LASTBITE

Final Report for the Distributed Systems Course 2024/2025

Pier Costante Babini
piercostante.babini@studio.unibo.it

Marco Morelli
marco.morelli15@studio.unibo.it

June 2026

Food waste represents one of the most significant sustainability challenges worldwide, with large quantities of edible products discarded daily across the food supply chain. Restaurants and food service activities contribute substantially to this issue, especially through unsold products remaining at the end of the day. This report details the design and implementation of LastBite, a distributed web application designed to mitigate food waste by enabling restaurants to sell surplus products at the end of the day. The project is built on top of the MEVN (MongoDB, Express.js, Vue.js, Node.js) stack and orchestrated using a Docker-based approach to ensure scalability and isolation. The system comprises multiple containerized services, including two backend instances, a Vue frontend, a MongoDB replica set, a Redis database for distributed locking and an Nginx reverse proxy for load balancing and fault tolerance. Furthermore, LastBite implements several security features required by the application domain, such as secure authentication, password hashing, Role-Based Access Control (RBAC) and distributed locks for handling product purchases and modifications. The system offers dedicated interfaces for restaurants, which manage products and orders, and for customers, who can search for nearby restaurants using geospatial filtering. Finally, the application also includes real-time notifications and an AI-powered "Food Advisor" assistant.

Disclaimer

During the preparation of this work, the authors used GitHub Copilot to assist with code completion and support the English translation and the spelling review of the present report. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

Contents

1 Concept

1.1 Type of product

The project implements a distributed full-stack web system, composed of multiple services for managing, publishing, and purchasing food products remaining at the end of the day from commercial activities. The system consists primarily of the following services:

- Backend (Node.js/Express) exposed as a REST/HTTP web service and WebSocket for notifications.
- Frontend (Vue 3) that provides the user interface visualization and interaction with the system.
- Database (MongoDB) for data persistence configured as a replica set.
- Reverse proxy (Nginx) for request routing and load balancing.
- External API integrations (Groq and OpenStreetMap) for using the AI chat assistant (Food Advisor) and resolving addresses into geographic coordinates.
- Redis used for the distributed locking system and event propagation among backend replicas.

1.2 Use case description

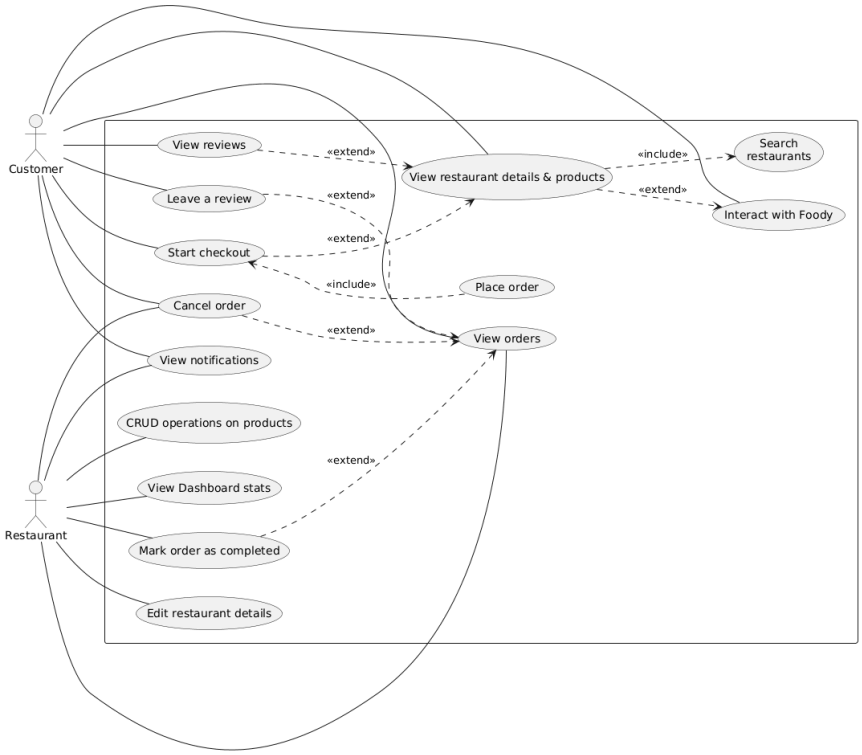

Figure 1: LastBite Actor diagram.

What the software does

LastBite allows users to participate in the recovery of unsold food, thereby reducing food waste. Users can indeed purchase products from restaurants and businesses in their area at a more affordable price. The application allows users to view a list of available businesses, select the venue where they want to make the purchase, and proceed with one or more orders. A dedicated section for restaurant owners is also included, designed to support the management of their own venue by allowing the insertion and administration of available products. This section also provides several metrics on sales performance to the restaurant owner, allowing them to view the sales history over time.

In the following and within the application, the account used to upload products will be referred to as "Restaurant / restaurant owner"; however, the sale of products by any business that wishes to do so is fully supported, such as bakeries, pastry shops, supermarkets, bars, etc.

Where are the users

Users can be in different scenarios:

- On the host machine: via `http://localhost:5173`, on the PC on which Docker is running.
- On the same LAN: via `http://<hostIP>:5173`, making sure that the firewall allows it.
- On an external network: in case of platform hosting, users can access it from their preferred devices.

1.3 When and how they interact

- Restaurant owners: they interact mainly when they want to upload unsold products. They then manage the preparation and pickup of orders on-demand (in real time) as soon as they are received.
- Customers: they browse the platform sporadically or during the opening hours of the businesses. They interact with the map, the checkout windows, and the AI assistant (Food Advisor) through a web browser.
- Maintenance activities take place periodically by accessing the Docker containers.

1.4 Users and roles

Restaurant owner: owner of a business, can:

- Monitor the status of sales and orders at a glance.
- Modify the information of their own business.
- Add and modify the details of the products for sale.
- Manage customer orders based on the physical availability of the products.

Customer: person who wants to purchase food, can:

- Register and consult the list of nearby restaurants.
- Search for a specific restaurant.
- Modify their own position to view other restaurants.
- View the products for sale from a restaurant.
- Complete an order.
- Leave a review after picking up an order.
- Interact with a chatbot to obtain information.

1.5 Stored data

Data persistence is provided by MongoDB, while the management of volatile and temporary data is entrusted to Redis. The stored information includes:

MongoDB:

- Users: credentials, email, location and roles (customer/restaurant).
- Restaurants: general information, location and opening hours.
- Products: item details, original and discounted price.
- Orders: references to the user and restaurant, order status, payment method and time.
- Notifications: recipient user, order reference, content and time.
- Reviews: recipient restaurant, authoring user, rating, order reference and comments.

Redis:

- Checkout lock status: temporary data and identifiers used to lock a product while a user is completing a purchase, preventing simultaneous purchases by other customers.

1.6 Why distribution is needed

The choice of a containerized and logically distributed architecture for LastBite is guided by the following practical requirements:

- **Isolation and security**: each service runs in its own container and the system applies strict role-based access controls, validating session JWT tokens and storing them securely (through HTTP-only cookies in the frontend), exposing the routes through Nginx.
- **Concurrency and scalability**: the purchase of limited products requires proper management of concurrent events. Furthermore, horizontal scaling must be managed, meaning that it must be possible to start multiple backend instances under the reverse proxy to absorb request peaks, without overloading frontend rendering or the database.
- **No single logical point of failure**: service separation prevents a rendering error in the client or a frontend update from bringing down the entire system. Nginx acts as a load balancer and fault-tolerance node, routing traffic to healthy containers.
- **Maintainability**: the use of Docker Compose makes the environment reproducible on any host. In fact, immutable images and file composition simplify testing, releases, and collaborative development.

2 Requirements Elicitation and Analysis

2.1 Functional requirements

The application provides a distinction between two types of users: customers and restaurant owners. This separation made it necessary to identify specific functional requirements for each category, while maintaining some common requirements shared by both. The common functional requirements, customer requirements, and restaurant owner requirements are listed below:

2.1.1 Common requirements

- Registration
- Secure login management
- Modification of own account data in the profile area
- Reception of real-time notifications based on events significant for the type of user

2.1.2 Customer requirements

- Possibility to search for a restaurant by name
  - Filter restaurants based on maximum distance
  - Sort restaurants by increasing distance or decreasing rating
- View restaurants on a geographic map
- View the details of a restaurant
  - Consultation of the restaurant's primary information, such as name, description, address, and pickup time
  - Consultation of a restaurant's reviews
  - Consultation of the products for sale in the restaurant
  - Possibility to filter the products for sale based on the price range and sort them by increasing or decreasing price
- Possibility to purchase a product by credit card or cash
- Management of concurrent operations on products: if two or more users try to purchase a product, only one of them must be able to proceed to complete the transaction
- Possibility to cancel an order within 5 minutes
- Possibility to consult ongoing, canceled, and completed orders
- Possibility to review a completed order

- Possibility to interact with a chatbot to obtain information about products and restaurants
- Possibility to modify one's own position for different search results

2.1.3 Restaurant Owner requirements

- Visualization of the statistics of one's own restaurant
  - Overview of key performance indicators (KPIs) on restaurant performance
  - Consultation of a chart showing sales trends over time, also filtering by the desired period
  - Visualization of a summary of the latest orders received
  - Consultation of the reviews of own restaurant, with the possibility to sort them by date or by increasing and decreasing rating
- Possibility to modify the information of own restaurant
- Possibility to add, modify, or delete products for sale
- Management of concurrent operations on products: management of the case in which a product is modified by the restaurant owner at the same instant as a user's purchase
- Possibility to manage received orders
  - Possibility to reject an order, optionally providing a reason to show to the customer
  - Possibility to complete an order
  - Possibility to view whether a review has been left for a specific order

2.2 Non-functional requirements

2.2.1 Usability

- User-friendly interface with a low learning curve
- Responsive design, also optimized for mobile viewing
- Visual feedback based on the performed action

Acceptance criteria: the interface must be usable within a couple of hours by users new to the service, the design must also be optimized for phone viewing, and the user must receive feedback (such as alerts, notifications, or error messages) after the actions performed.

2.2.2 Security

Passwords must be securely encrypted, and authentication tokens must be stored securely on the client. *Acceptance criteria*: algorithms equal or equivalent to Argon2 must be used for password storage, and methods equal or similar to HTTP-only cookies must be used for access token storage.

2.2.3 Responsiveness

The order status and AI assistant chats must be updated in near real time. *Acceptance criteria*: these data must be shown to the user within a maximum of 30 seconds from the moment they request them or navigate to the page that contains them.

2.3 Implementation Requirements

- **Technology stack**: the application must be implemented using the MEVN stack:
  - MongoDB: for data persistence, configured as a replica set for redundancy and high availability.
  - Express.js: backend framework to implement RESTful APIs and manage asynchronous business and concurrency logic.
  - Vue.js: frontend framework to create interactive interfaces.
  - Node.js: used to implement backend services, enabling asynchronous and scalable processing.
- **Communication model**:
  - HTTP/REST: used for calls between frontend and backend.
  - WebSocket: used for real-time updates to connected clients.
  - Redis: used to manage checkout locks and as an adapter for the WebSocket system in case of horizontal backend scaling.
- **Integrations and AI**:
  - External services: they must be integrated for the management of the virtual assistant "Food Advisor" and the resolution of the geographic coordinates of addresses. The integration must reside exclusively on the backend side, through the Groq API and the OSM API.
- **Cross-platform compatibility**: the entire project is containerized through Docker and Docker Compose, to allow a reproducible environment on major operating systems (Windows, MacOS, Linux).

2.4 Relevant Distributed System Features

Below is the analysis of the distributed system features in relation to the project:

- Transparency: the system hides distribution details from users. The consumer sees "LastBite" as a single portal and is completely unaware of how many backend containers are processing their request, or which node of the MongoDB Replica Set is responding to the query.
- Fault tolerance: failures must not compromise the operability and integrity of the system. If the main Redis or MongoDB container goes down, the Replica Set configurations for Mongo and Sentinel for Redis take over (failover). A maximum recovery time of 1 minute is acceptable to avoid service interruption.
- Scalability: the app will be prepared to accommodate a growing number of users and data. This can be done by adding backend containers to handle WebSocket connections and HTTP requests without degrading response times.
- Security: the app will handle sensitive data such as credit card numbers and user data. Within it, two different roles are available, customer and restaurant, with different permissions (RBAC), and the main operations of each role require authentication.
- Resource sharing: several users might try to purchase the same product simultaneously; for this reason, the correct management of these resources through a distributed locking system is essential.
- Openness: the system will interact with some external APIs: with the Groq API for managing an AI chatbot and with OpenStreetMap for resolving geographic coordinates. To do this, the request formats defined in the API specifications will be used.
- Evolvability: the system may require updates, such as the introduction of new features in the future. In this regard, the separation into independent containers and the solid modular structure in the code guarantee excellent long-term maintainability.
- Performance: no strict requirements are defined in terms of bandwidth consumption. However, delicate operations such as purchasing a product require fast communication between frontend and backend.
- Economy: the use of Docker to optimize resource usage on the host and the choice of open-source technologies and efficient LLM models (through Groq, known for low-cost/high-speed inference) significantly reduce development and deployment costs.

