enigmaX

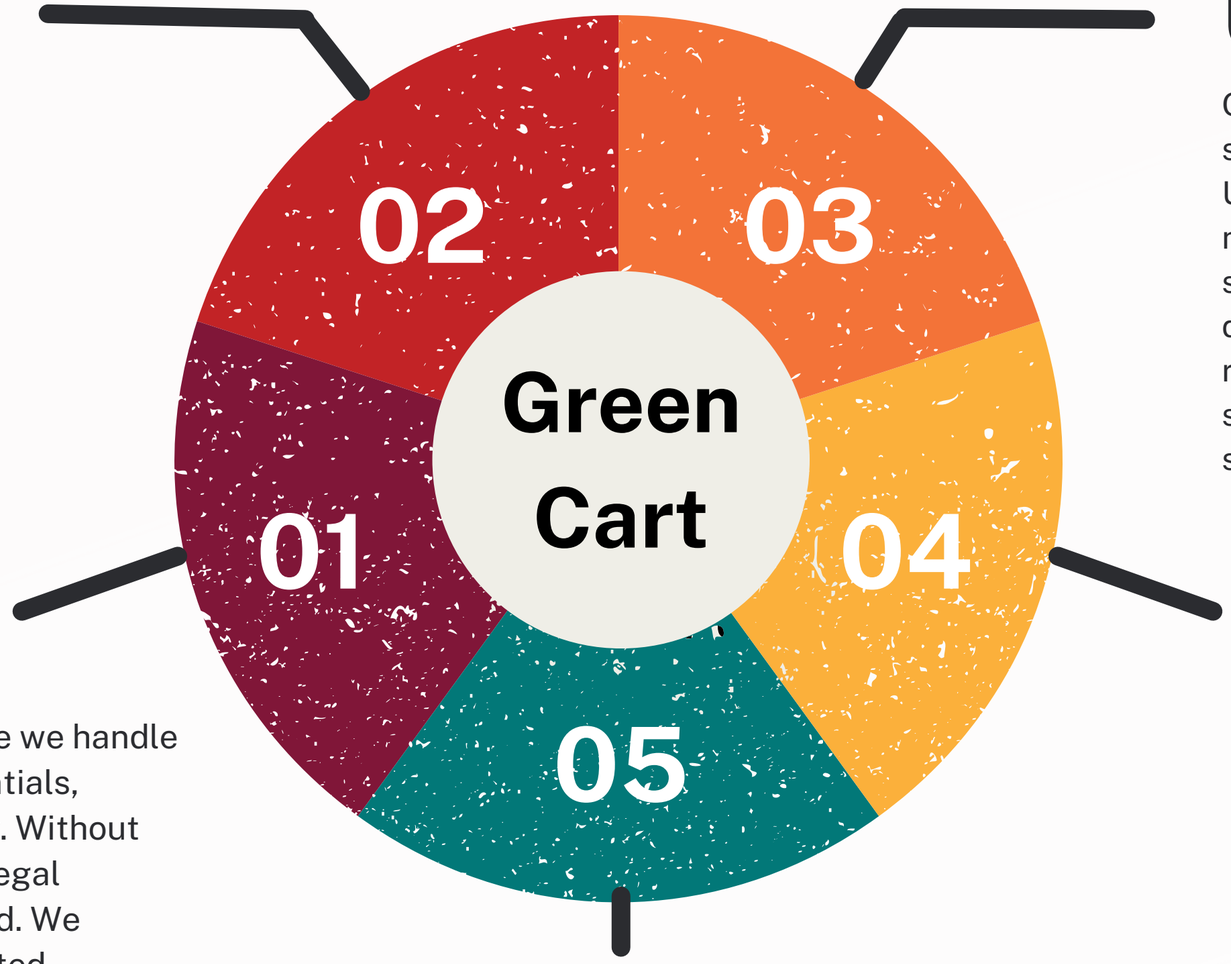
GreenCart Demo 2

Capstone 2025

5 Core Quality Requirements

Performance

Users expect a fast and responsive experience — especially when browsing, searching, or checking out. Delays can lead to drop-offs and poor engagement. To address this, we optimized backend queries, use Redis caching, and designed a non-blocking async backend to keep interaction times low even under load.



