- LABORATORY WORK REPORT #1 & #2
  - Lab 1: Environment Setup + First Project (Launch, DevTools, Logging, Port)
  - Lab 2: Advanced CRUD Operations, Database Relationships, pgAdmin Integration
  - 1. OBJECTIVE
  - 2. TECHNICAL SPECIFICATION
    - 2.1 Environment Requirements:
    - 2.2 Functional Requirements:
    - 2.3 Advanced Requirements:
  - 3. PROJECT ARCHITECTURE
    - 3.1 Project Structure:
    - 3.2 Technology Stack:
    - 3.3 Advanced Project Structure:
    - 3.4 Advanced Technology Stack:
  - 4. DATA MODEL
    - 4.1 SQLAlchemy Models:
      - Owner Entity:
      - Car Entity:
    - 4.2 Advanced SQLAlchemy Features:
      - Relationship Configuration:
      - Advanced Car Model:
    - 4.3 Table Relationships:
  - 5. ENVIRONMENT SETUP
    - 5.1 Installing Dependencies:
    - 5.2 Configuration (.env):
    - 5.3 Database Setup:
    - 5.4 Advanced Configuration:
    - 5.5 Database Optimization:
    - 5.6 pgAdmin Database Management:
  - 6. API IMPLEMENTATION
    - 6.1 FastAPI Endpoints:
      - Application Status:
      - CRUD Operations for Cars:
      - CRUD Operations for Owners:
      - Advanced Queries:
    - 6.2 Advanced CRUD Operations:
      - Car CRUD:

        - Owner CRUD:
- 7. LOGGING AND DEBUGGING
    - 7.1 Logging Configuration:
    - 7.2 DevTools and Automatic Reload:
    - 7.3 Swagger UI:
    - 7.4 Advanced Logging:
    - 7.5 pgAdmin Monitoring:
- 8. TESTING
    - 8.1 Testing Core Functions:
    - 8.2 Test Request Examples:
        - Creating Owner:
        - Creating Car:
        - Advanced Search:
    - 8.3 Advanced Testing:
    - 8.4 Test Examples:
        - Advanced Car Search:
        - Owner Statistics:
        - Query Testing:
- 9. RESULTS
    - 9.1 Achieved Goals:
    - 9.2 Key Implementation Features:
    - 9.3 Advanced Features:
- 10. CONCLUSION
- 11. APPENDICES
    - 11.1 requirements.txt file:
    - 11.2 Launch script (run.ps1):
    - 11.3 Advanced requirements.txt:
    - 11.4 Configuration Example:

LABORATORY WORK REPORT #1 & #2

Lab 1: Environment Setup + First Project (Launch, DevTools, Logging, Port)

Lab 2: Advanced CRUD Operations, Database Relationships, pgAdmin Integration

---

1. OBJECTIVE

Lab 1: Master the creation of a web application in Python using FastAPI, setting up a development environment, connecting to a PostgreSQL database, creating entities and CRUD operations.

Lab 2: Learn advanced CRUD operations, database relationship management, custom query implementation, and database administration through pgAdmin.

---

2. TECHNICAL SPECIFICATION

2.1 Environment Requirements:

- Python 3.11+
- FastAPI web framework
- PostgreSQL database
- pgAdmin for database management
- Uvicorn ASGI server
- SQLAlchemy ORM

2.2 Functional Requirements:

- Creating REST API with CRUD operations
- Working with two related entities: Owner and Car
- Advanced queries with filtering and sorting
- DEBUG level logging
- Automatic reload during development

2.3 Advanced Requirements:

- Entity relationship mapping (One-to-Many)
- Advanced CRUD operations with custom queries
- Database schema management through pgAdmin
- Custom query methods with filtering and sorting
- Pagination and sorting support
- Error handling and validation

---

3. PROJECT ARCHITECTURE

3.1 Project Structure:

```
lab-fullstack/
├── app/
│   ├── __init__.py
│   ├── main.py          # Main FastAPI application
│   ├── models.py        # SQLAlchemy models
│   ├── schemas.py       # Pydantic schemas
│   ├── crud.py          # CRUD operations
│   └── db.py            # Database connection
├── sql/
│   └── init.sql         # SQL initialization script
├── requirements.txt     # Python dependencies
├── .env                 # Environment variables
├── .env.example         # Configuration example
├── run.ps1              # Launch script
└── README.md            # Documentation
```

3.2 Technology Stack:

- **Backend**: Python 3.11, FastAPI, Uvicorn
- **Database**: PostgreSQL, SQLAlchemy 2.x
- **ORM**: SQLAlchemy with Declarative Base
- **Validation**: Pydantic v2
- **Environment**: python-dotenv
- **Database Client**: psycopg (PostgreSQL adapter)

