VINEFOX Maintenance & Development Guide

🛠️ Development & Maintenance Overview

Current Version: vinefox (5f3b53c)
Target Audience: Developers, DevOps Engineers, Project Managers
Maintenance Level: Production Critical

🚀 Development Environment Setup

Prerequisites

- **Node.js:** Version 18.17 or higher (LTS recommended)
- **npm:** Version 9 or higher
- **Git:** Version 2.30 or higher
- **PostgreSQL:** Version 14 or higher (production), SQLite (development)
- **Vercel CLI:** For deployment management

Local Development Setup

```
# Clone repository
git clone https://github.com/bbensaid/store-wine.git
cd store-wine

# Install dependencies
npm install

# Set up environment variables
cp .env.example .env.local
# Edit .env.local with your local configuration

# Set up database
npx prisma generate
npx prisma db push
npm run seed

# Start development server
npm run dev
```

Environment Variables

```
# Database
DATABASE_URL="postgresql://username:password@localhost:5432/wine_store"
DIRECT_URL="postgresql://username:password@localhost:5432/wine_store"

# Authentication (Clerk)
NEXT_PUBLIC_CLERK_PUBLISHABLE_KEY=pk_test_...
CLERK_SECRET_KEY=sk_test_...

# Payment Processing (Stripe)
STRIPE_SECRET_KEY=sk_test_...
NEXT_PUBLIC_STRIPE_PUBLISHABLE_KEY=pk_test_...

# Application
NEXT_PUBLIC_APP_URL=http://localhost:3000
```

Critical Security Notes:

- **Never commit environment files** to version control
- **Use different keys** for development, staging, and production
- **Rotate keys regularly** for security
- **Monitor key usage** for unauthorized access

🔄 Development Workflow

Git Workflow

```
# 1. Create feature branch
git checkout -b feature/new-feature-name

# 2. Make changes and commit
git add .
git commit -m "feat: add new feature description"

# 3. Push and create pull request
git push origin feature/new-feature-name

# 4. Code review and merge
# Merge via GitHub pull request interface

# 5. Clean up
git checkout main
git pull origin main
git branch -d feature/new-feature-name
```

Commit Message Convention

```
type(scope): description

Examples:
feat(auth): add two-factor authentication
fix(cart): resolve quantity update bug
docs(api): update API documentation
style(navbar): improve mobile navigation
refactor(products): optimize database queries
test(cart): add cart functionality tests
chore(deps): update dependencies
```

Commit Types:

- **feat:** New feature
- **fix:** Bug fix
- **docs:** Documentation changes
- **style:** Code style changes (formatting, etc.)
- **refactor:** Code refactoring
- **test:** Adding or updating tests
- **chore:** Maintenance tasks

Code Review Checklist

- ☐ **Functionality:** Does the code work as intended?
- ☐ **Performance:** Are there any performance implications?
- ☐ **Security:** Are there any security vulnerabilities?
- ☐ **Accessibility:** Does it meet accessibility standards?
- ☐ **Testing:** Are there appropriate tests?
- ☐ **Documentation:** Is the code well-documented?

- ☐ **Error Handling:** Is error handling comprehensive?
- ☐ **Mobile Responsiveness:** Does it work on mobile devices?

🚨 Critical Issues & Risk Mitigation

Known Critical Issues

1. Authentication Rate Limiting

Issue: Individual components making separate authentication calls can trigger rate limits.

Solution: Batch authentication calls at parent component level.

```
// ❌ WRONG: Individual component auth calls
const ProductCard = () => {
  const { userId } = useUser(); // This can cause rate limiting
  // ...
};

// ✅ CORRECT: Parent component auth management
const ProductsGrid = () => {
  const { userId } = useUser(); // Single auth call
  return (
    <div>
      {products.map((product) => (
        <ProductCard key={product.id} product={product} userId={userId} />
      ))}
    </div>
  );
};
```

Prevention:

- Always pass user context down from parent components
- Use React Context for global user state
- Implement authentication caching strategies

2. Double Logo Issue

Issue: Logo appears twice on mobile devices due to both navbar and hero logos.

