Demo 1: Architectural Requirements

GreenCart

Client: BBD Software

Team member	Student Number
Nikhil Govender	u22504193
Shayden Naidoo	u23599236
Corné de Lange	u23788862
Tshegofatso Mahlase	u22580833
Samvit Prakash	u23525119

Non-Functional Requirements

Quality Requirements

QR1: Performance

QR1.1: The system needs to return product search results within 2 seconds under normal load (more or less 100 concurrent users)

QR1.2: The system needs to ensure that page loading times and product listings are less than 3 seconds on a 4G network

QR1.3: Checkout procedures (payment and confirmation needs to be done within 5 seconds in 95% of cases)

QR2: Reliability

QR2.1: The system needs to be available 99% of the time on a monthly basis.

QR2.2: The system needs to recover from server failure within 60 sec using an auto restart or failover mechanisms

QR2.3: If Data loss during a crash occurs then it needs to be limited to less than 1 transaction lost as regular backups and atomic operations will be implemented.

QR3: Scalability

QR3.1: The system needs to support up to 10 000 active users per day without failure or loss in performance.

QR3.2: The architecture needs to support horizontal scaling of the backend services via AWS or Superbase.

QR3.3: The system needs to be stress tested and able to handle at least 500 concurrent sessions for e.g. Flash sale events

QR4: Security

QR4.1: All private data (passwords and payment information) needs to be encrypted in transit and at rest.

QR4.2: The system will implement Firebase Auth and JWT tokens for secure, stateless authentication.

QR4.3: The system will comply with POPIA and GDPR regulations, allowing the user the ability to delete data upon request

QR4.4: CSRF tokens will be implemented on all forms and state changing requests to prevent forgery.

QR4.5: The system will lock out user accounts for 30 minutes after 5 failed login attempts to prevent brute force attacks

QR5: Maintainability

QR5.1: The codebase will follow a modular architecture using Python (backend) and React (frontend).

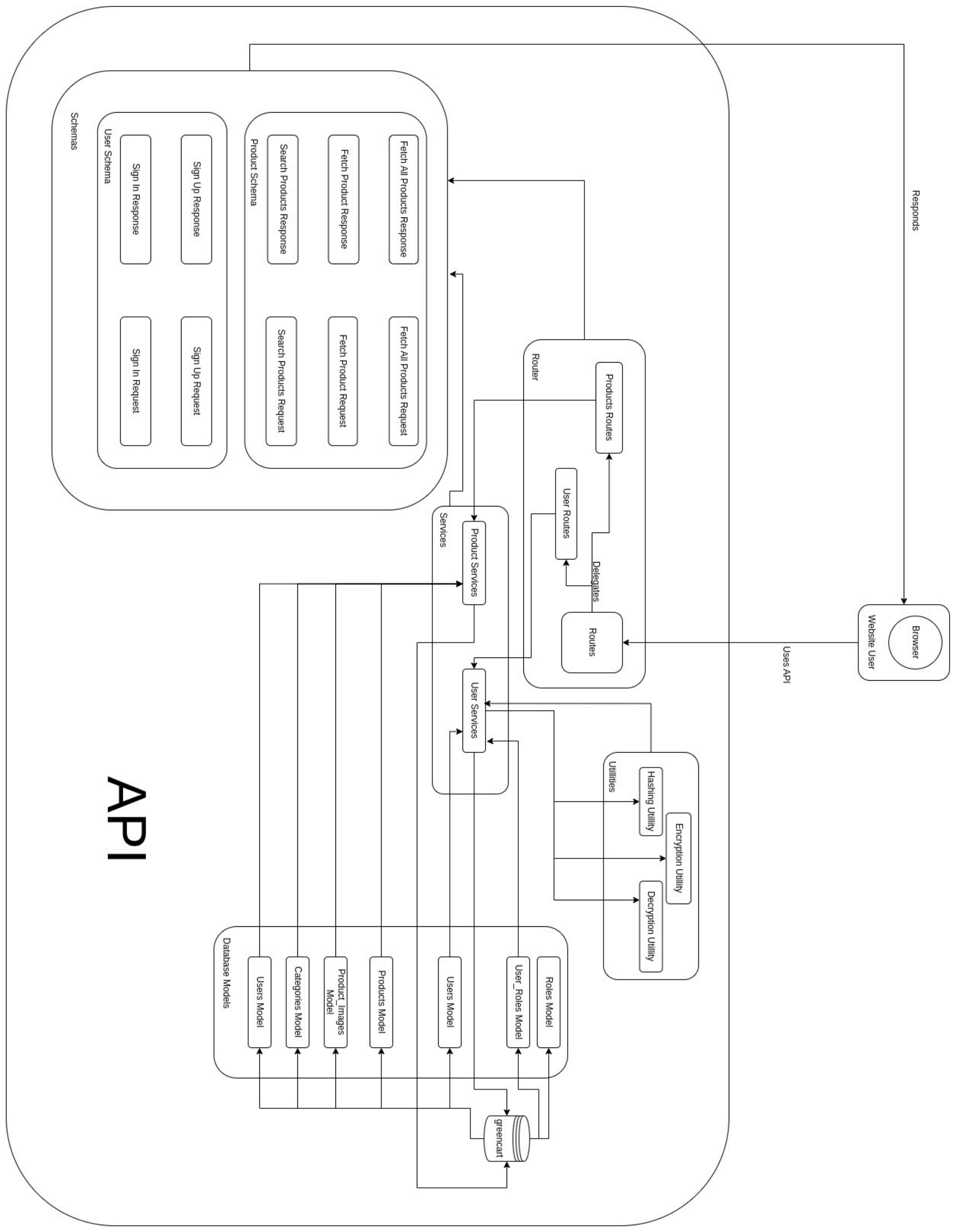
QR5.2: At least 80% unit test coverage needs to be achieved across all services using PyTest and Jest.

