TODO REST API Tutorial

PHP 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

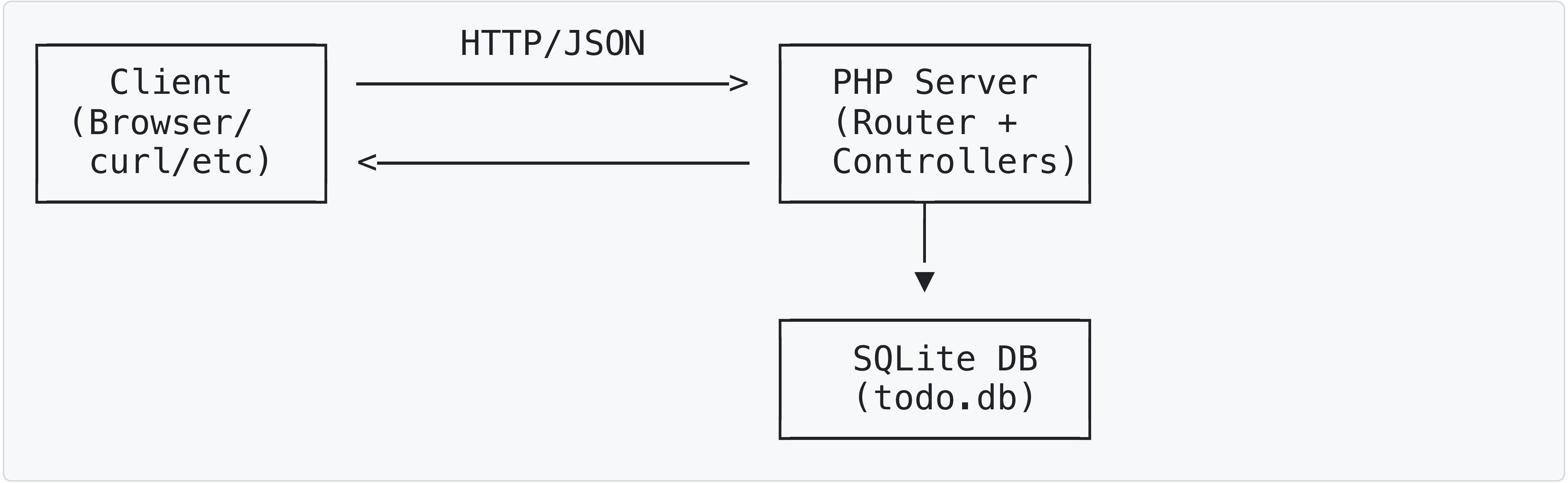
What is this API?

- A secure REST API for managing TODO lists and tasks
- Built with PHP 8.1+ 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
- ✅ SQL injection prevention
- ✅ XSS protection

Architecture



Database Tables:

- `users` - User accounts (username, email, password hash)
- `lists` - TODO lists (name, description)
- `tasks` - Individual tasks (title, description, completed, etc.)

Tech Stack

Component	Technology	Purpose
Language	PHP 8.1+	Server-side logic
Database	SQLite	Data persistence
Authentication	JWT	Stateless auth tokens
Password	bcrypt (cost 12)	Secure password hashing
Web Server	PHP Built-in / NGINX	HTTP handling
Testing	PHPUnit	Unit testing
API Testing	Bruno / curl	Manual testing

2. Authentication Flow

JWT-based Authentication with Token Blacklisting

Authentication Overview

What is JWT?

- **J**SON **W**eb **T**oken - industry standard for secure tokens
- Contains user info + expiration + signature
- Stateless (no server-side sessions needed)
- Sent as `Authorization: Bearer <token>` header

Our Implementation:

- Tokens expire after **1 hour** (configurable)
- Passwords hashed with **bcrypt** (cost factor 12)
- Tokens **blacklisted on logout** for security
- Protected endpoints require valid, non-blacklisted token

Authentication Endpoints

Endpoint	Method	Protected	Description
`/api/v1/auth/signup`	POST	No	Create new account
`/api/v1/auth/login`	POST	No	Login and get token
`/api/v1/auth/logout`	POST	**Yes**	Logout and blacklist token
`/api/v1/users/profile`	GET	**Yes**	Get current user info

Protected = Requires `Authorization: Bearer <token>` header

Signup Example

Request:

```
curl -X POST http://localhost:8000/api/v1/auth/signup \
  -H "Content-Type: application/json" \
  -d '{
    "username": "alice",
    "email": "alice@example.com",
    "password": "password123"
  }'
```

Response (201 Created):

