Complete every assignment in the order shown, push all work to a public GitHub repository named fastapi-intern-exercise, and notify me when everything is live. Each assignment below has two short parts—What to do and What to submit—so there is no confusion about either the work or the deliverables.

Assignment 1 – Environment & Bootstrap

What to do: Set up Python and VS Code as shown in the video; create a virtual environment; add all packages to requirements.txt; write a single command (make dev on macOS/Linux or setup.ps1 on Windows) that installs the dependencies and launches the FastAPI server on localhost:8000; open http://127.0.0.1:8000/docs to confirm the app runs.

What to submit: Commit the setup script, the requirements.txt, and a screenshot saved as docs/setup\_proof.png that shows the Swagger UI in your browser.

Assignment 2 – Core API & Database

What to do: Build two new resources—/authors and /books—using FastAPI, Pydantic, SQLAlchemy, and PostgreSQL; model a one-to-many relationship (one author, many books); implement full CRUD plus query-parameter filtering; keep the routers, schemas, and models in their own modules.

What to submit: Commit the code and Alembic migrations, then export and add a Postman collection (postman\_collection.json) that demonstrates every CRUD call for both endpoints.

Assignment 3 – Authentication & Permissions

What to do: Add JWT-based login at /login; hash passwords; create a /votes route so a logged-in user can like or unlike a book; ensure only the owner of a book can update or delete it; write pytest tests that cover login, token expiry, permission checks, and the vote logic.

What to submit: Commit all auth code and tests; include a console screenshot (or the HTML coverage report) proving the tests pass with at least 80 % coverage.

Assignment 4 – Automated Testing & CI

What to do: Expand the test suite to at least twenty pytest cases covering routes, schemas, and utilities; add a GitHub Actions workflow that checks out the code, spins up a Postgres service, installs requirements, and runs the tests on every push; make the build fail if any test fails.

What to submit: Push the .github/workflows/ci.yml file and add a build-status badge to the README that links to the latest successful run.

Assignment 5 – Containerisation & Deployment

What to do: Write a Dockerfile for the FastAPI app and a docker-compose.yml that includes Postgres and automatically runs Alembic migrations; push the built image to Docker Hub; deploy the stack to either Heroku or an Ubuntu VM with HTTPS enabled; expose a health endpoint at /api/health that returns {"status":"ok"}.

What to submit: Provide the live URL of the deployed app, commit a screenshot of the production Swagger UI, and add a short SUBMISSION.md that lists the Docker Hub image tag along with demo login credentials for testing protected routes.

**Main.py file**

*from* fastapi *import* FastAPI, Depends, HTTPException, status, Query

*from* sqlalchemy.orm *import* Session

*from* sqlalchemy *import* text, desc, asc

*from* database *import* engine, get\_db, recreate\_tables

*from* typing *import* List, Optional

*import* models

*from* pydantic *import* BaseModel

*from* datetime *import* datetime

*# Create database tables with new schema (only if they don't exist)*

*from* database *import* Base

Base.metadata.create\_all(*bind*=engine)

*# Pydantic models for request/response validation*

class PostCreate(BaseModel):

    title: str

    content: str

    published: bool = True

class PostUpdate(BaseModel):

    title: Optional[str] = None

    content: Optional[str] = None

    published: Optional[bool] = None

class PostResponse(BaseModel):

    id: int

    title: str

    content: str

    published: bool

    created\_at: datetime

    class Config:

        from\_attributes = True

*# User Pydantic models*

class UserCreate(BaseModel):

    email: str

    password: str

    first\_name: Optional[str] = None

    last\_name: Optional[str] = None

    is\_active: bool = True

    is\_verified: bool = False

class UserUpdate(BaseModel):

    email: Optional[str] = None

    password: Optional[str] = None

    first\_name: Optional[str] = None

    last\_name: Optional[str] = None

    is\_active: Optional[bool] = None

    is\_verified: Optional[bool] = None

class UserResponse(BaseModel):

    id: int

    email: str

    first\_name: Optional[str]

    last\_name: Optional[str]

    is\_active: bool

    is\_verified: bool

    created\_at: datetime

    updated\_at: datetime

    class Config:

        from\_attributes = True

app = FastAPI(*title*="Blog API", *version*="1.0.0", *description*="A comprehensive Blog API with PostgreSQL integration")

@app.get("/")

def root():