QR5.3: All APIs will be documented using OpenAPI and updated automatically in CI/CD pipelines.

QR5.4: The system will be deployed via GitHub actions enabling automated build, test and staging deployments.

QR5.5: All configuration and business logic needs to be separated into environment files and service layers to support easy updates and rollbacks.

Architectural Diagram



Architectural Patterns

Module-view controller (MVC):

- it separates the concerns of user interface, application logic, and data handling.
- Model
    - This represents the business logic and data of the application
    - We store Sustainability data models (carbon footprint, certifications)
    - We store Product listings and metadata (certifications, ratings, analytics)
- View
    - This is interface elements that the user interacts with
    - We are building our front end using React.js and vite and designed it using figma
    - We display:
        - Product pages
        - Dashboards (eco-impact, sales)
        - Cart and checkout flow
        - Sustainability visualizations (graphs, badges, etc.)
- Control:
    - Handles user input, interacts with models, and updates the view.
    - Implemented through **RESTful APIs** using Python and FastAPI.
    - Handles:
        - Authentication logic (login, register, JWT token handling)
        - Search/filtering logic
        - Checkout processing
        - Carbon footprint calculations and recommendation logic

Service orientated Architecture (SOA) :

- The system is divided into independent services—authentication, product management, sustainability insights, recommendations engine, order processing, and carbon offsetting

Microservices

- This is present in the separation of authentication, product management, recommendation engine and the sustainability insights into subsystems.
- It supports horizontal scaling and the independent deployment of services

Event-Driven Architecture

- This is present for the sustainability insights subsystems like carbon footprint tracking reports which require purchases to occur this will also trigger eco-recommendation updates based on the behavior of the user.
- This enables asynchronous processing for issuing donation badges and allowing the system to generate sustainability reports based on the carbon footprint tacking of the user through their purchasing histories.

Client-Server Architecture

- Our system follows the classic client server architecture as the frontend will be communicating with backend services via HTTP REST API endpoints.

Design Patterns

Factory Design Pattern

- The dynamic instantiation of user types is an example of the factory method
- E.g. a guest, retailer and eco shopper are instantiated when the specific user first uses the platform with different access to different features

Strategy Design Pattern

- The Strategy design pattern occurs in multiple situations throughout our system. These can include:
    - When a user chooses a specific delivery method (standard, express or pickup).
    - In recommendation engine: selecting different recommendation algorithms (carbon reduction, price efficiency, certification-first).

Observer Design Pattern

- Real time updates and notifications are provided to the user when monitoring their cart and their order status alerting the user when their ordered item is on its way and when it is delivered.
- The user is also alerted when there are major sustainability changes in their sustainability report.

Decorator Design Pattern

- This is for enhancing the product with extra metadata like certifications and carbon ratings. The retailer is also required to provide important details like descriptions and materials that the product is made of about their products and can modify these details without modifying the base structure of the product.

