E-Commerce Platform Project

CS3365 – Software Engineering

Project #1 and #2

Dr. Maaz Amjad

Spring 2025

Table of Contents

Introduction

In Project 1 & 2, students will delve into the essential components of software engineering by constructing a **full-stack E-Commerce Platform**. These projects are meticulously crafted to provide you with practical experience in modern web development, bridging the gap between theoretical concepts we covered in class and real-world applications.

Purpose of the Projects

The primary aim of these projects is to enhance your understanding of how sophisticated software systems are built using industry-standard technologies. By engaging in the development of a front-end interface and back-end integration, you will explore the critical processes of modern software development, testing, and deployment, which are integral to successful software engineering.

How These Projects will Help You Understand Software Engineering

Front-End Development (Project #1) – Individual Project

- Component-Based Architecture: Learn how modern UI is constructed using reusable components in React.
- State Management: Implement state handling techniques for dynamic user interfaces.
- Testing Fundamentals: Apply unit testing principles to ensure code quality and functionality.

Advanced Features and Integration (Project #2) – Group Project

- Full-Stack Implementation: Connect front-end interfaces with back-end services through APIs.
- Database Integration: Implement data persistence using NoSQL database technologies.
- Quality Assurance: Apply multiple testing methodologies to ensure system reliability and performance.

Educational Outcomes

By completing these projects, you will:

- **Gain Practical Skills:** Develop hands-on experience in writing components that are critical to modern web applications.
- **Understand Software Architecture:** Build a foundational knowledge of how full-stack applications work internally, preparing you for more advanced topics in software engineering.
- **Enhance Problem-Solving Abilities:** Tackle common challenges in software development, such as state management, API integration, and user experience optimization.
- **Connect Theory with Practice:** Apply theoretical concepts from lectures to practical coding tasks, solidifying your understanding through implementation.

These projects are not just academic exercises; they are essential steps toward mastering the complexities of modern software engineering. By the end of this journey, students will have a deeper appreciation for the intricacies involved in building comprehensive web applications, a skill set highly valued in the field of computer science and software engineering.

E-Commerce Platform Overview

Features and Requirements

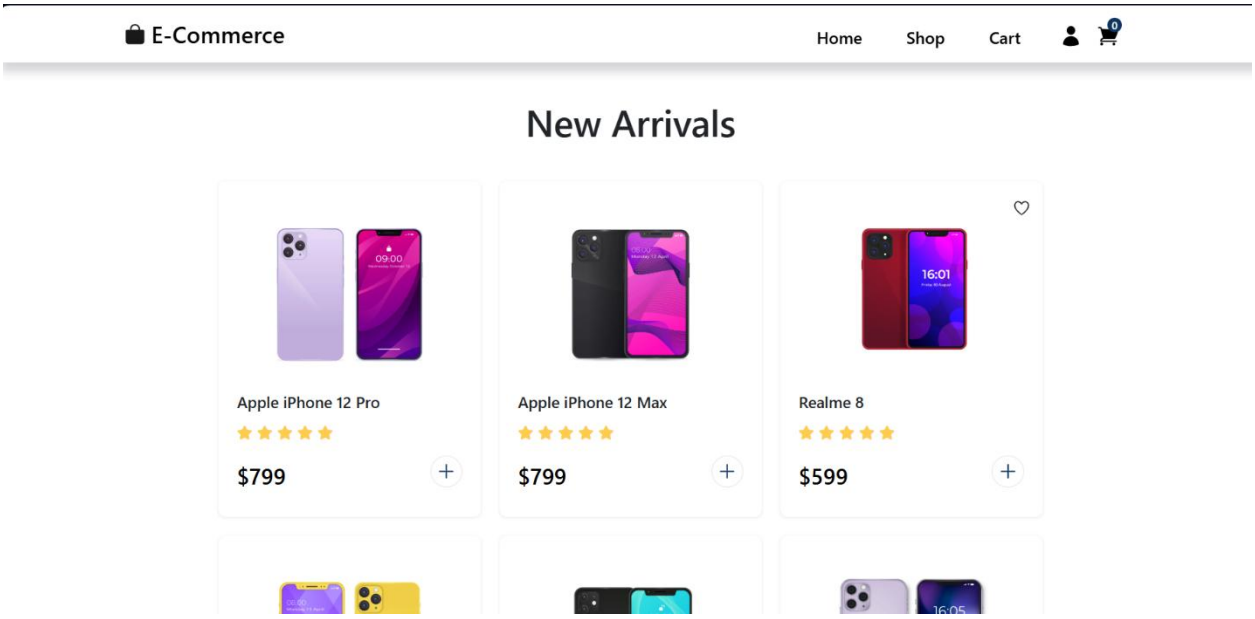
In these projects, you will be tasked with developing a comprehensive E-Commerce Platform capable of handling product browsing, user interactions, and transaction processing. The platform will involve both front-end and back-end components with the following core functionality:

1. User Interface Components:
    - Product Catalog Display
    - Shopping Cart Management
    - User Authentication (Login/Registration)
    - Search and Filter Functionality
    - Product Detail Views
    - Checkout Process Flow
2. Back-End Systems:
    - RESTful API Services
    - Database Integration
    - User Profile Management
    - Order Processing Logic
    - Payment Gateway Integration
    - Security Implementation
3. Testing Requirements:
    - Unit Tests for Components and Functions
    - Integration Tests for API Endpoints
    - End-to-End Tests for Critical User Flows
    - Performance Testing for Load Handling

Keep in mind that the requirements establish a framework for your implementation, but creative solutions that enhance functionality or user experience beyond these requirements are encouraged.

Example Expected Outcome

The image provided below illustrates an example of the expected front-end implementation for the E-commerce platform. It features a clean and user-friendly **React.js interface**, including reusable components such as product cards with clear pricing, ratings, and interactive elements like a favorite button and add-to-cart functionality. This UI demonstrates the emphasis on responsiveness, clarity, and ease of navigation, aligning with modern design principles essential for an engaging online shopping experience.


E-Commerce
Home
Shop
Cart
0
New Arrivals
09:00
Apple iPhone 12 Pro
$799
Apple iPhone 12 Max
$799
16:01
Realme 8
$599

Project #1 – Front-End Development

Problem

Develop a responsive front-end interface using React.js that provides the core e-commerce functionalities and user experience. This front-end should be designed with component reusability and state management best practices in mind, while ensuring proper unit testing coverage.

The e-commerce platform should allow users to browse products, add items to a cart, manage their cart, and proceed through a checkout flow. While the back-end integration will be implemented in Project #2, the front-end should be designed to easily connect with APIs in the future.

Your application should be structured according to modern React development practices, with clear separation of concerns and organization of components, hooks, and utilities.

Project Requirements

Your solution must conform to the following rules:

1. Component Structure:
    - Implement a minimum of 8 reusable React components including:
        - Navigation Bar
        - Product Card
        - Product List/Grid
        - Shopping Cart
        - Search Component
        - Filter Component
        - User Authentication Forms
        - Checkout Form
    - Organize components using a logical directory structure
2. Routing and Navigation:
    - Implement React Router with the following routes:
        - Home/Product Listing Page
        - Product Details Page
        - Shopping Cart Page
        - Checkout Page
        - User Profile Page
        - Login/Registration Pages
3. State Management:
    - Use appropriate state management techniques (useState, useContext, or Redux)
    - Implement a central store for product data and cart information

    - Maintain user session information in state
4. Responsive Design:
    - Ensure the application is fully responsive for mobile, tablet, and desktop viewports
    - Implement proper CSS organization (CSS Modules, Styled Components, or similar approach)
5. Testing Requirements:
    - Write unit tests for all components using Jest and React Testing Library
    - Achieve at least 70% test coverage for React components
    - Include tests for critical user flows (e.g., add to cart, checkout)
6. Mock Data Integration:
    - Create and utilize mock product data for display
    - Implement mock services that simulate API calls for future backend integration
7. Code Quality:
    - Adhere to React best practices and patterns
    - Include proper error handling for user interactions
    - Provide comprehensive documentation via comments and README
8. Your solution must print out "E-Commerce Project :: <Your Name>" in the footer of each page.
9. Code that fails to compile or run for any reason will result in significant point deductions. Ensure what you turn in compiles and works by testing the exact files you turn in.
    - It is not the responsibility of the grader to correct your code.