*return* {"message": "Blog API is running!", "version": "1.0.0", "docs": "/docs"}

@app.get("/posts", *response\_model*=List[PostResponse])

def get\_posts(

*db*: Session = Depends(get\_db),

*skip*: int = Query(0, *ge*=0, *description*="Number of posts to skip"),

*limit*: int = Query(100, *ge*=1, *le*=1000, *description*="Maximum number of posts to return"),

*published*: Optional[bool] = Query(None, *description*="Filter by published status"),

*search*: Optional[str] = Query(None, *description*="Search in title and content")

):

    """Get all posts with filtering, pagination, and search capabilities"""

    query = db.query(models.Post)

*# Apply filters*

*if* published is not None:

        query = query.filter(models.Post.published == published)

*if* search:

        search\_term = f"%{search}%"

        query = query.filter(

            (models.Post.title.ilike(search\_term)) |

            (models.Post.content.ilike(search\_term))

        )

*# Apply pagination*

    posts = query.offset(skip).limit(limit).all()

*return* posts

@app.get("/posts/{post\_id}", *response\_model*=PostResponse)

def get\_post(*post\_id*: int, *db*: Session = Depends(get\_db)):

    """Get a specific post by ID"""

    post = db.query(models.Post).filter(models.Post.id == post\_id).first()

*if* not post:

*raise* HTTPException(*status\_code*=404, *detail*=f"Post with id {post\_id} not found")

*return* post

@app.post("/posts", *response\_model*=PostResponse, *status\_code*=status.HTTP\_201\_CREATED)

def create\_post(*post*: PostCreate, *db*: Session = Depends(get\_db)):

    """Create a new post with validation"""

    new\_post = models.Post(

*title*=post.title,

*content*=post.content,

*published*=post.published

    )

    db.add(new\_post)

    db.commit()

    db.refresh(new\_post)

*return* new\_post

@app.put("/posts/{post\_id}", *response\_model*=PostResponse)

def update\_post(*post\_id*: int, *post*: PostUpdate, *db*: Session = Depends(get\_db)):

    """Update a post with partial updates support"""

    db\_post = db.query(models.Post).filter(models.Post.id == post\_id).first()

*if* not db\_post:

*raise* HTTPException(*status\_code*=404, *detail*=f"Post with id {post\_id} not found")

*# Only update provided fields*

*if* post.title is not None:

        db\_post.title = post.title

*if* post.content is not None:

        db\_post.content = post.content

*if* post.published is not None:

        db\_post.published = post.published

    db.commit()

    db.refresh(db\_post)

*return* db\_post

@app.delete("/posts/{post\_id}", *status\_code*=status.HTTP\_204\_NO\_CONTENT)

def delete\_post(*post\_id*: int, *db*: Session = Depends(get\_db)):

    """Delete a post"""

    post = db.query(models.Post).filter(models.Post.id == post\_id).first()

*if* not post:

*raise* HTTPException(*status\_code*=404, *detail*=f"Post with id {post\_id} not found")

    db.delete(post)

    db.commit()

*return* None

*# User CRUD endpoints*

@app.get("/users", *response\_model*=List[UserResponse])

def get\_users(

*db*: Session = Depends(get\_db),

*skip*: int = Query(0, *ge*=0, *description*="Number of users to skip"),

*limit*: int = Query(100, *ge*=1, *le*=1000, *description*="Maximum number of users to return"),

*is\_active*: Optional[bool] = Query(None, *description*="Filter by active status"),

*is\_verified*: Optional[bool] = Query(None, *description*="Filter by verification status")

):

    """Get all users with filtering and pagination"""

    query = db.query(models.User)

*# Apply filters*

*if* is\_active is not None:

        query = query.filter(models.User.is\_active == is\_active)

*if* is\_verified is not None:

        query = query.filter(models.User.is\_verified == is\_verified)

*# Apply pagination*

    users = query.offset(skip).limit(limit).all()

*return* users

@app.get("/users/{user\_id}", *response\_model*=UserResponse)

def get\_user(*user\_id*: int, *db*: Session = Depends(get\_db)):

    """Get a specific user by ID"""

    user = db.query(models.User).filter(models.User.id == user\_id).first()

*if* not user:

*raise* HTTPException(*status\_code*=404, *detail*=f"User with id {user\_id} not found")