Adapter Design Pattern

- The adapter design pattern is used when we integrate third party services e.g. PayPal, Stripe and Google pay for payment methods, external sustainable APIs and email verification services.

Constraints

- **Hosting**
  - Run everything **locally** for Demo 1.
- **Docker**
  - Backend can run with uvicorn.
  - Front-end can run with npm run dev.
- **External services**
  - No real payment or sustainability APIs yet – return mocked data (fetched from db).
- **Data limits**
  - Seed database ≤ 200 products (≈ ≤ 5 MB).
- **Security basics**
  - JWT in **HTTP-only cookies**; no tokens in localStorage.
  - Collect only email + name + hashed password (POPIA friendly).
- **Licensing / cost**
  - Use only free, open-source tools and free-tier resources.
- **CI gate**
  - Every PR must pass linting and unit tests.

Technology Requirements

Frontend Development: React, Vite & Figma

Our frontend is developed using **React** paired with **Vite** for fast build times and modern module bundling. This setup provides a responsive and efficient user experience. **Figma** is our primary tool for UI/UX design, enabling seamless collaboration between designers and developers and ensuring pixel-perfect implementation of user flows.

Pros:

- Fast Development and Build Times
- Component Reusability with React
- Streamlined Design-to-Code Workflow

Cons:

- Learning Curve for Vite & Modern React
- Performance Optimization Needed for Large Apps
- Design Handoff Still Requires Manual Adjustments

Backend Development: Python (FastAPI)

We use **Python** with the **FastAPI** framework to build high-performance, asynchronous backend services. FastAPI's built-in support for OpenAPI schemas and automatic validation makes it ideal for scalable, maintainable API development. It enables rapid iteration and consistent integration across the platform.

Pros:

- High Performance with Async Support
- Automatic Documentation with OpenAPI

- Type Hints and Validation Built-In

Cons:

- Steeper Learning Curve for Async Concepts
- Smaller Community Compared to Flask/Django
- Limited Out-of-the-Box Features

API & Security: FastAPI, JWT & OpenAPI

Our RESTful API layer is built with **FastAPI**, leveraging **JWT (JSON Web Tokens)** for stateless, secure authentication between services. **OpenAPI** is used for generating standardized API documentation and client interfaces. This approach ensures clear integration contracts, improved developer experience, and better testing coverage.

pros:

- Stateless Authentication with JWT
- Standardized and Auto-Generated Docs with OpenAPI
- Improved Testability and Integration Clarity

cons:

- JWT Token Management can be complex
- Security Risks if improper JWT implementation (e.g token forgery or exposure)
- OpenAPI and JWT may add complexity and setup time

Database Architecture: PostgreSQL

Our backend uses **PostgreSQL** as the primary relational database. It offers robust querying capabilities, strong consistency, and support for complex relationships, making it well-suited for managing structured data such as products, orders, and user profiles. Redis remains a future consideration for caching high-traffic endpoints if needed.

pros:

- Advanced Querying and Data Integrity
- Scalable and Reliable for Transactional Workloads
- Extensibility and Ecosystem Support

cons:

- Performance Bottlenecks Under High Read Load (without caching)
- Manual Scaling and Optimization
- Requires More Rigid Schema Planning

Testing Strategy: Pytest & Jest

Testing is core to our development workflow. **Pytest** is used for backend unit and integration testing, ensuring stability and correctness of services. On the frontend, **Jest** is employed to validate React component behavior and UI logic, maintaining quality throughout the user experience.

pros:

- both have comprehensive test coverage
- Fast Feedback for Developers
- Rich Ecosystem and Plugin Support

cons:

- Steeper Learning Curve for Advanced Features
- Maintenance Overhead
- Flaky Tests and CI Slowdowns

Deployment & Hosting: Microsoft Azure, Firebase, Supabase, and AWS

We are exploring a multi-option deployment strategy to balance flexibility, scalability, and ease of use.
Our current preferences include **Microsoft Azure** for enterprise-grade control, **Firebase** or **Supabase** for rapid deployment and frontend-friendly services, and **AWS** for its production-grade ecosystem and service breadth.

Microsoft Azure

Why Azure?
We are leaning toward Azure for enterprise-level scalability and configuration control—particularly if we need deep integration, compliance, or hybrid infrastructure.