Solution: Hide desktop logo on mobile screens.

```
// Desktop Logo - hidden on mobile to prevent double logo issue
<div className="hidden md:flex items-center">
  <Logo />
</div>

// Mobile Logo - only visible on mobile
<div className="md:hidden">
  <Logo />
</div>
```

Prevention:

- Use responsive design classes consistently
- Test on multiple device sizes
- Implement proper mobile/desktop component separation

3. Vercel Build Errors

Issue: Unused imports and dependencies causing build failures.

Solution: Regular dependency cleanup and import optimization.

```
# Check for unused dependencies
npm audit
npm outdated

# Remove unused dependencies
npm uninstall unused-package

# Clean up imports
# Remove unused import statements
# Use tree-shaking effectively
```

Prevention:

- Regular dependency audits
- Automated import cleanup
- Build testing before deployment

4. Database Performance Issues

Issue: Slow queries and inefficient database operations.

Solution: Implement proper indexing and query optimization.

```
-- Add composite indexes for common query patterns
CREATE INDEX "Wine_search_idx" ON "Wine"("name", "type", "price");
CREATE INDEX "Wine_featured_idx" ON "Wine"("featured", "createdAt");

-- Monitor slow queries
SELECT query, mean_time, calls
FROM pg_stat_statements
ORDER BY mean_time DESC
LIMIT 10;
```

Prevention:

- Regular query performance monitoring
- Database index optimization
- Query result caching strategies

Risk Mitigation Strategies

1. Database Backup Strategy

```
# Automated daily backups
#!/bin/bash
DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="/backups/wine_store"
DB_NAME="wine_store"

# Create backup
pg_dump $DB_NAME > $BACKUP_DIR/backup_$DATE.sql

# Compress backup
gzip $BACKUP_DIR/backup_$DATE.sql
```

```
# Keep only last 7 days of backups
find $BACKUP_DIR -name "backup_*.sql.gz" -mtime +7 -delete
```

Backup Schedule:

- **Daily:** Full database backup
- **Weekly:** Full backup with compression
- **Monthly:** Full backup with verification
- **Before Deployments:** Pre-deployment backup

2. Rollback Procedures

```
# Quick rollback to previous version
git log --oneline -5  # Identify target commit
git reset --hard <commit-hash>  # Rollback to specific commit
git push --force origin main  # Force push (use with caution)

# Database rollback
# Restore from backup if necessary
psql $DB_NAME < backup_YYYYMMDD_HHMMSS.sql
```

Rollback Triggers:

- Critical functionality broken
- Performance degradation
- Security vulnerabilities
- User experience issues

3. Monitoring & Alerting

```
// Performance monitoring setup
export const config = {
  api: {
    responseLimit: false,
    bodyParser: {
      sizeLimit: "10mb",
    },
  },
};

// Response time monitoring
export async function GET(request: NextRequest) {
  const startTime = Date.now();

  try {
    const result = await fetchData();
    const responseTime = Date.now() - startTime;

    // Alert on slow responses
    if (responseTime > 2000) {
      console.warn(`Slow API Response: ${responseTime}ms`);
      // Send alert to monitoring system
    }

    return NextResponse.json(result);
  } catch (error) {
    // Error logging and alerting
    console.error("API Error:", error);
    // Send error alert
```

```
  }
}
```

Monitoring Thresholds:

- **Response Time:** > 2 seconds (warning), > 5 seconds (critical)
- **Error Rate:** > 5% (warning), > 10% (critical)
- **Database Connections:** > 80% (warning), > 95% (critical)
- **Memory Usage:** > 80% (warning), > 90% (critical)

🧪 Testing Strategy

Testing Levels

1. Unit Testing

```
// Component testing with React Testing Library
import { render, screen, fireEvent } from "@testing-library/react";
import { AddToCart } from "../AddToCart";

describe("AddToCart Component", () => {
  it("should add item to cart when button is clicked", async () => {
    const mockAddToCart = jest.fn();

    render(<AddToCart wineId="1" onAddToCart={mockAddToCart} />);

    const addButton = screen.getByRole("button", { name: /add to cart/i });
    fireEvent.click(addButton);

    expect(mockAddToCart).toHaveBeenCalledWith("1", 1);
  });

  it("should update quantity when input changes", () => {
    render(<AddToCart wineId="1" onAddToCart={jest.fn()} />);

    const quantityInput = screen.getByRole("spinbutton");
    fireEvent.change(quantityInput, { target: { value: "3" } });

    expect(quantityInput).toHaveValue(3);
  });
});
```