3.3 Advanced Project Structure:

```
lab-fullstack/
├── app/
│   ├── __init__.py
│   ├── main.py          # Main FastAPI application
│   ├── models.py        # SQLAlchemy models
│   ├── schemas.py       # Pydantic schemas
│   ├── crud.py          # Advanced CRUD operations
│   └── db.py            # Database connection
├── sql/
│   ├── init.sql         # SQL initialization script
│   └── queries.sql      # Custom SQL queries
├── tests/
│   ├── test_crud.py     # CRUD operation tests
│   └── test_queries.py  # Query tests
├── requirements.txt     # Python dependencies
├── .env                 # Environment variables
├── .env.example         # Configuration example
├── run.ps1              # Launch script
└── README.md            # Documentation
```

3.4 Advanced Technology Stack:

- **Backend**: Python 3.11, FastAPI, Uvicorn
- **Database**: PostgreSQL, SQLAlchemy 2.x
- **ORM**: SQLAlchemy with Declarative Base
- **Validation**: Pydantic v2
- **Environment**: python-dotenv
- **Database Client**: psycopg (PostgreSQL adapter)
- **Testing**: pytest, httpx
- **Database Management**: pgAdmin

---

4. DATA MODEL

4.1 SQLAlchemy Models:

Owner Entity:

```python
class Owner(Base):
    __tablename__ = "owner"
    ownerid: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    firstname: Mapped[str] = mapped_column(String(100))
    lastname: Mapped[str] = mapped_column(String(100))
    cars: Mapped[list["Car"]] = relationship(
        back_populates="owner",
        cascade="all, delete-orphan"
    )
```

Car Entity:

```python
class Car(Base):
    __tablename__ = "car"
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    brand: Mapped[str] = mapped_column(String(100))
    model: Mapped[str] = mapped_column(String(100))
    color: Mapped[str] = mapped_column(String(40))
    registrationNumber: Mapped[str] = mapped_column(String(40))
    modelYear: Mapped[int] = mapped_column(Integer)
    price: Mapped[int] = mapped_column(Integer)
    owner_id: Mapped[int] = mapped_column(ForeignKey("owner.ownerid"))
    owner: Mapped["Owner"] = relationship(back_populates="cars")
```

4.2 Advanced SQLAlchemy Features:

Relationship Configuration:

```python
# Advanced relationship configuration
class Owner(Base):
    __tablename__ = "owner"
    ownerid: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    firstname: Mapped[str] = mapped_column(String(100), nullable=False)
    lastname: Mapped[str] = mapped_column(String(100), nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime, default=datetime.utcnow)
    cars: Mapped[list["Car"]] = relationship(
        back_populates="owner",
        cascade="all, delete-orphan",
        lazy="select"
    )
```

Advanced Car Model:

```
class Car(Base):
    __tablename__ = "car"
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    brand: Mapped[str] = mapped_column(String(100), nullable=False)
    model: Mapped[str] = mapped_column(String(100), nullable=False)
    color: Mapped[str] = mapped_column(String(40), nullable=False)
    registrationNumber: Mapped[str] = mapped_column(String(40), unique=True,
nullable=False)
    modelYear: Mapped[int] = mapped_column(Integer, nullable=False)
    price: Mapped[int] = mapped_column(Integer, nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime, default=datetime.utcnow)
    owner_id: Mapped[int] = mapped_column(ForeignKey("owner.ownerid"),
nullable=False)
    owner: Mapped["Owner"] = relationship(back_populates="cars", lazy="select")
```

4.3 Table Relationships:

- **One-to-Many**: One owner can have multiple cars
- **Cascade Delete**: When owner is deleted, all their cars are deleted
- **Lazy Loading**: Cars are loaded on demand to optimize performance
- **Unique Constraints**: Registration numbers must be unique
- **Timestamps**: Automatic creation timestamps for audit trails

---

5. ENVIRONMENT SETUP

5.1 Installing Dependencies:

```
# Create virtual environment
py -m venv .venv

# Activate environment
.\.venv\Scripts\Activate.ps1

# Install dependencies
pip install -r requirements.txt
```

5.2 Configuration (.env):

```
APP_PORT=8000
DB_URL=postgresql+psycopg://postgres:YOUR_PASSWORD@localhost:5432/cardb
APP_NAME=Lab1 FastAPI
APP_VERSION=1.0.0
```

5.3 Database Setup:

- Create `cardb` database in PostgreSQL
- Configure user and access rights
- Initialize tables through SQLAlchemy

5.4 Advanced Configuration:

```
# Advanced .env configuration
APP_PORT=8000
DB_URL=postgresql+psycopg://postgres:YOUR_PASSWORD@localhost:5432/cardb
APP_NAME=Lab1 FastAPI
APP_VERSION=1.0.0
LOG_LEVEL=DEBUG
ENABLE_SWAGGER=true
CORS_ORIGINS=http://localhost:3000,http://localhost:8080
```