Pros:

- **Enterprise-Level Scalability and Control**
  Offers granular infrastructure control, load balancing, and virtual machines for complex deployments.

- **Robust Compliance and Security**
  Supports GDPR, HIPAA, and other compliance standards—ideal for secure platforms.

- **Rich Ecosystem Integration**
  Integrates with Azure DevOps, Active Directory, GitHub Actions, and more.

Cons:

- **Steep Learning Curve**
  Azure's vast service offerings can be overwhelming to small/student teams.

- **Student Account Friction**
  Azure credits can be hard to activate or limited under student accounts.

- **Overhead for Simple Apps**
  Customization may be excessive for MVPs or lightweight projects.

Firebase

Why Firebase?
Firebase is ideal for MVPs, frontend hosting, and integrating with Firebase Auth and Firestore out of the box.

Pros:

- **Instant HTTPS Hosting**
  Built-in SSL, CDN, and deploy via terminal.
- **Backend-as-a-Service**
  Works seamlessly with Firebase Auth, Firestore, and Functions.
- **Generous Free Tier**
  Supports early-stage development without cost.

Cons:

- **Limited Backend Control**
  Custom server-side logic is constrained to Cloud Functions.
- **Vendor Lock-In**
  Firebase's architecture can make switching providers complex.
- **Scaling Limitations**
  May underperform for heavy compute or enterprise-scale operations.

Supabase

Why Supabase?
Supabase offers a modern, SQL-first, open-source backend with built-in auth and realtime features.

Pros:

- **PostgreSQL-Powered**
  Relational data with full SQL support and familiarity.

- **Built-In Auth & Realtime**
  Subscriptions, JWT, and row-level security.

- **Open Source**
  Can be self-hosted to avoid long-term vendor lock-in.

Cons:

- **Young Ecosystem**
  May lack mature documentation or stability in some modules.

- **Limited Job Scheduling/Compute**
  Not ideal for complex backend logic or batch processing.

- **SLA Uncertainty**
  Still growing as a managed production platform.

Amazon Web Services (AWS)

Why AWS?
AWS is the most mature and feature-rich cloud provider, suitable for scaling GreenCart into a production-ready, globally available service.

Pros:

- **Unmatched Service Breadth**
  Offers everything from serverless (Lambda) to container orchestration (ECS/EKS), S3 storage, global CDNs, and AI/ML services.

- **Highly Scalable & Reliable**
  Proven to handle millions of users with auto-scaling and regional replication.

- **Pay-as-You-Go with Free Tier**
  Ideal for startups that may scale over time; the free tier includes EC2, Lambda, and S3 usage.

- **Seamless CI/CD & DevOps Integration**
  Integrates with CodePipeline, GitHub Actions, and other tools for streamlined deployment.

Cons:

- **Configuration Complexity**
  The sheer number of options and services can be daunting without prior experience.