Usablility

Our platform targets eco-conscious shoppers with diverse digital skill levels. Usability ensures these users can easily navigate, filter products, and make sustainable choices without confusion. We designed with accessibility and responsiveness in mind, and included help sections to guide users through sustainability-focused features.

Security

Security is our top priority because we handle sensitive user data such as credentials, personal details, and order history. Without strong protection, user trust and legal compliance would be compromised. We implement authentication, encrypted transport, hashed credentials, and strict access control to prevent breaches.

Modularity

GreenCart has multiple evolving features — from sustainability tracking to retailer integration. A modular structure enables safe, parallel development and maintenance. It ensures new features or fixes can be added without breaking existing functionality, supporting agile workflows and testability.

Scalability

5 Core Quality Requirements

1. Security

Quantification:

Protect user data and platform access points from unauthorized access and manipulation.

Architectural Strategy:

GreenCart implements a Layered Architecture that enforces strict separation between the user interface, application logic, and data layers. Security is enforced at the API layer using JWT-based authentication, which ensures that each request is securely validated before access is granted. Sensitive user data is protected using bcrypt hashing, and HTTPS secures data in transit. The system also uses the Security Filter pattern, sanitizing inputs before they reach the business logic, reducing the risk of SQL injection and XSS attacks. Additionally, Role-Based Access Control (RBAC) ensures different user roles have access only to what they need, and suspicious access attempts are logged for monitoring and auditing.

5 Core Quality Requirements

2. Performance

Quantification:

Ensure 95% of API requests complete within 400ms, product pages load in under 1.5s, and checkout finishes within 3s.

Architectural Strategy:

GreenCart uses the Cache-Aside pattern with Redis to store frequently requested data, such as product listings and cart contents, which significantly reduces latency and alleviates database load. The backend is structured using a Service-Oriented Architecture (SOA), where performance-critical services like product and cart management are separated, allowing them to be monitored and optimized independently. FastAPI's asynchronous request handling ensures non-blocking performance, enabling the backend to handle multiple simultaneous user requests efficiently. This architecture ensures that even during peak traffic periods, the platform remains responsive and performant.

5 Core Quality Requirements

3. Usability

Quantification:

Enable users of all experience levels to intuitively navigate and interact with the system across all devices.

Architectural Strategy:

GreenCart’s frontend is designed using a Component-Based Architecture, implemented in React, which ensures that each UI element (e.g., filters, navigation, product cards) is reusable and consistently styled. The layout follows the Model-View-Controller (MVC) pattern conceptually, where the view layer reflects application state changes driven by user interactions and backend responses. This clear separation ensures that usability changes (e.g., help sections, labels, accessibility features) can be updated without disrupting application logic. Tooltips, mobile responsiveness, and sustainability education features are built directly into components, ensuring a smooth experience across desktop, tablet, and mobile platforms.

5 Core Quality Requirements

4. Modularity

Quantification:

Ensure new features can be added with integration into no more than two modules and ⩾80% unit test coverage per service.

Architectural Strategy:

GreenCart adopts a Microservices Architecture where features like authentication, product management, cart handling, and order processing are implemented as isolated services. This design supports independent development, deployment, and scaling of each module. RESTful APIs define strict service boundaries, enforcing loose coupling and high cohesion. The frontend mirrors this modularity with a Component-Based UI, ensuring that visual updates are localized. The use of Service-Oriented Architecture (SOA) principles supports inter-service communication via standard interfaces, promoting safe integration and parallel development across the team.

5 Core Quality Requirements

5. Scalability

Quantification:

Support ⩾100 concurrent users during campaigns, with product query latency ⩽600ms and cart/order latency ⩽500ms.