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
  "user": {
    "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
    "username": "alice",
    "email": "alice@example.com",
    "createdAt": "2025-11-07T10:00:00Z"
```

Signup - Validation Rules

Username:

- Required, 3-50 characters
- Must be unique
- Trimmed of whitespace

Email:

- Required, valid email format
- Must be unique
- Trimmed of whitespace

Password:

- Required, minimum 8 characters

Login Example

Request:

```
curl -X POST http://localhost:8000/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "alice",
    "password": "password123"
  }'
```

Response (200 OK):

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
  "user": {
    "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
    "username": "alice",
    "email": "alice@example.com",
    "createdAt": "2025-11-07T10:00:00Z"
```

Get User Profile Example

Request (with Bearer token):

```
TOKEN="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."

curl -X GET http://localhost:8000/api/v1/users/profile \
  -H "Authorization: Bearer $TOKEN"
```

Response (200 OK):

```
{
  "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "username": "alice",
  "email": "alice@example.com",
  "createdAt": "2025-11-07T10:00:00Z",
  "updatedAt": null
}
```

Error Responses:

Logout Example

Request:

```
TOKEN="your-jwt-token-here"

curl -X POST http://localhost:8000/api/v1/auth/logout \
  -H "Authorization: Bearer $TOKEN"
```

Response (204 No Content):

- Empty body
- Token is now **blacklisted** and cannot be reused

What happens:

1. Token is added to `token_blacklist` database
2. Future requests with this token will get `401 Unauthorized`

Token Blacklisting - Why?

Problem Without Blacklisting:

```
User logs in  ——> Gets token (valid 1 hour)
User logs out ——> Client deletes token
❌ Problem: If attacker copied the token,
            they can still use it!
```

Solution With Blacklisting:

```
User logs in  ——> Gets token (valid 1 hour)
User logs out ——> Token added to blacklist DB
✅ Solution: Even if attacker has token,
            server rejects it (blacklisted!)
```

Security Benefits:

- ✅ Logout immediately invalidates token

3. API Endpoints Overview

All 14 Endpoints at a Glance

Complete Endpoint List

Authentication (4 endpoints):

- POST `/api/v1/auth/signup` - Create account
- POST `/api/v1/auth/login` - Login
- POST `/api/v1/auth/logout` ⚠️ - Logout (protected)
- GET `/api/v1/users/profile` ⚠️ - Get profile (protected)

Lists (5 endpoints):

- GET `/api/v1/lists` - Get all lists
- POST `/api/v1/lists` - Create list
- GET `/api/v1/lists/:id` - Get single list
- PATCH `/api/v1/lists/:id` - Update list
- DELETE `/api/v1/lists/:id` - Delete list

Complete Endpoint List (cont.)

Tasks (5 endpoints):

- GET `/api/v1/lists/:listId/tasks` - Get tasks in list
- POST `/api/v1/lists/:listId/tasks` - Create task
- GET `/api/v1/tasks/:id` - Get single task
- PATCH `/api/v1/tasks/:id` - Update task
- DELETE `/api/v1/tasks/:id` - Delete task

Health Check (1 endpoint):

- GET `/api/v1/health` - Check API health

⚠️ = **Protected** (requires Bearer token)

4. Detailed API Examples

Lists & Tasks CRUD Operations

Health Check

Request:

```
curl -X GET http://localhost:8000/api/v1/health
```

Response (200 OK):

```
{
  "status": "healthy",
  "timestamp": "2025-11-07T23:48:15+00:00",
  "service": "PHP TODO REST API",
  "version": "v1",
  "checks": {
    "database": {"status": "healthy", "message": "Database connection successful"},
    "php": {"status": "healthy", "version": "8.1.33"},
    "disk": {"status": "healthy", "free_space_mb": 5629.3},
    "memory": {"status": "healthy", "memory_limit": "128M"}
  }
}
```

Use case: Monitor API health and dependencies

Get All Lists

Request:

```
curl -X GET http://localhost:8000/api/v1/lists
```

Response (200 OK):

```
[
  {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "name": "Groceries",
    "description": "Weekly shopping list",
    "createdAt": "2025-11-07T10:00:00Z",
    "updatedAt": null
  },
  {
    "id": "660e8400-e29b-41d4-a716-446655440001",
    "name": "Work Tasks",
    "description": "Q4 project deliverables",
    "createdAt": "2025-11-07T11:00:00Z"
```

Create List

Request:

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

Response (201 Created):