Unit Testing Coverage:

- **Components:** Individual component functionality
- **Utilities:** Helper functions and calculations
- **Hooks:** Custom React hooks
- **API Functions:** Data fetching and manipulation

2. Integration Testing

```
// API endpoint testing
import { createMocks } from "node-mocks-http";
import { GET, POST } from "../app/api/products/route";

describe("/api/products", () => {
  it("should return products with pagination", async () => {
    const { req, res } = createMocks({
      method: "GET",
```

```
      query: { page: "1", limit: "10" },
    });

    const response = await GET(req);
    const data = await response.json();

    expect(response.status).toBe(200);
    expect(data.wines).toBeDefined();
    expect(data.pagination).toBeDefined();
  });

  it("should handle invalid input gracefully", async () => {
    const { req, res } = createMocks({
      method: "POST",
      body: { invalid: "data" },
    });

    const response = await POST(req);

    expect(response.status).toBe(400);
  });
});
```

Integration Testing Focus:

- **API Endpoints:** Request/response handling
- **Database Operations:** CRUD operations
- **Authentication Flow:** User login/logout
- **Payment Processing:** Stripe integration

3. End-to-End Testing

```
// E2E testing with Playwright
import { test, expect } from "@playwright/test";

test("complete purchase flow", async ({ page }) => {
  // Navigate to product page
  await page.goto("/products/1");

  // Add to cart
  await page.click('[data-testid="add-to-cart"]');
  await expect(page.locator('[data-testid="cart-count"]')).toHaveText("1");

  // Go to cart
  await page.click('[data-testid="cart-button"]');
  await expect(page.locator('[data-testid="cart-items"]')).toBeVisible();

  // Proceed to checkout
  await page.click('[data-testid="checkout-button"]');
  await expect(page.locator('[data-testid="checkout-form"]')).toBeVisible();
});
```

E2E Testing Scenarios:

- **User Registration:** Complete sign-up flow
- **Product Purchase:** Add to cart → checkout → payment
- **Search & Filtering:** Product discovery workflows
- **Mobile Experience:** Touch interactions and responsive design

Testing Best Practices

1. Test Data Management

```
// Test data factories
export const createMockWine = (overrides = {}) => ({
  id: 1,
  name: "Test Wine",
  type: "Red",
  price: 2500,
  regionId: 1,
  ...overrides,
});

export const createMockUser = (overrides = {}) => ({
  id: "user_123",
  email: "test@example.com",
  firstName: "Test",
  lastName: "User",
  ...overrides,
});
```

2. Mocking Strategies

```
// API mocking
jest.mock("@/lib/prisma", () => ({
  wine: {
    findMany: jest.fn(),
    findUnique: jest.fn(),
    create: jest.fn(),
    update: jest.fn(),
    delete: jest.fn(),
  },
}));

// Authentication mocking
jest.mock("@clerk/nextjs", () => ({
  useUser: () => ({
    userId: "test_user_id",
    user: { id: "test_user_id", email: "test@example.com" },
    isLoaded: true,
  }),
}));
```

🔧 Maintenance Procedures

Regular Maintenance Tasks

1. Dependency Updates

```
# Weekly dependency check
npm audit
npm outdated

# Monthly dependency updates
npm update

# Quarterly major version updates
npx npm-check-updates -u
npm install
```

Update Strategy:

- **Security Updates:** Immediate installation
- **Minor Updates:** Weekly review and installation
- **Major Updates:** Monthly review with testing
- **Breaking Changes:** Quarterly review with migration planning

2. Database Maintenance

```sql
-- Weekly database optimization
ANALYZE "Wine";
ANALYZE "Cart";
ANALYZE "Order";
ANALYZE "Review";

-- Monthly cleanup
DELETE FROM "CartItem"
WHERE "cartId" NOT IN (SELECT id FROM "Cart");

DELETE FROM "OrderItem"
WHERE "orderId" NOT IN (SELECT id FROM "Order");

-- Quarterly maintenance
VACUUM ANALYZE;
REINDEX DATABASE wine_store;
```

Maintenance Schedule:

- **Daily:** Backup verification
- **Weekly:** Performance optimization
- **Monthly:** Data cleanup and integrity checks
- **Quarterly:** Major maintenance and optimization