*return* user

@app.post("/users", *response\_model*=UserResponse, *status\_code*=status.HTTP\_201\_CREATED)

def create\_user(*user*: UserCreate, *db*: Session = Depends(get\_db)):

    """Create a new user with validation"""

*# Check if email already exists*

    existing\_user = db.query(models.User).filter(models.User.email == user.email).first()

*if* existing\_user:

*raise* HTTPException(*status\_code*=400, *detail*="Email already registered")

    new\_user = models.User(

*email*=user.email,

*password*=user.password,  *# In production, hash this password*

*first\_name*=user.first\_name,

*last\_name*=user.last\_name,

*is\_active*=user.is\_active,

*is\_verified*=user.is\_verified

    )

    db.add(new\_user)

    db.commit()

    db.refresh(new\_user)

*return* new\_user

@app.put("/users/{user\_id}", *response\_model*=UserResponse)

def update\_user(*user\_id*: int, *user*: UserUpdate, *db*: Session = Depends(get\_db)):

    """Update a user with partial updates support"""

    db\_user = db.query(models.User).filter(models.User.id == user\_id).first()

*if* not db\_user:

*raise* HTTPException(*status\_code*=404, *detail*=f"User with id {user\_id} not found")

*# Check if email is being updated and if it already exists*

*if* user.email and user.email != db\_user.email:

        existing\_user = db.query(models.User).filter(models.User.email == user.email).first()

*if* existing\_user:

*raise* HTTPException(*status\_code*=400, *detail*="Email already registered")

*# Only update provided fields*

*if* user.email is not None:

        db\_user.email = user.email

*if* user.password is not None:

        db\_user.password = user.password  *# In production, hash this password*

*if* user.first\_name is not None:

        db\_user.first\_name = user.first\_name

*if* user.last\_name is not None:

        db\_user.last\_name = user.last\_name

*if* user.is\_active is not None:

        db\_user.is\_active = user.is\_active

*if* user.is\_verified is not None:

        db\_user.is\_verified = user.is\_verified

    db.commit()

    db.refresh(db\_user)

*return* db\_user

@app.delete("/users/{user\_id}", *status\_code*=status.HTTP\_204\_NO\_CONTENT)

def delete\_user(*user\_id*: int, *db*: Session = Depends(get\_db)):

    """Delete a user"""

    user = db.query(models.User).filter(models.User.id == user\_id).first()

*if* not user:

*raise* HTTPException(*status\_code*=404, *detail*=f"User with id {user\_id} not found")

    db.delete(user)

    db.commit()

*return* None

*# Advanced query endpoints*

@app.get("/posts/search/advanced")

def advanced\_search(

*db*: Session = Depends(get\_db),

*title\_contains*: Optional[str] = Query(None, *description*="Title contains text"),

*content\_contains*: Optional[str] = Query(None, *description*="Content contains text"),

*published\_only*: bool = Query(False, *description*="Show only published posts"),

*sort\_by*: str = Query("created\_at", *description*="Sort by field (id, title, created\_at)"),

*sort\_order*: str = Query("desc", *description*="Sort order (asc, desc)")

):

    """Advanced search with multiple filters and sorting"""

    query = db.query(models.Post)

*# Apply filters*

*if* title\_contains:

        query = query.filter(models.Post.title.ilike(f"%{title\_contains}%"))

*if* content\_contains:

        query = query.filter(models.Post.content.ilike(f"%{content\_contains}%"))

*if* published\_only:

        query = query.filter(models.Post.published == True)

*# Apply sorting*

*if* sort\_by == "id":

        sort\_field = models.Post.id

*elif* sort\_by == "title":

        sort\_field = models.Post.title

*elif* sort\_by == "created\_at":

        sort\_field = models.Post.created\_at

*else*:

        sort\_field = models.Post.created\_at

*if* sort\_order.lower() == "asc":

        query = query.order\_by(asc(sort\_field))

*else*:

        query = query.order\_by(desc(sort\_field))

    posts = query.all()

*return* {"count": len(posts), "posts": posts}

@app.get("/posts/stats")

def get\_posts\_stats(*db*: Session = Depends(get\_db)):

    """Get statistics about posts"""

    total\_posts = db.query(models.Post).count()

    published\_posts = db.query(models.Post).filter(models.Post.published == True).count()

    draft\_posts = total\_posts - published\_posts

*# Get latest post*

    latest\_post = db.query(models.Post).order\_by(desc(models.Post.created\_at)).first()

