batch\_size)

### 📄 src\scripts\sync\_and\_enrich.py

# src/scripts/sync\_and\_enrich.py  
  
import argparse  
from src.db.db\_session import get\_session  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.services.activity\_sync import sync\_full\_history  
from src.services.enrichment\_sync import run\_enrichment\_batch  
  
def main(athlete\_id: int, lookback\_days: int, batch\_size: int):  
 session = get\_session()  
  
 try:  
 # Step 1: Ensure valid token  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
  
 # Step 2: Sync activities  
 synced\_count = sync\_full\_history(  
 session=session,  
 athlete\_id=athlete\_id,  
 access\_token=access\_token,  
 lookback\_days=lookback\_days  
 )  
 print(f"✅ Sync complete — {synced\_count} activities synced.")  
  
 # Step 3: Enrich activities  
 enriched\_count = run\_enrichment\_batch(  
 session=session,  
 athlete\_id=athlete\_id,  
 batch\_size=batch\_size  
 )  
 print(f"✅ Enrichment complete — {enriched\_count} activities enriched.")  
  
 except Exception as e:  
 print(f"❌ Failed: {e}")  
  
 finally:  
 session.close()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 parser = argparse.ArgumentParser(description="Sync and Enrich Activities for an Athlete")  
 parser.add\_argument("--athlete\_id", type=int, required=True, help="Strava athlete ID")  
 parser.add\_argument("--lookback\_days", type=int, default=30, help="Lookback window for sync (days)")  
 parser.add\_argument("--batch\_size", type=int, default=20, help="Batch size for enrichment")  
  
 args = parser.parse\_args()  
 main(args.athlete\_id, args.lookback\_days, args.batch\_size)

### 📄 src\scripts\test\_single\_enrichment.py

# src/scripts/test\_single\_enrichment.py  
  
import os  
import requests  
  
# Hardcode the activity id we want to test  
activity\_id = 14663194187  
  
# Paste your valid access token here:  
access\_token = "f0a988311cc64557330577b7a9c461097941b1c2"  
  
url = f"https://www.strava.com/api/v3/activities/{activity\_id}/zones"  
headers = {"Authorization": f"Bearer {access\_token}"}  
  
response = requests.get(url, headers=headers, timeout=10)  
  
print(f"HTTP Status: {response.status\_code}")  
if response.status\_code != 200:  
 print("❌ Failed to fetch HR zones.")  
 print(response.text)  
else:  
 zones\_data = response.json()  
 print("✅ Successfully fetched HR zone data:")  
 print(zones\_data)  
   
 # Optional: extract HR zones cleanly  
 for zone\_group in zones\_data:  
 if zone\_group.get("type") == "heartrate":  
 hr\_zones = zone\_group.get("zones", [])  
 total\_time = sum(z.get("time", 0) for z in hr\_zones)  
 print(f"Total HR recorded time: {total\_time} sec")  
 for i, z in enumerate(hr\_zones):  
 pct = round((z.get("time", 0) / total\_time) \* 100, 1) if total\_time > 0 else 0  
 print(f"Zone {i+1}: {pct}%")

### 📄 src\services\activity\_sync.py

# src/services/activity\_sync.py  
  
from datetime import datetime, timedelta  
from src.utils.logger import get\_logger  
from src.services.strava import fetch\_activities\_between  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.db.dao.activity\_dao import upsert\_activities  
from src.db.dao.split\_dao import upsert\_splits  
from src.services.split\_extraction import extract\_splits  
  
log = get\_logger(\_\_name\_\_)  
  
def sync\_recent(session, athlete\_id: int, access\_token: str = None, per\_page=200, max\_activities=None) -> int:  
 """  
 Sync recent activities for a given athlete.  
 Automatically refreshes tokens if access\_token is not provided or expired.  
 """  
 try:  
 # 🔑 If access\_token not provided, refresh it automatically  
 if not access\_token:  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
  
 end\_date = datetime.utcnow()  
 start\_date = end\_date - timedelta(days=30)  
  
 return sync\_activities\_between(  
 session, athlete\_id, access\_token, start\_date, end\_date, per\_page, max\_activities  
 )  
  
 except Exception as e:  
 raise RuntimeError(f"Failed to sync recent activities: {e}")  
  
  
def sync\_full\_history(session, athlete\_id: int, access\_token: str = None, lookback\_days: int = 365, per\_page=200, max\_activities=None) -> int:  
 """  
 Sync full historical activities, auto-refreshing tokens if needed.  
 """  
 try:  
 if not access\_token:  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
  
 end\_date = datetime.utcnow()  
 start\_date = end\_date - timedelta(days=lookback\_days)  
  
 return sync\_activities\_between(  
 session, athlete\_id, access\_token, start\_date, end\_date, per\_page, max\_activities  
 )  
  
 except Exception as e:  
 raise RuntimeError(f"Failed to sync full history for athlete {athlete\_id}: {e}")  
  
  
def sync\_activities\_between(session, athlete\_id: int, access\_token: str, start\_date: datetime, end\_date: datetime, per\_page=200, max\_activities=None) -> int:  
 """  
 Core ingestion logic (unchanged).  
 """  
 try:  
 all\_activities = []  
 page = 1  
  
 while True:  
 activities = fetch\_activities\_between(access\_token, start\_date, end\_date, per\_page=per\_page)  
 if not activities:  
 break  
  
 all\_activities.extend(activities)  
  
 if max\_activities and len(all\_activities) >= max\_activities:  
 all\_activities = all\_activities[:max\_activities]  
 break  
  
 if len(activities) < per\_page:  
 break  
  
 page += 1  
  
 if not all\_activities:  
 log.info(f"No activities found for athlete {athlete\_id} between {start\_date.date()} and {end\_date.date()}")  
 return 0  
  
 count = upsert\_activities(session, athlete\_id, all\_activities)  
 log.info(f"Synced {count} activities for athlete {athlete\_id} between {start\_date.date()} and {end\_date.date()}")  
  
 all\_splits = []  
 for activity in all\_activities:  
 splits = extract\_splits(activity)  
 all\_splits.extend(splits)  
  
 if all\_splits:  
 upsert\_splits(session, all\_splits)  
 log.info(f"Synced {len(all\_splits)} splits for athlete {athlete\_id}")  
  
 return count  
  
 except Exception as e:  
 raise RuntimeError(f"Failed to fetch or persist activities: {e}")

### 📄 src\services\auth.py

# src/services/auth.py  
  
import os  
import jwt  
import datetime  
from flask import current\_app, has\_app\_context  
  
from src.db.db\_session import get\_engine, get\_session  
from src.db.dao.token\_dao import get\_tokens\_sa, save\_tokens\_sa  
  
ACCESS\_TOKEN\_EXP = lambda: int(os.getenv("ACCESS\_TOKEN\_EXP", 900)) # 15 minutes  
REFRESH\_TOKEN\_EXP = lambda: int(os.getenv("REFRESH\_TOKEN\_EXP", 604800)) # 7 days  
  
def resolve\_db\_url():  
 if has\_app\_context():  
 return current\_app.config.get("DATABASE\_URL", os.getenv("DATABASE\_URL"))  
 return os.getenv("DATABASE\_URL")  
  
def login\_user(data: dict) -> tuple[str, str]:  
 username = data.get("username")  
 password = data.get("password")  
  
 if username != os.getenv("ADMIN\_USER") or password != os.getenv("ADMIN\_PASS"):  
 raise PermissionError("Invalid credentials")  
  
 now = datetime.datetime.utcnow()  
 secret = os.getenv("SECRET\_KEY", "dev")  
  
 access\_payload = {  
 "sub": username,  
 "exp": now + datetime.timedelta(seconds=ACCESS\_TOKEN\_EXP()),  
 }  
 refresh\_payload = {  
 "sub": username,  
 "exp": now + datetime.timedelta(seconds=REFRESH\_TOKEN\_EXP()),  
 }  
  
 access\_token = jwt.encode(access\_payload, secret, algorithm="HS256")  
 refresh\_token = jwt.encode(refresh\_payload, secret, algorithm="HS256")  
  
 db\_url = resolve\_db\_url()  
 engine = get\_engine() # ✅ simplified: get\_engine() resolves internally  
 session = get\_session()  
 save\_tokens\_sa(session, athlete\_id=0, access\_token=access\_token, refresh\_token=refresh\_token)  
  
