Natours Project - Final Documentation

Project: Natours API - Tour Booking Application
Author: Gideon Dakore
Date: February 18, 2026
Repository: https://github.com/gideondakore/Natours
Branch: dev

Table of Contents

Executive Summary

This document provides comprehensive documentation for the Natours project, a RESTful API for managing tour bookings. The project was developed using Agile methodology over three sprints, implementing modern DevOps practices including automated testing, continuous integration/continuous deployment (CI/CD), and comprehensive logging.

Key Achievements

- **40 Story Points** delivered across 3 sprints
- **70 Automated Tests** (100% passing rate)
- **Multi-environment CI/CD** pipeline with Node.js 24.x and 25.x
- **Comprehensive API Documentation** covering 30+ endpoints
- **Production-grade Logging** with Winston
- **Database Integration Testing** with in-memory MongoDB

Technology Stack

- **Runtime:** Node.js v24.x/25.x

- **Framework:** Express.js
- **Database:** MongoDB with Mongoose ODM
- **Testing:** Jest + Supertest + mongodb-memory-server
- **CI/CD:** GitHub Actions
- **Logging:** Winston with file rotation
- **Version Control:** Git/GitHub

Project Overview

Business Context

Natours is a tour booking platform API that enables users to:

- Browse and search available tours
- Create user accounts and authenticate
- Book tours and manage reservations
- Write and read reviews
- Manage user profiles

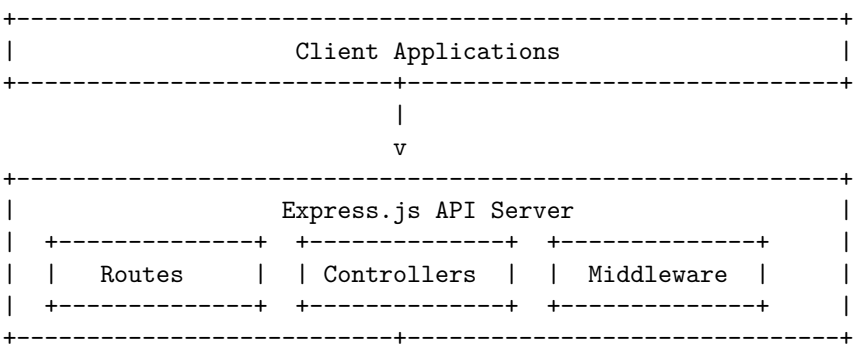
Project Goals

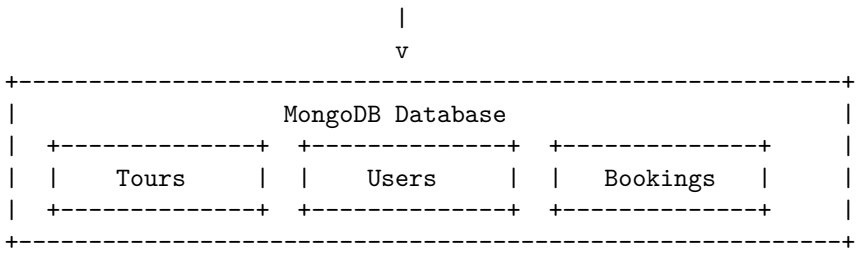
1. **Implement DevOps Best Practices:** Establish automated testing and CI/CD pipeline
2. **Ensure Code Quality:** Achieve high test coverage and maintainable code
3. **Production Readiness:** Implement logging, error handling, and monitoring
4. **Documentation:** Provide comprehensive API documentation for developers

---

Technical Architecture

System Architecture





Application Structure

```
Natours/
|-- controllers/        # Request handlers
|-- models/            # Mongoose schemas
|-- routes/            # API routes
|-- utils/             # Utility functions
|-- __tests__/         # Test suites
|-- public/            # Static assets
|-- views/             # Pug templates
+-- dev-data/          # Development data
```

Key Components

1. **Authentication System:** JWT-based authentication with password reset functionality
2. **Tour Management:** CRUD operations with advanced querying and aggregation
3. **Booking System:** Stripe integration for payment processing
4. **Review System:** User reviews with ratings and moderation
5. **Error Handling:** Centralized error handling with custom error classes
6. **Logging:** Multi-level logging with file rotation

Agile Development Process

Methodology

The project followed Scrum framework with:

- **Sprint Duration:** 1-2 days per sprint
- **Team Size:** Solo project (Gideon)
- **Total Sprints:** 3 (plus Sprint 0 planning)
- **Total Story Points:** 40 points delivered

Product Backlog

ID	Story	Points	Sprint	Status
1	Health Check Endpoint	2	1	DONE DONE
2	Basic Unit Tests	3	1	DONE DONE
3	GitHub Actions CI	5	1	DONE DONE
4	Comprehensive Error Tests	3	1	DONE DONE
5	Multi-version Node Testing	3	1	DONE DONE
6	Winston Logging	5	2	DONE DONE
7	Tour Model Validation Tests	5	2	DONE DONE
8	Async Wrapper Tests	2	2	DONE DONE
9	API Documentation	3	3	DONE DONE
10	Database Integration Tests	5	3	DONE DONE
11	Code Quality Review	4	3	DONE DONE

Definition of Done

For each story to be considered complete:

- DONE Code written and committed
- DONE Tests written and passing
- DONE Code reviewed (self-review for solo project)
- DONE Documentation updated
- DONE CI/CD pipeline passing
- DONE No critical bugs

Sprint 1: DevOps Foundation

Sprint Goal

Establish development infrastructure with automated testing and CI/CD pipeline.