3. Performance Monitoring

```typescript
// Performance metrics collection
export async function performanceMiddleware(
  req: NextRequest,
  next: NextFunction
) {
  const startTime = Date.now();

  try {
    const response = await next();

    const responseTime = Date.now() - startTime;

    // Log performance metrics
    console.log(`API Performance: ${req.url} - ${responseTime}ms`);

    // Send to monitoring service
    await sendToMonitoring({
      endpoint: req.url,
      responseTime,
      status: response.status,
      timestamp: new Date().toISOString(),
    });

    return response;
  } catch (error) {
    const responseTime = Date.now() - startTime;
```

```
    // Log error metrics
    console.error(`API Error: ${req.url} - ${responseTime}ms`, error);

    throw error;
  }
}
```

Performance Metrics:

- **Response Times:** API endpoint performance
- **Database Queries:** Query execution times
- **Memory Usage:** Application memory consumption
- **Error Rates:** Application error frequency

Emergency Procedures

1. Critical Bug Fixes

```
# 1. Create hotfix branch
git checkout -b hotfix/critical-bug-fix

# 2. Make minimal fix
# Edit only necessary files

# 3. Test thoroughly
npm run test
npm run build

# 4. Deploy immediately
git commit -m "hotfix: resolve critical bug"
git push origin hotfix/critical-bug-fix

# 5. Merge to main
git checkout main
git merge hotfix/critical-bug-fix
git push origin main

# 6. Tag release
git tag -a v1.0.1 -m "Hotfix release"
git push origin v1.0.1
```

2. Database Recovery

```
# 1. Stop application
pm2 stop wine-store

# 2. Restore from backup
psql $DB_NAME < backup_YYYYMMDD_HHMMSS.sql

# 3. Verify data integrity
psql $DB_NAME -c "SELECT COUNT(*) FROM \"Wine\";"
psql $DB_NAME -c "SELECT COUNT(*) FROM \"Cart\";"

# 4. Restart application
pm2 start wine-store

# 5. Monitor for issues
# Check logs and performance metrics
```

3. Performance Emergency

```
// Emergency performance mode
export const emergencyPerformanceMode = {
  // Disable non-essential features
  disableSearch: process.env.EMERGENCY_MODE === "true",
  disableFavorites: process.env.EMERGENCY_MODE === "true",
  disableReviews: process.env.EMERGENCY_MODE === "true",

  // Reduce data complexity
  limitProductImages: 1,
  limitProductDetails: true,
  enableAggressiveCaching: true,
};

// Apply emergency settings
if (emergencyPerformanceMode.disableSearch) {
  // Disable search functionality
  // Show maintenance message
}
```

📊 Monitoring & Alerting

Application Monitoring

1. Health Checks

```
// Health check endpoint
export async function GET() {
  try {
    // Database health check
    await prisma.$queryRaw`SELECT 1`;

    // External service health checks
    const stripeHealth = await checkStripeHealth();
    const clerkHealth = await checkClerkHealth();

    return NextResponse.json({
      status: "healthy",
      timestamp: new Date().toISOString(),
      services: {
        database: "healthy",
        stripe: stripeHealth ? "healthy" : "unhealthy",
        clerk: clerkHealth ? "healthy" : "unhealthy",
      },
    });
  } catch (error) {
    return NextResponse.json(
      { status: "unhealthy", error: error.message },
      { status: 503 }
    );
  }
}
```

2. Error Tracking

```
// Centralized error handling
export class ApplicationError extends Error {
  constructor(
```

```
    message: string,
    public code: string,
    public statusCode: number = 500,
    public context?: any
  ) {
    super(message);
    this.name = "ApplicationError";
  }
}

// Error logging and alerting
export function logError(error: ApplicationError, context?: any) {
  console.error("Application Error:", {
    message: error.message,
    code: error.code,
    statusCode: error.statusCode,
    context,
    timestamp: new Date().toISOString(),
    stack: error.stack,
  });

  // Send to error tracking service
  sendToErrorTracking(error, context);

  // Send alerts for critical errors
  if (error.statusCode >= 500) {
    sendCriticalErrorAlert(error, context);
  }
}
```