3 Design

3.1 Architecture

The system adopts a *distributed client-server architectural style*, in which the frontend acts as the client and multiple redundant backend instances expose the same application

APIs behind an Nginx reverse proxy. The client does not communicate directly with the database, Redis, or internal services: all requests pass through the backend, which coordinates authentication, authorization, data access, distributed locks, and real-time notifications.

This choice is motivated by:

- Clear separation of responsibilities: the backend manages business logic, data access, and coordination between services; the frontend deals exclusively with presentation, user interaction, and API calls.
- Redundancy and availability: the presence of multiple equivalent backends allows the reverse proxy to route requests to a working instance even if an application container stops or becomes temporarily unreachable.
- Horizontal scalability: new backend instances can be added behind Nginx without modifying the frontend. This makes it possible to absorb peaks of HTTP requests and WebSocket connections while keeping the interface exposed to the client stable.
- Distribution transparency: the user sees a single application endpoint, while the internal distribution among backend, replicated database, Redis, and proxy remains hidden. This simplifies the user experience and reduces coupling between client and infrastructure.

Communication for normal operations takes place through HTTP REST APIs, while real-time communication, for sending notifications and managing checkout, takes place through Socket.IO with a Redis adapter to synchronize events between multiple backend instances.

3.2 Infrastructure

The system architecture will be distributed into several infrastructural components. Clients will access the application through their web browsers by connecting to the Nginx proxy service (`proxy`), which will act both as reverse proxy and load balancer. It will distribute all the traffic between two backend instances (`backend-1` and `backend-2`) that will handle the core application logic.

For persistent data storage, the application will rely on a MongoDB replica set composed of `mongo1` (primary), `mongo2` (secondary), and `mongo-arbiter`.

To support distributed locks, the infrastructure will include Redis, composed of `redis-master`, `redis-replica-1`, and `redis-replica-2`. These nodes will be monitored by three Redis Sentinel services (`redis-sentinel-1`, `redis-sentinel-2`, and `redis-sentinel-3`), which will provide automatic failover capabilities.

All internal services will communicate exclusively through the isolated Docker network `inner`, while only the frontend service (`frontend`) and the Nginx proxy service will be exposed through the `outer` network. Service discovery will be entirely managed through Docker DNS, allowing containers to communicate using their service names. Figure 2 shows the infrastructure diagram of LastBite; the diagram contains the essential

components of the infrastructure, without yet showing implementation details such as Docker networks, ports, etc.

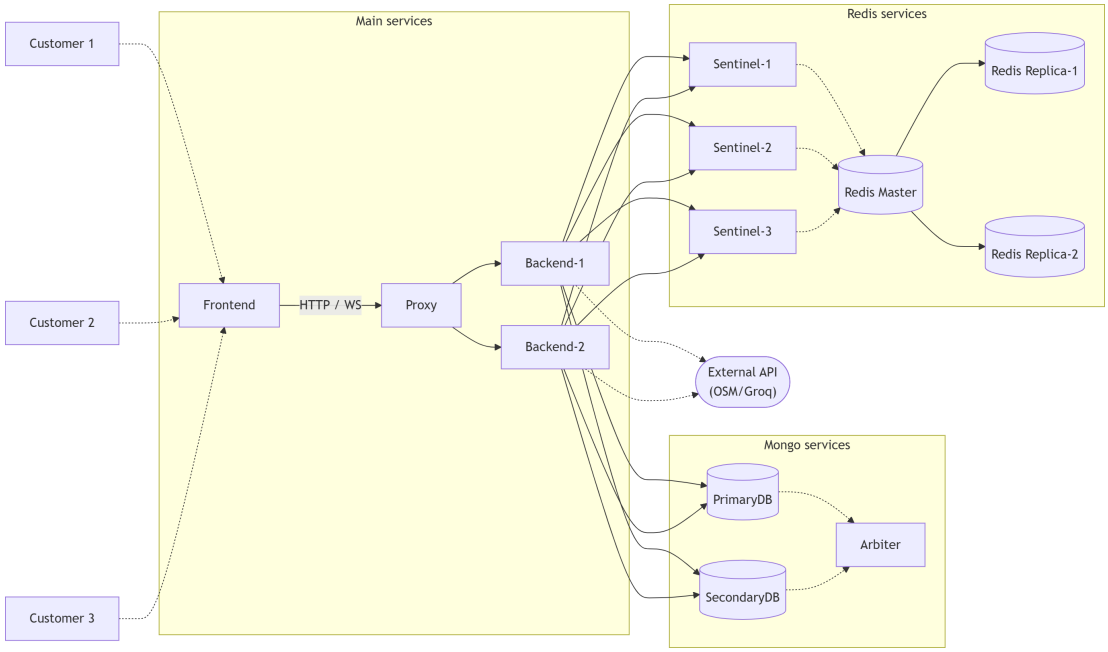

Figure 2: LastBite Infrastructure diagram.

3.3 Modelling

3.3.1 Domain entities

The main entities belonging to the application domain and their relationships are the following:

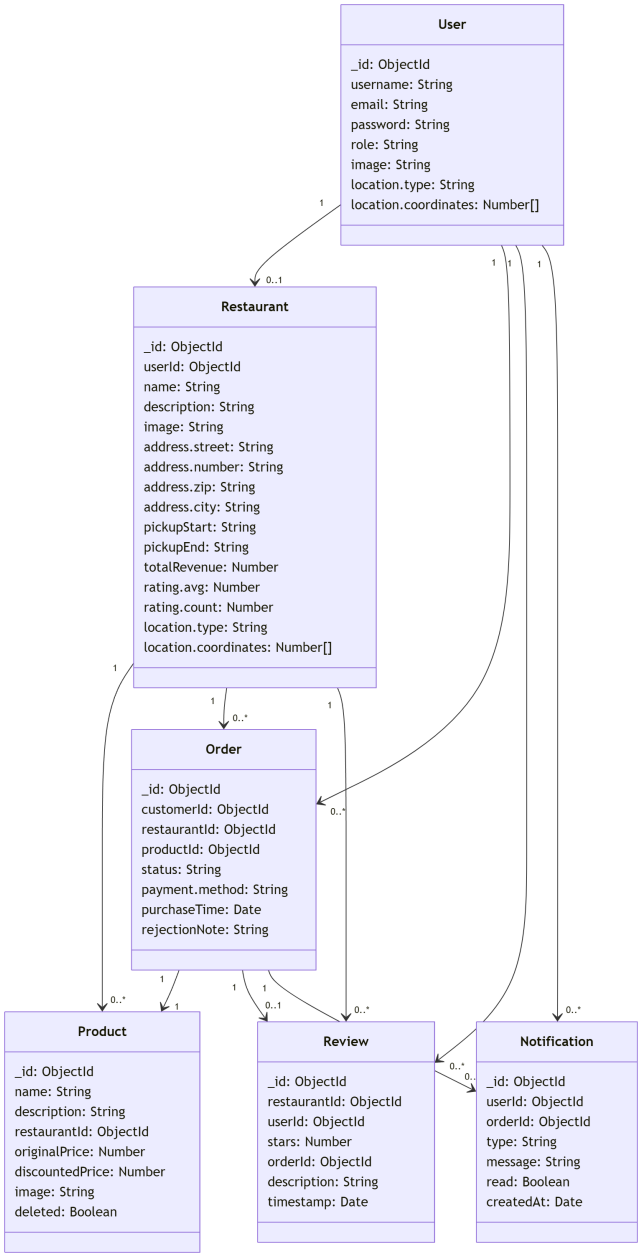

Figure 3: LastBite domain entities Class diagram.

Within LastBite, customer users are mapped as domain `Users`, while a User with `role: restaurant` is used to represent restaurant owners. The latter also owns a `Restaurant`, associated with the restaurant-owner user through `userId`. A restaurant can have several associated `Products` with various information, including `deleted`, which indicates whether the product has been deleted (logical deletion). Each `Order` corresponds to only one product. Orders can be rejected, optionally providing a reason,

or completed. If they are completed, customers can optionally leave a `Review` after pickup. During order status updates or after leaving a review, users receive specific `Notifications` based on their role.

3.3.2 Mapping between domain entities and infrastructural components

Persistent domain entities, such as users, restaurants, products, orders, notifications, and reviews, are stored in MongoDB because they represent the main state of the application and must survive container restarts or crashes. In particular, orders and products require strong consistency during purchase operations, which is why the backend coordinates changes through transactions and controlled database access.

The temporary state related to checkout, on the other hand, is not saved as permanent data: product locks are maintained in Redis with a TTL, so as to lock a shared resource only for the time needed to complete the purchase. If the customer abandons checkout or the process fails, the lock expires automatically and the product becomes available again.

User representations and inputs remain on the client: the frontend maintains the interface state, search filters, selected position, and current screens. Live updates, such as notifications, order status changes, and product availability, are propagated from the backend to clients through Socket.IO; however, the authoritative source of state always remains server-side, between MongoDB and Redis.

3.3.3 Domain events

Domain events represent significant changes in the application state, that is, actions that modify persistent entities, trigger notifications, or require coordination between distributed components. In the case of LastBite, the main events are:

- User registered: a new customer or restaurant-owner account is created.
- Login performed: the user is authenticated and receives a session token.
- Restaurant updated: the restaurant owner modifies the information of their business, such as description, address, or opening hours.
- Product published: a restaurant owner adds a new product available for purchase.
- Product modified or removed: the information of a product changes or the product is no longer available.
- Checkout started: a customer selects a product and the backend acquires a distributed lock in Redis.
- Checkout expired: the temporary lock is not renewed in time, for example if the checkout page is closed, and the product becomes available again.
- Order created: the customer confirms the purchase and the system records a new order.

- Order accepted, completed, or canceled: the restaurant owner or the customer modifies the order status.
- Review submitted: the customer publishes a rating after the order has been completed.
- Notification generated: the backend creates and sends a notification related to relevant changes, such as new orders or status updates.

3.3.4 Message classification

Commands

Method	Endpoint	Description
POST	/api/users	Registration of a new user
POST	/api/users/auth	User login
POST	/api/users/validate	Validation of the current JWT token
POST	/api/users/logout	User logout
PATCH	/api/users/:userId	User profile update
POST	/api/restaurants	Registration of a new restaurant
PATCH	/api/restaurants/:id	Restaurant data update
POST	/api/products	Creation of a new product for a restaurant
PATCH	/api/products/:id	Update of an existing product
DELETE	/api/products/:id	Logical deletion of a product
POST	/api/orders/checkout/start	Checkout start: acquires the lock on the product
POST	/api/orders/checkout/complete	Checkout completion: creates the order
PATCH	/api/orders/:id/complete	Restaurant marks the order as completed
PATCH	/api/orders/:id/cancel	Cancellation of an order
POST	/api/reviews	Creation of a new review for an order
PATCH	/api/notifications/:id/read	Marks a notification as read
PATCH	/api/notifications/read-all	Marks all user notifications as read
POST	/api/chat	Message sent to the AI chatbot

Queries

Method	Endpoint	Description
GET	/health	Service health check
GET	/api/users/me	Retrieves the authenticated user's profile
GET	/api/restaurants	Restaurant search

GET	/api/restaurants/me	Retrieves the authenticated user's restaurant
GET	/api/restaurants/dashboard	Restaurant dashboard statistics
GET	/api/restaurants/:id	Restaurant details with dish list
GET	/api/orders	List of authenticated user's orders
GET	/api/notifications	List of authenticated user's notifications
GET	/api/reviews/:restaurantId	List of reviews for a restaurant
GET	/uploads/*	Static serving of uploaded files

Events (WebSocket)

Recipient	Event	Room	Description
Server	connection	–	WebSocket connection with authentication via JWT cookie. The client is assigned to a room.
Server	checkout:heartbeat	–	Heartbeat sent by the client to keep the lock on a product active during checkout.
Server	disconnect	–	Client disconnection: releases all user locks.
Client	product:update	public restaurant_{id}	A product has been created, updated, or deleted.
Client	product:locked	public restaurant_{id}	A product has been locked during an ongoing checkout.
Client	product:sold	public restaurant_{id}	A product has been sold; the backend emits this event to the public room and to the restaurant-specific room.
Client	product:unlocked	public restaurant_{id}	The lock on a product has expired or has been released.
Client	product:checkout_conflict	user_{id}	Notifies the customer that the product in checkout has been modified or deleted by the restaurant.