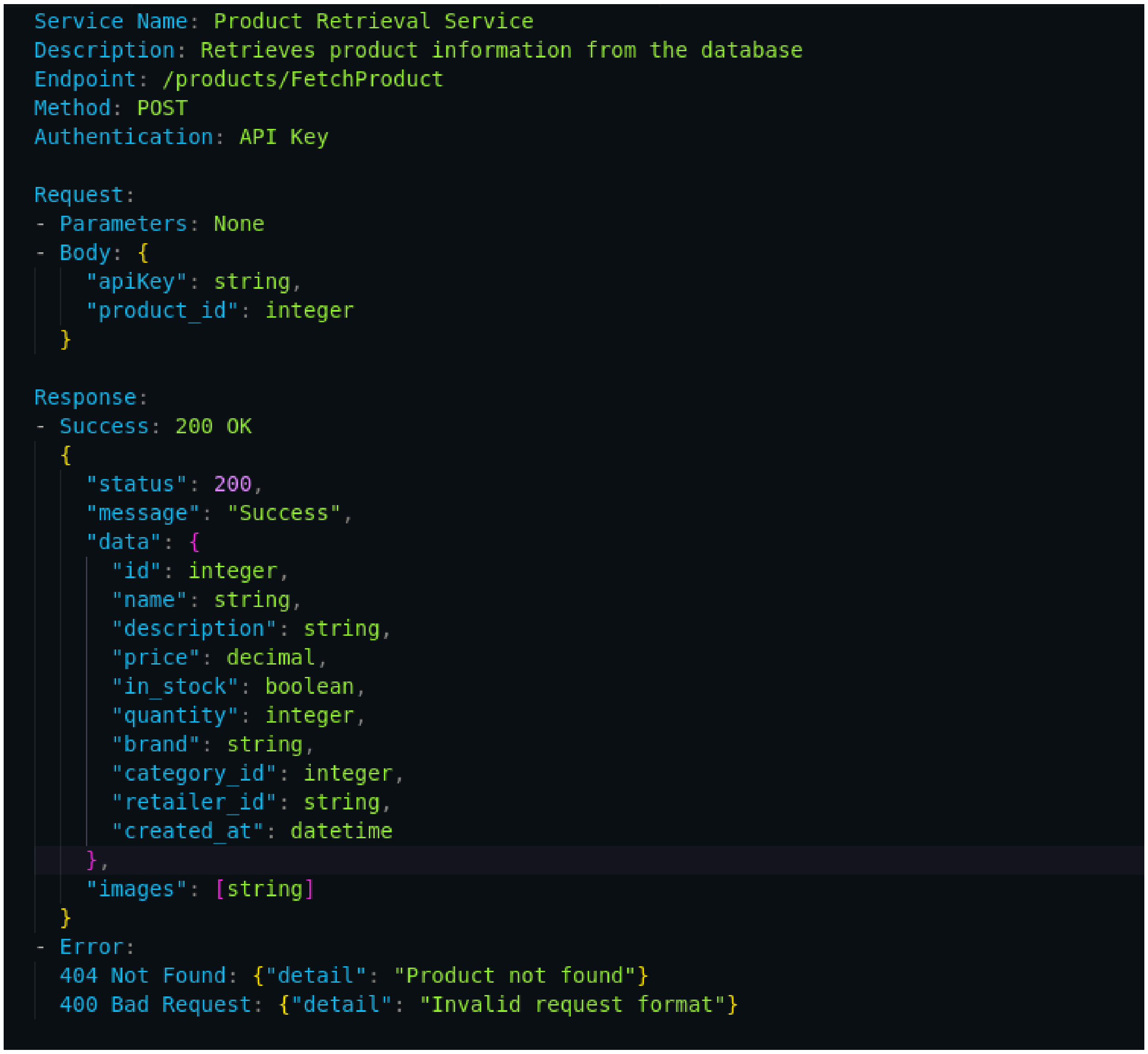
- **Costs Can Escalate**
  While affordable at first, costs rise quickly if services aren't optimized (e.g., leaving EC2 on).

- **Learning Curve**
  Deep understanding required for IAM, VPCs, and services like CloudFormation or CloudWatch.