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Groceries",
  "description": "Weekly shopping list",
  "createdAt": "2025-11-07T10:00:00Z",
  "updatedAt": null
}
```

Get Single List

Request:

```
curl -X GET http://localhost:8000/api/v1/lists/550e8400-e29b-41d4-a716-446655440000
```

Response (200 OK):

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Groceries",
  "description": "Weekly shopping list",
  "createdAt": "2025-11-07T10:00:00Z",
  "updatedAt": null
}
```

Error Responses:

- `400` - Invalid UUID format

Update List

Request:

```
curl -X PATCH http://localhost:8000/api/v1/lists/550e8400-e29b-41d4-a716-446655440000 \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Updated Groceries",
    "description": "Monthly shopping list"
  }'
```

Response (200 OK):

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Updated Groceries",
  "description": "Monthly shopping list",
  "createdAt": "2025-11-07T10:00:00Z",
  "updatedAt": "2025-11-07T11:30:00Z"
}
```

Delete List

Request:

```
curl -X DELETE http://localhost:8000/api/v1/lists/550e8400-e29b-41d4-a716-446655440000
```

Response (204 No Content):

- Empty body
- List and **all associated tasks** are deleted

Error Responses:

- `400` - Invalid UUID format
- `404` - List not found

⚠️ **Warning:** This operation is permanent and cascades to tasks!

Get Tasks in List

Request:

```
curl -X GET http://localhost:8000/api/v1/lists/550e8400-e29b-41d4-a716-446655440000/tasks
```

Response (200 OK):

```
[
  {
    "id": "660e8400-e29b-41d4-a716-446655440001",
    "listId": "550e8400-e29b-41d4-a716-446655440000",
    "title": "Buy milk",
    "description": "2 liters, skim",
    "completed": false,
    "dueDate": "2025-11-08T18:00:00Z",
    "priority": "medium",
    "categories": ["groceries", "dairy"],
    "createdAt": "2025-11-07T10:05:00Z",
    "updatedAt": null
  }
```

Create Task

Request:

```
curl -X POST http://localhost:8000/api/v1/lists/550e8400-e29b-41d4-a716-446655440000/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Buy milk",
    "description": "2 liters, skim",
    "dueDate": "2025-11-08T18:00:00Z",
    "priority": "medium",
    "categories": ["groceries", "dairy"]
  }'
```

Validation:

- `title` : Required, 1-255 chars
- `description` : Optional, max 2000 chars
- `completed` : Optional, boolean (default: false)
- `dueDate` : Optional, ISO 8601 datetime

Create Task Response

Response (201 Created):

```json
{
  "id": "660e8400-e29b-41d4-a716-446655440001",
  "listId": "550e8400-e29b-41d4-a716-446655440000",
  "title": "Buy milk",
  "description": "2 liters, skim",
  "completed": false,
  "dueDate": "2025-11-08T18:00:00Z",
  "priority": "medium",
  "categories": ["groceries", "dairy"],
  "createdAt": "2025-11-07T10:05:00Z",
  "updatedAt": null
}
```

Error Responses:

- `400` - Validation error or invalid UUID

Get Single Task

Request:

```
curl -X GET http://localhost:8000/api/v1/tasks/660e8400-e29b-41d4-a716-446655440001
```

Response (200 OK):

```
{
  "id": "660e8400-e29b-41d4-a716-446655440001",
  "listId": "550e8400-e29b-41d4-a716-446655440000",
  "title": "Buy milk",
  "description": "2 liters, skim",
  "completed": false,
  "dueDate": "2025-11-08T18:00:00Z",
  "priority": "medium",
  "categories": ["groceries", "dairy"],
  "createdAt": "2025-11-07T10:05:00Z",
  "updatedAt": null
}
```

Update Task

Request:

```
curl -X PATCH http://localhost:8000/api/v1/tasks/660e8400-e29b-41d4-a716-446655440001 \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Buy organic milk",
    "completed": true,
    "priority": "high"
  }'
```

Response (200 OK):

```
{
  "id": "660e8400-e29b-41d4-a716-446655440001",
  "listId": "550e8400-e29b-41d4-a716-446655440000",
  "title": "Buy organic milk",
  "description": "2 liters, skim",
  "completed": true,
  "dueDate": "2025-11-08T18:00:00Z",
  "priority": "high"
```

Delete Task

Request:

```
curl -X DELETE http://localhost:8000/api/v1/tasks/660e8400-e29b-41d4-a716-446655440001
```

Response (204 No Content):

- Empty body
- Task is permanently deleted

Error Responses:

- `400` - Invalid UUID format
- `404` - Task not found

5. Security Features

Built-in Protection Against Common Vulnerabilities

Security Overview

What We Protect Against:

1. **SQL Injection** ⚠️ Most critical web vulnerability
2. **XSS (Cross-Site Scripting)** ⚠️ Code injection attacks
3. **Password Leaks** ⚠️ Credential theft
4. **Token Theft** ⚠️ Session hijacking
5. **Invalid Input** ⚠️ Data corruption

How We Protect:

- Prepared statements (SQL injection)
- HTML entity escaping (XSS)
- Bcrypt hashing (passwords)
- Token blacklisting (logout security)

SQL Injection Prevention

❌ Bad (Vulnerable to SQL Injection):

```php
$query = "SELECT * FROM users WHERE username = '$username'";
// Attacker input: "admin' OR '1'='1"
// Result: Bypasses authentication!
```

✅ Good (Using Prepared Statements):

```php
$stmt = $db->prepare("SELECT * FROM users WHERE username = :username");
$stmt->execute([':username' => $username]);
// User input is treated as data, not code
// No SQL injection possible!
```

All database operations use prepared statements with PDO.

XSS Protection

❌ Bad (Vulnerable to XSS):

```
echo "<h1>" . $_POST['name'] . "</h1>";
// Attacker input: "<script>alert('XSS')</script>"
// Result: JavaScript executes!
```

✅ Good (HTML Entity Escaping):

```
echo "<h1>" . htmlspecialchars($_POST['name'], ENT_QUOTES, 'UTF-8') . "</h1>";
// Attacker input: "<script>alert('XSS')</script>"
// Result: Displayed as text, not executed
```

All user input is escaped before output.

Password Security

Implementation:

```
// Hashing on signup
$hashedPassword = password_hash($password, PASSWORD_BCRYPT, ['cost' => 12]);

// Verification on login
if (password_verify($inputPassword, $storedHash)) {
    // Password correct
}
```

Features:

- **Bcrypt** algorithm (industry standard)
- **Cost factor 12** (2^12 = 4096 iterations)
- **Salt** automatically generated and stored
- **One-way** hash (cannot be reversed)

Token Security

Security Features:

1. **Signed** with secret key (prevents tampering)
2. **Expiration** after 1 hour (limits exposure)
3. **Blacklisting** on logout (prevents reuse)
4. **Stateless** (no server sessions to steal)

Token Validation:

```
1. Check blacklist → Reject if blacklisted
2. Verify signature → Reject if tampered
3. Check expiration → Reject if expired
4. Allow request if all pass
```

Input Validation

UUID Validation:

```
if (!preg_match('/^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$/i', $id)) {
    return 400; // Bad Request
}
```

String Validation:

```
$name = trim($input['name']); // Remove whitespace
if (empty($name)) {
    return 400; // Cannot be empty
}
if (strlen($name) > 255) {
    return 400; // Too long
}
```

Enum Validation:

```
if (!in_array($priority, ['low', 'medium', 'high'])) {
```

Error Response Format

All errors return consistent JSON:

```
{
  "error": "Human-readable error message",
  "code": "ERROR_CODE",
  "details": {
    "field": "Additional context"
  }
}
```

HTTP Status Codes:

- `200` - Success (GET/PATCH)
- `201` - Created (POST)
- `204` - No Content (DELETE)
- `400` - Bad Request (validation error)

6. Testing & Deployment

How to Test and Deploy the API

Local Setup

Prerequisites:

- PHP 8.1 or higher
- Composer (PHP package manager)
- SQLite3

Installation Steps:

```
# 1. Clone repository
git clone https://github.com/yourusername/NKU-640.git
cd NKU-640/homework4/php-version

# 2. Install dependencies
composer install

# 3. Copy environment file
cp .env.example .env
```

Environment Configuration

Edit `.env` file:

```
# Debug mode (show detailed errors)
DEBUG_MODE=true

# Log level (error, warning, info, debug)
LOG_LEVEL=debug

# Database path
DATABASE_PATH=data/todo.db

# JWT configuration
JWT_SECRET=your-secret-key-change-in-production
JWT_EXPIRY=3600
```

⚠️ **Production Settings:**

- Set `DEBUG_MODE=false`

Testing with curl

Complete workflow script:

```
#!/bin/bash

# 1. Sign up
SIGNUP=$(curl -s -X POST http://localhost:8000/api/v1/auth/signup \
  -H "Content-Type: application/json" \
  -d '{"username":"alice","email":"alice@test.com","password":"pass123456"}')

TOKEN=$(echo $SIGNUP | grep -o '"token":"[^"]*"' | cut -d'"' -f4)
echo "Token: $TOKEN"

# 2. Create list
LIST=$(curl -s -X POST http://localhost:8000/api/v1/lists \
  -H "Content-Type: application/json" \
  -d '{"name":"Groceries","description":"Weekly shopping"}')

LIST_ID=$(echo $LIST | grep -o '"id":"[^"]*"' | cut -d'"' -f4)
echo "List ID: $LIST_ID"
```

Testing with curl (cont.)