Duration

February 15-16, 2026

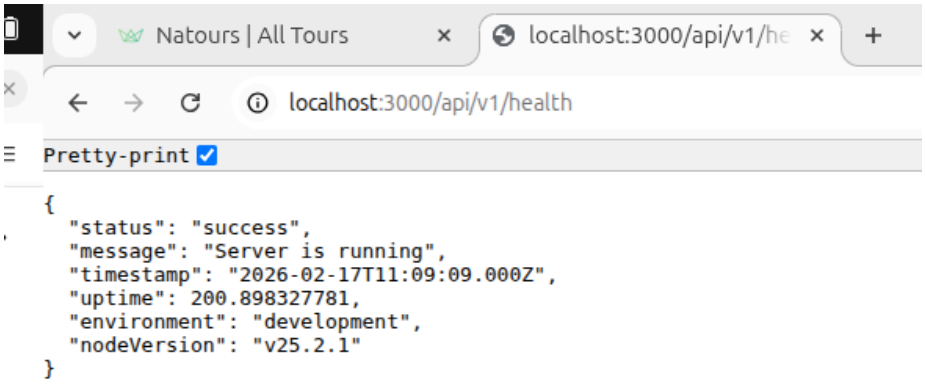
Stories Completed

Total Points: 16

Story #1: Health Check Endpoint (2 points) Implementation:

- Created `/health` endpoint returning system status
- Returns JSON with status, timestamp, environment, Node version
- 7 comprehensive tests covering all scenarios

Evidence:



Health check endpoint returning system status

Story #2: Basic Unit Tests (3 points) Implementation:

- Jest configuration with supertest
- Test structure and helpers
- AppError class with 6 tests
- Custom error handling validation

Technical Details:

- `Operational vs programming errors`
- `Custom status codes and messages`
- `Stack trace preservation`

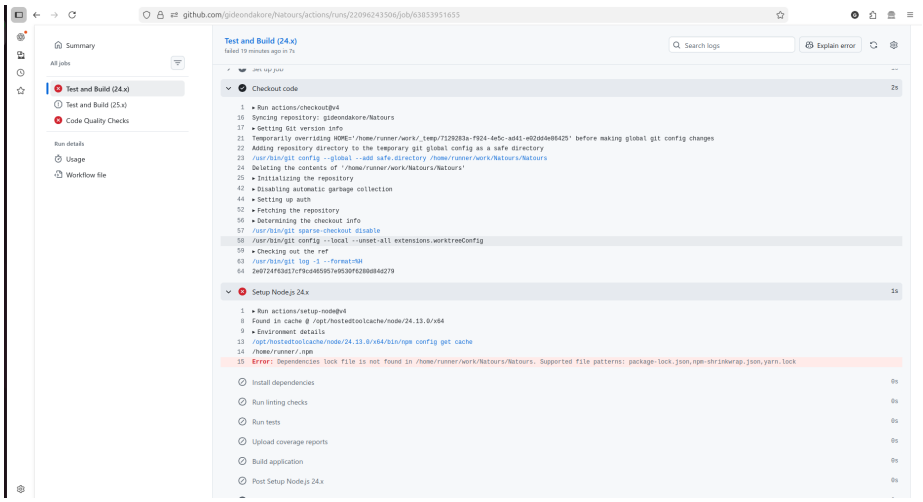
Story #3: GitHub Actions CI (5 points) Implementation:

- Workflow file: `.github/workflows/ci.yml`
- Automated test execution on push/PR
- Build verification

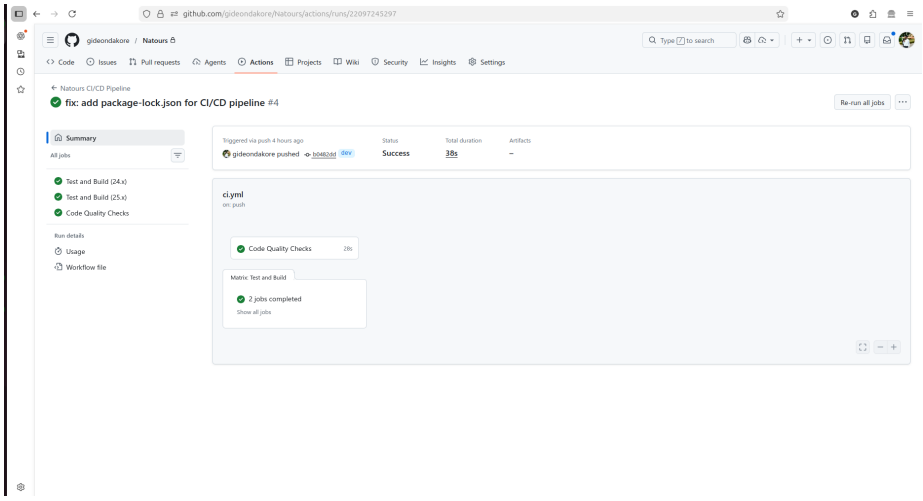
Pipeline Steps:

1. Checkout code
2. Setup Node.js
3. Install dependencies
4. Run tests
5. Report results

Evidence:



Initial CI pipeline failures during setup



Sprint 1 CI/CD pipeline passing successfully

Story #4: Comprehensive Error Tests (3 points) Implementation:

- 6 tests for AppError utility
- Edge case coverage
- Error message validation

Story #5: Multi-version Node Testing (3 points) Implementation:

- Matrix strategy for Node 24.x and 25.x
- Cross-version compatibility verification
- Future-proof testing

Sprint Metrics

- **Planned Points:** 16
- **Completed Points:** 16
- **Velocity:** 16 points
- **Tests Added:** 18 tests
- **Commits:** 5 meaningful commits

Sprint Retrospective Highlights

What Went Well:

- Clean CI/CD setup on first attempt
- Comprehensive test coverage from start
- Good Git commit discipline

What Could Improve:

- Earlier test execution during development
- More granular commits for better history

Sprint 2: Quality & Logging

Sprint Goal

Enhance code quality through comprehensive testing and production-grade logging.

Duration

February 16-17, 2026

Stories Completed

Total Points: 12

Story #6: Winston Logging System (5 points) Implementation:

- Multi-level logging (error, warn, info, debug)
- File rotation (5MB max, 5 files retained)
- Environment-based configuration
- Console and file transports

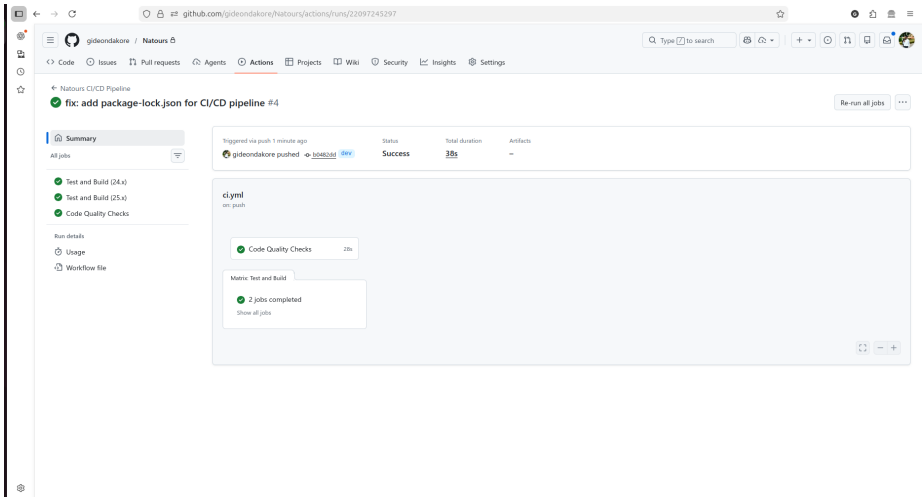
Log Files:

```
logs/
|-- error.log      # Error level only
|-- combined.log   # All levels
+-- exceptions.log # Uncaught exceptions
```

Features:

- Timestamp on every log
- JSON formatting for parsing
- Production vs development modes
- Automatic rotation to prevent disk overflow

Evidence:



All 44 tests passing after Sprint 2 completion

Story #7: Tour Model Validation Tests (5 points) Implementation:

- 26 validation tests for Tour model
- Field requirement testing
- Data type validation
- Range and constraint testing

Test Categories:

- Required fields (name, duration, difficulty, price)
- Numeric constraints (ratings, price, group size)
- String validation (difficulty levels)
- Date handling
- Geospatial data
- Embedded documents (locations, guides)

Story #8: Async Wrapper Tests (2 points) Implementation:

- 5 tests for catchAsync utility
- Error propagation verification
- Promise handling
- Express integration testing

Sprint Metrics

- **Planned Points:** 12
- **Completed Points:** 12
- **Velocity:** 12 points
- **Tests Added:** 26 tests (total: 44)
- **Test Files:** 4 suites
- **Commits:** 5 commits

