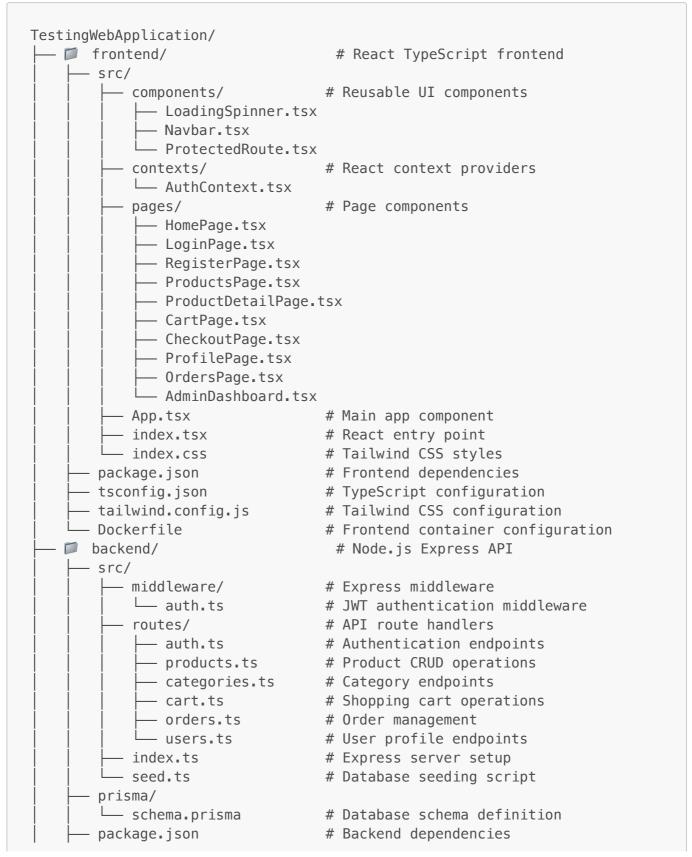
# **E-Commerce Test Application**

# Comprehensive Guide for Page Object Model Testing

# Project Structure





# Key Features for Testing

# **Authentication System**

- **V** User Registration: Complete form with validation
  - Email validation
  - Password strength requirements
  - o Duplicate email handling
  - Success/error feedback
- **V** User Login: Authentication with error handling
  - Email and password validation
  - Invalid credential handling
  - JWT token management
  - Automatic logout on token expiration

### Role-Based Access Control

- USER role: Standard e-commerce functionality
- o ADMIN role: Administrative dashboard access
- Protected route redirects
- o Permission-based UI elements

## **E-Commerce Features**

### • V Product Catalog

- Product listing with pagination
- Search functionality
- Category filtering
- Price range filtering
- Product detail views

# Shopping Cart

- Add products to cart
- Update item quantities
- o Remove items from cart
- Cart persistence across sessions
- o Real-time cart total calculations

#### Checkout Process

- Order summary review
- Order placement
- o Order confirmation
- Order history tracking

# • V User Profile Management

- View profile information
- o Update account details
- o Order history access

### • **V** Admin Dashboard

- Product management (CRUD operations)
- User management
- o Order monitoring
- Category management

# Testing-Friendly Design

### ▼ Comprehensive data-testid Attributes

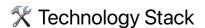
- o All interactive elements tagged
- Consistent naming conventions
- o Form fields and buttons identified
- Navigation elements tagged

### Z Error States and Loading States

- Loading spinners for async operations
- Error message displays
- Form validation feedback
- Network error handling

### • **V** Clear Page Navigation Structure

- o Predictable URL patterns
- o Breadcrumb navigation
- Protected route handling
- o Role-based menu items



# Frontend Technologies

- React 18: Latest React version with modern hooks
- TypeScript: Type safety and better development experience
- Tailwind CSS: Utility-first CSS framework for styling
- React Router: Client-side routing and navigation
- React Query: Server state management and caching
- Axios: HTTP client for API requests
- React Hot Toast: User-friendly notification system

### **Backend Technologies**

- Node.js: JavaScript runtime environment
- Express: Web application framework
- TypeScript: Type-safe server-side development
- Prisma ORM: Database toolkit and query builder
- PostgreSQL: Robust relational database
- JWT: JSON Web Tokens for authentication
- bcryptjs: Password hashing and validation
- Joi: Request validation library

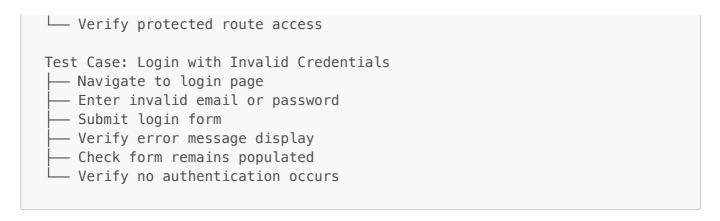
#### Infrastructure

- **Docker Compose**: Multi-container application orchestration
- PostgreSQL Database: Persistent data storage
- **Nginx**: Production-ready web server (configurable)
- VS Code: Integrated development environment support