Performance Monitoring

1. Core Web Vitals

```
// Web Vitals monitoring
import { getCLS, getFID, getFCP, getLCP, getTTFB } from "web-vitals";

function sendToAnalytics(metric: any) {
  // Send to analytics service
  console.log("Web Vital:", metric);

  // Alert on poor performance
  if (metric.name === "LCP" && metric.value > 2500) {
    sendPerformanceAlert("Poor LCP performance", metric);
  }
}

// Monitor all web vitals
getCLS(sendToAnalytics);
getFID(sendToAnalytics);
getFCP(sendToAnalytics);
getLCP(sendToAnalytics);
getTTFB(sendToAnalytics);
```

2. Database Performance

```
// Database performance monitoring
export async function monitorDatabasePerformance() {
  try {
    // Monitor query performance
    const slowQueries = await prisma.$queryRaw`
```

```
      SELECT query, mean_time, calls
      FROM pg_stat_statements
      WHERE mean_time > 100
      ORDER BY mean_time DESC
      LIMIT 10
    `;

    if (slowQueries.length > 0) {
      console.warn("Slow database queries detected:", slowQueries);
      sendDatabasePerformanceAlert(slowQueries);
    }

    // Monitor connection pool
    const connectionStats = await prisma.$queryRaw`
      SELECT count(*) as active_connections
      FROM pg_stat_activity
      WHERE state = 'active'
    `;

    if (connectionStats[0].active_connections > 80) {
      sendConnectionPoolAlert(connectionStats[0]);
    }
  } catch (error) {
    console.error("Database monitoring error:", error);
  }
}
```

🔒 Security Maintenance

Security Audits

1. Regular Security Reviews

```
# Weekly security checks
npm audit
npm audit fix

# Monthly dependency security review
npx audit-ci

# Quarterly penetration testing
# Use automated security scanning tools
```

2. Authentication Security

```
// Authentication security monitoring
export async function monitorAuthenticationSecurity() {
  try {
    // Monitor failed login attempts
    const failedLogins = await getFailedLoginAttempts();

    if (failedLogins.length > 10) {
      sendSecurityAlert("Multiple failed login attempts detected");
    }

    // Monitor suspicious activity
    const suspiciousActivity = await detectSuspiciousActivity();

    if (suspiciousActivity.length > 0) {
      sendSecurityAlert("Suspicious activity detected", suspiciousActivity);
```

```
    }
  } catch (error) {
    console.error("Security monitoring error:", error);
  }
}
```

3. Data Protection

```
// Data encryption and protection
export function encryptSensitiveData(data: string): string {
  // Implement encryption for sensitive data
  return crypto
    .createCipher("aes-256-cbc", process.env.ENCRYPTION_KEY)
    .update(data, "utf8", "hex");
}

export function decryptSensitiveData(encryptedData: string): string {
  // Implement decryption for sensitive data
  return crypto
    .createDecipher("aes-256-cbc", process.env.ENCRYPTION_KEY)
    .update(encryptedData, "hex", "utf8");
}
```

📈 Performance Optimization

Continuous Performance Improvement

1. Bundle Analysis

```
# Weekly bundle analysis
npm run build
npx @next/bundle-analyzer

# Monitor bundle size trends
# Identify large dependencies
# Optimize imports and code splitting
```

2. Image Optimization

```
// Image optimization monitoring
export function monitorImagePerformance() {
  // Monitor image loading times
  // Track image format usage
  // Optimize image delivery
  // Implement lazy loading strategies
}
```

3. Caching Strategies

```
// Implement strategic caching
export const cacheConfig = {
  // API response caching
  apiCache: {
    products: 300, // 5 minutes
    regions: 3600, // 1 hour
```

```
    userProfile: 1800, // 30 minutes
  },

  // Static asset caching
  staticCache: {
    images: 86400, // 24 hours
    css: 3600, // 1 hour
    js: 3600, // 1 hour
  },
};
```

📚 Documentation Maintenance

Documentation Standards

1. Code Documentation

```
/**
 * Adds a wine to the user's shopping cart
 *
 * @param wineId - The unique identifier of the wine
 * @param quantity - The quantity to add (default: 1)
 * @param userId - The authenticated user's ID
 *
 * @returns Promise<Cart> - The updated cart object
 *
 * @throws {AuthenticationError} When user is not authenticated
 * @throws {ValidationError} When wineId or quantity is invalid
 * @throws {DatabaseError} When database operation fails
 *
 * @example
 * ```typescript
 * const cart = await addToCart('123', 2, 'user_456');
 * console.log(`Added ${cart.numItemsInCart} items to cart`);
 * ```
 */
export async function addToCart(
  wineId: string,
  quantity: number = 1,
  userId: string
): Promise<Cart> {
  // Implementation details...
}
```