Client	`order:update`	`user_{customerId}` `restaurant_{id}`	Order status update.
Client	`notification`	`restaurant_{id}` `user_{id}`	New real-time push notification, for example for a new order, order status change, or new review.

WebSocket events can be directed to a specific customer or restaurant, using the room with the respective IDs, or to the `public` room, to which all customers are connected.

3.3.5 System state

The system state includes all the information necessary to represent users, available resources, and ongoing operations. In particular, it includes:

- User data: account, credentials, roles, personal profile, and selected geographic position.
- Restaurant data: business information, address, coordinates, opening hours, statistics, and associated owner.
- Product data: name, description, original price, discounted price, availability, image, and restaurant of ownership.
- Order data: customer, restaurant, purchased product, order status, and timestamps of the main operations.
- Review data: author, reviewed restaurant, reference order, rating, and comment.
- Notification data: recipient, content, associated event or order, and read status.
- Temporary state: checkout locks maintained in Redis to coordinate concurrent access to products during purchase.

3.4 Interaction

The frontend communicates with the backend through RESTful APIs and socket channels for real-time event propagation. Most requests are processed following the request-response pattern, while WebSockets use Publish/Subscribe. Below is an example of a product purchase by a customer. The example is simplified for clarity, omitting the login phase and the WebSocket connection with the backend, starting from a restaurant page, without representing some less interesting secondary operations. The example also does not represent some failure cases, such as the failure to renew the lock once checkout has started, which would lead to a redirect to the Home page.

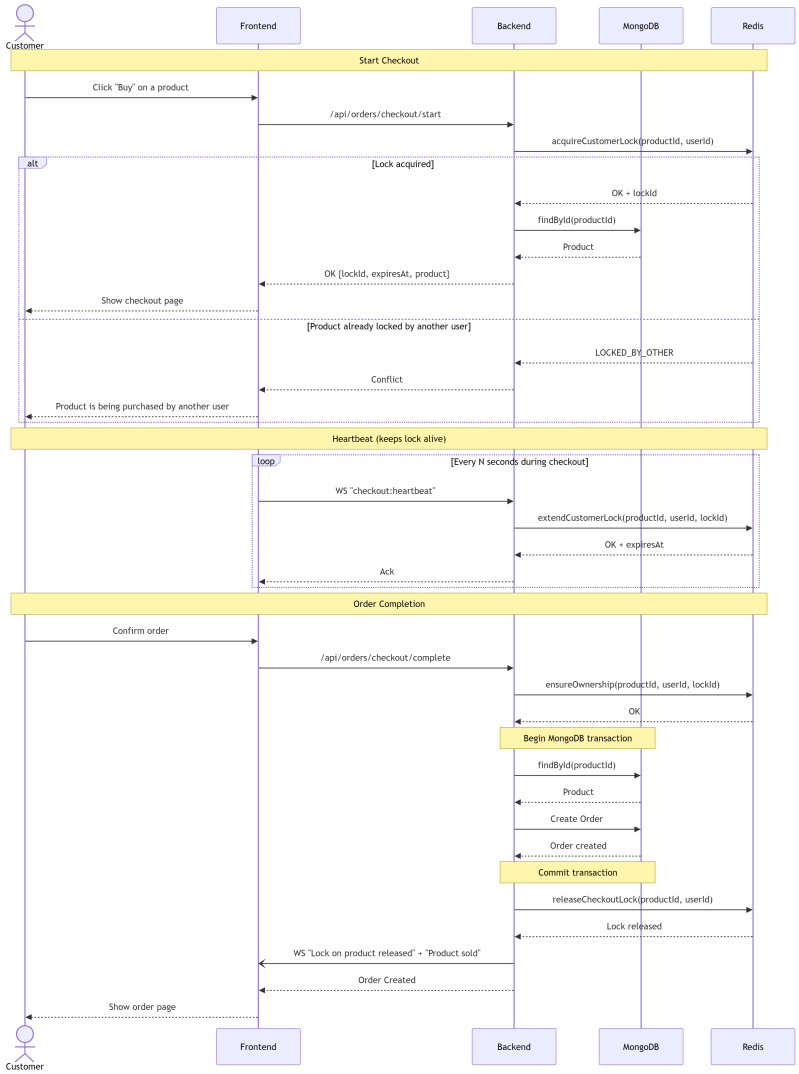

Figure 4: Sequence diagram of a product purchase.

The Frontend therefore communicates with the Backend mainly through HTTP calls, and WebSocket events for real-time operations. A WebSocket event generated by a client connected to a specific backend is propagated to the other backends through a Redis adapter for WebSocket.

3.5 Behaviour

The behaviour of the system is organized around components with different responsibilities: some maintain persistent or temporary state, while others process requests without storing local sessions.

- **Frontend**: it is stateless with respect to the application domain. It maintains only

temporary local state, such as the authenticated user, notifications, and current checkout. It reacts to REST responses and WebSocket events by updating the user interface, without directly modifying the persistent state of the system.

- **Backend**: it is stateless with respect to sessions, because each request is authenticated through the JWT in the cookie. It is the main component that interprets REST commands and WebSocket events, applies the business logic, checks permissions, and decides which updates to perform on MongoDB and Redis.
- **MongoDB**: it is the persistent stateful component. It stores users, restaurants, products, orders, reviews, and notifications. The state is modified by the backend during operations such as registration, profile update, order creation, order completion, and review insertion.
- **Redis**: it is stateful but temporary. It maintains checkout locks with TTL, information necessary for synchronizing real-time events, and pub/sub channels. The backend updates Redis when a user starts, maintains, completes, or abandons a checkout.
- **Nginx**: it is stateless. It receives HTTP and WebSocket requests from clients and forwards them to the available backend instances, contributing to load balancing and fault tolerance.

State updates are therefore concentrated in the backend: the frontend sends the user's intentions, the backend validates them and updates MongoDB for persistent data or Redis for temporary data. After each relevant change, the backend emits WebSocket events to notify the interested clients and keep the views synchronized.

The diagram in Figure 5 shows the set of possible states for the Customer user within LastBite.

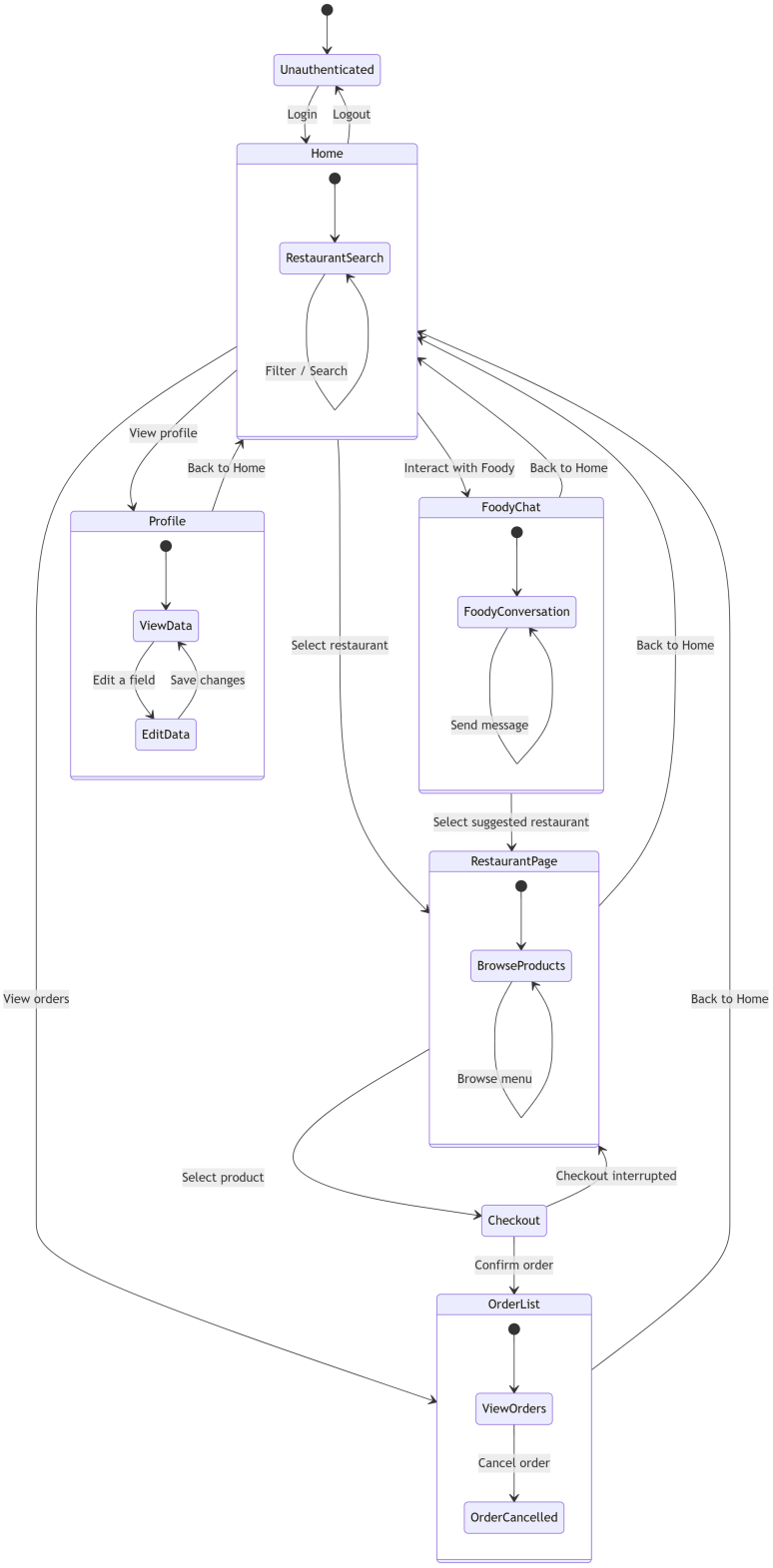

Figure 5: State diagram for the Customer user.

3.6 Data and Consistency Issues

The system must store both persistent data and temporary data shared between components. Persistent data represent the actual application state; temporary data, on the other hand, are used to coordinate concurrent operations and real-time communication.

- **Data to store**: MongoDB stores users, restaurants, products, orders, reviews, and notifications. These data must be persistent because they represent accounts, the product catalog, order history, and interactions between customers and restaurant owners. Images uploaded by users are saved in the shared Docker volume `shared_uploads`, while the corresponding path is stored in the database. Redis instead stores temporary data, in particular checkout locks with TTL and the information necessary for coordinating WebSocket events between backends.

- **Persistence model**: the main database is MongoDB, so a document-based model is used. This choice is suitable because the application's entities are naturally represented as JSON documents: for example, a restaurant contains address, coordinates, opening hours, and statistics; an order contains references to user, restaurant, product, status, and timestamps. The document model reduces the need for complex joins, supports nested data, and enables geospatial queries through `2dsphere` indexes, useful for searching nearby restaurants. Redis instead uses a key-value model, more suitable for temporary locks, TTL, atomic operations, and pub/sub.

- **Components that query the database**: the backend is the only component that directly accesses MongoDB and Redis. The frontend never queries the databases: it sends REST requests or WebSocket events to the backend. Read queries occur when a user consults their profile, restaurants, products, orders, notifications, or reviews. Write queries occur when a user registers, updates the profile, creates or modifies products, completes a checkout, changes the status of an order, or leaves a review. Centralizing access in the backend makes it possible to apply authentication, authorization, and validation before each operation.

- **Concurrency**: concurrent reads are allowed, because multiple users can consult restaurants, products, or orders without modifying the shared state. Concurrent writes, on the other hand, are critical especially during the purchase of a product available in limited quantity. The backend first acquires a lock on Redis through atomic operations; only the client that obtains the lock can complete the checkout. The final write on MongoDB then takes place in a controlled manner, updating order, product, and notifications consistently.

- **Data shared between components**: the two backend instances share MongoDB for persistent state and Redis for locks, pub/sub, and synchronization of Socket.IO events. This sharing is necessary because a client can be connected to any backend instance: without shared state, two backends could simultaneously accept the same purchase or fail to correctly propagate a notification to a client connected to the

other instance. The `shared_uploads` volume is also shared between backends, so images uploaded through one instance are also available from the other.