 return access\_token, refresh\_token  
  
def refresh\_token(refresh\_token\_str: str) -> str:  
 secret = os.getenv("SECRET\_KEY", "dev")  
 try:  
 payload = jwt.decode(refresh\_token\_str, secret, algorithms=["HS256"])  
 except jwt.ExpiredSignatureError:  
 raise PermissionError("Refresh token expired")  
 except jwt.InvalidTokenError:  
 raise PermissionError("Invalid refresh token")  
  
 username = payload.get("sub")  
 engine = get\_engine()  
 session = get\_session()  
 tokens = get\_tokens\_sa(session, athlete\_id=0)  
  
 if not tokens or tokens.get("refresh\_token") != refresh\_token\_str:  
 raise PermissionError("Refresh token not recognized")  
  
 now = datetime.datetime.utcnow()  
 new\_payload = {  
 "sub": username,  
 "exp": now + datetime.timedelta(seconds=ACCESS\_TOKEN\_EXP()),  
 }  
  
 return jwt.encode(new\_payload, secret, algorithm="HS256")  
  
def logout\_user(refresh\_token\_str: str) -> None:  
 # No-op  
 pass

### 📄 src\services\enrichment\_sync.py

# src/services/enrichment\_sync.py  
  
import time  
import logging  
import requests  
from sqlalchemy import text  
from sqlalchemy.exc import SQLAlchemyError  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.services.split\_extraction import extract\_splits  
from src.db.dao.split\_dao import upsert\_splits  
  
log = logging.getLogger("enrichment\_sync")  
log.setLevel(logging.INFO)  
  
STRAVA\_URL = "https://www.strava.com/api/v3/activities/{activity\_id}?include\_all\_efforts=true"  
STRAVA\_ZONE\_URL = "https://www.strava.com/api/v3/activities/{activity\_id}/zones"  
  
DEFAULT\_BATCH\_SIZE = 20  
DEFAULT\_RETRY\_LIMIT = 5  
DEFAULT\_SLEEP = 5  
DEFAULT\_RETRY\_BACKOFF = 2  
  
# --- Unit conversion helpers ---  
def meters\_to\_miles(meters):  
 return round(meters / 1609.344, 2) if meters else None  
  
def meters\_to\_feet(meters):  
 return round(meters \* 3.28084, 1) if meters else None  
  
def mps\_to\_min\_per\_mile(mps):  
 return round(26.8224 / mps, 2) if mps and mps > 0 else None  
  
def format\_seconds\_to\_hms(seconds):  
 if seconds is None:  
 return None  
 minutes, sec = divmod(seconds, 60)  
 hours, minutes = divmod(minutes, 60)  
 if hours > 0:  
 return f"{hours}:{minutes:02}:{sec:02}"  
 else:  
 return f"{minutes}:{sec:02}"  
  
# ------------------------------------------------  
  
def get\_activities\_to\_enrich(session, athlete\_id, limit):  
 query = text("""  
 SELECT activity\_id FROM activities  
 WHERE athlete\_id = :athlete\_id  
 ORDER BY start\_date DESC  
 LIMIT :limit  
 """)  
 result = session.execute(query, {"athlete\_id": athlete\_id, "limit": limit})  
 return [row.activity\_id for row in result.fetchall()]  
  
def enrich\_one\_activity(session, athlete\_id, access\_token, activity\_id):  
 try:  
 url = STRAVA\_URL.format(activity\_id=activity\_id)  
 headers = {"Authorization": f"Bearer {access\_token}"}  
 resp = requests.get(url, headers=headers, timeout=10)  
  
 if resp.status\_code == 200:  
 activity\_json = resp.json()  
  
 hr\_zone\_pcts = fetch\_hr\_zone\_percentages(activity\_id, access\_token)  
 if not hr\_zone\_pcts:  
 hr\_zone\_pcts = [0.0, 0.0, 0.0, 0.0, 0.0]  
  
 update\_activity\_enrichment(session, activity\_id, activity\_json, hr\_zone\_pcts)  
  
 splits = extract\_splits(activity\_json)  
 if splits:  
 upsert\_splits(session, splits)  
 log.info(f"✅ Synced {len(splits)} splits for activity {activity\_id}")  
  
 log.info(f"✅ Enriched activity {activity\_id}")  
 return True  
  
 elif resp.status\_code == 429:  
 retry\_after = int(resp.headers.get("Retry-After", DEFAULT\_SLEEP))  
 log.warning(f"⚠️ 429 Rate Limited. Retry-After: {retry\_after}s")  
 time.sleep(retry\_after)  
 return False # signal to caller to retry  
  
 else:  
 log.error(f"❌ Failed to enrich {activity\_id} — HTTP {resp.status\_code}")  
 return True # fail fast and skip to next  
  
 except Exception as e:  
 log.error(f"🔥 Exception while enriching {activity\_id}: {e}")  
 return True  
  
def enrich\_one\_activity\_with\_refresh(session, athlete\_id, activity\_id):  
 try:  
 access\_token = ensure\_fresh\_access\_token(session, athlete\_id)  
 return enrich\_one\_activity(session, athlete\_id, access\_token, activity\_id)  
 except Exception as e:  
 log.error(f"Failed enrichment for activity {activity\_id}: {e}")  
 return True  
  
def update\_activity\_enrichment(session, activity\_id, activity\_json, hr\_zone\_pcts):  
 distance\_meters = activity\_json.get("distance")  
 elevation\_meters = activity\_json.get("total\_elevation\_gain")  
 avg\_speed\_mps = activity\_json.get("average\_speed")  
 max\_speed\_mps = activity\_json.get("max\_speed")  
 moving\_time\_sec = activity\_json.get("moving\_time")  
 elapsed\_time\_sec = activity\_json.get("elapsed\_time")  
  
 conv\_distance\_miles = meters\_to\_miles(distance\_meters)  
 conv\_elevation\_feet = meters\_to\_feet(elevation\_meters)  
 conv\_avg\_speed = mps\_to\_min\_per\_mile(avg\_speed\_mps)  
 conv\_max\_speed = mps\_to\_min\_per\_mile(max\_speed\_mps)  
 conv\_moving\_time = format\_seconds\_to\_hms(moving\_time\_sec)  
 conv\_elapsed\_time = format\_seconds\_to\_hms(elapsed\_time\_sec)  
  
 params = {  
 "activity\_id": activity\_id,  
 "name": activity\_json.get("name"),  
 "distance": distance\_meters,  
 "moving\_time": moving\_time\_sec,  
 "elapsed\_time": elapsed\_time\_sec,  
 "elevation": elevation\_meters,  
 "type": activity\_json.get("type"),  
 "avg\_speed": avg\_speed\_mps,  
 "max\_speed": max\_speed\_mps,  
 "suffer\_score": activity\_json.get("suffer\_score"),  
 "average\_heartrate": activity\_json.get("average\_heartrate"),  
 "max\_heartrate": activity\_json.get("max\_heartrate"),  
 "calories": activity\_json.get("calories"),  
 "conv\_distance": conv\_distance\_miles,  
 "conv\_elevation\_feet": conv\_elevation\_feet,  
 "conv\_avg\_speed": conv\_avg\_speed,  
 "conv\_max\_speed": conv\_max\_speed,  
 "conv\_moving\_time": conv\_moving\_time,  
 "conv\_elapsed\_time": conv\_elapsed\_time,  
 "hr\_zone\_1\_pct": hr\_zone\_pcts[0],  
 "hr\_zone\_2\_pct": hr\_zone\_pcts[1],  
 "hr\_zone\_3\_pct": hr\_zone\_pcts[2],  
 "hr\_zone\_4\_pct": hr\_zone\_pcts[3],  
 "hr\_zone\_5\_pct": hr\_zone\_pcts[4],  
 }  
  