2. API Documentation

```
/**
 * @api {GET} /api/products Get Products
 * @apiName GetProducts
 * @apiGroup Products
 * @apiVersion 1.0.0
 *
 * @apiQuery {Number} [page=1] Page number for pagination
 * @apiQuery {Number} [limit=12] Number of items per page
 * @apiQuery {String} [search] Search term for wine names
 * @apiQuery {String} [type] Filter by wine type
 * @apiQuery {String} [region] Filter by wine region
 * @apiQuery {Number} [minPrice] Minimum price filter
 * @apiQuery {Number} [maxPrice] Maximum price filter
 *
```

```
 * @apiSuccess {Object[]} wines Array of wine objects
 * @apiSuccess {Object} pagination Pagination metadata
 * @apiSuccess {Number} pagination.page Current page
 * @apiSuccess {Number} pagination.total Total number of wines
 * @apiSuccess {Number} pagination.totalPages Total number of pages
 *
 * @apiError {Object} 500 Internal server error
 * @apiErrorExample {json} Error-Response:
 *     HTTP/1.1 500 Internal Server Error
 *     {
 *       "error": "Failed to fetch products"
 *     }
 */
```

Documentation Maintenance Schedule

1. Code Documentation

- **Inline Comments:** Update with code changes
- **Function Documentation:** Review monthly
- **API Documentation:** Update with API changes
- **Architecture Documentation:** Review quarterly

2. User Documentation

- **User Guides:** Update with feature changes
- **API References:** Maintain with API updates
- **Troubleshooting:** Update based on user feedback
- **Release Notes:** Create for each release

🚀 Deployment & Release Management

Deployment Process

1. Pre-deployment Checklist

- ☐ **Code Review:** All changes reviewed and approved
- ☐ **Testing:** All tests passing
- ☐ **Performance:** Performance benchmarks met
- ☐ **Security:** Security scan completed
- ☐ **Documentation:** Documentation updated
- ☐ **Backup:** Database backup completed
- ☐ **Rollback Plan:** Rollback procedure prepared

2. Deployment Steps

```
# 1. Create release branch
git checkout -b release/v1.1.0

# 2. Update version numbers
npm version patch  # or minor/major

# 3. Final testing
npm run test
npm run build
npm run lint

# 4. Merge to main
git checkout main
git merge release/v1.1.0
```

```
# 5. Tag release
git tag -a v1.1.0 -m "Release v1.1.0"
git push origin v1.1.0

# 6. Deploy to production
vercel --prod
```

3. Post-deployment Verification

- [ ] **Health Check:** Application health verified
- [ ] **Functionality:** Core features tested
- [ ] **Performance:** Performance metrics checked
- [ ] **Monitoring:** Monitoring systems verified
- [ ] **User Feedback:** Monitor user feedback
- [ ] **Error Logs:** Check for new errors

Release Management

1. Release Types

- **Patch Releases:** Bug fixes and minor improvements
- **Minor Releases:** New features and enhancements
- **Major Releases:** Breaking changes and major features

2. Release Notes Template

```
# Release v1.1.0

## 🆕 New Features

- Added advanced product filtering
- Implemented user favorites system
- Enhanced mobile navigation

## 🐛 Bug Fixes

- Fixed cart quantity update issue
- Resolved mobile logo duplication
- Fixed search performance issues

## 🔧 Improvements

- Optimized database queries
- Enhanced error handling
- Improved loading states

## 📱 Mobile Experience

- Better touch interactions
- Improved responsive design
- Enhanced mobile performance

## 🔒 Security

- Updated authentication flow
- Enhanced input validation
- Improved error logging

## 📚 Documentation

- Updated API documentation
```

```
- Added component guides
- Enhanced troubleshooting guides
```

This maintenance and development guide provides comprehensive guidance for maintaining and developing the VINEFOX wine store application. Regular updates to this document ensure it remains current with the evolving codebase and development practices.