*return* {

        "total\_posts": total\_posts,

        "published\_posts": published\_posts,

        "draft\_posts": draft\_posts,

        "latest\_post\_date": latest\_post.created\_at *if* latest\_post *else* None

    }

@app.get("/posts/recent/{limit}")

def get\_recent\_posts(*limit*: int, *db*: Session = Depends(get\_db)):

    """Get recent posts with custom limit"""

*if* limit < 1 or limit > 50:

*raise* HTTPException(*status\_code*=400, *detail*="Limit must be between 1 and 50")

    posts = db.query(models.Post).order\_by(desc(models.Post.created\_at)).limit(limit).all()

*return* {"recent\_posts": posts, "limit": limit}

@app.get("/posts/published")

def get\_published\_posts(

*db*: Session = Depends(get\_db),

*skip*: int = Query(0, *ge*=0),

*limit*: int = Query(100, *ge*=1, *le*=1000)

):

    """Get only published posts"""

    posts = db.query(models.Post).filter(models.Post.published == True).offset(skip).limit(limit).all()

*return* {"published\_posts": posts, "count": len(posts)}

@app.get("/posts/drafts")

def get\_draft\_posts(

*db*: Session = Depends(get\_db),

*skip*: int = Query(0, *ge*=0),

*limit*: int = Query(100, *ge*=1, *le*=1000)

):

    """Get only draft posts"""

    posts = db.query(models.Post).filter(models.Post.published == False).offset(skip).limit(limit).all()

*return* {"draft\_posts": posts, "count": len(posts)}

*# Raw SQL query endpoint for advanced users*

@app.get("/query/raw")

def execute\_raw\_query(

*sql\_query*: str = Query(..., *description*="Raw SQL query to execute"),

*db*: Session = Depends(get\_db)

):

    """Execute raw SQL queries (use with caution)"""

*try*:

*# Only allow SELECT queries for security*

*if* not sql\_query.strip().upper().startswith('SELECT'):

*raise* HTTPException(*status\_code*=400, *detail*="Only SELECT queries are allowed")

        result = db.execute(text(sql\_query))

        columns = result.keys()

        rows = [dict(zip(columns, row)) *for* row *in* result.fetchall()]

*return* {

            "query": sql\_query,

            "columns": list(columns),

            "rows": rows,

            "row\_count": len(rows)

        }

*except* Exception *as* e:

*raise* HTTPException(*status\_code*=400, *detail*=f"Query execution failed: {str(e)}")

*# Database health check*

@app.get("/health")

def health\_check(*db*: Session = Depends(get\_db)):

    """Check database connection and table status"""

*try*:

*# Test database connection*

        db.execute(text("SELECT 1"))

*# Get table info*

        total\_posts = db.query(models.Post).count()

        total\_users = db.query(models.User).count()

*return* {

            "status": "healthy",

            "database": "connected",

            "tables": {

                "posts": {

                    "total\_records": total\_posts

                },

                "users": {

                    "total\_records": total\_users

                }

            },

            "timestamp": "2025-01-09T10:50:00Z"

        }

*except* Exception *as* e:

*raise* HTTPException(*status\_code*=500, *detail*=f"Database health check failed: {str(e)}")

*# Sample data creation for testing*

@app.post("/sample-data")

def create\_sample\_data(*db*: Session = Depends(get\_db)):

    """Create sample posts for testing purposes"""

    sample\_posts = [

        {

            "title": "Getting Started with FastAPI",

            "content": "FastAPI is a modern, fast web framework for building APIs with Python 3.7+ based on standard Python type hints.",

            "published": True

        },

        {

            "title": "PostgreSQL with SQLAlchemy",

            "content": "Learn how to integrate PostgreSQL with SQLAlchemy ORM for robust database operations in your Python applications.",

            "published": True

        },

        {

            "title": "Building REST APIs",

            "content": "A comprehensive guide to building RESTful APIs with proper HTTP methods, status codes, and data validation.",

            "published": False

        },

        {

            "title": "Database Design Best Practices",

            "content": "Essential principles for designing efficient and scalable database schemas for modern applications.",

            "published": True

        },

        {

            "title": "API Testing with Postman",

            "content": "How to effectively test your APIs using Postman, including authentication, environment variables, and automated testing.",

            "published": True

        }

    ]

    created\_posts = []