3.7 Fault-Tolerance

Fault tolerance is achieved by combining data replication, redundant services, health checks, timeouts, and explicit error handling. The goal is to prevent the failure of a single container from making the entire system unavailable.

- **Replication, sharing, and redundancy**: MongoDB is configured as a replica set with two data nodes and an arbiter. The primary handles writes, while the secondary maintains a replicated copy of the data; if the primary fails, the replica set can elect a new primary. The arbiter does not store data, but participates in the election quorum. Redis runs with one master, two replicas, and three Sentinels: the replicas maintain a copy of the master's state and the Sentinels are used to promote a replica in case of master failure. In addition, the two backend instances share MongoDB, Redis, and the `shared_uploads` volume, so they can serve the same requests even if one of the two instances is unavailable.
- **Heartbeat, timeout, and retry**: Docker Compose uses health checks to verify that MongoDB, Redis, Sentinel, and the backend are ready. The backends expose `/health`, which checks connectivity to MongoDB and Redis; Nginx and the other services can therefore avoid sending traffic to unhealthy containers. During checkout, the frontend sends a periodic WebSocket heartbeat to keep the Redis lock on the product active: if the heartbeat stops, the lock TTL expires and the product becomes available again. Proxy-side timeouts and connection retries are also present: Nginx can switch to another backend instance if one does not respond, while the backend retries the initial connection to data services when they are not yet ready.
- **Error handling**: if a backend instance fails, Nginx stops using it and routes new requests to the other instance, which accesses the same shared data. If MongoDB loses the primary, the replica set elects a new primary; during the election, some writes may be temporarily delayed, but the persistent state remains available after failover. If the Redis master fails, the Sentinels promote a replica and the backend Redis client reconnects to the new master. If a client interrupts checkout or loses the WebSocket connection, the lock is released on disconnection or expires automatically through TTL.
- **Consistency during failures**: critical operations, such as order creation, are executed by the backend in a controlled manner: first the Redis lock is acquired, then the necessary MongoDB documents are updated. This prevents duplicate purchases even with multiple active backends. Errors during application operations are intercepted and transformed into consistent HTTP responses; WebSocket events are used to realign clients when a product is locked, unlocked, or when the status of an order changes.

3.8 Availability

System availability is improved through lightweight caching, load balancing, and service separation. The goal is to keep the main functionalities operational even in the presence of traffic peaks or partial failures.

- **Caching**: the system mainly uses browser-side caching and in-memory frontend caching. Uploaded images, such as product photos, avatars, and restaurant images, are served by the backend with HTTP cache headers, so the browser can reuse them for several hours without requesting them on each navigation. This reduces traffic and latency, because most images rarely change. The frontend also maintains some temporary data in Pinia stores, such as the authenticated user, notifications, and checkout state, avoiding repeated REST requests during the same session. Persistent caching is not used for domain data, because orders, product availability, and locks must remain up to date.
- **Load balancing**: Nginx balances requests between the backends. This makes it possible to distribute the load of REST calls and WebSocket connections across multiple instances, preventing a single backend from becoming a bottleneck. If one instance does not respond, Nginx can forward traffic to the other.
- **Network partitioning**: if a backend loses the connection to MongoDB, it cannot read or write persistent data and therefore rejects operations that require the database. If it loses the connection to Redis, functions that depend on locks, such as concurrent checkout, cannot be guaranteed and are blocked or fail. If the partition involves MongoDB, the replica set prioritizes consistency: writes are accepted only by the node that can act as primary with a valid quorum. If the partition involves Redis, the Sentinels try to promote a reachable replica, restoring the master when possible.
- **Availability choices**: the system prioritizes consistency in critical operations, especially checkout and orders. It is preferable to temporarily reject a purchase rather than sell the same product twice. For non-critical operations, such as consulting restaurants, products, or already cached images, the system can continue to offer partial availability even if some services are not reachable.

3.9 Security

System security is based on token-based authentication, role-based authorization, and the use of cryptographic primitives to protect credentials and sessions.

- **Authentication**: users log in through the backend, which verifies email and password and generates a JWT token. The token is saved in the browser as an `HTTP-only` cookie, so it is not accessible from the frontend JavaScript code. Every protected REST request and every WebSocket connection are authenticated by the backend by verifying the token present in the cookie. In this way, the backend

does not need to maintain server-side sessions and each request can be validated independently.

- **Authorization**: the system uses RBAC, that is, Role-Based Access Control. The main roles are `customer` and `restaurant`. A customer can search for restaurants, view products, start and complete checkout, consult their own orders, cancel them within the expected limit, leave reviews, and manage their own profile. A restaurant can manage only its own restaurant, create and modify its own products, view received orders, complete or cancel them, and consult its own statistics. The backend applies these controls through middleware, verifying both the role and ownership of the requested resource.
- **Resource access control**: in addition to the role, the system checks that the user is operating on data within their competence. For example, a restaurant owner cannot modify products of another restaurant, and a customer cannot consult or modify orders belonging to other users. Database queries are therefore filtered using identifiers such as `userId` and `restaurantId`, avoiding cross-account access.
- **Cryptographic schemes**: passwords are not stored in plain text, but are hashed with Argon2 before persistence in MongoDB. JWTs are signed with a secret key configured through an environment variable, and the backend verifies their signature and validity before accepting a request.

4 Implementation

This section summarizes the main technology-dependent implementation choices adopted to realize the design described in the previous sections.

4.1 Network Protocols

The system uses different protocols depending on the type of communication required:

- **HTTP REST**: used for synchronous communication between the Vue frontend and the Express backend. The REST APIs cover authentication, users, restaurants, products, orders, reviews, notifications, and chatbot.
- **WebSocket**: used for real-time bidirectional communication. It is used for notifications, order updates, and product status during checkout.
- **TCP**: underlying transport protocol for HTTP, WebSocket, MongoDB, and Redis.
- **Redis Pub/Sub**: used internally by the Socket.IO Redis adapter to propagate events between `backend-1` and `backend-2`.
- **Docker DNS**: used for internal service discovery. Containers communicate through the names defined in `docker-compose.yml`.

4.2 In-Transit Data Representation

The data exchanged between frontend and backend are represented mainly in JSON, a choice consistent with the MEVN stack and with the use of REST. Image uploads instead use `multipart/form-data`, handled on the backend side with Multer. The authentication JWT travels as a compact Base64URL string inside an `HTTP-only` cookie.

4.3 Database Queries

The backend is the only component that directly queries the databases. MongoDB is the main document-based database and is used for persistent data such as users, restaurants, products, orders, reviews, and notifications. It is queried through Mongoose, which provides schemas, validation, NoSQL queries, indexes, and transactions. Redis is instead used as a key-value database for temporary data and distributed coordination, in particular checkout locks, TTLs, and pub/sub. MongoDB was chosen because the document model adapts well to nested JSON entities and geospatial queries on restaurants; Redis because it offers native atomic operations and TTLs, necessary to implement secure checkout locks.

4.4 Authentication

Authentication is implemented through JWT. At login, the backend verifies the credentials and generates a signed token using the `jose` library. The token is sent to the browser as an `HTTP-only` cookie; the frontend does not read it directly, but the browser automatically attaches it to subsequent requests.

The backend verifies the token both for protected REST APIs and during the Socket.IO handshake. In this way, REST and WebSocket share the same authentication mechanism and the backend remains stateless with respect to sessions.

4.5 Authorization

Authorization is implemented through RBAC, Role-Based Access Control. The main roles are:

- `customer`: can search for restaurants, view products, purchase, consult their own orders, cancel them within the expected limit, leave reviews, and manage their own profile.
- `restaurant`: can manage its own restaurant, create and modify its own products, view and update received orders, and consult statistics.

In addition to the role, the backend checks resource ownership through middleware: for example, a restaurant can modify only its own products and a customer can access only their own orders.

4.6 Technological Details

- **Backend**: Node.js with Express, organized into routes, controllers, models, services, validators, and utils. Express exposes the REST APIs and integrates Socket.IO for real-time communication. All exposed endpoints validate incoming requests with Joi before serving them.
- **Frontend**: Vue 3 with Composition API, Vite, Vue Router, Pinia, Axios, and Socket.IO client. Pinia maintains local authentication, notification, and checkout state. An initial local validation with Zod is performed before forwarding requests to the backend.
- **Database**: MongoDB with Mongoose for document modeling, validation, queries, and transactions. `2dsphere` indexes support geospatial restaurant search.
- **Locking and real-time coordination**: Redis for distributed locks, TTLs, and pub/sub. The Socket.IO Redis adapter synchronizes events between multiple backends.
- **Upload**: Multer handles `multipart/form-data` files; images are saved in the shared volume `shared_uploads` and served as static files.
- **Security**: Argon2 is used for password hashing; `jose` is used to generate and verify signed JWTs.
- **AI integration**: the Food Advisor is implemented through the Groq SDK. The backend classifies the message intent and then queries MongoDB to produce the response.
- **Deployment**: Docker Compose orchestrates frontend, two backends, Nginx, MongoDB replica set, Redis master/replicas, and Sentinel.

The main functionalities have been implemented. Any future extensions or incomplete functionalities are discussed in section 9.

5 Validation

System validation was carried out by combining automatic tests on the backend and manual tests on the distributed and real-time parts. The automatic tests mainly verify controllers, middleware, models, and HTTP integration; the manual tests instead cover concurrent checkout, Socket.IO, user interface, and integration with external services.

5.1 Automatic Testing

The project includes an automatic test suite based on Vitest and Supertest. Vitest runs the tests, while Supertest sends real HTTP requests to the Express application, making it possible to verify routing, middleware, authentication, authorization, and controllers.

To avoid dependencies on external infrastructure, the tests use `mongodb-memory-server`, which starts an in-memory MongoDB instance and resets it between test cases.

The individual backend components are tested at the HTTP integration level: each test exercises a route and verifies the interaction between input validation, middleware, controller, and database. This approach was preferred over isolated unit tests because many critical functionalities depend on cooperation between multiple application layers.

- **Users**: the tests verify registration, login, session validation, profile update, and logout. The rationale is to check that a user can authenticate correctly, receive the session cookie, and access protected routes. The corner cases include wrong credentials, missing token, duplicate username, and duplicate email. These tests validate the requirements for registration, secure login, and profile management.

- **Restaurants**: the tests verify creation and update of the restaurant profile, retrieval of restaurants, and geospatial search. The rationale is to check that only users with the restaurant role can manage a restaurant and that customers can correctly consult the results. The corner cases include unauthorized access, missing data, and attempts to modify resources belonging to others.

- **Products**: the tests verify creation, modification, logical deletion, and retrieval of products. The rationale is to ensure that only the owner restaurant can modify its own products and that deleted products are no longer returned in public queries. These tests validate the restaurant owner's product management requirements and product visualization for the customer.

- **Orders**: the tests verify creation, completion, and cancellation of orders within or beyond the expected time window. The rationale is to check that the purchase flow keeps the order and product state consistent. The corner cases include cancellations after the maximum time and operations on invalid orders.

- **Reviews**: the tests verify review creation and restaurant rating update. The rationale is to check that a review can be left only on completed orders belonging to the correct user. These tests validate the post-order review requirement.

- **Notifications**: the tests verify retrieval of unread notifications, marking a single notification as read, and marking all notifications as read. The rationale is to check that each user sees only their own unread notifications, that a user cannot mark as read a notification belonging to another user, and that the request on a nonexistent notification responds with an error. The corner cases include the user with no notifications, bulk marking with zero unread notifications, and the attempt of unauthenticated access.

- **Utility**: the tests verify the behavior of utility functions, such as the one for formatting order IDs, creating AppError, and validating the discounted price of a product. For formatOrderId, the correct ID format and edge cases with null input are tested. For AppError, the correct setting of fields, inheritance from Error, and

the fromCode factory method are verified. For validateDiscountedPrice, the cases in which the discounted price is lower than, equal to, or higher than the original price and edge cases with null values are tested. The rationale is to guarantee the correctness of cross-cutting utility functions used throughout the backend. These tests are purely unit tests and do not require a database.

The integration between components is tested by sending real HTTP requests to the Express app and verifying the effects on the in-memory MongoDB. In this way, not only an isolated function is tested, but also the communication between routes, middleware, controllers, and database. The suite does not require MongoDB or Redis to be installed locally. To run the automatic tests, the following commands can be executed:

```
cd backend
npm test
```

To run them in watch mode during development:

```
cd backend
npm run test:watch
```

For a complete end-to-end test of the system, it is necessary to start a production-like environment with Docker Compose, including frontend, proxy, two backends, MongoDB replica set, Redis, and Sentinel. In this environment, user scenarios can be automated with tools such as Playwright or Cypress: registration, login, restaurant search, checkout, order update, and review. This E2E automation was not integrated into the project; the equivalent checks were carried out manually as acceptance tests.