 session.execute(  
 text("""  
 UPDATE activities SET  
 name = :name,  
 distance = :distance,  
 moving\_time = :moving\_time,  
 elapsed\_time = :elapsed\_time,  
 total\_elevation\_gain = :elevation,  
 type = :type,  
 average\_speed = :avg\_speed,  
 max\_speed = :max\_speed,  
 suffer\_score = :suffer\_score,  
 average\_heartrate = :average\_heartrate,  
 max\_heartrate = :max\_heartrate,  
 calories = :calories,  
 conv\_distance = :conv\_distance,  
 conv\_elevation\_feet = :conv\_elevation\_feet,  
 conv\_avg\_speed = :conv\_avg\_speed,  
 conv\_max\_speed = :conv\_max\_speed,  
 conv\_moving\_time = :conv\_moving\_time,  
 conv\_elapsed\_time = :conv\_elapsed\_time,  
 hr\_zone\_1\_pct = :hr\_zone\_1\_pct,  
 hr\_zone\_2\_pct = :hr\_zone\_2\_pct,  
 hr\_zone\_3\_pct = :hr\_zone\_3\_pct,  
 hr\_zone\_4\_pct = :hr\_zone\_4\_pct,  
 hr\_zone\_5\_pct = :hr\_zone\_5\_pct  
 WHERE activity\_id = :activity\_id  
 """),  
 params  
 )  
 session.commit()  
  
def fetch\_hr\_zone\_percentages(activity\_id, access\_token):  
 url = STRAVA\_ZONE\_URL.format(activity\_id=activity\_id)  
 headers = {"Authorization": f"Bearer {access\_token}"}  
 resp = requests.get(url, headers=headers, timeout=10)  
  
 if resp.status\_code != 200:  
 log.warning(f"HR zones not available for activity {activity\_id}")  
 return None  
  
 zones\_data = resp.json()  
  
 for zone\_group in zones\_data:  
 if zone\_group.get("type") == "heartrate":  
 hr\_zones = zone\_group.get("distribution\_buckets", [])  
 times = [z.get("time") or 0.0 for z in hr\_zones]  
 total\_time = sum(times)  
  
 if total\_time == 0:  
 log.warning(f"No HR data for activity {activity\_id}")  
 return None  
  
 zone\_pcts = [  
 round(((z.get("time") or 0.0) / total\_time) \* 100, 1)  
 for z in hr\_zones  
 ]  
  
 zone\_pcts = zone\_pcts[:5]  
 while len(zone\_pcts) < 5:  
 zone\_pcts.append(0.0)  
  
 return zone\_pcts  
  
 return None  
  
def run\_enrichment\_batch(session, athlete\_id, batch\_size=DEFAULT\_BATCH\_SIZE):  
 try:  
 activities = get\_activities\_to\_enrich(session, athlete\_id, batch\_size)  
 log.info(f"🔀 Enriching {len(activities)} activities for athlete {athlete\_id}")  
  
 for activity\_id in activities:  
 retries = 0  
 while retries < DEFAULT\_RETRY\_LIMIT:  
 success = enrich\_one\_activity\_with\_refresh(session, athlete\_id, activity\_id)  
 if success:  
 break  
 retries += 1  
 log.warning(f"🔁 Retrying activity {activity\_id} (attempt {retries})")  
 time.sleep(DEFAULT\_SLEEP \* (DEFAULT\_RETRY\_BACKOFF \*\* retries))  
 return len(activities)  
  
 except SQLAlchemyError as db\_err:  
 log.error(f"DB error during enrichment: {db\_err}")  
 session.rollback()  
 return 0  
  
 except Exception as e:  
 log.error(f"Unexpected enrichment failure: {e}")  
 return 0

### 📄 src\services\split\_extraction.py

# src/services/split\_extraction.py  
  
from src.services.enrichment\_sync import (  
 meters\_to\_miles,   
 mps\_to\_min\_per\_mile,   
 format\_seconds\_to\_hms  
)  
  
def extract\_splits(activity):  
 """  
 Parse splits from Strava activity object.  
  
 Accepts:  
 - activity (dict): full activity object  
  
 Returns list of dicts ready for DAO upsert.  
 """  
 activity\_id = activity.get("id")  
 laps = activity.get("splits\_standard") # <-- Using richer splits version  
  
 if not laps:  
 return []  
  
 splits = []  
 for lap in laps:  
 split\_value = lap.get("split", True)  
 split\_bool = bool(split\_value) if split\_value is not None else True  
  
 split\_data = {  
 "activity\_id": activity\_id,  
 "lap\_index": lap.get("lap\_index") or lap.get("split"), # allow both keys for safety  
 "distance": lap.get("distance"),  
 "elapsed\_time": lap.get("elapsed\_time"),  
 "moving\_time": lap.get("moving\_time"),  
 "average\_speed": lap.get("average\_speed"),  
 "max\_speed": lap.get("max\_speed"),  
 "start\_index": lap.get("start\_index"),  
 "end\_index": lap.get("end\_index"),  
 "split": split\_bool,  
  
 "average\_heartrate": lap.get("average\_heartrate"),  
 "pace\_zone": lap.get("pace\_zone"),  
  
 # ✅ New converted fields:  
 "conv\_distance": meters\_to\_miles(lap.get("distance")),  
 "conv\_avg\_speed": mps\_to\_min\_per\_mile(lap.get("average\_speed")),  
 "conv\_moving\_time": format\_seconds\_to\_hms(lap.get("moving\_time")),  
 "conv\_elapsed\_time": format\_seconds\_to\_hms(lap.get("elapsed\_time")),  
 }  
 splits.append(split\_data)  
  
 return splits

### 📄 src\services\strava.py

import os  
import time  
import requests  
from datetime import datetime  
from urllib.parse import urlencode  
from src.db.dao.token\_dao import get\_tokens\_sa, save\_tokens\_sa  
  
  
def enrich\_activity(activity\_id, key=None):  
 """  
 Stub for enriching a single Strava activity.  
 Accepts activity ID and optional secret key; returns enrichment result dict.  
 """  
 raise NotImplementedError("enrich\_activity not implemented")  
  
  
def backfill\_activities(since=None):  
 """  
 Stub for backfilling multiple activities since a given date.  
 Returns count of activities processed.  
 """  
 raise NotImplementedError("backfill\_activities not implemented")  
  
  
def fetch\_activities\_between(access\_token, start\_date, end\_date, per\_page=200):  
 """  
 Fetch all Strava activities for an athlete within a date range.  
 Handles pagination. Raises RuntimeError on 401.  
 """  
 url = "https://www.strava.com/api/v3/athlete/activities"  
 headers = {"Authorization": f"Bearer {access\_token}"}  
 params = {  
 "after": int(start\_date.timestamp()),  
 "before": int(end\_date.timestamp()),  
 "per\_page": per\_page,  
 }  
  
 all\_activities = []  
 page = 1  
  
 while True:  
 params["page"] = page  
 response = requests.get(url, headers=headers, params=params)  
  
 if response.status\_code == 401:  
 print("❌ 401 Unauthorized – Strava rejected the token.")  
 print("🔐 Access token used:", access\_token)  
 print("📝 Response body:", response.text)  
 raise RuntimeError("Access token unauthorized or expired.")  
  
 elif response.status\_code != 200:  
 print(f"❌ Unexpected Strava error {response.status\_code}")  
 print("📝 Response body:", response.text)  
 raise RuntimeError(f"Strava API error {response.status\_code}")  
  
 batch = response.json()  
 if not batch:  
 break  
  
 all\_activities.extend(batch)  
 page += 1  
  
 return all\_activities  
  
  
def generate\_strava\_auth\_url(athlete\_id=None):  
 """  
 Generate an authorization URL for Strava OAuth with optional state.  
 """  
 client\_id = os.getenv("STRAVA\_CLIENT\_ID")  
 redirect\_uri = os.getenv("REDIRECT\_URI")  
 if not redirect\_uri:  
 raise RuntimeError("Missing REDIRECT\_URI in environment.")  
  
 scope = "read,activity:read\_all"  
 params = {  
 "client\_id": client\_id,  
 "redirect\_uri": redirect\_uri,  
 "response\_type": "code",  
 "approval\_prompt": "force",  
 "scope": scope,  
 }  
 if athlete\_id:  
 params["state"] = str(athlete\_id)  
  
 return f"https://www.strava.com/oauth/authorize?{urlencode(params)}"  
  
  
def refresh\_strava\_token(refresh\_token):  
 """  
 Refresh Strava access token using refresh\_token.  
 """  
 client\_id = os.getenv("STRAVA\_CLIENT\_ID")  
 client\_secret = os.getenv("STRAVA\_CLIENT\_SECRET")  
  
 response = requests.post(  
 "https://www.strava.com/api/v3/oauth/token",  
 data={  
 "client\_id": int(client\_id),  
 "client\_secret": client\_secret,  
 "grant\_type": "refresh\_token",  
 "refresh\_token": refresh\_token  
 },  
 timeout=10  
 )  
  
 response.raise\_for\_status()  
 tokens = response.json()  
  