Component Description

1. Navigation Bar Component:

- Display site logo and name
- Provide links to main pages (Home, Cart, Profile)
- Include a search bar component
- Show cart item count
- Responsive design with mobile menu toggle

2. Product Card Component:

- Display product image
- Show product name, price, and brief description
- Include "Add to Cart" button
- Display product rating if available
- Handle image loading states and errors

3. Product List/Grid Component:

- Arrange Product Cards in a responsive grid layout
- Implement pagination or infinite scroll
- Allow toggling between grid and list views
- Include sorting functionality (by price, popularity, etc.)
- Demonstrate proper performance optimization

4. Shopping Cart Component:

- List all items added to cart with quantities
- Allow quantity adjustments and item removal
- Calculate and display subtotal, taxes, and total
- Include "Proceed to Checkout" button
- Handle empty cart state appropriately

5. Search Component:

- Provide text input for product search
- Implement search suggestions/autocomplete
- Display search results or "No results" message
- Include search history functionality
- Demonstrate debouncing for performance

6. Filter Component:

- Allow filtering products by category
- Include price range selection
- Provide options for additional filters (size, color, etc.)
- Allow multiple simultaneous filters
- Maintain filter state across page navigation

7. User Authentication Forms:

- Implement login form with email/password fields
- Create registration form with validation
- Handle form submission and error states
- Include password reset functionality
- Store authentication state appropriately

8. Checkout Form:

- Create multi-step checkout process
- Gather shipping information

- Collect payment details
- Provide order summary
- Implement form validation
- Show confirmation message upon submission

Example Implementation

The example implementation below represents a basic structure for the Product Card component in React:

```jsx
// ProductCard.jsx
import React from 'react';
import PropTypes from 'prop-types';
import './ProductCard.css';

const ProductCard = ({ product, onAddToCart }) => {
  const { id, name, price, image, description, rating } = product;

  return (
    <div className="product-card" data-testid="product-card">
      <div className="product-image">
        <img src={image} alt={name} onError={(e) => {
          e.target.onerror = null;
          e.target.src = '/placeholder-image.jpg';
        }} />
        {rating && (
          <div className="product-rating">
            <span>★</span> {rating.toFixed(1)}
          </div>
        )}
      </div>
      <div className="product-info">
        <h3 className="product-name">{name}</h3>
        <p className="product-price">${price.toFixed(2)}</p>
        <p className="product-description">{description.substring(0, 100)}...</p>
        <button
          className="add-to-cart-button"
          onClick={() => onAddToCart(id)}
        >
          Add to Cart
        </button>
      </div>
    </div>
  );
};
```

```
ProductCard.propTypes = {
  product: PropTypes.shape({
    id: PropTypes.number.isRequired,
    name: PropTypes.string.isRequired,
    price: PropTypes.number.isRequired,
    image: PropTypes.string.isRequired,
    description: PropTypes.string.isRequired,
    rating: PropTypes.number
  }).isRequired,
  onAddToCart: PropTypes.func.isRequired
};

export default ProductCard;
```

