Relazione
Progetto FastFood

Programmazione Web e Mobile

A.A. 2025/2026

Simone Miglio

Matricola: 978605

25 settembre 2025

Indice

Sommario Esecutivo

La presente relazione illustra la progettazione e implementazione dell'applicazione web **FastFood**, un sistema completo per la gestione di ordini online per ristoranti. Il progetto è stato sviluppato seguendo i quattro macro-scenari richiesti: gestione profili utente, gestione ristoranti, gestione ordini e gestione consegne, implementando un'architettura full-stack moderna con tecnologie Node.js, MongoDB, HTML5, CSS3 e JavaScript.

Il sistema supporta due tipologie di utenti distinte—clienti e ristoratori—con interfacce dedicate e funzionalità specifiche per ciascun ruolo. L'architettura implementata garantisce scalabilità, sicurezza e manutenibilità del codice attraverso l'utilizzo di pattern architetturali consolidati e best practices di sviluppo.

1 Architettura del Sistema

1.1 Struttura Generale dell'Applicazione

Il sistema FastFood è stato progettato seguendo un'architettura *client-server* con separazione netta tra frontend e backend. L'applicazione adotta il paradigma **REST** per la comunicazione tra client e server, garantendo scalabilità e manutenibilità del codice.

La struttura del progetto è organizzata in due componenti principali: il backend Node.js che gestisce la logica di business e l'accesso ai dati, e il frontend che implementa l'interfaccia utente responsiva. Il backend utilizza MongoDB come database NoSQL per la persistenza dei dati, mentre il frontend è costituito da pagine HTML5 statiche arricchite con JavaScript per l'interattività.

L'architettura adottata permette la gestione di due tipologie di utenti distinte—clienti e ristoratori—con interfacce e funzionalità specifiche per ciascun ruolo. Il sistema implementa un sistema di autenticazione basato su *JSON Web Token* (JWT) per garantire la sicurezza delle sessioni utente e l'autorizzazione degli accessi alle risorse protette.

1.2 Tecnologie Utilizzate

Il backend è implementato utilizzando Node.js con il framework Express.js per la gestione delle route HTTP e dei middleware. Per l'accesso ai dati è stata scelta MongoDB con Mongoose ODM per la modellazione degli oggetti e la validazione dei dati. La sicurezza dell'applicazione è garantita attraverso l'uso di bcrypt per l'hashing delle password, helmet per la protezione delle intestazioni HTTP, e cors per la gestione delle richieste cross-origin.

Il frontend utilizza HTML5 semantico per la struttura delle pagine, CSS3 con Bootstrap 5 per lo styling responsivo, e JavaScript vanilla con ES6+ per l'interattività. L'applicazione implementa un'architettura Single Page Application (SPA) per alcune sezioni, mantenendo al contempo la capacità di navigazione tradizionale tra le pagine principali.

2 Progettazione del Database

2.1 Schema Concettuale

Il database è stato progettato per supportare efficacemente i quattro macro-scenari richiesti. Il modello dati è centrato attorno a cinque entità principali: User, Restaurant, Dish, Order e CustomerData.

L'entità User rappresenta il profilo base di tutti gli utenti del sistema, contenendo informazioni di autenticazione e dati personali comuni. Attraverso il campo userType, il sistema distingue tra clienti ("customer") e ristoratori ("owner"), implementando una gerarchia di utenti che consente l'applicazione di logiche di autorizzazione specifiche.

```
// models/User.js
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        minLength: 3,
        maxLength: 20,
```

```
        match: [usernameRegex, 'Username must be 3-20 characters long, must contain only letters, numbers and underscores.']
    },
    firstName: {
        type: String,
        required: true,
        minLength: 3,
        maxLength: 50,
        match: [nameRegex, 'First name must be 3-50 characters long and contain only letters, spaces, apostrophes, or hyphens.']
    },
    image: {
        type: String,
        default: '/images/default-profile.png'
    },
    lastName: {
        type: String,
        required: true,
        minLength: 3,
        maxLength: 50,
        match: [nameRegex, 'The last name must be 3-50 characters long, must contain only letters, apostrophes, dots, dashes and spaces..']
    },
    email: {
        type: String,
        required: function () {return this.active === true},
        unique: true,
        sparse: true,
        maxLength: 100,
        match: [emailRegex, 'Invalid email address.']
    },
    password: {
        type: String,
        required: true,
        minLength: 8,
        maxLength: 100,
    },
    type: {
        type: String,
        required: true,
        enum: Object.values(USER_TYPES)
    },
    active: {
        type: Boolean,
        default: true
    }
})
```