Architectural Strategy:

GreenCart is designed using a Service-Oriented Architecture (SOA), where core functionalities such as product management, cart handling, and order processing are implemented as separate services that communicate through well-defined REST APIs. This separation allows performance-critical services to be scaled or optimized independently without affecting the rest of the system. FastAPI's asynchronous request handling enables the backend to serve many concurrent users efficiently by avoiding blocking operations. To further reduce load, Redis caching is implemented using the Cache-Aside pattern, ensuring that frequently accessed data is served quickly without querying the database repeatedly. Combined with the system's layered structure, which separates routing, business logic, and persistence, this architecture supports seamless scalability in response to growth in user traffic and data volume.

Technology Choices

Frontend Development

Options Considered:

React with Vite
Angular
Vue.js
Final Choice: React with Vite

Justification:

React offers a component-based architecture that aligns with our usability and modularity goals. Vite provides fast dev server start-up and optimized production builds, improving productivity and performance. This stack integrates seamlessly with our Figma-based UI design and supports responsive, accessible user interfaces.

Technology Choices

Backend Development

Options Considered:

FastAPI (Python)
Express.js (Node.js)
Spring Boot (Java)
Final Choice: FastAPI

Justification:

FastAPI provides asynchronous request handling and automatic documentation via OpenAPI. It's lightweight, fast, and ideal for modular service design — aligning with our performance and scalability requirements. Python also supports data processing needs for future sustainability features.

Technology Choices

Database

Options Considered:
PostgreSQL
MongoDB
MySQL
Final Choice: PostgreSQL

Justification:
PostgreSQL offers robust relational data handling, support for JSON fields, and strong ACID compliance. It aligns with our need for structured product and user data, while still offering flexibility for sustainability metadata and future reporting features.

Live Demo

Unit Testing

Frontend

```yaml
name: Run Frontend Tests Before Merge

on:
  pull_request:
    branches:
      - main

jobs:
  frontend_tests:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        # Checks out your repository code into the runner's environment.
        # This is essential for the workflow to access your project files.
        uses: actions/checkout@v3

      - name: Set up Node.js
        # Sets up the Node.js environment required for npm commands.
        # '20.x' is a common stable version, you can adjust this if needed.
        uses: actions/setup-node@v3
        with:
          node-version: '20.x'
          cache: 'npm' # Caches npm dependencies to speed up subsequent runs

      - name: Install and Test Frontend
        run: |
          cd frontend
          npm ci
          npm run test:run
```

Backend

```python
def test_cart_add_item_logic():
    product = Product(id=10, name="Test Product", price=100)
    cart_item = CartItem(product_id=10, quantity=2)
    cart = Cart(user_id=1, items=[cart_item])

    assert cart.items[0].product_id == 10
    assert cart.items[0].quantity == 2

def test_cart_total_price_computation():
    cart = Cart(user_id=1)
    cart.items = [
        CartItem(product_id=1, quantity=2),
        CartItem(product_id=2, quantity=1)
    ]

    prices = {1: 50, 2: 80}
    total = sum(prices[item.product_id] * item.quantity for item in cart.items)
    assert total == 180
```

Integration Test

```python
def test_add_item_to_cart():
    global actual_user_id
    ensure_cart_exists()

    response = client.post(f"/cart/add?user_id={actual_user_id}", json={
        "product_id": 1,
        "quantity": 2
    })

    assert response.status_code in [200, 201]
    data = response.json()
    assert data["user_id"] == actual_user_id
    assert any(item["product_id"] == 1 and item["quantity"] >= 2 for item in data["items"])


def test_view_cart():
    global actual_user_id
    ensure_cart_exists()

    response = client.get(f"/cart/{actual_user_id}")
    assert response.status_code == 200
    data = response.json()

    assert data["user_id"] == actual_user_id
    assert isinstance(data["items"], list)
```

Q&A