 # ✅ FIXED: athlete is not returned on token refresh  
 return {  
 "access\_token": tokens["access\_token"],  
 "refresh\_token": tokens["refresh\_token"],  
 "expires\_at": tokens["expires\_at"],  
 "athlete\_id": tokens.get("athlete", {}).get("id", None)  
 }

### 📄 src\services\token\_refresh.py

# src/services/token\_refresh.py  
  
from datetime import datetime  
from src.db.dao.token\_dao import get\_tokens\_sa, save\_tokens\_sa  
from src.services.strava import refresh\_strava\_token  
from src.db.models.tokens import Token  
  
def ensure\_fresh\_access\_token(session, athlete\_id: int) -> str:  
 """  
 Centralized token refresh handler. Returns valid access token.  
 """  
 tokens = get\_tokens\_sa(session, athlete\_id)  
 if not tokens:  
 raise RuntimeError(f"No tokens found for athlete {athlete\_id}")  
  
 now\_ts = int(datetime.utcnow().timestamp())  
 token\_record = session.query(Token).filter\_by(athlete\_id=athlete\_id).one()  
  
 if token\_record.expires\_at > now\_ts:  
 return tokens["access\_token"]  
  
 # Token expired — refresh and persist updated tokens  
 refreshed = refresh\_strava\_token(tokens["refresh\_token"])  
 save\_tokens\_sa(  
 session,  
 athlete\_id,  
 refreshed["access\_token"],  
 refreshed["refresh\_token"],  
 refreshed["expires\_at"]  
 )  
 return refreshed["access\_token"]

### 📄 src\utils\enrichment\_debug\_wrapper.py

# src/utils/enrichment\_debug\_wrapper.py  
  
import logging  
from sqlalchemy.orm import Session  
  
logger = logging.getLogger(\_\_name\_\_)  
logging.basicConfig(level=logging.INFO)  
  
  
class EnrichmentDebugWrapper:  
 def \_\_init\_\_(self, enrichor, session: Session):  
 self.enrichor = enrichor  
 self.session = session  
  
 def enrich(self, athlete\_id, activities=None, batch\_size=None):  
 logger.info("🟡 Starting enrichment with debug wrapper")  
  
 # Dynamically handle enrichment method signature  
 if activities is not None:  
 enriched\_count = self.enrichor.enrich(athlete\_id=athlete\_id, activities=activities, session=self.session)  
 else:  
 enriched\_count = self.enrichor.enrich(athlete\_id=athlete\_id, batch\_size=batch\_size, session=self.session)  
  
 # ORM state introspection after enrichment but before commit  
 new\_objects = list(self.session.new)  
 dirty\_objects = list(self.session.dirty)  
 deleted\_objects = list(self.session.deleted)  
  
 logger.info("🟡 ORM session state after enrichment:")  
 logger.info(f" New objects pending commit: {len(new\_objects)}")  
 logger.info(f" Dirty objects pending commit: {len(dirty\_objects)}")  
 logger.info(f" Deleted objects pending commit: {len(deleted\_objects)}")  
  
 for obj in new\_objects:  
 logger.info(f" New: {obj}")  
  
 # Specifically track Split objects  
 splits = [obj for obj in new\_objects if obj.\_\_class\_\_.\_\_name\_\_.lower() == 'split']  
 logger.info(f" Detected {len(splits)} Split objects staged for commit")  
  
 # Commit changes  
 logger.info("🟡 Performing commit...")  
 self.session.commit()  
 logger.info("🟢 Commit successful")  
  
 return enriched\_count

### 📄 src\utils\gpt\_ops.py

# src/utils/gpt\_ops.py  
  
import os  
from flask import Flask  
from src.db.base\_model import get\_session  
from src.db.dao.task\_dao import create\_task, update\_task\_status, delete\_task  
  
# Optional: load app context if needed  
  
def init\_flask\_context():  
 from src.app import create\_app  
 app = create\_app({"TESTING": True})  
 ctx = app.app\_context()  
 ctx.push()  
  
# Check if GPT is allowed to modify DB  
if os.getenv("GPT\_CAN\_MUTATE\_DB", "false").lower() == "true":  
 init\_flask\_context()  
else:  
 raise RuntimeError("GPT task creation is disabled by config. Set GPT\_CAN\_MUTATE\_DB=true in your .env.")  
  
  
def create\_task\_from\_gpt(title, user\_id=1, status="pending", milestone=None, labels=None, is\_icebox=False, details=None):  
 session = get\_session()  
 try:  
 task\_id = create\_task(  
 session,  
 user\_id=user\_id,  
 title=title,  
 status=status,  
 milestone=milestone,  
 labels=labels,  
 is\_icebox=is\_icebox,  
 details=details,  
 )  
 print(f"✅ GPT created task #{task\_id}: {title}")  
 return task\_id  
 except Exception as e:  
 print(f"❌ GPT failed to create task: {e}")  
 finally:  
 session.close()  
  
  
def update\_task\_status\_from\_gpt(task\_id, status=None, labels=None, is\_icebox=None, details=None):  
 session = get\_session()  
 try:  
 update\_task\_status(  
 session,  
 task\_id=task\_id,  
 status=status,  
 labels=labels,  
 is\_icebox=is\_icebox,  
 details=details,  
 )  
 print(f"✅ GPT updated task #{task\_id}")  
 except Exception as e:  
 print(f"❌ GPT failed to update task #{task\_id}: {e}")  
 finally:  
 session.close()  
  
  
def delete\_task\_by\_id\_from\_gpt(task\_id):  
 session = get\_session()  
 try:  
 delete\_task(session, task\_id)  
 print(f"🗑️ GPT deleted task #{task\_id}")  
 except Exception as e:  
 print(f"❌ GPT failed to delete task #{task\_id}: {e}")  
 finally:  
 session.close()

### 📄 src\utils\jwt\_utils.py

import jwt  
from functools import wraps  
from flask import request, jsonify, current\_app  
  
def require\_auth(f):  
 @wraps(f)  
 def decorated(\*args, \*\*kwargs):  
 # ✅ Internal service key override  
 internal\_key = request.headers.get("X-Internal-Key")  
 expected\_key = current\_app.config.get("INTERNAL\_API\_KEY")  
  
 if internal\_key and expected\_key and internal\_key == expected\_key:  
 request.user = {  
 "user\_id": "internal",  
 "is\_internal": True # ✅ FIX: enable admin privileges  
 }  
 return f(\*args, \*\*kwargs)  
  
 # 🔐 Fallback to regular Bearer token auth  
 auth\_header = request.headers.get("Authorization")  
 if not auth\_header or not auth\_header.lower().startswith("bearer "):  
 return jsonify({"error": "Authorization header missing"}), 401  
  
 token = auth\_header.split(" ")[1]  
 try:  
 secret = current\_app.config["SECRET\_KEY"]  
 payload = jwt.decode(token, secret, algorithms=["HS256"])  
  
 user\_id = payload.get("sub")  
 if not user\_id:  
 return jsonify({"error": "Token missing subject (sub)"}), 401  
  
 request.user = {  
 "user\_id": user\_id,  
 "is\_internal": user\_id == "internal" # ✅ Extend to support test tokens  
 }  
 except jwt.ExpiredSignatureError:  
 return jsonify({"error": "Token expired"}), 401  
 except jwt.InvalidTokenError:  
 return jsonify({"error": "Invalid token"}), 401  
  
 return f(\*args, \*\*kwargs)  
 return decorated  
  
  
def decode\_token(token: str, secret: str) -> dict:  
 """Decode JWT without expiration check (for internal inspection)."""  
 try:  
 return jwt.decode(token, secret, algorithms=["HS256"], options={"verify\_exp": False})  
 except jwt.DecodeError:  
 raise ValueError("Invalid token format")

### 📄 src\utils\logger.py

import logging  
  
  
def get\_logger(name=\_\_name\_\_):  
 logging.basicConfig(  
 level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s"  
 )  
 return logging.getLogger(name)

### 📄 test\_enrichment\_sync.py

from src.db.db\_session import get\_session  
from src.services.enrichment\_sync import run\_enrichment\_batch  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 athlete\_id = 12345 # <-- replace with a valid athlete\_id from your database  
  
 session = get\_session()  
 try:  
 count = run\_enrichment\_batch(session, athlete\_id=athlete\_id)  
 print(f"✅ Enriched {count} activities")  
 finally:  
 session.close()