*for* post\_data *in* sample\_posts:

        new\_post = models.Post(\*\*post\_data)

        db.add(new\_post)

        created\_posts.append(new\_post)

    db.commit()

*# Refresh all posts to get IDs*

*for* post *in* created\_posts:

        db.refresh(post)

*return* {

        "message": f"Created {len(created\_posts)} sample posts",

        "posts": created\_posts

    }

@app.post("/sample-users")

def create\_sample\_users(*db*: Session = Depends(get\_db)):

    """Create sample users for testing purposes"""

    sample\_users = [

        {

            "email": "john.doe@example.com",

            "password": "hashed\_password\_123",  *# In production, this would be properly hashed*

            "first\_name": "John",

            "last\_name": "Doe",

            "is\_active": True,

            "is\_verified": True

        },

        {

            "email": "jane.smith@example.com",

            "password": "hashed\_password\_456",

            "first\_name": "Jane",

            "last\_name": "Smith",

            "is\_active": True,

            "is\_verified": False

        },

        {

            "email": "admin@example.com",

            "password": "admin\_hashed\_password",

            "first\_name": "Admin",

            "last\_name": "User",

            "is\_active": True,

            "is\_verified": True

        },

        {

            "email": "test.user@example.com",

            "password": "test\_password\_hash",

            "first\_name": "Test",

            "last\_name": "User",

            "is\_active": False,

            "is\_verified": False

        },

        {

            "email": "developer@example.com",

            "password": "dev\_password\_hash",

            "first\_name": "Developer",

            "last\_name": "Name",

            "is\_active": True,

            "is\_verified": True

        }

    ]

    created\_users = []

*for* user\_data *in* sample\_users:

*# Check if user already exists*

        existing\_user = db.query(models.User).filter(models.User.email == user\_data["email"]).first()

*if* existing\_user:

*continue*  *# Skip if user already exists*

        new\_user = models.User(\*\*user\_data)

        db.add(new\_user)

        created\_users.append(new\_user)

    db.commit()

*# Refresh all users to get IDs*

*for* user *in* created\_users:

        db.refresh(user)

*return* {

        "message": f"Created {len(created\_users)} sample users",

        "users": created\_users

    }

*# Function to create sample data directly (for terminal use)*

def create\_sample\_data\_direct():

    """Create sample posts directly in the database"""

*from* sqlalchemy.orm *import* sessionmaker

*# Create session*

    SessionLocal = sessionmaker(*autocommit*=False, *autoflush*=False, *bind*=engine)

    db = SessionLocal()

*# Sample data*

    sample\_posts = [

        {

            'title': 'Getting Started with FastAPI',

            'content': 'FastAPI is a modern, fast web framework for building APIs with Python 3.7+ based on standard Python type hints.',

            'published': True

        },

        {

            'title': 'PostgreSQL with SQLAlchemy',

            'content': 'Learn how to integrate PostgreSQL with SQLAlchemy ORM for robust database operations in your Python applications.',

            'published': True

        },

        {

            'title': 'Building REST APIs',

            'content': 'A comprehensive guide to building RESTful APIs with proper HTTP methods, status codes, and data validation.',

            'published': False

        },

        {

            'title': 'Database Design Best Practices',

            'content': 'Essential principles for designing efficient and scalable database schemas for modern applications.',

            'published': True

        },

        {

            'title': 'API Testing with Postman',

            'content': 'How to effectively test your APIs using Postman, including authentication, environment variables, and automated testing.',

            'published': True

        }

    ]

*# Insert sample data*

*for* post\_data *in* sample\_posts:

        new\_post = models.Post(\*\*post\_data)

        db.add(new\_post)

*# Commit changes*

    db.commit()

    print('Sample posts inserted successfully!')

*# Close session*

    db.close()

def create\_sample\_users\_direct():

    """Create sample users directly in the database"""

*from* sqlalchemy.orm *import* sessionmaker

*# Create session*

    SessionLocal = sessionmaker(*autocommit*=False, *autoflush*=False, *bind*=engine)

    db = SessionLocal()