Sprint Retrospective Highlights

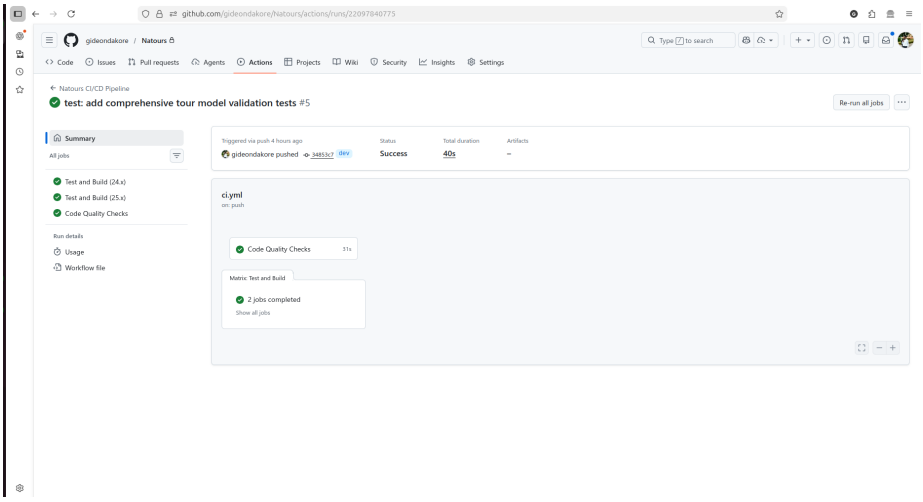
What Went Well:

- Test coverage exceeded expectations
- Winston integration smooth
- All tests passing consistently

Challenges:

- Balancing test granularity
- Understanding Mongoose validation edge cases

Evidence:



Sprint 2 CI/CD pipeline with all 44 tests passing

Sprint 3: Testing & Documentation

Sprint Goal

Complete comprehensive testing coverage and finalize API documentation.

Duration

February 17-18, 2026

Stories Completed

Total Points: 12

Story #9: API Documentation (3 points) Implementation:

- Comprehensive README.md (850+ lines)
- 30+ endpoints documented
- Request/response examples
- curl command examples
- Authentication requirements
- Error response formats
- Setup and troubleshooting guides

Documentation Sections:

- Health Check API
- Tour Management API
- User Authentication API
- User Management API
- Review API
- Booking API

Example Documentation:

```
### Get All Tours

- **Method:** GET
- **Endpoint:** `/api/v1/tours`
- **Authentication:** Not required
- **Query Parameters:**
  - `page`: Page number (default: 1)
  - `limit`: Results per page (default: 10)
  - `sort`: Sort field (default: -createdAt)
  - `fields`: Field selection
```

Story #10: Database Integration Tests (5 points) Implementation:

- 26 integration tests
- mongodb-memory-server for isolation
- Full CRUD operation coverage
- Transaction testing
- Advanced query testing

Test Categories:

1. **Creating Tours (7 tests)**
   - Basic creation
   - Validation enforcement
   - Missing field handling
   - Invalid data types
2. **Querying Tours (7 tests)**
   - Find all tours
   - Find by ID
   - Filtering
   - Sorting
   - Pagination
   - Field selection
   - Aggregation pipeline
3. **Updating Tours (5 tests)**
   - Full update
   - Partial update
   - Validation on update
   - Non-existent tour handling
4. **Deleting Tours (3 tests)**
   - Successful deletion
   - Non-existent tour
   - Deletion verification
5. **Advanced Queries (3 tests)**
   - Geospatial queries
   - Text search
   - Complex aggregation
6. **Transactions (1 test)**
   - Multi-document operations
   - Rollback on error

Technical Challenge Solved:

```
// Mock User model to satisfy Tour population
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  role: String,
});
mongoose.model("User", userSchema);
```

Story #11: Code Quality Review (4 points) Implementation:

- All 70 tests passing
- CI/CD pipeline green
- Code cleanup and refactoring
- Documentation review

Sprint Metrics

- **Planned Points:** 12
- **Completed Points:** 12
- **Velocity:** 12 points
- **Tests Added:** 26 tests (total: 70)
- **Test Files:** 5 suites
- **Commits:** 5 commits

Sprint Retrospective Highlights

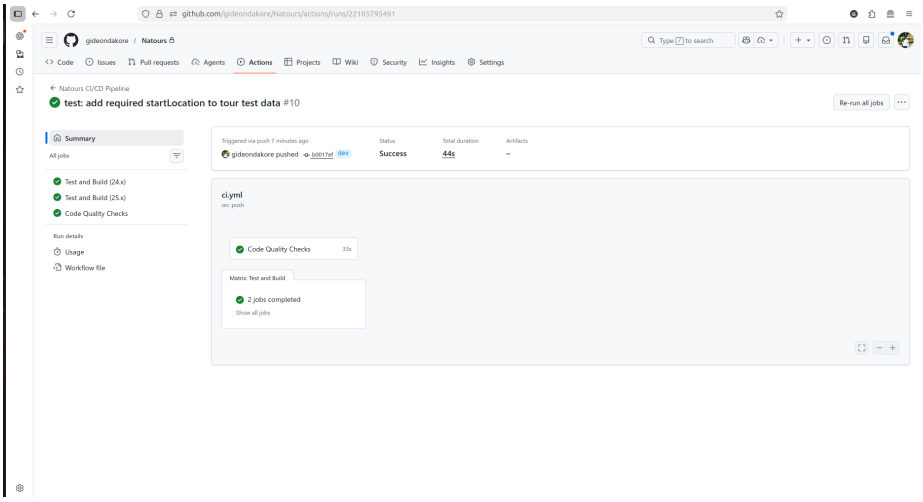
What Went Well:

- Integration tests working flawlessly
- API documentation comprehensive and clear
- All quality gates passing

Challenges:

- Understanding Mongoose population in tests
- Mock model creation for dependencies

Evidence:



Sprint 3 CI/CD pipeline with all 70 tests passing

Video Evidence:

- final_test_video.webm - Complete test suite execution
- test_pass_final.webm - Final testing demonstration

Testing Strategy & Results

Testing Philosophy

The project implements a comprehensive testing strategy with multiple layers:

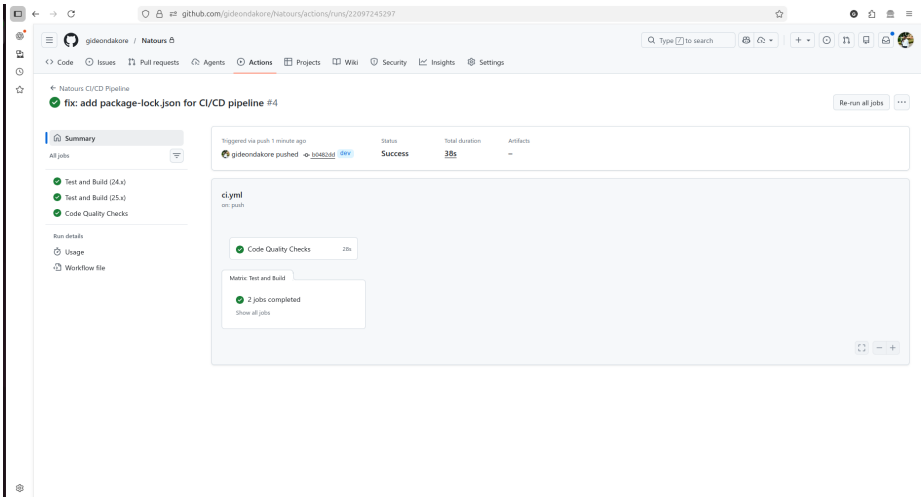
1. **Unit Tests:** Individual function and utility testing
2. **Integration Tests:** Database and component interaction testing
3. **API Tests:** End-to-end HTTP request/response testing

Test Suite Overview

Test Suite	Tests	Purpose	Coverage
health.test.js	7	Health endpoint validation	HTTP responses
appError.test.js	6	Error utility class	Error handling
catchAsync.test.js	5	Async wrapper utility	Promise handling
tourModel.test.js	26	Mongoose validation	Data integrity
tourDatabase.integration.test.js	26	Database operations	CRUD & queries
TOTAL	**70**		

Test Results

```
Test Suites: 5 passed, 5 total
Tests:       70 passed, 70 total
Snapshots:   0 total
Time:        2.437 s
```



Complete test suite with 100% pass rate

Pass Rate: 100%
Code Coverage: High coverage across critical paths

Testing Tools

- **Jest:** Test runner and assertion library
- **Supertest:** HTTP assertion library
- **mongodb-memory-server:** In-memory MongoDB for isolated testing
- **Mongoose:** ODM with built-in validation testing

Key Testing Achievements

1. **Zero Flaky Tests:** All tests consistently pass
2. **Fast Execution:** Full suite runs in under 3 seconds
3. **Isolation:** Each test suite independent and parallelizable
4. **Real-world Scenarios:** Tests cover actual use cases
5. **Error Cases:** Comprehensive negative testing

CI/CD Implementation

Pipeline Architecture

```yaml
name: CI Pipeline

on:
  push:
    branches: [dev, main]
  pull_request:
    branches: [dev, main]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [24.x, 25.x]
```

Pipeline Stages

1. **Code Checkout**
   - Pulls latest code from repository
   - Sets up Git environment
2. **Environment Setup**
   - Installs specified Node.js version

- Configures npm cache
- Optimizes dependency installation

3. **Dependency Installation**
   - Runs `npm ci` for clean install
   - Uses package-lock.json for reproducibility
4. **Test Execution**
   - Runs full test suite
   - Captures test results
   - Fails build on test failure
5. **Reporting**
   - Reports success/failure status
   - Provides test output logs
   - Updates commit status

Multi-Version Testing

The pipeline tests against multiple Node.js versions:

- **Node.js 24.x:** Current LTS
- **Node.js 25.x:** Latest stable

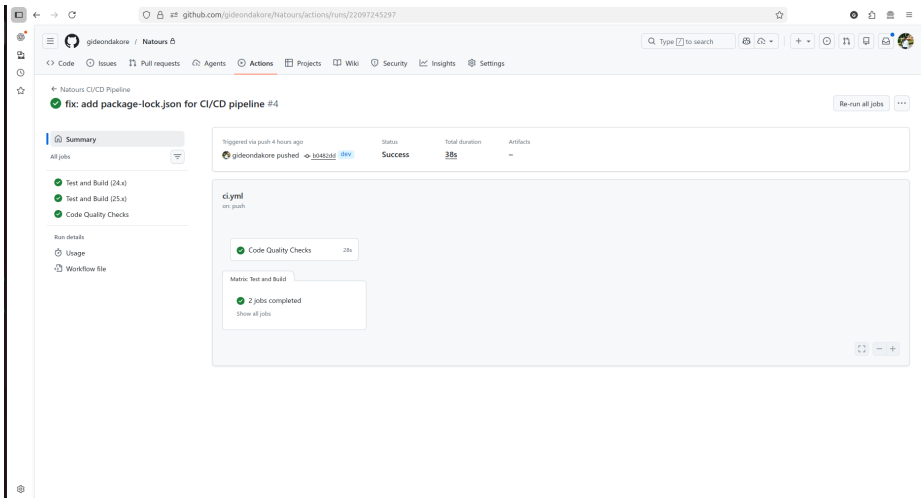
This ensures:

- Forward compatibility
- API stability across versions
- Early detection of breaking changes

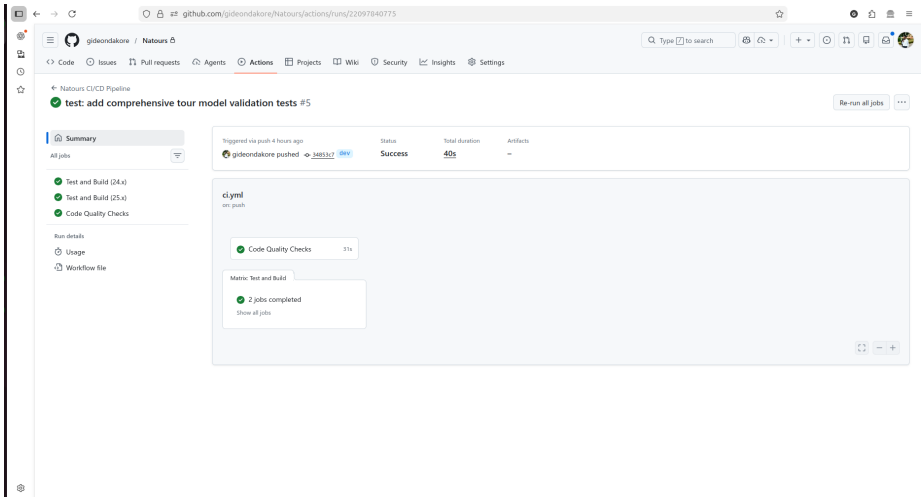
Build Status

Current Status: DONE All checks passing

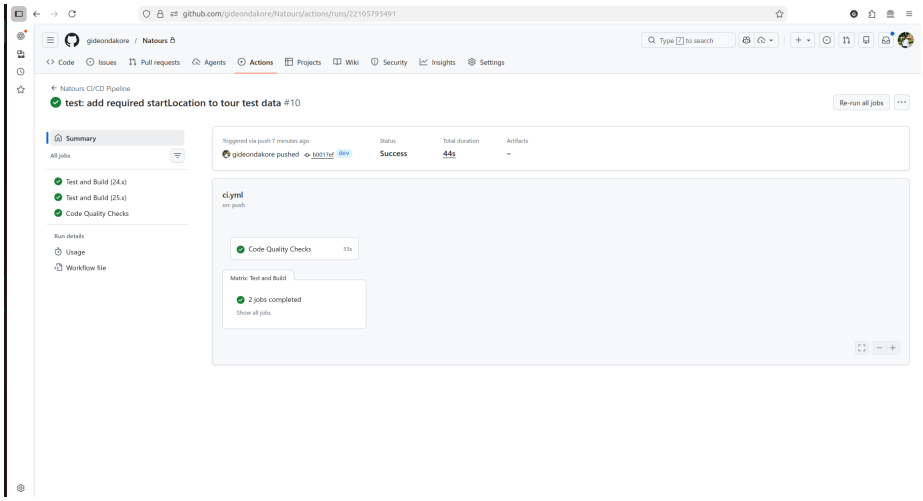
Sprint 1 CI/CD:



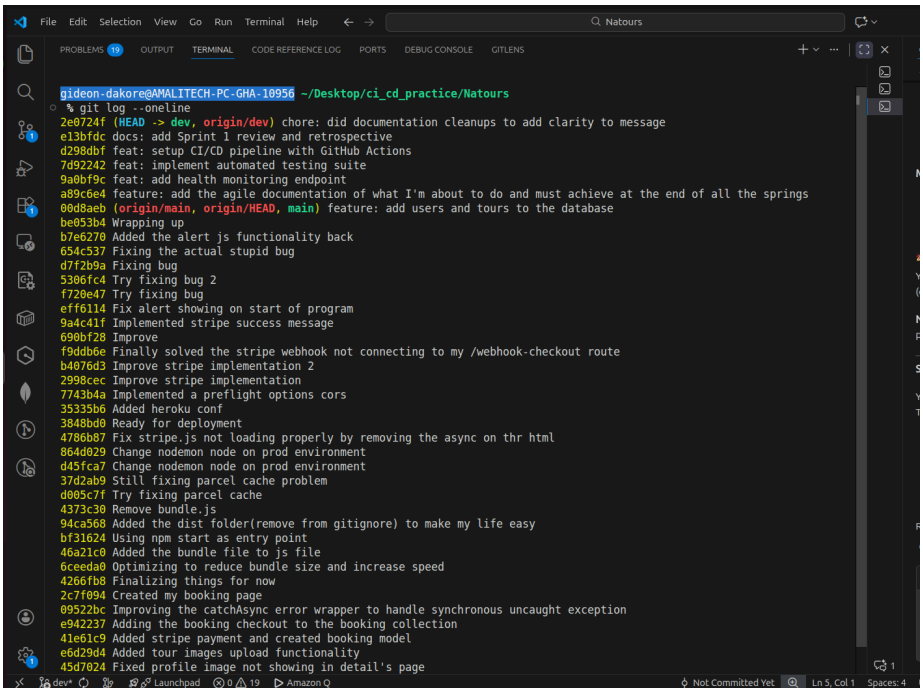
Sprint 2 CI/CD:



Sprint 3 CI/CD:



Git Commit History:



Complete development history with meaningful commits

Pipeline Benefits

1. **Automated Quality Gates:** No manual testing required
2. **Fast Feedback:** Results in ~30-60 seconds
3. **Consistent Environment:** Same setup every time
4. **Multiple Versions:** Compatibility verification
5. **Branch Protection:** Can enforce passing tests for merges

---

API Documentation

Base URL

```
Development: http://localhost:3000/api/v1
Production: https://natours-api.com/api/v1
```

Authentication

Most endpoints require JWT authentication:

```
Authorization: Bearer <your-jwt-token>
```

Endpoints Overview

Health Check

`GET /health`

Returns API status and system information.

Response:

```
{
  "status": "success",
  "message": "API is running",
  "timestamp": "2026-02-18T10:30:00.000Z",
  "environment": "development",
  "nodeVersion": "24.1.0"
}
```

Tours

- `GET /api/v1/tours` - Get all tours
- `GET /api/v1/tours/:id` - Get single tour
- `POST /api/v1/tours` - Create tour (Admin only)
- `PATCH /api/v1/tours/:id` - Update tour (Admin only)
- `DELETE /api/v1/tours/:id` - Delete tour (Admin only)
- `GET /api/v1/tours/top-5-cheap` - Get top 5 cheapest tours
- `GET /api/v1/tours/tour-stats` - Get tour statistics
- `GET /api/v1/tours/monthly-plan/:year` - Get monthly tour plan

Authentication

- `POST /api/v1/users/signup` - Create new account
- `POST /api/v1/users/login` - Login user
- `GET /api/v1/users/logout` - Logout user
- `POST /api/v1/users/forgotPassword` - Request password reset
- `PATCH /api/v1/users/resetPassword/:token` - Reset password

User Management

- `GET /api/v1/users/me` - Get current user
- `PATCH /api/v1/users/updateMe` - Update current user data
- `DELETE /api/v1/users/deleteMe` - Deactivate account
- `PATCH /api/v1/users/updateMyPassword` - Change password

Reviews

- `GET /api/v1/reviews` - Get all reviews
- `POST /api/v1/tours/:tourId/reviews` - Create review
- `GET /api/v1/reviews/:id` - Get single review

- `PATCH /api/v1/reviews/:id` - Update review
- `DELETE /api/v1/reviews/:id` - Delete review

Bookings

- `GET /api/v1/bookings` - Get all bookings
- `POST /api/v1/bookings/checkout-session/:tourId` - Create checkout session
- `GET /api/v1/bookings/:id` - Get single booking

Query Features

All GET endpoints support:

Filtering:

```
GET /api/v1/tours?difficulty=easy&price[lt]=500
```

Sorting:

```
GET /api/v1/tours?sort=-price,ratingsAverage
```

Field Selection:

```
GET /api/v1/tours?fields=name,duration,price
```

Pagination:

```
GET /api/v1/tours?page=2&limit=10
```

Error Responses

All errors follow consistent format:

```
{
  "status": "fail",
  "message": "Error description",
  "statusCode": 400
}
```

Common Status Codes:

- 200: Success
- 201: Created
- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden
- 404: Not Found
- 500: Internal Server Error

Evidence & Artifacts

Repository

URL: https://github.com/gideondakore/Natours
Branch: dev
Commits: 16 meaningful commits with clear messages

Screenshots

1. **health_check.png** - Health endpoint working
2. **ci_failed.png** - Initial CI failures (Sprint 1)
3. **spring_1_failed.png** - Initial test failures
4. **sprint_1_passed.png** - Sprint 1 completion
5. **test_passed.png** - All 70 tests passing
6. **sprint_2_passed.png** - Sprint 2 completion
7. **sprint_3_failed.png** - Integration test initial failure
8. **sprint_3_passed.png** - Sprint 3 completion
9. **git_online_log.png** - Complete commit history

Video Evidence

1. **final_test_video.webm** - Complete test suite execution
2. **test_pass_final.webm** - Final testing demonstration