```
# 3. Create task
curl -X POST http://localhost:8000/api/v1/lists/$LIST_ID/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "title":"Buy milk",
    "priority":"medium",
    "categories":["groceries","dairy"]
  }'

# 4. Get all tasks
curl -X GET http://localhost:8000/api/v1/lists/$LIST_ID/tasks

# 5. Get profile (requires token)
curl -X GET http://localhost:8000/api/v1/users/profile \
  -H "Authorization: Bearer $TOKEN"

# 6. Logout
curl -X POST http://localhost:8000/api/v1/auth/logout \
  -H "Authorization: Bearer $TOKEN"
```

Unit Testing

Run tests with PHPUnit:

```
# Run all tests
./vendor/bin/phpunit

# Run with coverage report
./vendor/bin/phpunit --coverage-html coverage

# Run specific test file
./vendor/bin/phpunit tests/ListControllerTest.php
```

Test Coverage:

- 39 unit tests covering all endpoints
- Authentication tests (signup, login, logout, profile)
- List CRUD tests
- Task CRUD tests

NGINX Deployment

NGINX Configuration:

```
server {
    listen 80;
    server_name yourdomain.com;
    root /var/www/todo-api/public;
    index index.php;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php/php8.1-fpm.sock;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

NGINX Deployment Steps

1. Install PHP and NGINX:

```
sudo apt update
sudo apt install nginx php8.1-fpm php8.1-sqlite3 php8.1-mbstring
```

2. Deploy code:

```
sudo mkdir -p /var/www/todo-api
sudo cp -r * /var/www/todo-api/
sudo chown -R www-data:www-data /var/www/todo-api
```

3. Configure NGINX:

```
sudo nano /etc/nginx/sites-available/todo-api
sudo ln -s /etc/nginx/sites-available/todo-api /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

Production Checklist

Before deploying to production:

✅ Set `DEBUG_MODE=false` in `.env`
✅ Use strong random `JWT_SECRET`
✅ Remove `.env` from git (use `.gitignore`)
✅ Enable HTTPS (SSL/TLS certificate)
✅ Set appropriate file permissions
✅ Configure CORS if needed
✅ Implement rate limiting
✅ Set up logging and monitoring
✅ Regular backups of database
✅ Update dependencies regularly

Monitoring & Logging

Check logs:

```
# Application logs
tail -f logs/app.log

# NGINX access logs
tail -f /var/log/nginx/access.log

# NGINX error logs
tail -f /var/log/nginx/error.log

# PHP-FPM logs
tail -f /var/log/php8.1-fpm.log
```

Health check endpoint:

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

Summary & Best Practices

Key Takeaways

What We Learned

API Design:

- ✅ RESTful principles (proper HTTP methods, status codes)
- ✅ Consistent endpoint structure
- ✅ Clear request/response formats
- ✅ Comprehensive error handling

Security:

- ✅ JWT authentication with blacklisting
- ✅ Bcrypt password hashing
- ✅ SQL injection prevention (prepared statements)
- ✅ XSS protection (HTML escaping)
- ✅ Input validation and sanitization

Best Practices

Do:

- ✅ Use prepared statements for all database queries
- ✅ Hash passwords with bcrypt
- ✅ Validate and sanitize all user input
- ✅ Return appropriate HTTP status codes
- ✅ Implement token expiration and blacklisting
- ✅ Write unit tests
- ✅ Use environment variables for secrets
- ✅ Log errors for debugging

Don't:

- ❌ Concatenate user input into SQL queries

Resources

Documentation:

- API Reference - Complete endpoint specification
- README - Setup and quick start guide
- Implementation Summary - Technical details

Code Repository:

- GitHub: https://github.com/yourusername/NKU-640/tree/main/homework4/php-version

Technologies:

- PHP Documentation
- JWT.io - JWT debugger
- PHPUnit - Testing framework

Questions?

Thank you for following this tutorial!

For questions or issues:

- Check the API Reference
- Review the README
- Examine test files in `tests/` directory
- Consult the course materials

Happy coding! 🚀