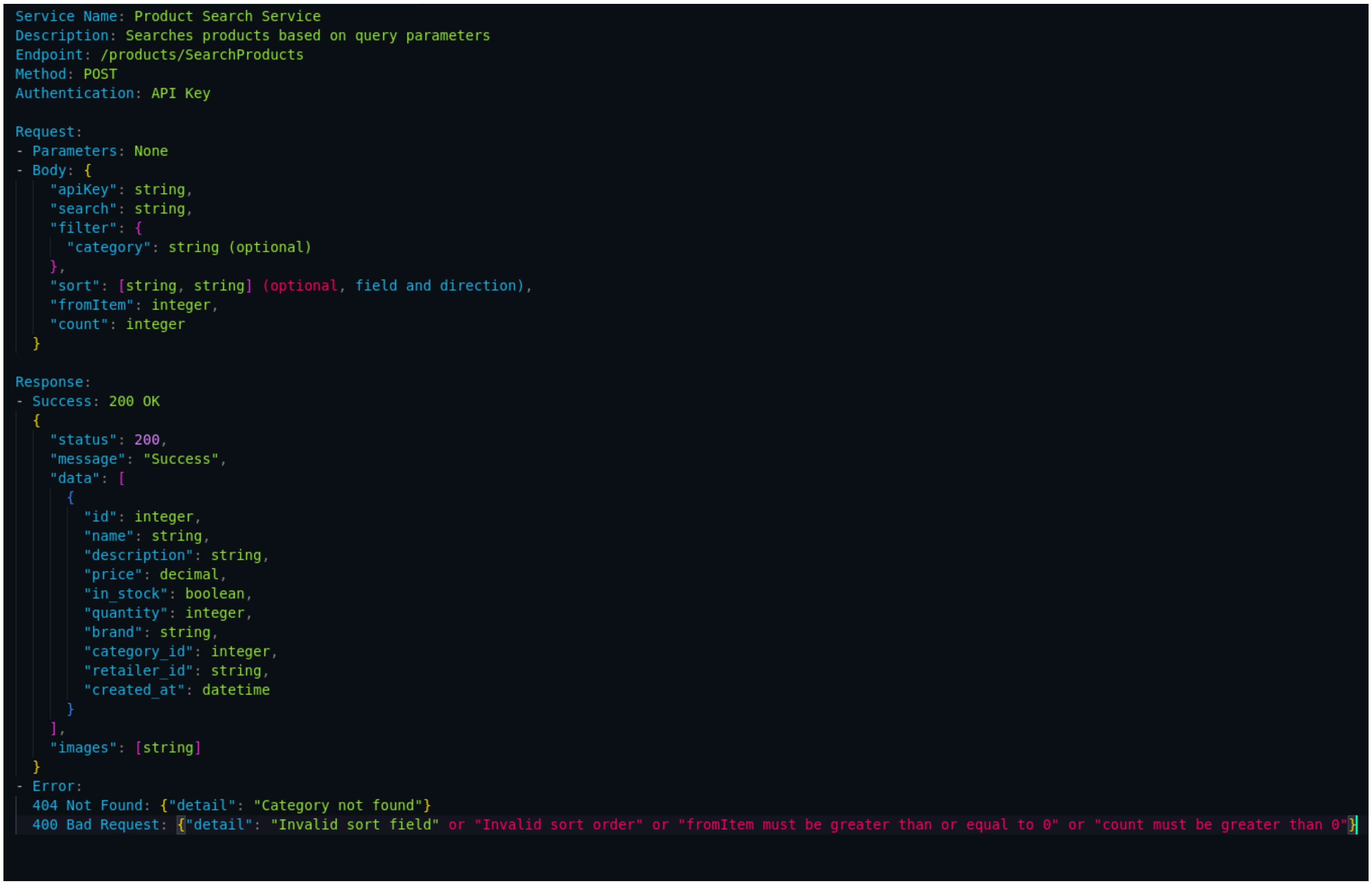
Version Control & CI/CD: GitHub & GitHub Actions

Code is managed via **GitHub**, with branch protection and pull request reviews enforcing team quality standards. **GitHub Actions** automates our CI/CD pipeline, running tests, linters, and deploy workflows on push, enabling rapid and reliable feature delivery.