*# Sample user data*

    sample\_users = [

        {

            'email': 'john.doe@example.com',

            'password': 'hashed\_password\_123',

            'first\_name': 'John',

            'last\_name': 'Doe',

            'is\_active': True,

            'is\_verified': True

        },

        {

            'email': 'jane.smith@example.com',

            'password': 'hashed\_password\_456',

            'first\_name': 'Jane',

            'last\_name': 'Smith',

            'is\_active': True,

            'is\_verified': False

        },

        {

            'email': 'admin@example.com',

            'password': 'admin\_hashed\_password',

            'first\_name': 'Admin',

            'last\_name': 'User',

            'is\_active': True,

            'is\_verified': True

        },

        {

            'email': 'test.user@example.com',

            'password': 'test\_password\_hash',

            'first\_name': 'Test',

            'last\_name': 'User',

            'is\_active': False,

            'is\_verified': False

        },

        {

            'email': 'developer@example.com',

            'password': 'dev\_password\_hash',

            'first\_name': 'Developer',

            'last\_name': 'Name',

            'is\_active': True,

            'is\_verified': True

        }

    ]

*# Insert sample data*

*for* user\_data *in* sample\_users:

*# Check if user already exists*

        existing\_user = db.query(models.User).filter(models.User.email == user\_data['email']).first()

*if* existing\_user:

            print(f'User with email {user\_data["email"]} already exists, skipping...')

*continue*

        new\_user = models.User(\*\*user\_data)

        db.add(new\_user)

        print(f'Added user: {user\_data["email"]}')

*# Commit changes*

    db.commit()

    print('Sample users inserted successfully!')

*# Close session*

    db.close()

*# Clear all data (use with caution)*

@app.delete("/clear-all")

def clear\_all\_data(*db*: Session = Depends(get\_db)):

    """Clear all posts from the database (use with caution)"""

*try*:

        deleted\_count = db.query(models.Post).delete()

        db.commit()

*return* {

            "message": f"Deleted {deleted\_count} posts",

            "deleted\_count": deleted\_count

        }

*except* Exception *as* e:

        db.rollback()

*raise* HTTPException(*status\_code*=500, *detail*=f"Failed to clear data: {str(e)}")

*if* \_\_name\_\_ == "\_\_main\_\_":

*import* uvicorn

    uvicorn.run(app, *host*="0.0.0.0", *port*=8001)

**models.py file**

*from* sqlalchemy *import* Column, Integer, String, Boolean, column, TIMESTAMP, UniqueConstraint

*from* database *import* Base

*from* sqlalchemy.sql.expression *import* text

class Post(Base):

    \_\_tablename\_\_ = "posts"

    id = Column(Integer, *primary\_key*=True, *nullable*=False)

    title = Column(String, *nullable*=False)

    content = Column(String, *nullable*=False)

    published = Column(Boolean, *server\_default*='True', *nullable*=False)

    created\_at = Column(TIMESTAMP(*timezone*=True), *nullable*=False, *server\_default*=text('now()'))

class User(Base):

    \_\_tablename\_\_ = "users"

    id = Column(Integer, *primary\_key*=True, *nullable*=False, *unique*=True)

    email = Column(String, *nullable*=False, *unique*=True)

    password = Column(String, *nullable*=False)

    first\_name = Column(String, *nullable*=True)

    last\_name = Column(String, *nullable*=True)

    is\_active = Column(Boolean, *server\_default*='True', *nullable*=False)

    is\_verified = Column(Boolean, *server\_default*='False', *nullable*=False)

    created\_at = Column(TIMESTAMP(*timezone*=True), *nullable*=False, *server\_default*=text('now()'))

    updated\_at = Column(TIMESTAMP(*timezone*=True), *nullable*=False, *server\_default*=text('now()'), *onupdate*=text('now()'))

*# Additional constraint to ensure email uniqueness at database level*

    \_\_table\_args\_\_ = (

        UniqueConstraint('email', *name*='unique\_user\_email'),

    )

**Database.py file**

*from* sqlalchemy *import* create\_engine

*from* sqlalchemy.ext.declarative *import* declarative\_base

*from* sqlalchemy.orm *import* sessionmaker

SQLALCHEMY\_DATABASE\_URL = 'postgresql://postgres:swagath12345@localhost/fastapi'

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

SessionLocal = sessionmaker(*autocommit*=False,*autoflush*=False,*bind*=engine)

Base = declarative\_base()

def get\_db():

    db = SessionLocal()

*try*:

*yield* db

*finally*:

        db.close()

def recreate\_tables():

    """Drop all tables and recreate them with the new schema"""

    Base.metadata.drop\_all(*bind*=engine)

    Base.metadata.create\_all(*bind*=engine)