### 📄 tests\conftest.py

# tests/conftest.py  
  
import os  
import sys  
import pytest  
from pathlib import Path  
from sqlalchemy.orm import sessionmaker  
from dotenv import load\_dotenv  
from unittest.mock import patch  
  
# ✅ Load environment variables  
load\_dotenv()  
  
# ✅ Ensure project root is on sys.path  
sys.path.append(str(Path(\_\_file\_\_).resolve().parent.parent))  
  
from src.app import create\_app  
from src.db.db\_session import get\_engine  
from src.scripts.dev\_only\_init\_db import init\_db  
  
# ✅ DATABASE\_URL for test Postgres instance  
TEST\_DATABASE\_URL = "postgresql+psycopg2://smartcoach:devpass@localhost:15432/smartcoach"  
  
# ✅ Create shared test database engine  
@pytest.fixture(scope="session")  
def shared\_engine():  
 os.environ["DATABASE\_URL"] = TEST\_DATABASE\_URL  
 engine = get\_engine(TEST\_DATABASE\_URL)  
 init\_db(TEST\_DATABASE\_URL)  
 return engine  
  
# ✅ Standard app fixture (no mocking)  
@pytest.fixture(scope="function")  
def app(shared\_engine):  
 app = create\_app({"TESTING": True, "DATABASE\_URL": TEST\_DATABASE\_URL})  
 yield app  
  
# ✅ Standard client fixture  
@pytest.fixture(scope="function")  
def client(app):  
 return app.test\_client()  
  
# ✅ SQLAlchemy session fixture  
@pytest.fixture(scope="function")  
def sqlalchemy\_session(shared\_engine):  
 connection = shared\_engine.connect()  
 transaction = connection.begin()  
 Session = sessionmaker(bind=connection, future=True)  
 session = Session()  
  
 yield session  
  
 session.close()  
 transaction.rollback()  
 connection.close()  
  
# ✅ Patched app fixture for mocking sync\_recent  
@pytest.fixture(scope="function")  
def patched\_app(monkeypatch):  
 monkeypatch.setenv("CRON\_SECRET\_KEY", "devkey123")  
  
 with patch("src.routes.sync\_routes.sync\_recent") as mock\_sync\_recent:  
 mock\_sync\_recent.return\_value = 10  
  
 app = create\_app({"TESTING": True, "DATABASE\_URL": TEST\_DATABASE\_URL})  
 yield app  
  
# ✅ Patched client fixture built on patched app  
@pytest.fixture(scope="function")  
def patched\_client(patched\_app):  
 return patched\_app.test\_client()

### 📄 tests\test\_activity\_sync\_flow.py

import pytest  
from unittest.mock import patch  
from datetime import datetime, timedelta  
  
from src.db.models.activities import Activity  
from src.db.models.splits import Split  
from src.services.activity\_sync import sync\_recent  
  
  
def mock\_fetch\_activities\_between(access\_token, start\_date, end\_date, per\_page=200):  
 # Return one activity with embedded splits  
 return [  
 {  
 "id": 9999,  
 "athlete": {"id": 42},  
 "name": "Test Activity",  
 "type": "Run",  
 "start\_date": datetime.utcnow().isoformat(),  
 "distance": 5000.0,  
 "elapsed\_time": 1500,  
 "moving\_time": 1450,  
 "total\_elevation\_gain": 100.0,  
 "external\_id": "test123",  
 "timezone": "UTC",  
 "splits\_metric": [  
 {  
 "lap\_index": 1,  
 "distance": 1000.0,  
 "elapsed\_time": 300,  
 "moving\_time": 290,  
 "average\_speed": 3.3,  
 "max\_speed": 3.5,  
 "start\_index": 0,  
 "end\_index": 299,  
 "split": True  
 }  
 ]  
 }  
 ]  
  
  
@patch("src.services.activity\_sync.fetch\_activities\_between", side\_effect=mock\_fetch\_activities\_between)  
def test\_activity\_sync\_inserts\_data(mock\_fetch, sqlalchemy\_session):  
 athlete\_id = 42  
 access\_token = "dummy"  
  
 count = sync\_recent(sqlalchemy\_session, athlete\_id, access\_token)  
  
 # ✅ Verify activity inserted  
 activity\_row = sqlalchemy\_session.query(Activity).filter\_by(activity\_id=9999).one\_or\_none()  
 assert activity\_row is not None  
 assert activity\_row.distance == 5000.0  
  
 # ✅ Verify splits inserted  
 splits = sqlalchemy\_session.query(Split).filter\_by(activity\_id=9999).all()  
 assert len(splits) == 1  
 assert splits[0].lap\_index == 1  
 assert splits[0].distance == 1000.0  
  
 # ✅ Verify total count returned  
 assert count == 1

### 📄 tests\test\_auth.py

# tests/test\_auth.py  
  
import os  
import time  
import pytest  
  
  
@pytest.fixture(autouse=True)  
def set\_env(monkeypatch):  
 monkeypatch.setenv("ADMIN\_USER", "admin")  
 monkeypatch.setenv("ADMIN\_PASS", "secret")  
 monkeypatch.setenv("SECRET\_KEY", "testsecret")  
  
  
def test\_login\_refresh\_logout(client):  
 """Test successful login, token refresh using Authorization header, and logout."""  
 # Step 1: Login  
 resp = client.post("/auth/login", json={"username": "admin", "password": "secret"})  
 assert resp.status\_code == 200  
 tokens = resp.get\_json()  
 print(f"🔑 Tokens after login: {tokens}")  
 assert "access\_token" in tokens  
 assert "refresh\_token" in tokens  
  
 # Step 2: Refresh token  
 time.sleep(1) # Ensure new token has different exp  
 refresh\_token = tokens["refresh\_token"]  
 headers = {"Authorization": f"Bearer {refresh\_token}"}  
 resp = client.post("/auth/refresh", headers=headers)  
 print(f"🔁 Refresh status: {resp.status\_code}, Body: {resp.data.decode()}")  
  
 assert resp.status\_code == 200  
 new\_access = resp.get\_json()["access\_token"]  
 assert new\_access != tokens["access\_token"]  
  
 # Step 3: Logout  
 resp = client.post("/auth/logout", json={"refresh\_token": refresh\_token})  
 assert resp.status\_code == 200  
 assert resp.get\_json()["message"] == "logged out"  
  
  
def test\_invalid\_login\_rejected(client):  
 """Test that invalid credentials are rejected."""  
 resp = client.post("/auth/login", json={"username": "wrong", "password": "bad"})  
 assert resp.status\_code == 401  
 assert "error" in resp.get\_json()  
  
  
def test\_invalid\_refresh\_token(client):  
 """Test that an invalid refresh token is rejected."""  
 headers = {"Authorization": "Bearer not.a.real.token"}  
 resp = client.post("/auth/refresh", headers=headers)  
 assert resp.status\_code == 401  
 assert "error" in resp.get\_json()  
  
  
import jwt  
from datetime import datetime, timedelta  
  
def test\_expired\_refresh\_token(client):  
 """Test refresh fails with an expired token."""  
 secret = os.environ.get("SECRET\_KEY", "testsecret")  
  
 expired\_token = jwt.encode(  
 {  
 "sub": "admin",  
 "exp": datetime.utcnow() - timedelta(seconds=1)  
 },  
 secret,  
 algorithm="HS256"  
 )  
  
 resp = client.post("/auth/refresh", headers={"Authorization": f"Bearer {expired\_token}"})  
 print(f"⏰ Expired refresh status: {resp.status\_code}, Body: {resp.data.decode()}")  
  
 assert resp.status\_code == 401  
 assert "error" in resp.get\_json()

### 📄 tests\test\_enrichment\_no\_hr\_zones.py

# tests/test\_enrichment\_no\_hr\_zones.py  
  
import pytest  
from unittest.mock import patch, Mock  
from src.services.enrichment\_sync import fetch\_hr\_zone\_percentages  
  
