

TypeShield Authenticator Tutorial

This beginner-friendly guide explains how the project works, step by step. It also explains each file and the important code sections so you can follow the flow without prior security or biometrics knowledge.

1) What This Project Does

TypeShield Authenticator combines a normal password with keystroke dynamics. That means it measures how you type (timing and rhythm) in addition to what you type.

- During registration, the app stores your password hash and a typing template.
- During login, it checks the password and compares your typing pattern.
- If the timing is too different, login is rejected even if the password is correct.

2) Quick Start (Beginner Setup)

Run these steps from the project folder:

Snippet 1: Local setup commands (requirements.txt)

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install -r requirements.txt
$ createdb keyrhythm
$ uvicorn app.main:app --reload
```

Open a browser and visit <http://127.0.0.1:8000>. Register first, then login.

3) Project Structure

Important files in this project:

- app/main.py: FastAPI entry point and web routes.
- app/config.py: Environment settings (database URL, secret key, threshold).
- app/database.py: SQLAlchemy engine and session helper.
- app/models.py: Database tables for users, templates, and attempts.
- app/schemas.py: Pydantic models that validate behaviour data.
- app/auth.py: Password hashing and JWT creation.
- app/behaviour.py: Scoring logic for typing similarity.
- app/utils.py: Helper math utilities.
- app/static/behaviour.js: Frontend capture of typing rhythm.
- app/templates/*.html: HTML pages for register, login, dashboard.

4) Configuration (app/config.py)

Settings are loaded from environment variables. The most important values are the database URL, secret key, and behaviour threshold.

Snippet 2: Settings model (app/config.py)

```
class Settings(BaseSettings):
    app_name: str = "TypeShield Authenticator"
    secret_key: str = Field("super-secret-key-change-me", env="SECRET_KEY")
    database_url: str = Field(DEFAULT_DB_URL, env="DATABASE_URL")
    behaviour_threshold: float = Field(75.0, env="BEHAVIOUR_THRESHOLD")
```

Beginner tip: put these values in a .env file so you do not hard-code secrets in

code.

5) Database Layer (app/database.py)

The database engine is created once, and FastAPI uses a session dependency for each request.

Snippet 3: Engine and session dependency (app/database.py)

```
engine = create_engine(settings.database_url, echo=False, pool_pre_ping=True)

def init_db() -> None:
    SQLAlchemyModel.metadata.create_all(engine)

def get_session() -> Iterator[Session]:
    session = Session(engine)
    try:
        yield session
        session.commit()
    except Exception:
        session.rollback()
        raise
    finally:
        session.close()
```

FastAPI injects the session into route handlers with Depends(get_session).

6) Data Models (app/models.py)

SQLModel classes become database tables. Each user has one behaviour template, and every login attempt is stored.

Snippet 4: User and BehaviourTemplate (app/models.py)

```
class User(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    username: str = Field(index=True, unique=True)
    hashed_password: str

class BehaviourTemplate(SQLModel, table=True):
    user_id: int = Field(foreign_key="user.id")
    dwell_times: List[float] = Field(default_factory=list, sa_column=Column(JSON))
    flight_times: List[float] = Field(default_factory=list, sa_column=Column(JSON))
    total_time: float
    error_count: int = Field(default=0)
```

7) Request Validation (app/schemas.py)

Pydantic ensures the timing values are valid and non-negative.

Snippet 5: BehaviourData validation (app/schemas.py)

```
class BehaviourData(BaseModel):
    dwell_times: List[float] = Field(default_factory=list)
    flight_times: List[float] = Field(default_factory=list)
    total_time: float
    error_count: int = 0
    device_type: str = "fine"

    @validator("total_time")
    def validate_total_time(cls, v: float) -> float:
```

```

    if v < 0:
        raise ValueError("total_time must be non-negative")
    return v

```

8) Password Hashing (app/auth.py)

Passwords are never stored in plain text. They are hashed with PBKDF2-SHA256.

Snippet 6: Password hashing helpers (app/auth.py)

```

pwd_context = CryptContext(schemes=["pbkdf2_sha256"], deprecated="auto")

def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)

```

9) Behaviour Scoring (app/behaviour.py)

The scoring function compares the stored template with the new attempt. It returns a score from 0 to 100 and checks a threshold.

Snippet 7: Similarity score (app/behaviour.py)

```

def similarity_score(stored: BehaviourTemplate, attempt: BehaviourData) -> tuple[float, dict]:
    dwell_component = dwell_score(stored, attempt)
    flight_component = flight_score(stored, attempt)
    total_component = total_time_score(stored, attempt)
    speed_component = speed_score(stored, attempt)
    error_component = error_score(stored, attempt)
    length_component = length_score(stored, attempt)
    combined = (
        weights["dwell"] * dwell_component
        + weights["flight"] * flight_component
        + weights["total"] * total_component
        + weights["speed"] * speed_component
        + weights["length"] * length_component
        + weights["error"] * error_component
    )
    score = round(clamp(combined), 2)
    return score, components

```

The `is_behaviour_match` function adds strong guards (key count and tempo) before accepting a score.

10) Utility Math (app/utils.py)

Two helpers handle clamping and average percentage difference between vectors.

Snippet 8: Clamp and average difference (app/utils.py)