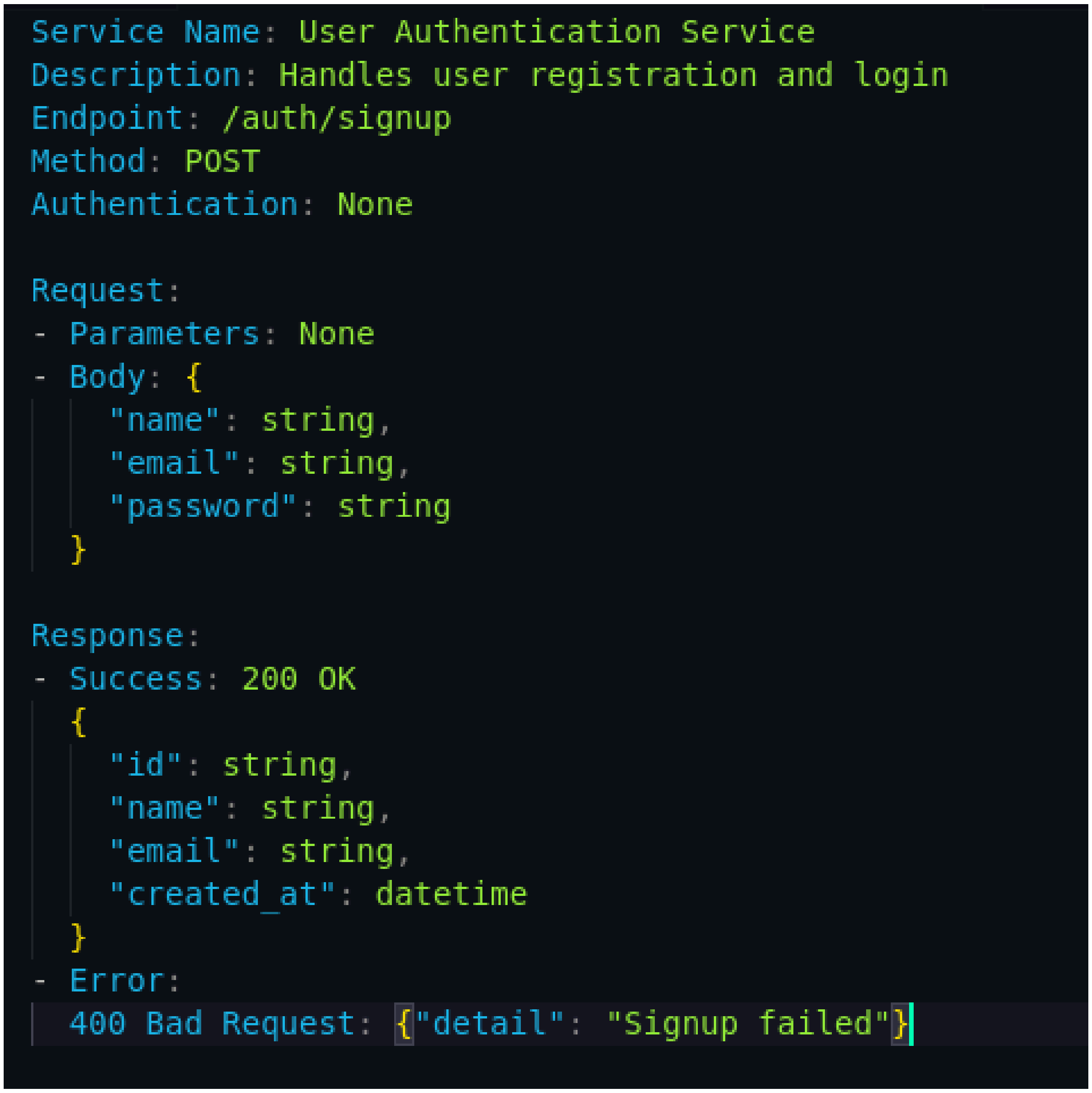
Service Contracts



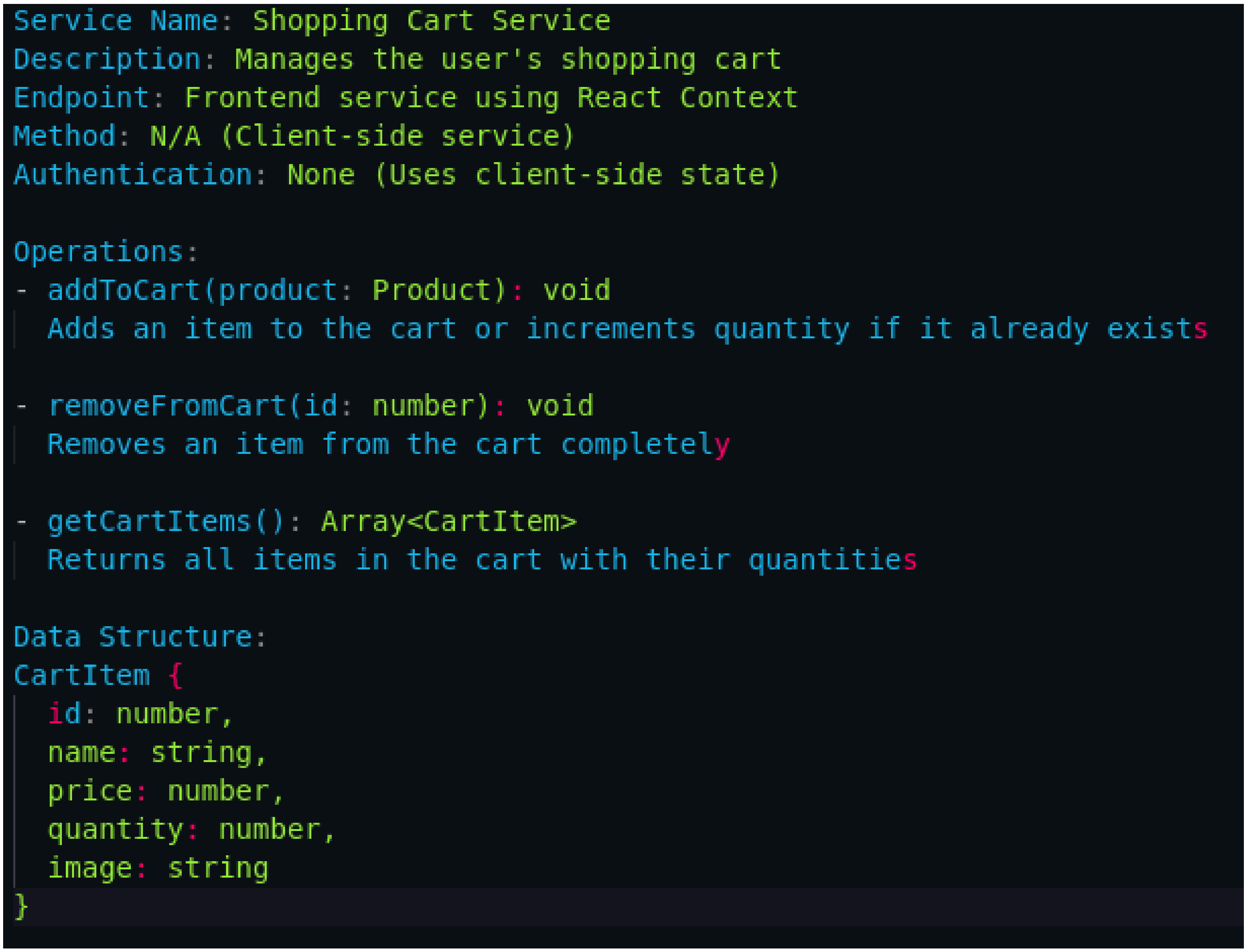
```
Service Name: Product Retrieval Service
Description: Retrieves product information from the database
Endpoint: /products/FetchProduct
Method: POST
Authentication: API Key

Request:
- Parameters: None
- Body: {
    "apiKey": string,
    "product_id": integer
  }

Response:
- Success: 200 OK
  {
    "status": 200,
    "message": "Success",
    "data": {
      "id": integer,
      "name": string,
      "description": string,
      "price": decimal,
      "in_stock": boolean,
      "quantity": integer,
      "brand": string,
      "category_id": integer,
      "retailer_id": string,
      "created_at": datetime
    },
    "images": [string]
  }
- Error:
  404 Not Found: {"detail": "Product not found"}
  400 Bad Request: {"detail": "Invalid request format"}
```



```
Service Name: Product Search Service
Description: Searches products based on query parameters
Endpoint: /products/SearchProducts
Method: POST
Authentication: API Key

Request:
- Parameters: None
- Body: {
    "apiKey": string,
    "search": string,
    "filter": {
      "category": string (optional)
    },
    "sort": [string, string] (optional, field and direction),
    "fromItem": integer,
    "count": integer
  }

Response:
- Success: 200 OK
  {
    "status": 200,
    "message": "Success",
    "data": [
      {
        "id": integer,
        "name": string,
        "description": string,
        "price": decimal,