And the corresponding test file:

```
// ProductCard.test.jsx
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import ProductCard from './ProductCard';

const mockProduct = {
  id: 1,
  name: 'Test Product',
  price: 19.99,
  image: '/test-image.jpg',
  description: 'This is a test product description that is relatively long.',
  rating: 4.5
};

test('renders product card with correct information', () => {
  const mockAddToCart = jest.fn();
  render(<ProductCard product={mockProduct} onAddToCart={mockAddToCart} />);

  expect(screen.getByText('Test Product')).toBeInTheDocument();
  expect(screen.getByText('$19.99')).toBeInTheDocument();
  expect(screen.getByText('This is a test product description that is relatively long...')).toBeInTheDocument();
  expect(screen.getByText('4.5')).toBeInTheDocument();
  expect(screen.getByRole('img')).toHaveAttribute('src', '/test-image.jpg');
  expect(screen.getByRole('img')).toHaveAttribute('alt', 'Test Product');
});

test('calls onAddToCart with product id when add to cart button is clicked', () => {
  const mockAddToCart = jest.fn();
  render(<ProductCard product={mockProduct} onAddToCart={mockAddToCart} />);
```

```
  fireEvent.click(screen.getByText('Add to Cart'));
  expect(mockAddToCart).toHaveBeenCalledWith(1);
});

test('handles missing rating gracefully', () => {
  const productWithoutRating = { ...mockProduct, rating: null };
  const mockAddToCart = jest.fn();
  render(<ProductCard product={productWithoutRating} onAddToCart={mockAddToCart} />);

  expect(screen.queryByText('★')).not.toBeInTheDocument();
});
```

Hints

1. Plan your component hierarchy before starting to code:
    - Draw a diagram of your components and how they interact
    - Decide on your data structures early in the process
2. Use a component library to accelerate development:
    - Material-UI, Chakra UI, or Ant Design can provide pre-styled components
    - Focus on functionality before customizing appearance
3. Implement a consistent state management approach:
    - For smaller applications, React Context with useReducer may be sufficient
    - For larger applications, consider Redux or another dedicated state library
4. Write tests alongside components, not after:
    - Practice Test-Driven Development (TDD) when possible
    - Focus on testing component behavior rather than implementation details
5. Use mock data during development:
    - Create a comprehensive mock data structure for products
    - Simulate API responses for different scenarios
6. Optimize your development workflow:
    - Set up hot module reloading
    - Use React developer tools for debugging
    - Implement linting and code formatting rules

What to turn into Blackboard

A zip archive (.zip) containing the following files:

- All source code files for your React application

- package.json and package-lock.json
- README.md with:
    - Setup instructions
    - Description of implemented features
    - Testing instructions
    - Any third-party libraries used and why
- Test files and configuration
- Screenshots of your application's key pages (min. 5 screenshots)

Project #2 – Advanced Features and System Integration

Problem Statement

Building upon your front-end implementation from Project #1, develop a fully integrated e-commerce platform by implementing back-end services, connecting to a database, and adding advanced features such as payment processing, user authentication, and order management. This project focuses on creating a cohesive system where front-end and back-end components work seamlessly together.

Your solution should demonstrate the best practices in API design, security implementation, and system integration while maintaining high standards for testing and code quality.

Project Requirements

Your full-stack application should conform to the following rules:

1. Back-End Development:
    - Implement a Node.js/Express.js server with RESTful API endpoints for:
        - Products (GET, POST, PUT, DELETE)
        - Users (GET, POST, PUT)
        - Orders (GET, POST, PUT)
        - Authentication (Register, Login, Logout)
    - Connect to a NoSQL database (MongoDB recommended)
    - Implement proper error handling and response formatting
    - Include middleware for authentication, logging, and error handling
2. Database Integration:
    - Design and implement appropriate data models for:
        - Products (with categories, images, pricing)
        - Users (personal information, addresses, payment methods)
        - Orders (items, quantities, status, shipping information)
        - Reviews (ratings, comments, user references)
    - Implement validation for data integrity
    - Create appropriate indexes for query optimization
3. Advanced Features:
    - Implement JWT-based authentication
    - Connect to at least one third-party API (e.g., payment gateway, shipping calculator)
    - Create an order processing workflow
    - Implement at least one complex feature from:
        - Product recommendation system