5.2 Acceptance test

Manual tests were performed on the core functionalities of the project, in particular those involving concurrency, real-time communication, graphical interface, and external services.

Concurrent checkout, already tested with automatic tests, was also tested by opening two browser sessions with two different customer accounts and attempting to purchase the same product at the same time. The expected result is that only one user acquires the lock and can proceed, while the other receives the indication that the product is temporarily unavailable. Abandoning the checkout page was also tested, verifying that the lock was released or expired through TTL.

Socket.IO events were tested by keeping a customer view and a restaurant dashboard open at the same time. The arrival of notifications, order updates, and product status updates in real time were verified.

The Food Advisor was manually tested with requests in Italian and English, including ambiguous or out-of-context formulations. It was not automated because it depends on an external call to the Groq model and therefore can produce non-deterministic responses.

6 Deployment

The deployment of LastBite was designed to be reproducible from scratch through Docker and Docker Compose. The goal is to allow the entire distributed architecture to be started without manually installing Node.js, MongoDB, Redis, or Nginx on the host machine: all these dependencies are run inside dedicated containers.

6.1 Required software

To install and run the project on a local machine, only the following tools are required:

- **Docker Desktop**: recent version with Docker Compose v2 support.
- **Git**: required to clone the project repository.
- **System shell**: PowerShell or Command Prompt on Windows, standard terminal on macOS/Linux.

It is not necessary to install specific versions of Node.js, npm, MongoDB, Redis, or Nginx on the host. The runtime versions are defined by the Docker images and the project's Dockerfiles: the backend and frontend are built from their respective folders, MongoDB uses the `mongo:latest` image, Redis uses `redis:7-alpine`, and the reverse proxy uses `nginx:latest`.

6.2 Installation from scratch

The installation starts from an environment where Docker Desktop is already installed and running. The steps are the following:

1. Clone the repository:

```
git clone https://github.com/piertv21/LastBite.git
cd LastBite
```

2. Create the local configuration file by copying the environment variables template:

```
cp .env.example .env
```

3. Open the `.env` file and configure at least the mandatory variables, in particular the `GROQ_API_KEY` key if the Food Advisor needs to be enabled.

4. Build and start all services:
   - on Windows it is possible to run `.\start.bat`;
   - on macOS/Linux it is possible to run manually:

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

5. Wait for initialization to complete. On the first startup, Docker downloads the base images, builds the custom backend and frontend images, initializes the MongoDB replica set, and waits for Redis Sentinel to be operational.

At the end of startup, the user should expect the application to be reachable at the following addresses:

- Frontend Vue: `http://localhost:5173`;
- Backend API through Nginx: `http://localhost:3000`;

6.3 Environment variables and configuration

The application configuration is centralized in the `.env` file, loaded by the backend and frontend services. The main variables are:

- `GROQ_API_KEY`: API key required to use the Food Advisor assistant. If missing or invalid, the AI part does not work correctly, while the rest of the application remains startable.
- `JWT_KEY`: secret used to sign and verify JSON Web Tokens. In a real environment, it must be replaced with a long, random string that is not publicly shared.
- `VITE_API_BASE_URL`: address used by the frontend to reach the APIs. Locally, the expected value is `http://localhost:3000`.
- `CHECKOUT_LOCK_TTL_MS`: duration of the Redis lock used during checkout to prevent concurrent purchases of the same product.

6.4 Engineering of docker-compose.yml

The `docker-compose.yml` file describes fourteen services and two Docker networks. The architectural choice locally replicates a distribution composed of multiple backend nodes, replicated database, a frontend and reverse proxy.

6.4.1 Application services

There are two backend instances, `backend-1` and `backend-2`, both built from the `./backend` folder. The two instances expose the same Node.js/Express application but differ in the `SEED_DATABASE` variable: only the first one performs the initial seed. Both mount the shared volume `shared_uploads` in `/root/app/src/uploads`, so files uploaded through one instance remain available to the other as well.

The frontend is built from the `./frontend` folder, runs `npm run dev`, and exposes port `5173`. The `proxy` service, based on `nginx:latest`, exposes host port `3000` and loads the configuration from `./nginx/nginx.conf`. Nginx forwards requests to the backends and allows traffic to be balanced between the two instances.

6.4.2 MongoDB replica set

The main persistence is managed by three MongoDB containers: `mongo1`, `mongo2`, and `mongo-arbiter`. The `mongo-init` service does not remain running: it waits for the three nodes to be healthy and then performs initialization, configuring `mongo1` and `mongo2` as data members and `mongo-arbiter` as arbiter. The backends depend on `mongo-init`, so they start only after the replica set has been configured.

6.4.3 Redis, replicas, and Sentinel

Redis is used both as in-memory support and for the distributed locking system. The compose defines one master, `redis-master`, and two replicas, `redis-replica-1` and `redis-replica-2`. The availability of the Redis master is monitored by three Sentinel instances: `redis-sentinel-1`, `redis-sentinel-2`, and `redis-sentinel-3`. Each Sentinel generates its own configuration file at runtime, identifies the address of `redis-master` through Docker DNS, and monitors the master with a quorum equal to 2. The parameters `down-after-milliseconds`, `failover-timeout`, and `parallel-syncs` respectively define the failure detection times, the failover timeout, and the number of replicas synchronized in parallel.

6.4.4 Dependencies, health checks, and restart policy

The startup order is not entrusted only to container creation, but to explicit health checks. MongoDB uses a ping through `mongosh`; Redis master uses `redis-cli ping`; replicas verify their role through `redis-cli INFO replication`; Sentinels check the presence of the master with `redis-cli -p 26379 SENTINEL masters`; the backends verify the `/health` endpoint of their own application.

The `depends_on` directives with `condition: service_healthy` ensure that frontend and proxy start only when both backends are healthy, and that the backends start only when MongoDB and Redis Sentinel are ready. The application and stateful services use `restart: unless-stopped`, so they are automatically restarted after crashes or a machine restart. `mongo-init`, instead, uses `restart: "no"` because it must be executed only once.

6.4.5 Networks and volumes

The deployment defines two bridge networks: `inner` and `outer`. The `inner` network connects all internal services and also allows internet access, necessary for the backend to contact external APIs such as OpenStreetMap and Groq. The `outer` network is used by the services exposed to the host, namely frontend and proxy.

The `redis_master_data`, `redis_replica_1_data`, and `redis_replica_2_data` volumes persist Redis data between restarts. The `shared_uploads` volume stores uploads shared between the two backend instances.

7 User Guide

This section provides a step-by-step guide to using the LastBite application. The platform supports two types of users: Customer and Restaurant. Each role has its own set of pages and functionalities, which are described in detail below. During the design process, several basic usability and *Human–computer interaction* principles were taken into account. The Nielsen heuristics[1] were applied whenever possible, and the final design was also informed by interviews with a heterogeneous group of users, aimed at collecting feedback and suggestions on the platform's usability.

7.1 Authentication

7.1.1 Registration

To create a new account, navigate to the **Register** page by clicking *"Register now"* on the Login screen.

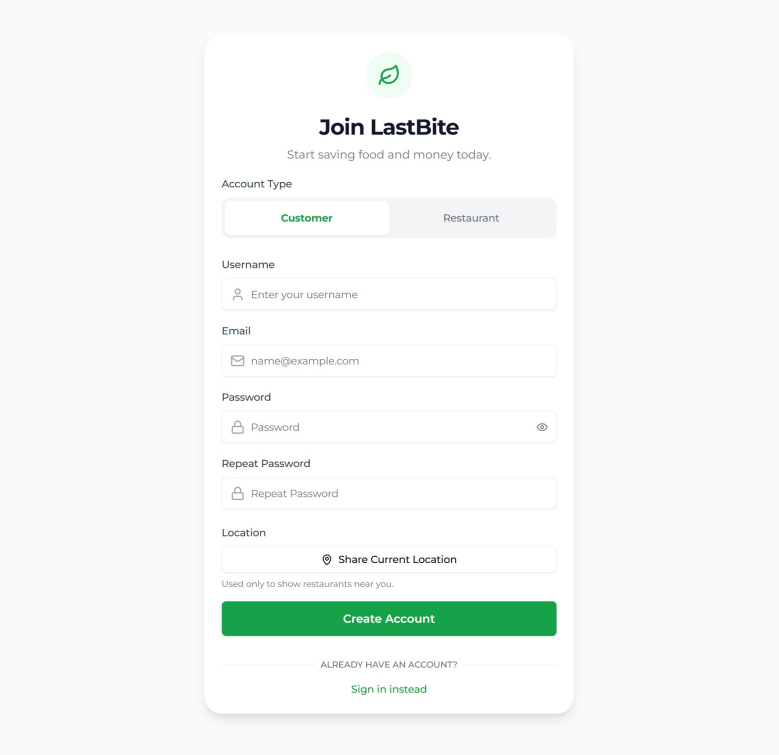

Figure 6: Customer registration form.

1. **Select the account type**: choose between **Customer** and **Restaurant** by clicking the corresponding tab at the top of the form.

[1] `https://www.trapstudio.it/ui-design-le-10-euristiche-di-nielsen/`

2. **Fill in the required fields**:
    - **Customer**: username, email, password, and location (obtained via the *"Get Location"* button, which uses the browser's GPS).
    - **Restaurant**: username, email, password, restaurant name, description, pickup start/end time, and full address (street, number, ZIP, city).
3. Click **"Create Account"** (or **"Create Restaurant Account"** for the restaurant form).

Expected outcome: upon successful registration, the user is redirected to the `/home` or `/dashboard` page, depending on their role. If any field is missing or invalid, an error alert is displayed at the top of the form.

7.1.2 Login

To sign in, open the **Login** page (the default landing page of the application).

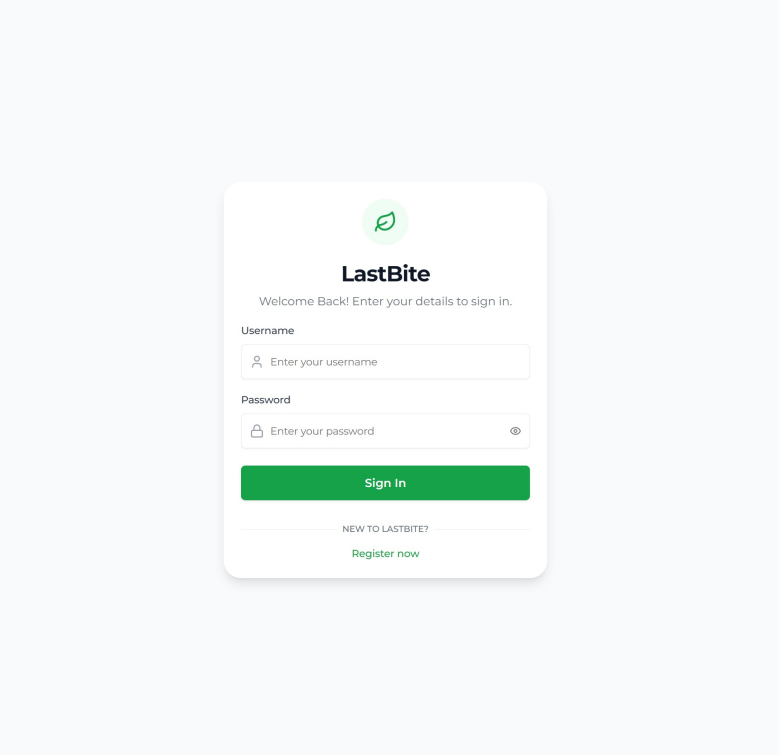

Figure 7: Login page.

1. Enter your **username** and **password** in the corresponding fields.
2. Click the **"Sign In"** button.

Expected outcome:

- If the user is a **Customer**, they are redirected to the `/home` page.
- If the user is a **Restaurant**, they are redirected to the `/dashboard` page.
- If credentials are invalid, a destructive error alert is shown above the form.

7.2 Customer Operations

After logging in as a Customer, the navigation bar provides access to: **Home**, **Orders**, **Foody** (Food Advisor), and **Profile**. A notification bell and user avatar are always visible in the top-right corner.

7.2.1 Searching for a Restaurant (Home)

The Home page is the main entry point for Customers. It features a hero section with a search bar and a grid of restaurant cards below.