```

def clamp(value: float, minimum: float = 0.0, maximum: float = 100.0) -> float:
    return max(minimum, min(maximum, value))

def average_percentage_difference(reference: List[float], sample: List[float]) -> float:
    if not reference or not sample:
        return 100.0
    diffs = []
    for ref_val, sample_val in zip(reference[:min_len], sample[:min_len]):
        denominator = ref_val if ref_val != 0 else 1e-6

```

```

        diffs.append(abs(ref_val - sample_val) / denominator * 100)
    return sum(diffs) / max_len if max_len else 100.0

```

11) FastAPI Routes (app/main.py)

The main file wires everything together: app startup, register, login, dashboard, and logout.

Snippet 9: App setup (app/main.py)

```

app = FastAPI(title=settings.app_name)
app.add_middleware(SessionMiddleware, secret_key=settings.secret_key)

@app.on_event("startup")
def on_startup() -> None:
    init_db()

```

On startup, init_db creates tables if they do not exist.

Snippet 10: Register flow (app/main.py)

```

@app.post("/register")
def register(...):
    statement = select(User).where(User.username == username)
    if session.exec(statement).first():
        return error

    behaviour_parsed = BehaviourData.parse_raw(behaviour_data)
    user = User(username=username, hashed_password=hash_password(password))
    session.add(user)
    session.flush()
    template = BehaviourTemplate(user_id=user.id, ...)
    session.add(template)
    session.commit()
    request.session["user_id"] = user.id
    return RedirectResponse(url="/dashboard", status_code=302)

```

Snippet 11: Login flow (app/main.py)

```

@app.post("/login")
def login(...):
    user = session.exec(select(User).where(User.username == username)).first()
    if not user or not verify_password(password, user.hashed_password):
        return invalid_credentials

    stored_template = session.exec(select(BehaviourTemplate).where(...)).first()
    behaviour_parsed = BehaviourData.parse_raw(behaviour_data)
    is_match, score, reasons = behaviour.is_behaviour_match(stored_template, behaviour_parsed)
    if not is_match:
        return behaviour_mismatch

    request.session["user_id"] = user.id
    request.session["score"] = score
    return RedirectResponse(url="/dashboard", status_code=302)

```

12) Frontend Capture (app/static/behaviour.js)

The browser script captures dwell time (keydown to keyup) and flight time (gap between keys). It also tracks errors (backspace).

Snippet 12: Keydown capture (app/static/behaviour.js)

```

passwordInput.addEventListener("keydown", (event) => {
    if (event.key === "Backspace") {
        errorCount += 1;
    }
});

```

```

        dwellTimes.pop();
        flightTimes.pop();
        return;
    }
    if (startTime === null) startTime = now;
    if (lastKeyUp !== null) flightTimes.push(now - lastKeyUp);
    pendingDown = now;
  });

```

13) Templates (HTML)

Jinja2 templates render the forms and dashboard. They also include the behaviour.js script on register and login pages.

Snippet 13: Login form captures behaviour (app/templates/login.html)

```

<form method="post" action="/login">
  <input type="text" name="username" required>
  <input type="password" id="password" name="password" required>
  <input type="hidden" id="behaviour_data" name="behaviour_data">
</form>
<script src="/static/behaviour.js"></script>

```

14) Step-by-Step Data Flow

Register

- User types password; behaviour.js records timing vectors.
- Form submits username, password, and behaviour JSON.
- Backend hashes password and stores BehaviourTemplate.
- Session is created and user is redirected to /dashboard.

Login

- Password is checked with verify_password.
- Stored template is loaded from database.
- Behaviour score is computed; must be above threshold.
- On success, session and score are stored for dashboard.

15) Common Beginner Questions

- Why store timing vectors? They capture how you type, not just what you type.
- Why JSON columns? Timing arrays fit naturally as JSON lists.
- What is behaviour_threshold? The minimum score required to accept a login.
- What happens on touch devices? The script switches to coarse capture.

16) Where To Explore Next

- Try changing BEHAVIOUR_THRESHOLD to see stricter or softer matching.
- Add more charts or logs to the dashboard.
- Store multiple templates per user for more robust matching.

Code Snippets Index

- Snippet 1: Local setup commands (requirements.txt) - page 1
- Snippet 2: Settings model (app/config.py) - page 1
- Snippet 3: Engine and session dependency (app/database.py) - page 2
- Snippet 4: User and BehaviourTemplate (app/models.py) - page 2

- Snippet 5: BehaviourData validation (app/schemas.py) - page 2
- Snippet 6: Password hashing helpers (app/auth.py) - page 3
- Snippet 7: Similarity score (app/behaviour.py) - page 3
- Snippet 8: Clamp and average difference (app/utils.py) - page 3
- Snippet 9: App setup (app/main.py) - page 4
- Snippet 10: Register flow (app/main.py) - page 4
- Snippet 11: Login flow (app/main.py) - page 4
- Snippet 12: Keydown capture (app/static/behaviour.js) - page 4
- Snippet 13: Login form captures behaviour (app/templates/login.html) - page 5