- Inventory management
- Discount/coupon system
- User reviews and ratings

4. API Integration:
    - Connect your React front-end to the back-end services
    - Implement proper state management for API interactions
    - Handle loading, error, and success states for all API calls
    - Secure API requests with JWT tokens
5. Testing Requirements:
    - Write unit tests for API endpoints using Mocha/Chai or Jest
    - Implement integration tests for critical workflows
    - Add end-to-end tests using Cypress for at least 3 user journeys
    - Achieve at least 70% test coverage for back-end code
6. Non-Functional Requirements:
    - Implement input validation and sanitization
    - Add basic security measures (HTTPS, authentication, authorization)
    - Optimize API performance (caching, pagination)
    - Document all API endpoints using Swagger/OpenAPI
7. Deployment:
    - Deploy the full application using appropriate platforms
    - Provide configuration for development and production environments
    - Include deployment instructions in documentation
8. Your solution must print out "E-Commerce Project :: <Your Name>" in the footer of each page.
9. Code that fails to compile or run for any reason will result in significant point deductions. Ensure what you turn in compiles and works by testing the exact files you turn in.
    - It is not the responsibility of the grader to correct your code.

Example Implementation

The example implementations below demonstrate how to create an Express.js API endpoint and connect it to your React front-end:

Backend API Endpoint (Node.js/Express):

```
// productController.js
const Product = require('../models/Product');
```

```
// Get all products with optional filtering
const getProducts = async (req, res) => {
  try {
    const { category, minPrice, maxPrice, search } = req.query;

    // Build filter object based on query parameters
    const filter = {};

    if (category) {
      filter.category = category;
    }

    if (minPrice || maxPrice) {
      filter.price = {};
      if (minPrice) filter.price.$gte = parseFloat(minPrice);
      if (maxPrice) filter.price.$lte = parseFloat(maxPrice);
    }

    if (search) {
      filter.$or = [
        { name: { $regex: search, $options: 'i' } },
        { description: { $regex: search, $options: 'i' } }
      ];
    }

    // Execute query with pagination
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    const products = await Product.find(filter)
      .sort({ createdAt: -1 })
      .skip(skip)
      .limit(limit);

    const total = await Product.countDocuments(filter);

    res.status(200).json({
      success: true,
      count: products.length,
      total,
      totalPages: Math.ceil(total / limit),
      currentPage: page,
      products
    });
```

```
  } catch (error) {
    console.error('Error fetching products:', error);
    res.status(500).json({
      success: false,
      message: 'Server Error',
      error: process.env.NODE_ENV === 'development' ? error.message : undefined
    });
  }
};

// Get a single product by ID
const getProductById = async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({
        success: false,
        message: 'Product not found'
      });
    }

    res.status(200).json({
      success: true,
      product
    });
  } catch (error) {
    console.error('Error fetching product:', error);

    // Handle case where ID format is invalid
    if (error.kind === 'ObjectId') {
      return res.status(404).json({
        success: false,
        message: 'Product not found'
      });
    }

    res.status(500).json({
      success: false,
      message: 'Server Error',
      error: process.env.NODE_ENV === 'development' ? error.message : undefined
    });
  }
};
```

```
module.exports = {
  getProducts,
  getProductById
  // Other controller methods would be included here
};
```