        "in_stock": boolean,
        "quantity": integer,
        "brand": string,
        "category_id": integer,
        "retailer_id": string,
        "created_at": datetime
      }
    ],
    "images": [string]
  }
- Error:
  404 Not Found: {"detail": "Category not found"}
  400 Bad Request: {"detail": "Invalid sort field" or "Invalid sort order" or "fromItem must be greater than or equal to 0" or "count must be greater than 0"}
```



```
Service Name: User Authentication Service
Description: Handles user registration and login
Endpoint: /auth/signup
Method: POST
Authentication: None

Request:
- Parameters: None
- Body: {
    "name": string,
    "email": string,
    "password": string
  }

Response:
- Success: 200 OK
  {
    "id": string,
    "name": string,
    "email": string,
    "created_at": datetime
  }
- Error:
  400 Bad Request: {"detail": "Signup failed"}
```



```
Service Name: Shopping Cart Service
Description: Manages the user's shopping cart
Endpoint: Frontend service using React Context
Method: N/A (Client-side service)
Authentication: None (Uses client-side state)

Operations:
- addToCart(product: Product): void
  Adds an item to the cart or increments quantity if it already exists

- removeFromCart(id: number): void
  Removes an item from the cart completely

- getCartItems(): Array<CartItem>
  Returns all items in the cart with their quantities

Data Structure:
CartItem {
  id: number,
  name: string,
  price: number,
  quantity: number,
  image: string
}
```