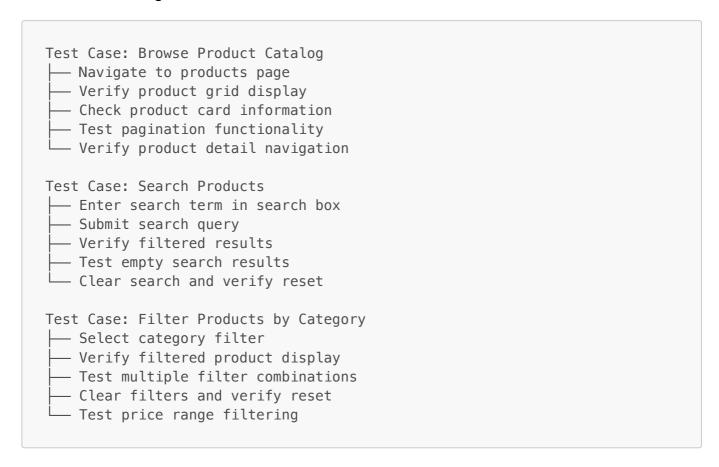
# **©** Perfect Test Scenarios

# 1. User Registration & Authentication Flow

Test Case: Complete User Registration  Navigate to registration page Fill out registration form Enter valid email address Enter secure password Enter first and last name Submit form Verify successful registration Check automatic login	
└── Verify user dashboard access	
Test Case: Login with Valid Credentials	

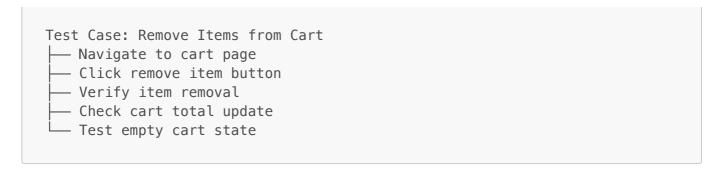


# 2. Product Browsing & Search

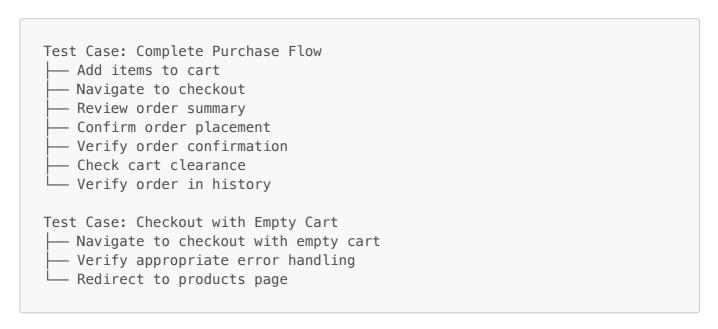


## 3. Shopping Cart Management

Test Case: Add Products to Cart  Navigate to product detail page Select product options (if any) Click "Add to Cart" button Verify cart icon update Navigate to cart page Verify product in cart	
Test Case: Update Cart Quantities  Navigate to cart page Modify item quantity Verify price recalculation Test quantity limits Verify cart total updates	



### 4. Checkout Process



# 5. Admin Functions

Test Case: Admin Dashboard Access  — Login with admin credentials  — Verify admin menu visibility  — Navigate to admin dashboard  — Check admin-only features  — Verify user management access	
Test Case: Product Management  — Access admin product management  — Create new product  — Edit existing product  — Delete product  — Verify changes in catalog	
Test Case: Role-Based Access Control  Login as regular user  Attempt admin dashboard access  Verify access denial  Check hidden admin menu items  Test protected admin routes	



# **Pre-configured Test Accounts**

### Sample Product Data

The application includes comprehensive sample data:

# **Categories:**

- Electronics (MacBook Pro, iPhone, Headphones, iPad)
- Clothing (T-Shirts, Denim Jackets, Running Shoes)
- Books (Programming, Design, Technical manuals)

### **Product Attributes:**

- Unique product IDs
- Names and descriptions
- Price points (\$29.99 \$2,499.00)
- Stock quantities
- High-quality placeholder images
- · Featured product flags
- Category associations

# API Endpoints Reference

### **Authentication Endpoints**

```
— Returns: { user, token }
— Status: 200 OK | 401 Unauthorized
```

### **Product Endpoints**

```
GET /api/products
Query: ?category&search&minPrice&maxPrice&page&limit
  - Returns: { products[], pagination }
Status: 200 OK
GET /api/products/:id
Returns: { product }
└── Status: 200 OK | 404 Not Found
POST /api/products (Admin Only)
Body: { name, description, price, categoryId, ... }
  - Returns: { product }
└── Status: 201 Created | 403 Forbidden
PUT /api/products/:id (Admin Only)
Body: { name, description, price, ... }
  - Returns: { product }
└── Status: 200 OK | 403 Forbidden | 404 Not Found
DELETE /api/products/:id (Admin Only)
 — Returns: { message }
└── Status: 200 OK | 403 Forbidden | 404 Not Found
```