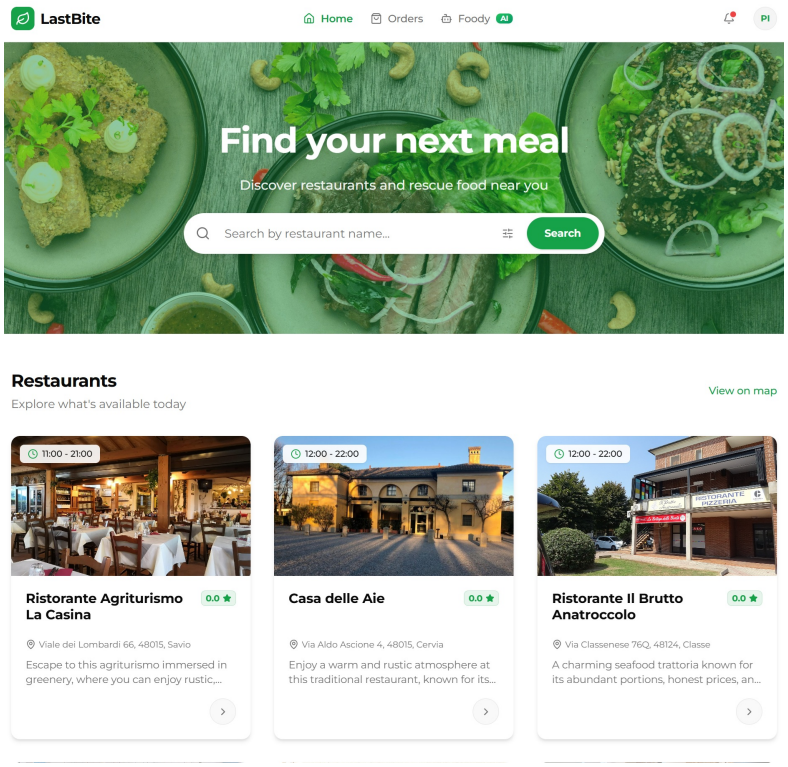

Figure 8: Customer Home page.

1. **Type a restaurant name** in the search bar located in the hero section (placeholder: *"Search by restaurant name..."*).

2. Optionally, adjust the **distance filter** by clicking the sliders icon next to the search bar:
   - A popover appears with a **distance slider** (0–50 km).
   - A **Sort By** selector is also available inside the popover, with options *Distance* and *Rating*.
3. Click the **"Search"** button or press **Enter** to apply the search.
4. Browse the results displayed as **restaurant cards** in a responsive grid (1, 2, or 3 columns depending on screen size).
5. Optionally, click **"View on map"** to switch to an interactive map view showing the restaurants' locations.
6. Click on a **restaurant card** to navigate to its detail page.

Expected outcome: the restaurant grid updates dynamically based on the search query and filters. Scrolling down triggers **infinite scrolling**, automatically loading more results. If no restaurants match the criteria, an empty state message is shown: *"No restaurants found – Try changing your search terms or filters."*.

7.2.2 Browsing Products and Purchasing (Restaurant Page)

After clicking on a restaurant card in the Home page, the user is taken to the **Restaurant Info** page.

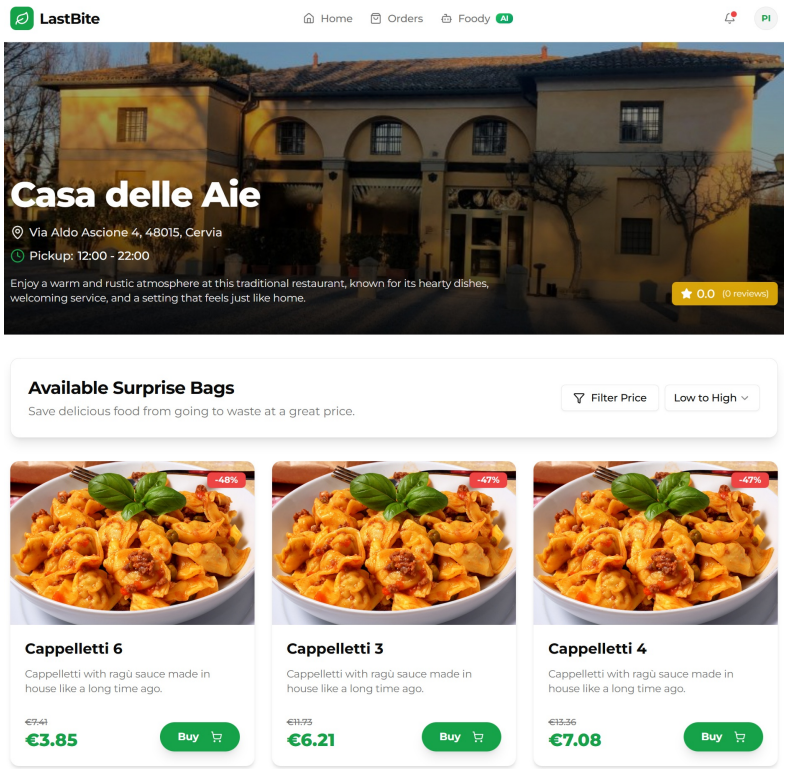

Figure 9: Restaurant detail page with available products.

1. **Browse the restaurant details**: the page displays a hero banner with the restaurant image, name, address, pickup time window, description, and average rating.
2. **Browse available products**: scroll down to the "Available Surprise Bags" section, which shows product cards in a grid. Each card displays the product image, name, description, original price (strikethrough), and discounted price.
3. Optionally, use the **"Filter Price"** button to set a price range (0–30 €, step 0.50 €) and the **sort selector** to order products by price ("Low to High" or "High to Low").
4. **Click the "Buy" button** on the desired product card.

The user is then redirected to the **Checkout** page:

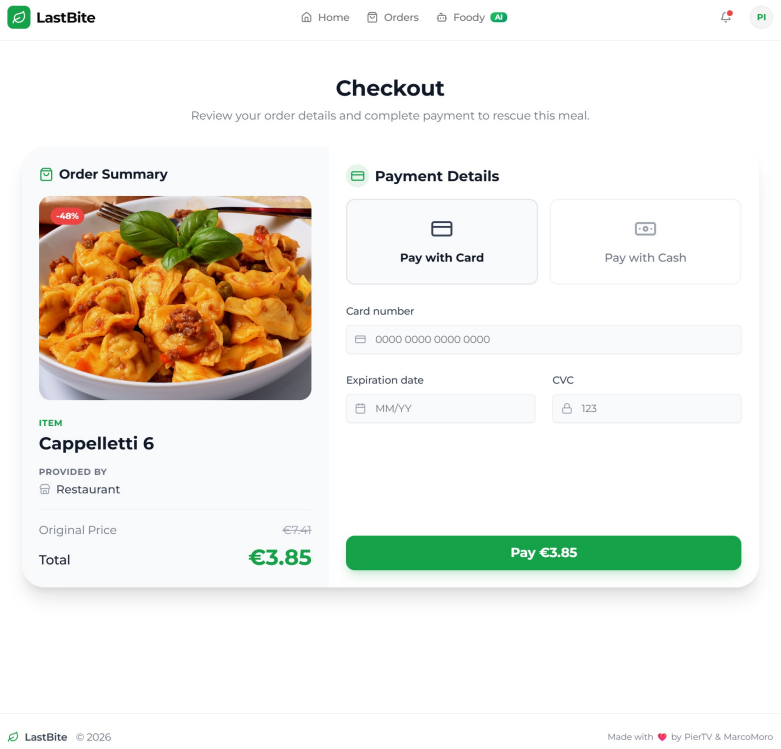

Figure 10: Checkout page.

5. Review the **order summary** on the left panel: product image (with discount percentage badge), product name, restaurant name, original price, and total.
6. **Select the payment method**: choose between **"Pay with Card"** and **"Pay with Cash"** by clicking the corresponding card.
   - **Card**: fill in card number (auto-formatted as `0000 0000 0000 0000`), expiration date (`MM/YY`), and CVC. Inline validation errors appear under each field if the data is invalid.
   - **Cash**: a message confirms that payment will be collected at pickup.
7. Click **"Pay €X.XX"** (card) or **"Confirm Order"** (cash) to complete the purchase.

Expected outcome: a success toast notification *"Order completed successfully"* is displayed, and the user is automatically redirected to the **Orders** page where the new order appears. The checkout session is maintained via WebSocket heartbeats; if the session expires, the user is redirected back to Home.

7.2.3 Viewing Orders

The **Orders** page allows Customers to monitor the status of their orders.

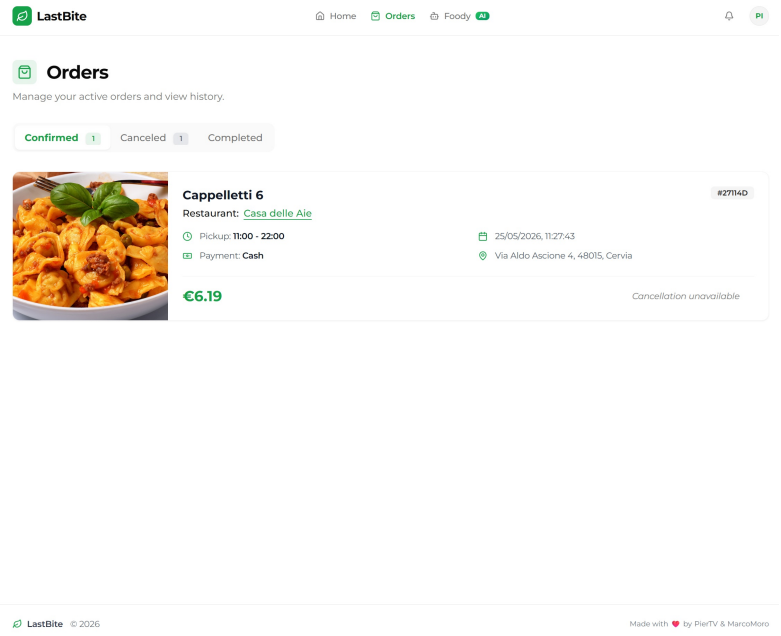

Figure 11: Customer order page.

1. Navigate to the **Orders** page from the navigation bar.
2. Orders are organized into three tabs:
   - **Confirmed** – orders awaiting pickup. Each tab label shows a count badge.
   - **Canceled** – orders that have been canceled (by the customer or rejected by the restaurant). If a rejection reason was provided, a *"View Reason"* link is available.
   - **Completed** – successfully completed orders. A **"Leave a Review"** button is shown on completed orders that have not yet been reviewed.
3. Each **order card** displays: product name and image, restaurant name (clickable link), pickup time window, purchase date, payment method (card/cash icon), restaurant address, and total price.

Expected outcome: the recently placed order appears under the *Confirmed* tab with its details. Order status updates are received in real-time via WebSocket. If the tab is empty, an appropriate empty state message is displayed.

7.2.4 Using the Food Advisor (Foody)

The **Food Advisor** page provides an AI-powered chat assistant called **Foody** that helps customers discover food and restaurants.

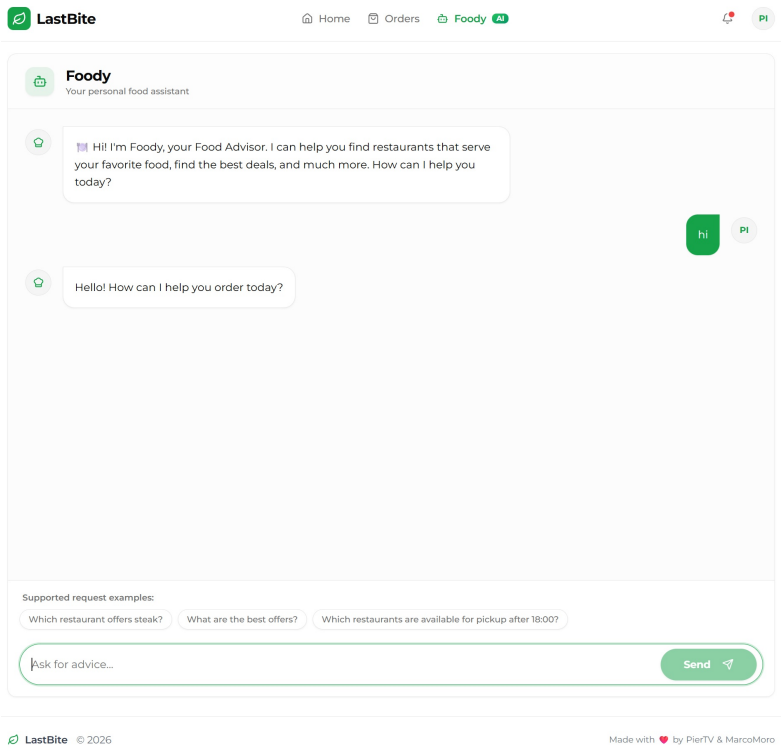

Figure 12: Food advisor page.