React Integration with API:

```
// hooks/useProducts.js
import { useState, useEffect } from 'react';
import axios from 'axios';

const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000/api';

const useProducts = (initialFilters = {}) => {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  const [filters, setFilters] = useState(initialFilters);
  const [pagination, setPagination] = useState({
    currentPage: 1,
    totalPages: 1,
    total: 0
  });

  const fetchProducts = async () => {
    setLoading(true);
    setError(null);

    try {
      // Build query string from filters
      const queryParams = new URLSearchParams();

      if (filters.category) queryParams.set('category', filters.category);
      if (filters.minPrice) queryParams.set('minPrice', filters.minPrice.toString());
      if (filters.maxPrice) queryParams.set('maxPrice', filters.maxPrice.toString());
      if (filters.search) queryParams.set('search', filters.search);
      if (filters.page) queryParams.set('page', filters.page.toString());
      if (filters.limit) queryParams.set('limit', filters.limit.toString());

      const response = await axios.get(`${API_URL}/products?${queryParams.toString()}`);

      setProducts(response.data.products);
      setPagination({
        currentPage: response.data.currentPage,
```

```
        totalPages: response.data.totalPages,
        total: response.data.total
      });
    } catch (err) {
      console.error('Error fetching products:', err);
      setError(err.response?.data?.message || 'Failed to fetch products');
    } finally {
      setLoading(false);
    }
  };

  // Update filters and trigger a new fetch
  const updateFilters = (newFilters) => {
    setFilters(prev => ({
      ...prev,
      ...newFilters,
      page: 1 // Reset to first page when filters change
    }));
  };

  // Change page
  const goToPage = (page) => {
    setFilters(prev => ({
      ...prev,
      page
    }));
  };

  // Fetch products when filters change
  useEffect(() => {
    fetchProducts();
  }, [filters]);

  return {
    products,
    loading,
    error,
    pagination,
    updateFilters,
    goToPage,
    refreshProducts: fetchProducts
  };
};

export default useProducts;
```

Hints

1. Plan your API architecture before coding:
    - Create a comprehensive API documentation
    - Define data models and relationships
    - Establish authentication and authorization rules
2. Implement strong security practices:
    - Sanitize all user inputs
    - Implement proper authentication and authorization
    - Follow security best practices for JWT implementation
3. Use environment variables for configuration:
    - Never hardcode sensitive information (API keys, database credentials)
    - Create separate configurations for development and production
4. Apply proper error handling throughout your application:
    - Create consistent error response formats
    - Log errors with appropriate detail
    - Present user-friendly error messages in the UI
5. Implement database operations carefully:
    - Use appropriate indexes for frequent queries
    - Write efficient database queries
    - Consider data validation at both application and database levels
6. Test across different environments:
    - Verify functionality in development and production settings
    - Test with various network conditions and loads
    - Validate all critical user flows end-to-end
7. Consider deployment requirements early:
    - Understand hosting platform constraints
    - Plan for database deployment and connection
    - Consider CI/CD pipeline integration

What to turn in to Blackboard

A zip archive (.zip) containing the following files:

- All source code files for your full-stack application (front-end and back-end)
- Database scripts or configuration files
- package.json and package-lock.json for both client and server

- .env.example file (without actual sensitive information)
- README.md with:
    - Comprehensive setup instructions
    - API documentation
    - Features implemented
    - Testing instructions
    - Deployment procedure
    - Third-party services/APIs used
- Test files and configuration
- Screenshots or screen recordings demonstrating core functionalities
- Link to deployed application (if applicable)

Extra Credit Opportunity (Optional): Performance Optimization Strategy (+15)

This extra credit opportunity requires you to develop and implement a comprehensive performance optimization strategy for your e-commerce platform. Your strategy should address both front-end and back-end performance and include measurable improvements.

Requirements for the extra credit:

1. Create a document detailing your optimization strategy, including:
    - Identified performance bottlenecks
    - Specific optimization techniques applied
    - Metrics and tools used for measurement
    - Before and after performance comparison
2. Implement at least 5 of the following optimizations:
    - Front-end code splitting and lazy loading
    - Image optimization pipeline
    - Effective caching strategies (browser, API, database)
    - Database query optimization
    - Server-side rendering or static generation for critical pages
    - API response optimization (compression, pagination, field selection)
    - Resource minification and bundling
    - Advanced React optimization techniques (memoization, virtualization)
3. Provide performance test results demonstrating measurable improvements:
    - Lighthouse scores before and after optimization
    - Load time measurements for key pages
    - API response time improvements
    - Database query performance enhancements

Your optimization strategy will be evaluated based on the comprehensiveness of your approach, the effectiveness of implemented techniques, and the measurable performance improvements achieved.