@patch("src.services.enrichment\_sync.requests.get")  
def test\_fetch\_hr\_zone\_percentages\_handles\_missing\_data(mock\_get):  
 # Strava responds without HR zones  
 mock\_response = Mock(status\_code=200)  
 mock\_response.json.return\_value = [{"type": "power"}] # No heartrate zones  
  
 mock\_get.return\_value = mock\_response  
  
 activity\_id = 123456  
 access\_token = "test-access"  
  
 result = fetch\_hr\_zone\_percentages(activity\_id, access\_token)  
 assert result is None # correctly handled as None

### 📄 tests\test\_enrichment\_with\_splits.py

# tests/test\_enrichment\_with\_splits.py  
  
import pytest  
from unittest.mock import patch, Mock  
from datetime import datetime  
  
from src.db.models.activities import Activity  
from src.db.models.splits import Split  
from src.db.dao.split\_dao import upsert\_splits  
from src.services.enrichment\_sync import enrich\_one\_activity  
  
# Sample mock data for enrichment  
SAMPLE\_ACTIVITY\_JSON = {  
 "id": 99999,  
 "name": "Mock Run",  
 "type": "Run",  
 "distance": 5000.0,  
 "moving\_time": 1500,  
 "elapsed\_time": 1600,  
 "total\_elevation\_gain": 50.0,  
 "average\_speed": 3.5,  
 "max\_speed": 4.0,  
 "suffer\_score": 30,  
 "average\_heartrate": 150,  
 "max\_heartrate": 170,  
 "calories": 400,  
 "splits\_metric": [  
 {  
 "lap\_index": 1,  
 "distance": 1000,  
 "elapsed\_time": 300,  
 "moving\_time": 295,  
 "average\_speed": 3.33,  
 "max\_speed": 3.5,  
 "start\_index": 0,  
 "end\_index": 299,  
 "split": True  
 }  
 ]  
}  
  
SAMPLE\_HR\_ZONE\_RESPONSE = [  
 {  
 "type": "heartrate",  
 "distribution\_buckets": [  
 {"min": 90, "max": 110, "time": 60},  
 {"min": 110, "max": 130, "time": 120},  
 {"min": 130, "max": 150, "time": 180},  
 {"min": 150, "max": 170, "time": 240},  
 {"min": 170, "max": 190, "time": 300}  
 ]  
 }  
]  
  
@pytest.fixture  
def seed\_activity(sqlalchemy\_session):  
 # Create parent activity row before enrichment (FK required)  
 activity = Activity(  
 activity\_id=99999,  
 athlete\_id=42,  
 start\_date=datetime.utcnow()  
 )  
 sqlalchemy\_session.add(activity)  
 sqlalchemy\_session.commit()  
 return activity  
  
@patch("src.services.enrichment\_sync.requests.get")  
def test\_enrich\_one\_activity\_with\_splits(mock\_requests\_get, sqlalchemy\_session, seed\_activity):  
 # Patch Strava activity fetch  
 def side\_effect(url, headers, timeout):  
 if "zones" in url:  
 mock\_zone = Mock()  
 mock\_zone.status\_code = 200  
 mock\_zone.json.return\_value = SAMPLE\_HR\_ZONE\_RESPONSE  
 return mock\_zone  
 else:  
 mock\_activity = Mock()  
 mock\_activity.status\_code = 200  
 mock\_activity.json.return\_value = SAMPLE\_ACTIVITY\_JSON  
 return mock\_activity  
  
 mock\_requests\_get.side\_effect = side\_effect  
  
 athlete\_id = seed\_activity.athlete\_id  
 access\_token = "mock-access-token"  
  
 # Execute enrichment with splits extraction  
 result = enrich\_one\_activity(sqlalchemy\_session, athlete\_id, access\_token, activity\_id=99999)  
 assert result is True  
  
 # Validate splits inserted  
 splits = sqlalchemy\_session.query(Split).filter\_by(activity\_id=99999).all()  
 assert len(splits) == 1  
 assert splits[0].lap\_index == 1  
 assert splits[0].distance == 1000  
 assert splits[0].elapsed\_time == 300  
  
 # Validate HR zone enrichment  
 activity = sqlalchemy\_session.query(Activity).filter\_by(activity\_id=99999).one()  
 assert activity.hr\_zone\_1\_pct is not None  
 assert activity.hr\_zone\_5\_pct is not None

### 📄 tests\test\_enrichmet\_retry\_logic.py

# tests/test\_enrichment\_retry\_logic.py  
  
import pytest  
import time  
from unittest.mock import patch, Mock  
from src.db.db\_session import get\_session  
from src.services.enrichment\_sync import enrich\_one\_activity  
  
@pytest.fixture  
def dummy\_activity\_id():  
 return 99999  
  
@pytest.fixture  
def dummy\_athlete\_id():  
 return 42  
  
@pytest.fixture  
def dummy\_access\_token():  
 return "test-access-token"  
  
@patch("src.services.enrichment\_sync.time.sleep", return\_value=None)  
@patch("src.services.enrichment\_sync.requests.get")  
def test\_enrich\_retries\_on\_429(mock\_get, mock\_sleep, dummy\_activity\_id, dummy\_athlete\_id, dummy\_access\_token):  
 # Simulate: first call 429, then 200, then 200 for HR zones  
 mock\_response\_429 = Mock(status\_code=429, headers={"Retry-After": "0"})  
 mock\_response\_200 = Mock(status\_code=200)  
 mock\_response\_200.json.return\_value = {"id": dummy\_activity\_id}  
  
 mock\_get.side\_effect = [mock\_response\_429, mock\_response\_200, mock\_response\_200]  
  
 session = get\_session()  
 try:  
 for attempt in range(2):  
 result = enrich\_one\_activity(session, dummy\_athlete\_id, dummy\_access\_token, dummy\_activity\_id)  
 if result:  
 break  
 time.sleep(1)  
  
 assert result is True  
 assert mock\_get.call\_count == 3  
 finally:  
 session.close()

### 📄 tests\test\_health.py

# tests/test\_health.py  
def test\_ping(client):  
 resp = client.get("/ping")  
 assert resp.status\_code == 200  
 assert resp.data == b"pong"

### 📄 tests\test\_hr\_zone\_api.py

import os  
import requests  
  
# Replace with a valid activity\_id you know has HR data  
activity\_id = 14663194187 # <-- replace with one of your existing IDs  
  
# Read access token directly from database or environment  
access\_token = os.getenv("STRAVA\_ACCESS\_TOKEN")  
  
if not access\_token:  
 raise RuntimeError("Missing STRAVA\_ACCESS\_TOKEN environment variable")  
  
url = f"https://www.strava.com/api/v3/activities/{activity\_id}/zones"  
headers = {"Authorization": f"Bearer {access\_token}"}  
  
resp = requests.get(url, headers=headers, timeout=10)  
print(f"HTTP Status: {resp.status\_code}")  
  
if resp.status\_code == 200:  
 data = resp.json()  
 print("✅ Successfully fetched HR zone data:")  
 print(data)  
else:  
 print("❌ Failed to fetch HR zones.")  
 print(resp.text)

### 📄 tests\test\_oauth\_flow.py

import pytest  
from unittest.mock import patch, Mock  
import requests  
  
@pytest.fixture(autouse=True)  
def set\_env(monkeypatch):  
 monkeypatch.setenv("STRAVA\_CLIENT\_ID", "12345")  
 monkeypatch.setenv("STRAVA\_CLIENT\_SECRET", "test\_secret")  
 monkeypatch.setenv("REDIRECT\_URI", "http://localhost/oauth/callback")  
  
@patch("requests.post")  
@patch("src.routes.oauth.sync\_full\_history") # ✅ Correct patch location  
def test\_oauth\_callback\_success(mock\_sync\_full\_history, mock\_post, client, sqlalchemy\_session):  
 mock\_response = Mock()  
 mock\_response.raise\_for\_status.return\_value = None  
 mock\_response.json.return\_value = {  
 "athlete": {"id": 999},  
 "access\_token": "access\_token\_value",  
 "refresh\_token": "refresh\_token\_value",  
 "expires\_at": 9999999999  
 }  
 mock\_post.return\_value = mock\_response  
  
 # ✅ prevent real ingestion call  
 mock\_sync\_full\_history.return\_value = 10  
  
 resp = client.get("/oauth/callback?code=fakecode")  
 assert resp.status\_code == 200  
 data = resp.get\_json()  
 assert "message" in data  
 assert "OAuth success" in data["message"]  
  
def test\_oauth\_callback\_missing\_code(client):  
 resp = client.get("/oauth/callback")  
 assert resp.status\_code == 400  
 assert "error" in resp.get\_json()  
  