1. Navigate to the **Foody** page from the navigation bar (marked with an *AI* badge).
2. The chat interface opens with Foody's avatar (ChefHat icon) and the header: *"Foody – Your personal food assistant"*.
3. **Type a message** in the input field at the bottom (placeholder: *"Ask for advice..."*) or click one of the **suggestion chips** displayed above the input.
4. Press **Enter** or click the **"Send"** button to send the message.
5. Foody responds with food recommendations. While the response is being generated, a **typing indicator** (three bouncing dots) is shown.
6. If Foody suggests a restaurant, a clickable link is available in the chat bubble to navigate directly to that restaurant's page.

Expected outcome: user messages appear on the right side and Foody's responses on the left. The chat area auto-scrolls to the latest message.

7.2.5 Editing Profile Information

The **Profile** page allows Customers to manage their personal information, location, and account security.

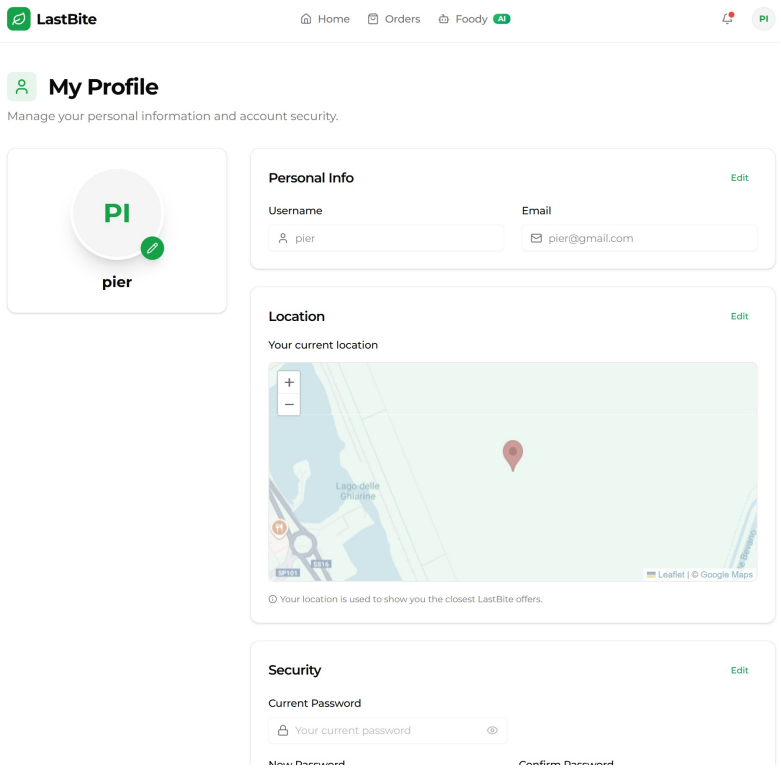

Figure 13: Profile page.

1. Navigate to the **Profile** page by clicking the user avatar in the top-right corner and selecting *"My Profile"* from the dropdown menu.
2. The page is divided into a sidebar (avatar and username) and a main area with three cards:
   - **Personal Info** – username and email fields.
   - **Location** (customer only) – an interactive map showing the current location.
   - **Security** – password change fields (current password, new password, confirm password).
3. To edit a section, click the **"Edit"** button on the corresponding card header. The fields become editable.
4. Modify the desired fields:
   - **Avatar**: click the avatar image to open a file picker and upload a new profile picture.
   - **Personal Info**: modify username and/or email.
   - **Location**: drag the pin on the map or click **"Use GPS"** to auto-detect the current position.

- **Security**: enter the current password, then the new password and its confirmation.

5. Click **"Save Changes"** at the bottom of the card to persist the modifications.

Expected outcome: a success toast notification (e.g., *"Profile updated successfully"*) confirms the changes. If validation fails (empty fields, invalid email, password mismatch), an error toast is displayed. Each section is saved independently.

7.2.6 Reading Notifications

Notifications keep the user informed about order updates and other events.

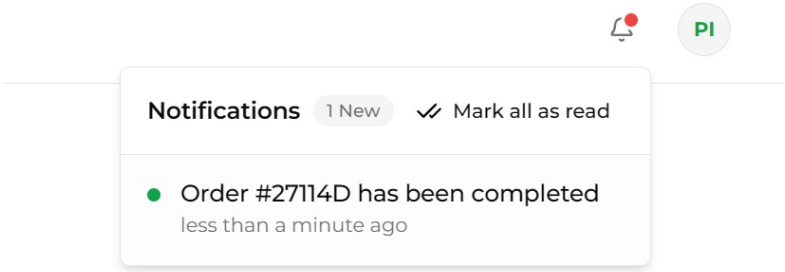

Figure 14: User notifications.

1. Click the **bell icon** in the top-right corner of the navigation bar. If there are unread notifications, a small red dot is displayed on the icon.
2. A **popover panel** opens, showing the list of notifications. Each notification displays:
   - The notification **message**.
   - A **relative timestamp** (e.g., *"2 minutes ago"*).
   - A **green dot** indicating unread status.
3. **Click on a notification** to mark it as read (the green dot disappears).
4. Optionally, click **"Mark all as read"** to clear all unread indicators at once.

Expected outcome: the unread count badge on the bell icon updates accordingly. If there are no notifications, the panel shows: *"You have no new notifications at the moment."*.

7.3 Restaurant Operations

After logging in as a Restaurant, the navigation bar provides access to: **Dashboard**, **Restaurant Info**, **Manage Products**, **Manage Orders**, and **Profile**. Notifications and the user avatar dropdown are also available.

7.3.1 Consulting Restaurant Statistics (Dashboard)

The **Dashboard** page provides an overview of the restaurant's performance.

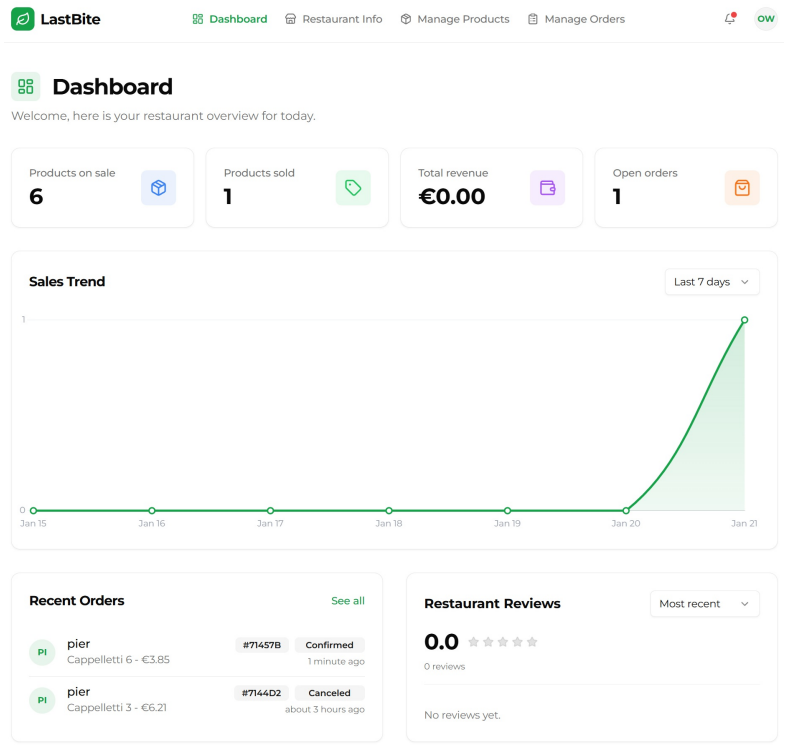

Figure 15: Restaurant dashboard page.

1. Upon login, the Dashboard is the default landing page. It displays:
   - **Four KPI cards** at the top:
     - **Products on sale** – number of currently listed products.
     - **Products sold** – total number of products sold.
     - **Total revenue** – total earnings formatted in EUR.
     - **Open orders** – number of orders awaiting action.
   - A **Sales Trend** line chart showing the number of sales over time, with a configurable period selector (*Last 7 days*, *Last 30 days*, *Last 3 months*).

- A **Recent Orders** list showing the latest orders with: customer avatar/name, product name, total price, order ID, status badge, and relative timestamp.
- A **Restaurant Reviews** panel displaying customer reviews with an average rating summary and individual reviews (sortable by recent/rating).

2. Use the **period selector** dropdown above the chart to change the time window.
3. Click **"See all"** next to "Recent Orders" to navigate to the full Orders page.

Expected outcome: the dashboard reflects real-time data. The chart dynamically updates when the period is changed. Review sorting updates the displayed reviews instantly.

7.3.2 Editing Restaurant Information

The **Restaurant Info** page allows restaurants to update their public information.

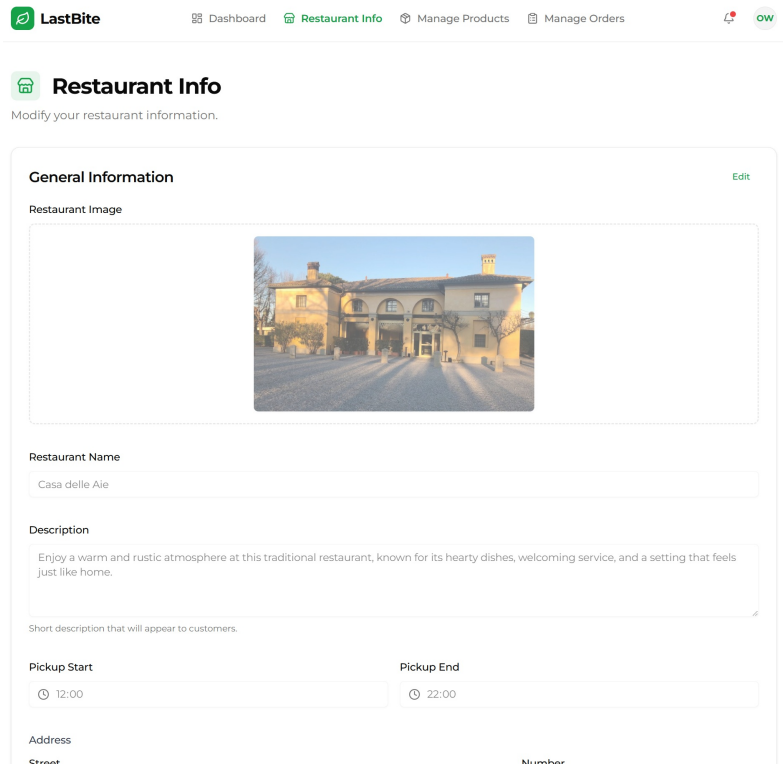

Figure 16: Restaurant info page.

1. Navigate to the **Restaurant Info** page from the navigation bar.
2. The page shows the current restaurant information in a card with the title "General Information". All fields are **read-only** by default.

3. Click the **"Edit"** button in the top-right corner of the card to enable editing.
4. Modify the desired fields:
   - **Restaurant Image** – click to upload a new image or remove the existing one via the image selector component.
   - **Restaurant Name** – text input.
   - **Description** – textarea for a short description visible to customers.
   - **Pickup Start / Pickup End** – time pickers defining the pickup window.
   - **Address** – street, number, ZIP, and city fields.
5. Click **"Save changes"** to persist the modifications.
6. To discard changes, click **"Cancel"** instead.

Expected outcome: a success toast *"Restaurant information updated successfully"* confirms the update. The form returns to read-only mode. If required fields are missing, a validation error toast is displayed.

7.3.3 Managing Products

The **Manage Products** page allows restaurants to create, edit, and delete their product listings.