5.5 Database Optimization:

```python
# Database connection optimization
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine(
    DATABASE_URL,
    poolclass=QueuePool,
    pool_size=10,
    max_overflow=20,
    pool_pre_ping=True,
    echo=True  # For development
)
```

5.6 pgAdmin Database Management:

- Connect to PostgreSQL server through pgAdmin
- Create and manage `cardb` database
- Monitor table creation and data insertion
- Execute custom SQL queries for testing

---

6. API IMPLEMENTATION

6.1 FastAPI Endpoints:

Application Status:

- `GET /hello` - Simple greeting
- `GET /api/status` - Application status with version

CRUD Operations for Cars:

- `GET /cars` - Get all cars
- `GET /cars/{id}` - Get car by ID
- `POST /cars` - Create new car
- `PUT /cars/{id}` - Update car
- `DELETE /cars/{id}` - Delete car

CRUD Operations for Owners:

- `GET /owners` - Get all owners
- `GET /owners/{id}` - Get owner by ID
- `POST /owners` - Create new owner
- `PUT /owners/{id}` - Update owner
- `DELETE /owners/{id}` - Delete owner

Advanced Queries:

- `GET /cars/search/brand/{brand}` - Search by brand
- `GET /cars/search/color/{color}` - Search by color
- `GET /cars/search/year/{year}` - Search by model year
- `GET /cars/search/price-range` - Search by price range
- `GET /cars/search/owner/{owner_id}` - Search by owner

- `POST /cars/search` - Advanced search with filtering
- `GET /owners/search/{search_term}` - Simple search by any field
- `POST /owners/search` - Advanced search with filtering
- `GET /cars/statistics` - Car statistics
- `GET /owners/statistics` - Owner statistics

6.2 Advanced CRUD Operations:

Car CRUD:

```
class CarCRUD:
    @staticmethod
    def get_cars_by_price_range(db: Session, min_price: int, max_price: int):
        return db.query(Car).filter(Car.price >= min_price, Car.price <=
max_price).all()

    @staticmethod
    def get_expensive_cars(db: Session, threshold: int = 50000):
        return db.query(Car).filter(Car.price > threshold).all()

    @staticmethod
    def get_cars_by_multiple_criteria(db: Session, brand: str = None, color: str =
None, year: int = None):
        query = db.query(Car)
        if brand: query = query.filter(Car.brand.ilike(f"%{brand}%"))
        if color: query = query.filter(Car.color.ilike(f"%{color}%"))
        if year: query = query.filter(Car.modelYear == year)
        return query.all()
```

Owner CRUD:

```
class OwnerCRUD:
    @staticmethod
    def get_owners_with_car_count(db: Session):
        return db.query(Owner,
func.count(Car.id).label('car_count')).outerjoin(Car).group_by(Owner.ownerid).all()

    @staticmethod
    def search_owners_advanced(db: Session, search_term: str):
        return db.query(Owner).filter(
            or_(Owner.firstname.ilike(f"%{search_term}%"), Owner.lastname.ilike(f"%
{search_term}%"))
        ).all()
```

7. LOGGING AND DEBUGGING

7.1 Logging Configuration:

```
import logging

# Logging setup
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s %(levelname)s %(name)s: %(message)s',
    handlers=[
        logging.FileHandler('app.log'),
        logging.StreamHandler()
    ]
)

log = logging.getLogger(__name__)
```

7.2 DevTools and Automatic Reload:

```
# Launch with automatic reload
uvicorn app.main:app --reload --host 127.0.0.1 --port 8000
```

7.3 Swagger UI:

- Automatic API documentation available at: `http://127.0.0.1:8000/docs`
- ReDoc documentation: `http://127.0.0.1:8000/redoc`

7.4 Advanced Logging:

```
import logging
from logging.handlers import RotatingFileHandler

def setup_logging():
    file_handler = RotatingFileHandler('app.log', maxBytes=10*1024*1024,
backupCount=5)
    console_handler = logging.StreamHandler()
    logging.basicConfig(
        level=logging.DEBUG,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[file_handler, console_handler]
    )
```

7.5 pgAdmin Monitoring:

- Database monitoring and query execution
- Table structure management

---

8. TESTING

8.1 Testing Core Functions:

✅ Application creation and launch
✅ PostgreSQL database connection
✅ Table creation through SQLAlchemy
✅ CRUD operations for both entities
✅ Advanced queries with filtering
✅ Search by various criteria
✅ Statistical queries
✅ Logging and debugging

8.2 Test Request Examples:

Creating Owner:

```
POST /owners
{
  "firstname": "John",
  "lastname": "Johnson"
}
```