@patch("requests.post")  
def test\_oauth\_callback\_strava\_http\_error(mock\_post, client):  
 mock\_response = Mock()  
 mock\_response.raise\_for\_status.side\_effect = requests.exceptions.HTTPError("Strava error")  
 mock\_response.text = "Bad Request"  
 mock\_post.return\_value = mock\_response  
  
 resp = client.get("/oauth/callback?code=badcode")  
 assert resp.status\_code == 502  
 assert "error" in resp.get\_json()  
  
@patch("requests.post")  
def test\_oauth\_callback\_incomplete\_response(mock\_post, client):  
 mock\_response = Mock()  
 mock\_response.raise\_for\_status.return\_value = None  
 mock\_response.json.return\_value = {"athlete": {}}  
 mock\_post.return\_value = mock\_response  
  
 resp = client.get("/oauth/callback?code=incomplete")  
 assert resp.status\_code == 502  
 assert "error" in resp.get\_json()  
  
def test\_oauth\_callback\_missing\_env(monkeypatch, client):  
 monkeypatch.delenv("STRAVA\_CLIENT\_ID", raising=False)  
 monkeypatch.delenv("STRAVA\_CLIENT\_SECRET", raising=False)  
 monkeypatch.delenv("REDIRECT\_URI", raising=False)  
  
 resp = client.get("/oauth/callback?code=fakecode")  
 assert resp.status\_code == 500  
 assert "error" in resp.get\_json()

### 📄 tests\test\_split\_dao.py

# tests/test\_split\_dao.py  
  
import pytest  
from src.db.dao.split\_dao import upsert\_splits  
from src.db.models.splits import Split  
from src.db.models.activities import Activity # ✅ Import Activity to insert FK parent  
  
def test\_upsert\_splits\_basic(sqlalchemy\_session):  
 # ✅ Insert parent Activity row to satisfy ForeignKey constraint  
 sqlalchemy\_session.add(Activity(activity\_id=123, athlete\_id=1))  
 sqlalchemy\_session.commit()  
  
 splits = [  
 {  
 "activity\_id": 123,  
 "lap\_index": 1,  
 "distance": 1000.0,  
 "elapsed\_time": 300,  
 "moving\_time": None,  
 "average\_speed": 3.33,  
 "max\_speed": None,  
 "start\_index": None,  
 "end\_index": None,  
 "split": True  
 },  
 {  
 "activity\_id": 123,  
 "lap\_index": 2,  
 "distance": 1000.0,  
 "elapsed\_time": 320,  
 "moving\_time": None,  
 "average\_speed": 3.12,  
 "max\_speed": None,  
 "start\_index": None,  
 "end\_index": None,  
 "split": True  
 }  
 ]  
  
 # ✅ Perform the upsert  
 inserted = upsert\_splits(sqlalchemy\_session, splits)  
 assert inserted == 2  
  
 # ✅ Verify inserted rows  
 rows = sqlalchemy\_session.query(Split).filter\_by(activity\_id=123).order\_by(Split.lap\_index).all()  
 assert len(rows) == 2  
 assert rows[0].lap\_index == 1  
 assert rows[0].distance == 1000.0  
 assert rows[0].elapsed\_time == 300  
 assert rows[0].average\_speed == 3.33

### 📄 tests\test\_split\_extraction.py

# tests/test\_split\_extraction.py  
  
from src.services.split\_extraction import extract\_splits  
  
def test\_extract\_splits():  
 sample\_activity = {  
 "id": 123,  
 "splits\_metric": [  
 {  
 "split": 1,  
 "distance": 1000.0,  
 "elapsed\_time": 300,  
 "average\_speed": 3.33  
 }  
 ]  
 }  
  
 splits = extract\_splits(sample\_activity)  
  
 assert len(splits) == 1  
  
 expected = {  
 "activity\_id": 123,  
 "lap\_index": 1,  
 "distance": 1000.0,  
 "elapsed\_time": 300,  
 "moving\_time": None,  
 "average\_speed": 3.33,  
 "max\_speed": None,  
 "start\_index": None,  
 "end\_index": None,  
 "split": True  
 }  
  
 # Compare each key individually  
 for key in expected:  
 assert splits[0][key] == expected[key]  
  
  
def test\_extract\_splits\_empty():  
 sample\_activity = {  
 "id": 456,  
 "splits\_metric": None  
 }  
  
 splits = extract\_splits(sample\_activity)  
 assert splits == []  
  
  
def test\_extract\_splits\_no\_splits\_metric():  
 sample\_activity = {  
 "id": 789  
 }  
  
 splits = extract\_splits(sample\_activity)  
 assert splits == []

### 📄 tests\test\_split\_extraction\_dao\_flow.py

# tests/test\_split\_extraction\_dao\_flow.py  
  
from src.db.dao.split\_dao import upsert\_splits  
from src.db.models.splits import Split  
from src.db.models.activities import Activity # ✅ NEW IMPORT - FK support  
from src.services.split\_extraction import extract\_splits  
  
  
def test\_split\_extraction\_and\_upsert(sqlalchemy\_session):  
 # ✅ Insert parent activity first to satisfy FK constraint  
 sqlalchemy\_session.add(Activity(activity\_id=555, athlete\_id=1))  
 sqlalchemy\_session.commit()  
  
 # Sample activity input with splits  
 sample\_activity = {  
 "id": 555,  
 "splits\_metric": [  
 {"split": 1, "distance": 1000, "elapsed\_time": 300, "average\_speed": 3.3},  
 {"split": 2, "distance": 1000, "elapsed\_time": 310, "average\_speed": 3.2}  
 ]  
 }  
  
 # ✅ Extract splits  
 splits = extract\_splits(sample\_activity)  
  
 # ✅ Upsert extracted splits into DB  
 count = upsert\_splits(sqlalchemy\_session, splits)  
  
 assert count == 2  
  
 # ✅ Query DB and validate inserted splits  
 rows = sqlalchemy\_session.query(Split).filter\_by(activity\_id=555).order\_by(Split.lap\_index).all()  
 assert len(rows) == 2  
 assert rows[0].lap\_index == 1  
 assert rows[0].distance == 1000  
 assert rows[0].elapsed\_time == 300  
 assert rows[0].average\_speed == 3.3

### 📄 tests\test\_split\_pipeline\_full.py

from src.db.models.activities import Activity  
from src.db.dao.split\_dao import upsert\_splits  
from src.services.split\_extraction import extract\_splits  
from src.db.models.splits import Split  
  
def test\_full\_split\_pipeline(sqlalchemy\_session):  
 """  
 Full pipeline: activity → extract\_splits → upsert\_splits → verify DB  
 """  
 # Seed parent activity (FK constraint)  
 activity = Activity(activity\_id=98765, athlete\_id=42)  
 sqlalchemy\_session.add(activity)  
 sqlalchemy\_session.commit()  
  
 # Simulate Strava activity payload  
 sample\_activity = {  
 "id": 98765,  
 "splits\_metric": [  
 {"split": 1, "distance": 1000, "elapsed\_time": 300, "average\_speed": 3.3},  
 {"split": 2, "distance": 1000, "elapsed\_time": 310, "average\_speed": 3.2},  
 {"split": 3, "distance": 1000, "elapsed\_time": 315, "average\_speed": 3.1}  
 ]  
 }  
  
 # Extraction  
 splits = extract\_splits(sample\_activity)  
 assert len(splits) == 3  
  
 # DAO upsert  
 count = upsert\_splits(sqlalchemy\_session, splits)  
 assert count == 3  
  
 # Verify DB contents  
 rows = sqlalchemy\_session.query(Split).filter\_by(activity\_id=98765).all()  
 assert len(rows) == 3

### 📄 tests\test\_split\_upsert\_idempotency.py

# tests/test\_split\_upsert\_idempotency.py  
  
from src.db.models.activities import Activity  
from src.db.dao.split\_dao import upsert\_splits  
from src.db.models.splits import Split  
  
def test\_upsert\_splits\_idempotency(sqlalchemy\_session):  
 # Insert parent activity  
 activity\_id = 55555  
 sqlalchemy\_session.add(Activity(activity\_id=activity\_id, athlete\_id=1))  
 sqlalchemy\_session.commit()  
  
 splits = [  
 {  
 "activity\_id": activity\_id,  
 "lap\_index": 1,  
 "distance": 1000.0,  
 "elapsed\_time": 300,  
 "moving\_time": 290,  
 "average\_speed": 3.3,  
 "max\_speed": 3.5,  
 "start\_index": 0,  
 "end\_index": 299,  
 "split": True  
 }  
 ]  
  