### Cart Management Endpoints

```
GET /api/cart
 — Headers: Authorization: Bearer <token>
  — Returns: { cartItems[] }
└── Status: 200 OK | 401 Unauthorized
POST /api/cart/add
Headers: Authorization: Bearer <token>
Body: { productId, quantity }
  - Returns: { cartItem }
└── Status: 201 Created | 401 Unauthorized
PUT /api/cart/:id
 — Headers: Authorization: Bearer <token>
— Body: { quantity }
  - Returns: { cartItem }
└── Status: 200 OK | 401 Unauthorized
DELETE /api/cart/:id
Headers: Authorization: Bearer <token>
```

```
—— Returns: { message }
—— Status: 200 OK | 401 Unauthorized
```

### **Order Management Endpoints**

```
GET /api/orders

- Headers: Authorization: Bearer <token>
- Returns: { orders[] }
- Status: 200 OK | 401 Unauthorized

POST /api/orders/checkout
- Headers: Authorization: Bearer <token>
- Returns: { order }
- Status: 201 Created | 400 Bad Request | 401 Unauthorized
```

# **d** Getting Started

# Prerequisites

Before running the application, ensure you have one of the following setups:

### **Option 1: Docker Setup (Recommended)**

- Docker Desktop installed and running
- Docker Compose available
- Minimum 4GB RAM allocated to Docker
- Ports 3000, 5000, and 5432 available

## **Option 2: Manual Setup**

- Node.js 18+ installed
- npm or yarn package manager
- PostgreSQL 12+ database server
- Git for version control

### Quick Start with Docker

### 1. Clone and Navigate

```
git clone <repository-url>
cd TestingWebApplication
```

### 2. Start All Services

```
docker-compose up --build
```

### 3. Wait for Services to Start

Database initialization: ~30 seconds

• Backend API server: ~60 seconds

• Frontend development server: ~90 seconds

### 4. Access the Application

Frontend: http://localhost:3000Backend API: http://localhost:5000

o Database: localhost:5432

### 5. Verify Setup

Navigate to http://localhost:3000

- o Click "Login" and use test credentials
- Explore the application features

# Manual Development Setup

### 1. Database Setup

```
# Install and start PostgreSQL
# Create database: ecommerce_db
# Create user: ecommerce_user
```

### 2. Backend Setup

```
cd backend
cp .env.example .env
# Edit .env with your database credentials
npm install
npx prisma migrate dev
npx prisma generate
npm run seed
npm run dev
```

### 3. Frontend Setup

```
cd frontend
npm install
npm start
```

# **Environment Configuration**

#### **Backend Environment Variables (.env)**

```
DATABASE_URL=postgresql://ecommerce_user:ecommerce_password@localhost:5432
/ecommerce_db
JWT_SECRET=your-super-secret-jwt-key-change-in-production
NODE_ENV=development
PORT=5000
FRONTEND_URL=http://localhost:3000
```

### Frontend Environment Variables (.env)

```
REACT_APP_API_URL=http://localhost:5000/api
```

## **VS Code Integration**

The project includes VS Code tasks for streamlined development:

#### **Available Tasks:**

- "Build and Run E-Commerce App": Starts Docker Compose
- Backend development server
- Frontend development server
- Database migrations
- Test execution

### **Usage:**

- 1. Open VS Code in project directory
- 2. Press Cmd/Ctrl + Shift + P
- 3. Type "Tasks: Run Task"
- 4. Select desired task from the list

**Troubleshooting Common Issues** 

#### **Port Conflicts:**

- Check if ports 3000, 5000, or 5432 are in use
- Stop conflicting services or change ports in docker-compose.yml

### **Docker Build Failures:**

- Ensure Docker Desktop is running
- Clear Docker cache: docker system prune -a
- Rebuild containers: docker-compose up --build --force-recreate

#### **Database Connection Issues:**

Verify PostgreSQL container is running

- Check database credentials in environment variables
- Wait for database initialization to complete

### **Frontend/Backend Communication:**

- Verify backend is running on port 5000
- · Check CORS configuration in backend
- Confirm API URL in frontend environment variables

# Additional Resources

## **Development Tools**

- VS Code Extensions: TypeScript, Tailwind CSS IntelliSense, Prisma
- Database Management: pgAdmin, DBeaver, or Prisma Studio
- API Testing: Postman, Thunder Client, or curl
- Container Management: Docker Desktop GUI

### **Testing Frameworks Integration**

This application is designed to work seamlessly with:

- Selenium WebDriver: Java, Python, C#, JavaScript
- Cypress: JavaScript/TypeScript end-to-end testing
- Playwright: Multi-browser automation
- WebDriverIO: Node.js testing framework
- TestCafe: JavaScript testing without WebDriver

### Next Steps for Test Implementation

- 1. Choose your preferred testing framework
- 2. Set up Page Object Model structure
- 3. Create base page classes
- 4. Implement page-specific classes
- 5. Write comprehensive test suites
- 6. Set up CI/CD integration
- 7. Generate test reports

This comprehensive guide provides everything needed to get started with Page Object Model testing using a modern, full-stack web application. The application includes real-world complexity while maintaining clean, testable architecture.