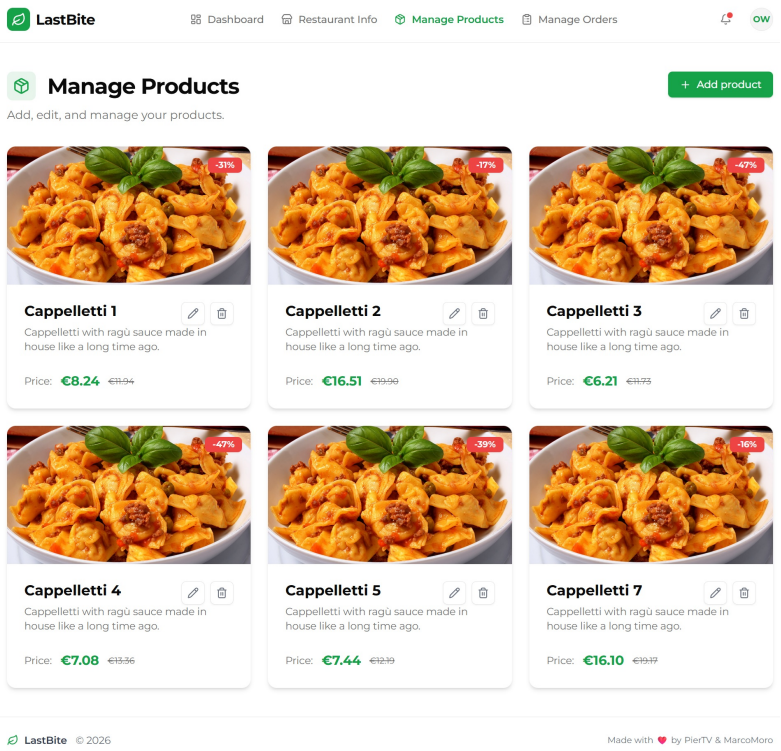

Figure 17: Restaurant products page.

Creating a product.

1. Navigate to the **Manage Products** page from the navigation bar.
2. Click the **"Add product"** button at the top-right of the page header.
3. A **dialog** appears with the title "Add New Product". Fill in the following fields:
   - **Product Image** – upload an image via the image selector.
   - **Name** – the product name.
   - **Description** – a text description of the product.
   - **Original Price** – the original price in EUR.
   - **Discounted Price** – the discounted sale price in EUR.
4. Click **"Save Product"** to create the product, or **"Cancel"** to discard.

Expected outcome: a success toast *"Product added successfully"* is shown and the new product card appears in the grid.

Editing a product.

1. On the Manage Products page, locate the product card you want to edit.
2. Click the **pencil icon** (edit button) on the product card.
3. The same dialog opens, pre-filled with the product's current data (title: "Edit Product").
4. Modify the desired fields and click **"Save Product"**.

Expected outcome: a success toast *"Product updated successfully"* is shown and the card reflects the changes.

Deleting a product.

1. On the Manage Products page, locate the product card you want to delete.
2. Click the **trash icon** (delete button) on the product card.
3. A **confirmation dialog** appears: *"Are you absolutely sure? This action cannot be undone. This will permanently delete the product from your menu."*
4. Click **"Delete"** to confirm, or **"Cancel"** to abort.

Expected outcome: a success toast *"Product deleted successfully"* is shown and the product card is removed from the grid.

Note: If a product is currently **being purchased** by a customer (locked), the edit and delete buttons are disabled and a toast notification informs the restaurant owner. Products that are locked appear grayed out in the grid.

7.3.4 Managing Orders (Complete or Reject)

The **Manage Orders** page allows restaurants to handle incoming orders.

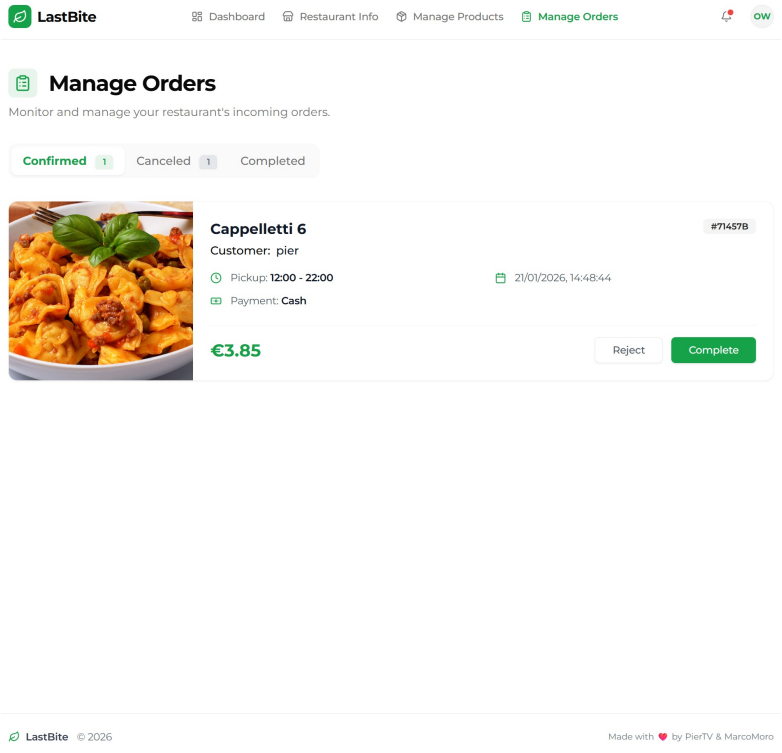

Figure 18: Restaurant order page.

1. Navigate to the **Manage Orders** page from the navigation bar.
2. Orders are organized into three tabs: **Confirmed**, **Canceled**, and **Completed**. Each tab shows a count badge.
3. Under the **Confirmed** tab, each order card displays: product name and image, customer username, pickup time window, purchase date, payment method, and total price.

Completing an order.

4. Click the **"Complete"** button on the order card.

Expected outcome: a success toast *"Order marked as completed"* is shown. The order moves from the *Confirmed* tab to the *Completed* tab. The customer receives a notification about the status change.

Rejecting an order.

4. Click the **"Reject"** button on the order card.
5. A **dialog** appears with the title "Reject Order" and the message: *"Are you sure you want to reject this order? You can optionally provide a reason for the customer."*
6. Optionally, type a **rejection reason** in the textarea (e.g., *"Sold out"*, *"Ingredient missing..."*).
7. Click **"Reject Order"** (destructive red button) to confirm, or **"Cancel"** to abort.

Expected outcome: a success toast *"Order rejected successfully"* is shown. The order moves to the *Canceled* tab. The customer receives a notification and can view the rejection reason (if provided) on their Orders page via the *"View Reason"* link.

8 Self-evaluation

8.1 Pier Costante Babini

I believe LastBite is a solid and well-structured project, one that I am genuinely proud of. The most technically challenging aspect was designing and implementing the distributed architecture from scratch: setting up the MongoDB replica set with an arbiter, the Redis Sentinel for high availability, the Nginx reverse proxy with load balancing across multiple backend instances, and ensuring that WebSocket state was correctly shared between nodes via the Redis adapter. On the backend, I designed and implemented the entire API layer: all REST endpoints for users, restaurants, products, orders, notifications, and reviews, including the Joi validation schemas, the JWT-based authentication system with HTTP-only cookies, and the middleware chain for role-based access control and resource ownership checks. I also implemented the chatbot service (Foody) with its intent classification, the review system and the geocoding integration via the OpenStreetMap Nominatim API.

On the frontend, I was responsible for the overall application architecture: project setup with Vue, Vite, Tailwind CSS, and shadcn-vue; the routing system with role-based guards; the Pinia stores for authentication and checkout state management; and the Axios API layer. I implemented the majority of the customer-facing pages (Login, Register, Home with search and map, Restaurant Info with product filtering, Checkout with the heartbeat-based session, Food Advisor chat, Orders with review system, and Profile with map-based location editing), the notification system with real-time WebSocket delivery, and a significant portion of the shared layout components (Navbar, Footer, PageHeader). I also carried out UI/UX refinements based on user testing sessions, such as redesigning the restaurant cards and adjusting product card interactions.

Regarding the infrastructure, I set up the Docker Compose environment, the Nginx load balancer configuration with WebSocket proxy support and the Redis distributed architecture (Sentinel + adapter). I also wrote the backend integration tests for users,

restaurants, orders, and the checkout lock service using Vitest, Supertest, and MongoDB Memory Server.

From a self-critical perspective, I recognize that the chatbot's intent set is still limited, and there is room for improvement in terms of frontend test coverage, which is currently absent. These are areas I would prioritize in a future iteration.

I believe Marco and I collaborated effectively throughout the project. We divided the work clearly, I focused primarily on the backend, infrastructure, and the customer-side frontend, while he concentrated on the restaurant-side frontend and cross-cutting refactoring and we were able to merge our contributions smoothly without significant conflicts.

8.2 Marco Morelli

I am satisfied with my contribution to LastBite, as I believe I played a key role in building several core features that are central to the restaurant experience and in improving the overall quality and consistency of the codebase.

On the frontend, I was primarily responsible for the restaurant-side interface. I designed and implemented the Dashboard page with its KPI cards and the interactive sales trend chart (using Chart.js and vue-chartjs), including the period selector and the recent orders panel. I built the Edit Restaurant page with the editable form and image management, and the Manage Products page with the full CRUD workflow: the product card grid, the add/edit dialog with image upload, and the delete confirmation dialog. I also implemented the Restaurant Order Page with the complete/reject workflow, including the rejection reason dialog.

Beyond the restaurant pages, I contributed significantly to cross-cutting frontend improvements. I implemented the responsive design across the entire application, ensuring a consistent experience on mobile, tablet, and desktop viewports. I introduced the password visibility toggle in the login, register, and profile forms, and enhanced the review system on the frontend by creating the component with rating aggregation and sortable review lists, which is used in both the Dashboard and the Restaurant Info page. I also worked on frontend validation using Zod, refactored several components to eliminate code duplication and improved image handling across the restaurant editing workflows.

On the backend, I contributed to the MongoDB replica set setup, configuring the replica set initialization and updating the Nginx configuration for the distributed environment. I improved the chatbot response handling and sanitization in the search controller and I wrote a substantial portion of the backend tests for notifications, products, reviews, and utility functions. I also created the seed script with realistic restaurant data, product images, and customer accounts for testing purposes.

Looking back, I think the main strength of the project lies in its polished user interface and the real-time interactivity provided by WebSockets, which make the application feel responsive and professional. A weakness I would highlight is the lack of advanced filtering capabilities (e.g., by cuisine type or dietary preferences) in the restaurant search, which would significantly enhance the customer experience. Additionally, while the product management interface is fully functional, adding drag-and-drop reordering and batch

operations would improve the restaurant owner experience.

I am very pleased with the collaboration between Pier and me. We maintained a clear separation of responsibilities while regularly syncing to ensure consistency, and the result is a cohesive and complete distributed application.

9 Future Works

This section discusses possible improvements, extensions, and optimizations that could be applied to the LastBite, as well as features that were considered but not implemented within the current scope of the project.

9.1 Real Payment Integration

The current checkout flow supports two payment methods – card and cash – but the card payment is **simulated**: the frontend performs client-side validation of the card number, expiration date, and CVC, but no actual payment processing occurs on the backend. The order is directly confirmed regardless of the payment method selected.

Proposed implementation. A real payment integration could be achieved by integrating a third-party payment gateway such as **Stripe** or **PayPal**.

9.2 Enhanced Food Advisor (Foody)

The current Food Advisor chatbot uses a two-stage architecture: an **intent classifier** (powered by the Groq API) categorizes the user's message into one of four intents.

Proposed improvements.

- **Conversation context**: pass the full chat history to the LLM to enable multi-turn conversations, follow-up questions, and clarification dialogs.
- **RAG (Retrieval-Augmented Generation)**: use vector embeddings of restaurant descriptions and product names to enable semantic search, allowing the chatbot to answer more nuanced queries by retrieving relevant context before generating responses.
- **Additional intents**: extend the intent classifier with new intents such as filtering by cuisine type, price range, or restaurant rating, and combine multiple filters in a single query.
- **Personalized recommendations**: leverage the user's order history and location to provide tailored suggestions.

9.3 Security Improvements

Several security enhancements could harden the platform:

- **Rate limiting**: the API currently has no rate-limiting mechanism. Implementing rate limiting (e.g., via an Express middleware such as `express-rate-limit` or at the Nginx reverse-proxy level) would protect against brute-force login attempts, abuse of the chatbot endpoint (which incurs external API costs), and denial-of-service attacks.
- **Account lockout**: implementing temporary account lockout after a configurable number of failed login attempts (e.g., 5 attempts within 10 minutes) would protect against credential stuffing attacks.

9.4 Frontend Improvements

- **Internationalization (i18n)**: the application is currently English-only. Integrating `vue-i18n` would allow the interface to be translated into multiple languages, broadening the user base. All user-facing strings are currently hardcoded in the Vue templates, so a migration to translation keys would be required.
- **Dark mode**: the UI uses a light-only design system. Adding a dark mode toggle (leveraging the existing Tailwind CSS `dark:` variants) would improve the user experience in low-light conditions and align with modern design expectations.