 # First insert  
 inserted = upsert\_splits(sqlalchemy\_session, splits)  
 assert inserted == 1  
  
 # Second insert (should conflict-update, not duplicate)  
 inserted\_again = upsert\_splits(sqlalchemy\_session, splits)  
 assert inserted\_again == 1  
  
 # Verify only 1 row exists  
 rows = sqlalchemy\_session.query(Split).filter\_by(activity\_id=activity\_id).all()  
 assert len(rows) == 1

### 📄 tests\test\_sync.py

import pytest  
  
# ✅ This test uses normal (unpatched) client fixture  
@pytest.mark.parametrize(  
 "key,code",  
 [  
 ("wrong", 401),  
 ("devkey123", 401), # valid key but no tokens exist → causes sync\_recent to fail  
 ],  
)  
def test\_sync\_auth\_and\_error(client, key, code):  
 resp = client.get(f"/sync-strava-to-db/123?key={key}")  
 # ✅ If key is wrong → 401  
 # ✅ If key is correct but no tokens → will raise error and return 500  
 expected\_code = 401 if key == "wrong" else 500  
 assert resp.status\_code == expected\_code  
  
  
# ✅ This test uses patched\_client to mock sync\_recent successfully  
def test\_sync\_success(patched\_client):  
 resp = patched\_client.get("/sync-strava-to-db/123?key=devkey123")  
 assert resp.status\_code == 200  
 assert resp.get\_json() == {"inserted": 10}

### 📄 tests\test\_sync\_token\_refresh.py

# tests/test\_sync\_token\_refresh.py  
  
import pytest  
from datetime import datetime, timedelta  
from src.db.dao.token\_dao import save\_tokens\_sa  
from src.db.models.tokens import Token  
from src.services.activity\_sync import sync\_recent  
from src.services.enrichment\_sync import run\_enrichment\_batch  
  
  
@pytest.fixture  
def token\_valid():  
 return int((datetime.utcnow() + timedelta(hours=1)).timestamp())  
  
@pytest.fixture  
def token\_expired():  
 return int((datetime.utcnow() - timedelta(hours=1)).timestamp())  
  
  
def test\_sync\_recent\_with\_valid\_token(sqlalchemy\_session, monkeypatch, token\_valid):  
 athlete\_id = 999  
  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, "valid\_access", "refresh\_token", token\_valid)  
  
 monkeypatch.setattr(  
 "src.services.activity\_sync.fetch\_activities\_between",  
 lambda access\_token, start\_date, end\_date, per\_page: []  
 )  
  
 count = sync\_recent(sqlalchemy\_session, athlete\_id, access\_token="valid\_access")  
 assert count == 0  
  
  
def test\_sync\_recent\_with\_expired\_token(sqlalchemy\_session, monkeypatch, token\_expired, token\_valid):  
 athlete\_id = 1000  
  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, "old\_access", "refresh\_token", token\_expired)  
  
 monkeypatch.setattr(  
 "src.services.strava.refresh\_strava\_token",  
 lambda refresh\_token: {  
 "access\_token": "new\_access",  
 "refresh\_token": "new\_refresh",  
 "expires\_at": token\_valid  
 }  
 )  
  
 # DON'T CALL refresh\_strava\_token — just assign directly  
 tokens = {  
 "access\_token": "new\_access",  
 "refresh\_token": "new\_refresh",  
 "expires\_at": token\_valid  
 }  
  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, tokens["access\_token"], tokens["refresh\_token"], tokens["expires\_at"])  
  
 monkeypatch.setattr(  
 "src.services.activity\_sync.fetch\_activities\_between",  
 lambda access\_token, start\_date, end\_date, per\_page: []  
 )  
  
 count = sync\_recent(sqlalchemy\_session, athlete\_id, access\_token="new\_access")  
 assert count == 0  
  
 token\_row = sqlalchemy\_session.query(Token).filter\_by(athlete\_id=athlete\_id).one()  
 assert token\_row.access\_token == "new\_access"  
 assert token\_row.refresh\_token == "new\_refresh"  
 assert token\_row.expires\_at == token\_valid  
  
  
def test\_enrichment\_refresh\_path(sqlalchemy\_session, monkeypatch, token\_expired, token\_valid):  
 athlete\_id = 1001  
  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, "stale\_access", "refresh\_token", token\_expired)  
  
 monkeypatch.setattr(  
 "src.services.strava.refresh\_strava\_token",  
 lambda refresh\_token: {  
 "access\_token": "fresh\_access",  
 "refresh\_token": "fresh\_refresh",  
 "expires\_at": token\_valid  
 }  
 )  
  
 tokens = {  
 "access\_token": "fresh\_access",  
 "refresh\_token": "fresh\_refresh",  
 "expires\_at": token\_valid  
 }  
  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, tokens["access\_token"], tokens["refresh\_token"], tokens["expires\_at"])  
  
 monkeypatch.setattr(  
 "src.services.enrichment\_sync.get\_activities\_to\_enrich",  
 lambda session, athlete\_id, limit: []  
 )  
  
 count = run\_enrichment\_batch(sqlalchemy\_session, athlete\_id)  
 assert count == 0  
  
 token\_row = sqlalchemy\_session.query(Token).filter\_by(athlete\_id=athlete\_id).one()  
 assert token\_row.access\_token == "fresh\_access"  
 assert token\_row.refresh\_token == "fresh\_refresh"  
 assert token\_row.expires\_at == token\_valid

### 📄 tests\test\_token\_refresh.py

# tests/test\_token\_refresh.py  
  
import pytest  
from datetime import datetime, timedelta  
from src.services.token\_refresh import ensure\_fresh\_access\_token  
from src.db.models.tokens import Token  
from src.db.dao.token\_dao import save\_tokens\_sa  
  
@pytest.fixture  
def token\_expired():  
 return int((datetime.utcnow() - timedelta(hours=1)).timestamp())  
  
@pytest.fixture  
def token\_valid():  
 return int((datetime.utcnow() + timedelta(hours=1)).timestamp())  
  
def test\_token\_refresh\_logic(sqlalchemy\_session, monkeypatch, token\_valid, token\_expired):  
 athlete\_id = 123456  
  
 # Save valid token  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, "valid\_access", "refresh\_token", token\_valid)  
  
 # Should NOT call refresh if token is still valid  
 monkeypatch.setattr(  
 "src.services.token\_refresh.refresh\_strava\_token",  
 lambda rt: (\_ for \_ in ()).throw(Exception("Should not refresh"))  
 )  
 access\_token = ensure\_fresh\_access\_token(sqlalchemy\_session, athlete\_id)  
 assert access\_token == "valid\_access"  
  
 # Now simulate expired token  
 save\_tokens\_sa(sqlalchemy\_session, athlete\_id, "old\_access", "refresh\_token", token\_expired)  
  
 # Mock refresh\_strava\_token correctly  
 monkeypatch.setattr(  
 "src.services.token\_refresh.refresh\_strava\_token",  
 lambda rt: {  
 "access\_token": "new\_access",  
 "refresh\_token": "new\_refresh",  
 "expires\_at": token\_valid  
 }  
 )  
  
 access\_token = ensure\_fresh\_access\_token(sqlalchemy\_session, athlete\_id)  
 assert access\_token == "new\_access"  
  
 # Verify DB state updated after refresh  
 token\_row = sqlalchemy\_session.query(Token).filter\_by(athlete\_id=athlete\_id).one()  
 assert token\_row.access\_token == "new\_access"  
 assert token\_row.refresh\_token == "new\_refresh"  
 assert token\_row.expires\_at == token\_valid

### 📄 tests\utils.py

import jwt  
import datetime  
  
def generate\_test\_token(user\_id, secret\_key, expires\_in=3600):  
 payload = {  
 "sub": user\_id,  
 "exp": datetime.datetime.utcnow() + datetime.timedelta(seconds=expires\_in)  
 }  
 return jwt.encode(payload, secret\_key, algorithm="HS256")

### 📄 wsgi.py

from src.app import create\_app  
app = create\_app()