TODO REST API Tutorial

Python FastAPI Implementation with SQLite

A Comprehensive Guide to Building Secure RESTful APIs

Table of Contents

1. Overview & Architecture

Understanding the TODO REST API System

Project Overview

What is this API?

- A secure REST API for managing TODO lists and tasks
- Built with Python FastAPI and SQLite database
- JWT-based authentication with token blacklisting
- 14+ endpoints covering authentication, lists, and tasks

Key Features:

- ✅ User authentication (signup, login, logout)
- ✅ CRUD operations for lists and tasks
- ✅ Bearer token authentication
- ✅ Argon2 password hashing (no length limits)
- ✅ Comprehensive input validation

Technology Stack

Backend Framework:

- **FastAPI** - Modern, fast web framework for Python
- **Uvicorn** - ASGI server for FastAPI
- **Pydantic v2** - Data validation and serialization

Database Layer:

- **SQLite** - File-based database (no server needed)
- **SQLAlchemy 2.0** - Modern ORM with async support
- **Alembic** - Database migrations

Security:

- **python-jose** - JWT token handling
- [illegible]

Project Structure

```
python-version/
├── app/
│   ├── main.py              # FastAPI application
│   ├── config.py            # Settings management
│   ├── database.py          # DB connection & session
│   ├── models/              # SQLAlchemy models
│   ├── schemas/             # Pydantic schemas
│   ├── routers/             # API endpoints
│   ├── services/            # Business logic
│   └── utils/               # Helper functions
├── tests/                   # Test suite
├── docker/                  # Docker configs
├── docs/                    # Documentation
└── scripts/                 # Utility scripts
```

Key Files:

- `main.py` – Application entry point with middleware

Database Design

Core Tables:

1. **users**
   - UUID primary key
   - Unique username and email
   - Password hash (Argon2)
   - Timestamps
2. **lists**
   - UUID primary key
   - Title and description
   - Optional user ownership
   - Timestamps

2. Authentication Flow

How JWT Authentication Works

JWT Authentication Process

1. User Registration (Signup):

```
# User provides credentials
{
  "username": "alice",
  "email": "alice@example.com",
  "password": "securepassword"
}

# Server response
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGc...",
  "user": { "id": "...", "username": "alice", ... }
}
```

2. Token Storage:

- Store JWT in HTTP-only cookies (recommended)

Token Lifecycle

Creation:

```
def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(hours=1)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm="HS256")
    return encoded_jwt
```

Validation:

```
def get_current_user(token: str = Depends(oauth2_scheme)):
    # 1. Decode token
    # 2. Check expiration
    # 3. Verify signature
    # 4. Check blacklist
    # 5. Get user from database
    return user
```

Security Features

Password Security:

- Argon2 hashing (winner of Password Hashing Competition)
- No 72-byte limit (unlike bcrypt)
- Configurable work factors
- Secure random salt generation

Token Security:

- HS256 algorithm with strong secrets
- 1-hour expiration (configurable)
- Token blacklisting prevents reuse
- Secure token transmission (HTTPS required in production)

Input Validation:

3. API Endpoints Overview

Complete REST API Structure

Endpoint Categories

Authentication (4 endpoints):

- `POST /api/v1/auth/signup` - User registration
- `POST /api/v1/auth/login` - User authentication
- `POST /api/v1/auth/logout` - Token blacklisting
- `GET /api/v1/users/profile` - User profile

Lists (5 endpoints):

- `GET /api/v1/lists` - Get all lists
- `POST /api/v1/lists` - Create list
- `GET /api/v1/lists/{id}` - Get list by ID
- `PATCH /api/v1/lists/{id}` - Update list
- `DELETE /api/v1/lists/{id}` - Delete list

HTTP Methods & Status Codes

Standard REST Methods:

- `GET` - Retrieve resources
- `POST` - Create new resources
- `PATCH` - Update existing resources
- `DELETE` - Remove resources

Common Status Codes:

- `200 OK` - Success
- `201 Created` - Resource created
- `204 No Content` - Success, no response body
- `400 Bad Request` - Invalid request
- `401 Unauthorized` - Authentication required

Request/Response Format

All requests use JSON:

```
{
  "title": "My Task",
  "description": "Task description",
  "priority": "high",
  "categories": ["work", "urgent"]
}
```

All responses use JSON:

```
{
  "id": "uuid-here",
  "title": "My Task",
  "description": "Task description",
  "completed": false,
  "priority": "high",
  "categories": ["work", "urgent"],
  "createdAt": "2025-12-01T10:00:00Z"
```

4. Detailed API Examples

Step-by-Step API Usage

Authentication Examples

User Registration

Request:

```
curl -X POST http://localhost:8000/api/v1/auth/signup \
  -H "Content-Type: application/json" \
  -d '{
    "username": "alice",
    "email": "alice@example.com",
    "password": "securepassword123"
  }'
```

Response (201):

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGc...",
  "user": {
    "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
```

User Login

Request:

```
curl -X POST http://localhost:8000/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "alice",
    "password": "securepassword123"
  }'
```

Response (200):

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGc...",
  "user": {
    "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
    "username": "alice",
    "email": "alice@example.com",
    "createdAt": "2025-12-01T10:00:00Z"
  }
```

Get User Profile

Request:

```
TOKEN="your-jwt-token-here"
curl -H "Authorization: Bearer $TOKEN" \
     http://localhost:8000/api/v1/users/profile
```

Response (200):

```
{
  "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "username": "alice",
  "email": "alice@example.com",
  "createdAt": "2025-12-01T10:00:00Z",
  "updatedAt": null
}
```

List Management Examples

Create a List

Request:

```
curl -X POST http://localhost:8000/api/v1/lists \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Weekly Groceries",
    "description": "Shopping list for the week"
  }'
```

Response (201):

```
{
  "id": "b2b3c4d5-e6f7-8901-bcde-f23456789012",
  "title": "Weekly Groceries",
  "description": "Shopping list for the week",
  "createdAt": "2025-12-01T10:15:00Z",
```

Get All Lists

Request:

```
curl http://localhost:8000/api/v1/lists
```

Response (200):

```
[
  {
    "id": "b2b3c4d5-e6f7-8901-bcde-f23456789012",
    "title": "Weekly Groceries",
    "description": "Shopping list for the week",
    "createdAt": "2025-12-01T10:15:00Z",
    "updatedAt": null
  }
]
```

Update a List

Request:

```
LIST_ID="b2b3c4d5-e6f7-8901-bcde-f23456789012"
curl -X PATCH http://localhost:8000/api/v1/lists/$LIST_ID \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Monthly Groceries",
    "description": "Updated shopping list"
  }'
```

Response (200):

```
{
  "id": "b2b3c4d5-e6f7-8901-bcde-f23456789012",
  "title": "Monthly Groceries",
  "description": "Updated shopping list",
  "createdAt": "2025-12-01T10:15:00Z",
  "updatedAt": "2025-12-01T10:30:00Z"
}
```

Task Management Examples

Create a Task

Request:

```
LIST_ID="b2b3c4d5-e6f7-8901-bcde-f23456789012"
curl -X POST http://localhost:8000/api/v1/lists/$LIST_ID/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Buy organic milk",
    "description": "Get 2% organic milk from Whole Foods",
    "priority": "high",
    "dueDate": "2025-12-01T18:00:00Z",
    "categories": ["groceries", "dairy", "organic"]
  }'
```

Response (201):

```
{
```

Get Tasks in a List

Request:

```
LIST_ID="b2b3c4d5-e6f7-8901-bcde-f23456789012"
curl http://localhost:8000/api/v1/lists/$LIST_ID/tasks
```

Response (200):

```
[
  {
    "id": "c3c4d5e6-f7g8-9012-cdef-g34567890123",
    "listId": "b2b3c4d5-e6f7-8901-bcde-f23456789012",
    "title": "Buy organic milk",
    "description": "Get 2% organic milk from Whole Foods",
    "completed": false,
    "priority": "high",
    "categories": ["groceries", "dairy", "organic"],
    "dueDate": "2025-12-01T18:00:00Z",
    "createdAt": "2025-12-01T10:45:00Z",
    "updatedAt": null
```

Update a Task

Request:

```
TASK_ID="c3c4d5e6-f7g8-9012-cdef-g34567890123"
curl -X PATCH http://localhost:8000/api/v1/tasks/$TASK_ID \
  -H "Content-Type: application/json" \
  -d '{
    "completed": true,
    "priority": "medium"
  }'
```

Response (200):

```
{
  "id": "c3c4d5e6-f7g8-9012-cdef-g34567890123",
  "listId": "b2b3c4d5-e6f7-8901-bcde-f23456789012",
  "title": "Buy organic milk",
  "description": "Get 2% organic milk from Whole Foods",
  "completed": true,
  "priority": "medium",
```

Health Check

Request:

```
curl http://localhost:8000/api/v1/health
```

Response (200):

```
{
  "status": "healthy",
  "timestamp": "2025-12-01T11:15:00Z",
  "service": "TODO REST API",
  "version": "1.0.0",
  "checks": {
    "database": {
      "status": "healthy",
      "message": "Database connection successful"
    },
    "python": {
      "status": "healthy",
      "version": "3.11"
    },
    "disk": {
      "status": "healthy",
      "free_space_mb": 146645.1,
```

5. Security Features

Built-in Security Measures

Password Security

Argon2 Hashing:

```python
# No 72-byte limit like bcrypt
pwd_context = CryptContext(
    schemes=["argon2"],
    deprecated="auto"
)

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

Benefits:

- Memory-hard algorithm (resistant to GPU attacks)
- Configurable time/memory cost
- No length restrictions

JWT Token Security

Secure Token Handling:

- HS256 algorithm with strong secrets
- Configurable expiration (default: 1 hour)
- Token blacklisting on logout
- Automatic cleanup of expired tokens

Token Blacklist Table:

```
CREATE TABLE token_blacklist (
    id TEXT PRIMARY KEY,
    token TEXT NOT NULL,
    expires_at TIMESTAMP NOT NULL
);
```

Input Validation

Pydantic v2 Validation:

```python
class TaskCreate(BaseModel):
    title: str = Field(..., min_length=1, max_length=255)
    description: Optional[str] = Field(None, max_length=2000)
    priority: Optional[Literal["low", "medium", "high"]] = "medium"
    categories: Optional[List[str]] = Field(None, max_length=10)

    @field_validator('title')
    @classmethod
    def title_not_empty(cls, v):
        if not v or not v.strip():
            raise ValueError('Title cannot be empty')
        return v.strip()
```

Validation Features:

- Type checking and conversion

SQL Injection Prevention

SQLAlchemy ORM Protection:

```
# Automatic parameterization
user = db.query(User).filter(User.username == username).first()

# Never do this (vulnerable):
# query = f"SELECT * FROM users WHERE username = '{username}'"
```

Additional Protections:

- No raw SQL queries in application code
- Prepared statements for all database operations
- Input sanitization at schema level
- Foreign key constraints

Rate Limiting

Nginx Configuration:

```
limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;
limit_req zone=api burst=20 nodelay;

location /api/ {
    limit_req zone=api;
    proxy_pass http://fastapi;
}
```

Protection Against:

- Brute force attacks
- DoS attacks
- API abuse
- Resource exhaustion

6. Testing & Deployment

Quality Assurance and Production Deployment

Testing Strategy

Test Categories:

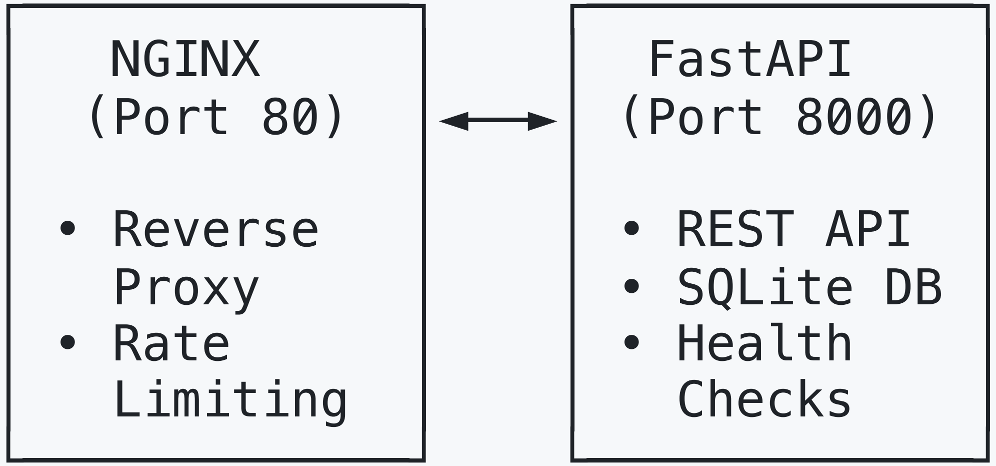
- Unit tests for individual functions
- Integration tests for API endpoints
- Authentication flow testing
- Database integration testing
- Error condition testing

Test Framework:

```
# pytest with async support
@pytest.mark.asyncio
async def test_create_task(client, db_session):
    # Test task creation
    response = client.post("/api/v1/lists/{list_id}/tasks", json=task_data)
    assert response.status_code == 201
```

Docker Deployment

Docker Architecture:



Deployment Steps:

```
# Build and run
docker-compose up -d

# Check health
curl http://localhost/api/v1/health
```

Production Checklist

Security:

- [ ] Change JWT_SECRET to strong random value
- [ ] Set DEBUG_MODE=false
- [ ] Enable HTTPS with SSL certificates
- [ ] Configure proper CORS settings
- [ ] Set secure file permissions

Performance:

- [ ] Enable connection pooling
- [ ] Configure database indexes
- [ ] Set appropriate rate limits
- [ ] Enable response compression

Development Workflow

Local Development:

```
# Install dependencies
uv sync

# Run development server
uv run uvicorn app.main:app --reload

# Run tests
uv run pytest --cov=app

# Format code
uv run black app/

# Lint code
uv run ruff check app/
```

Code Quality:

Common Issues & Solutions

Database Connection Issues:

```
# Check database file
ls -la data/todo.db

# Reset database
rm data/todo.db
# Restart application to recreate tables
```

Import Errors:

```
# Reinstall dependencies
uv sync --force

# Check Python version
python --version  # Should be 3.11+
```

Port Conflicts:

Next Steps

Enhancement Ideas:

1. **PostgreSQL Support** - For production scaling
2. **Redis Caching** - For improved performance
3. **API Versioning** - Support multiple API versions
4. **WebSocket Support** - Real-time task updates
5. **Background Jobs** - Email notifications, reminders
6. **Admin Panel** - User management interface
7. **API Documentation** - Enhanced OpenAPI specs
8. **Monitoring** - Application metrics and alerting

Learning Outcomes:

- REST API design principles

This tutorial provides a comprehensive guide to building secure, scalable REST APIs with Python FastAPI. The implementation demonstrates modern development practices suitable for both learning and production use.