Creating Car:

```
POST /cars
{
  "brand": "Ford",
  "model": "Mustang",
  "color": "Red",
  "registrationNumber": "ADF-1121",
  "modelYear": 2023,
  "price": 59000,
  "owner_id": 1
}
```

Advanced Search:

```
POST /cars/search
{
  "brand": "Ford",
  "color": "Red",
  "minPrice": 50000,
  "sort_by": "price",
  "sort_order": "desc"
}
```

8.3 Advanced Testing:

✅ Complex queries with joins
✅ Performance monitoring
✅ Error handling and validation
✅ Database relationship testing

8.4 Test Examples:

Advanced Car Search:

```
POST /cars/search/advanced
{
  "brand": "Ford",
  "color": "Red",
  "minPrice": 50000,
  "maxPrice": 100000,
  "year": 2023,
  "sort_by": "price",
  "sort_order": "desc",
  "limit": 10
}
```

Owner Statistics:

```
GET /owners/statistics
Response:
{
  "total_owners": 5,
  "owners_with_cars": 3,
  "total_cars": 8,
  "average_cars_per_owner": 2.67
}
```

Query Testing:

```
# Test advanced queries
expensive_cars = CarCRUD.get_expensive_cars(db, threshold=50000)
ford_red_cars = CarCRUD.get_cars_by_multiple_criteria(db, brand="Ford",
color="Red")
owner_stats = OwnerCRUD.get_owners_with_car_count(db)
```

---

9. RESULTS

9.1 Achieved Goals:

- ✅ Development environment setup with Python 3.11 and FastAPI
- ✅ PostgreSQL database connection with pgAdmin
- ✅ SQLAlchemy models created with proper relationships
- ✅ All CRUD operations implemented
- ✅ Advanced queries with filtering and sorting added
- ✅ DEBUG level logging configured
- ✅ Automatic reload during development implemented
- ✅ Automatic API documentation created

9.2 Key Implementation Features:

- Using SQLAlchemy 2.x with modern `Mapped` syntax
- Pydantic v2 for data validation
- Cascade delete when owner is removed
- Case-insensitive search using `ILIKE`
- Pagination and sorting support
- Error handling with detailed messages

9.3 Advanced Features:

- ✅ Complex queries with joins and aggregations
- ✅ Performance monitoring and logging
- ✅ Database optimization and connection pooling
- ✅ Advanced error handling and validation

---

10. CONCLUSION

Lab 1: During the first laboratory work, a web application on FastAPI with PostgreSQL connection was successfully created. All required functions have been implemented: CRUD operations, advanced queries, logging, and automatic API documentation. The application is ready for further development and can serve as a foundation for more complex projects.

Lab 2: The second laboratory work focused on advanced CRUD operations, complex database queries, performance optimization, and comprehensive database management. Advanced SQLAlchemy features were implemented, including custom query methods, relationship optimization, performance monitoring, and sophisticated error handling.

Both laboratory works demonstrate proficiency in modern Python web development with FastAPI and SQLAlchemy. The applications work stably, and the code is written following best practices for Python development. The projects showcase advanced database integration techniques, performance optimization, and comprehensive API development.

---

11. APPENDICES

11.1 requirements.txt file:

```
fastapi==0.115.*
uvicorn[standard]==0.30.*
python-dotenv==1.0.*
sqlalchemy==2.0.*
pydantic==2.*
psycopg[binary]==3.2.*
```

11.2 Launch script (run.ps1):

```
param([int]$Port)

# 1) Create venv if missing
if (-not (Test-Path .venv)) { py -m venv .venv }

# 2) Activate venv
.\.venv\Scripts\Activate.ps1

# 3) Install deps (idempotent)
pip install -r requirements.txt

# 4) Determine port: prefer CLI arg > .env > default 8000
$resolvedPort = 8000
if ($Port) { $resolvedPort = $Port }
elseif (Test-Path .env) {
  $line = Get-Content .env | Where-Object { $_ -match '^APP_PORT=' } | Select-
Object -First 1
  if ($line) { $resolvedPort = ($line -split '=',2)[1] }
}

# 5) Run dev server with auto-reload
uvicorn app.main:app --reload --port $resolvedPort
```

11.3 Advanced requirements.txt:

```
fastapi==0.115.*
uvicorn[standard]==0.30.*
python-dotenv==1.0.*
sqlalchemy==2.0.*
```

```
pydantic==2.*
psycopg[binary]==3.2.*
pytest==7.4.*
httpx==0.25.*
pytest-asyncio==0.21.*
```

11.4 Configuration Example:

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    app_name: str = "Lab1 FastAPI"
    app_port: int = 8000
    db_url: str
    log_level: str = "DEBUG"

    class Config:
        env_file = ".env"

settings = Settings()
```