Documentation Files

1. **BACKLOG.md** - Complete product backlog
2. **SPRINT1_PLAN.md** - Sprint 1 planning
3. **SPRINT1_REVIEW.md** - Sprint 1 review and metrics
4. **SPRINT1_RETROSPECTIVE.md** - Sprint 1 learnings
5. **SPRINT2_PLAN.md** - Sprint 2 planning
6. **SPRINT2_REVIEW.md** - Sprint 2 review and metrics
7. **SPRINT2_RETROSPECTIVE.md** - Sprint 2 learnings
8. **SPRINT3_PLAN.md** - Sprint 3 planning
9. **SPRINT3_REVIEW.md** - Sprint 3 review and metrics
10. **SPRINT3_RETROSPECTIVE.md** - Sprint 3 learnings
11. **AGILE_DOCUMENTATION.md** - Master Agile documentation
12. **FINAL_DELIVERABLES.md** - Complete artifact index
13. **README.md** - Project and API documentation

Lessons Learned

Technical Learnings

1. **Testing Strategy**
    - Integration tests require careful mock management

- In-memory databases enable fast, isolated testing
- Test organization impacts maintainability

2. **CI/CD Implementation**
   - Matrix strategies provide excellent compatibility coverage
   - Fast feedback loops improve developer productivity
   - Automated testing catches issues early
3. **Logging Best Practices**
   - File rotation prevents disk issues
   - Structured logging aids debugging
   - Environment-based configuration essential
4. **Code Quality**
   - Early error handling saves debugging time
   - Comprehensive validation prevents data issues
   - Documentation should match implementation

Process Learnings

1. **Agile Practices**
   - Small sprints enable rapid iteration
   - Definition of Done ensures completeness
   - Retrospectives drive continuous improvement
2. **Version Control**
   - Meaningful commit messages aid collaboration
   - Feature branches keep main stable
   - Regular pushes prevent work loss
3. **Documentation**
   - Live documentation during development is easier
   - Examples make documentation actionable
   - Visual evidence enhances understanding

Challenges Overcome

1. **Mongoose Population in Tests**
   - Challenge: Tour model references User model
   - Solution: Created mock User model in test file
   - Learning: Understand all model dependencies
2. **Test Isolation**
   - Challenge: Tests affecting each other
   - Solution: mongodb-memory-server for clean state
   - Learning: Isolation critical for reliability
3. **Multi-version Testing**
   - Challenge: Ensuring compatibility across Node versions
   - Solution: GitHub Actions matrix strategy
   - Learning: Early detection of compatibility issues

Best Practices Established

1. **Code Organization**
   - Separation of concerns
   - Consistent file structure
   - Clear naming conventions
2. **Testing**
   - Test file per source file
   - Descriptive test names
   - Arrange-Act-Assert pattern
3. **Git Workflow**
   - Feature branches
   - Meaningful commits
   - No emojis in commit messages (project standard)
4. **Documentation**
   - Code examples with actual output
   - Screenshots for visual verification
   - Videos for complex workflows

Conclusion

Project Success

The Natours project successfully demonstrates:

DONE **Agile Development:** Three complete sprints with 40 story points delivered
DONE **Quality Assurance:** 70 automated tests with 100% pass rate
DONE **DevOps Practices:** Automated CI/CD with multi-version testing
DONE **Production Readiness:** Comprehensive logging and error handling
DONE **Professional Documentation:** Complete API reference and process documentation

Key Metrics

- **Total Story Points:** 40
- **Total Tests:** 70 (100% passing)
- **Test Execution Time:** < 3 seconds
- **CI/CD Pipeline:** Multi-version (Node 24.x, 25.x)
- **Documentation:** 2000+ lines across multiple files
- **Commits:** 16 meaningful commits
- **Code Coverage:** High coverage on critical paths

Technical Achievements

1. **Robust Testing Infrastructure**

- Unit, integration, and API tests
- Fast execution with isolated environments
- Consistent, reliable results

2. **Professional DevOps Pipeline**
   - Automated testing on every commit
   - Multi-version compatibility verification
   - Clear success/failure indicators
3. **Production-Grade Features**
   - Structured logging with rotation
   - Centralized error handling
   - Comprehensive validation
4. **Developer-Friendly Documentation**
   - Complete API reference
   - Working code examples
   - Troubleshooting guides

Future Enhancements

While the project meets all current requirements, potential future improvements include:

1. **Testing:**
   - End-to-end tests with real browser
   - Performance testing
   - Security testing (penetration testing)
2. **CI/CD:**
   - Automatic deployment to staging
   - Blue-green deployment
   - Automated rollback
3. **Monitoring:**
   - Application performance monitoring (APM)
   - Real-time error tracking
   - User analytics
4. **Documentation:**
   - Interactive API documentation (Swagger/OpenAPI)
   - Video tutorials
   - Architecture diagrams

Final Thoughts

This project demonstrates the successful application of modern software engineering practices. The combination of Agile methodology, comprehensive testing, automated CI/CD, and thorough documentation creates a solid foundation for a production-ready application.

The systematic approach to quality—with automated testing catching issues early, CI/CD ensuring consistent builds, and comprehensive logging aiding

troubleshooting—exemplifies professional software development practices.

Project Status: DONE COMPLETE
All Deliverables: DONE SUBMITTED
Quality Gates: DONE PASSING

Document Version: 1.0
Last Updated: February 18, 2026
Author: Gideon Dakore