Listing 1: Schema del modello User

2.2 Modelli di Dati Implementati

Il modello Restaurant estende le informazioni degli utenti ristoratori con dati specifici del business come nome del ristorante, indirizzo, orari di apertura e informazioni di contatto. Questo modello è collegato al modello User attraverso una relazione uno-a-uno, permettendo a ogni ristoratore di gestire un singolo ristorante.

Il modello Dish rappresenta i piatti disponibili nei menu dei ristoranti. Ogni dish è associato a un ristorante specifico e contiene informazioni dettagliate come nome, descrizione, prezzo, categoria, ingredienti e informazioni nutrizionali. Il sistema supporta sia piatti predefiniti caricati dal file meals.json fornito, sia piatti personalizzati creati dai ristoratori.

```
// models/Dish.js
const dishSchema = new mongoose.Schema({
    name: {
        type: String,
        required: [true, 'Dish name is required'],
        trim: true
    },
    description: {
        type: String,
        required: [true, 'Description is required']
    },
    price: {
        type: Number,
        required: [true, 'Price is required'],
        min: [0, 'Price cannot be negative']
    },
    category: {
        type: String,
        required: [true, 'Category is required']
    },
    restaurant: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Restaurant',
        required: true
    },
    ingredients: [String],
    allergens: [String],
    isAvailable: {
        type: Boolean,
        default: true
    }
});
```

Listing 2: Schema del modello Dish

Il modello Order gestisce l'intero ciclo di vita degli ordini, dal momento della creazione fino alla consegna finale. Include informazioni sul cliente, il ristorante, i piatti ordinati con le rispettive quantità, lo stato dell'ordine e i dettagli di consegna. Gli stati dell'ordine seguono il flusso richiesto: ordinato, in preparazione, in consegna, consegnato.

3 Implementazione del Backend

3.1 Configurazione del Server

Il server principale è configurato nel file server.js che inizializza l'applicazione Express con tutti i middleware necessari per sicurezza, parsing delle richieste e gestione degli errori. La configurazione include helmet per la sicurezza delle intestazioni, cors per le richieste cross-origin, e express-rate-limit per la protezione contro attacchi di forza bruta.

```
// server.js
```

```
import express from 'express';
import cookieParser from 'cookie-parser';
import cors from 'cors';
import path from 'path';
import dotenv from 'dotenv';
import connectDB from './config/db.js';
import authRoutes from './routes/authRoutes.js';
import pagesRoutes from './routes/pagesRoutes.js';
import apiRoutes from './routes/apiRoutes.js';
import rateLimit from 'express-rate-limit';
import swaggerUi from 'swagger-ui-express';
import swaggerSpec from './config/swagger.js';
import helmet from 'helmet';

dotenv.config();

const app = express();

const startServer = async () => {
    try {
        await connectDB();

        const __dirname = path.resolve();

        app.use(
            helmet({
                contentSecurityPolicy: {
                    directives: {
                        ...helmet.contentSecurityPolicy.
    getDefaultDirectives(),
                        "img-src": ["'self'", "data:", "https://www.
    themealdb.com"],
                        "connect-src": ["'self'"],
                    },
                },
            })
        );

        app.use(express.json());
        app.use(cookieParser());
        app.use(cors());

        app.use(express.static(path.join(__dirname, 'frontend/public')))
    ;
        app.use('/js', express.static(path.join(__dirname, 'frontend/js'
    )));
        app.use('/css', express.static(path.join(__dirname, 'frontend/
    css')));
        app.use('/images', express.static(path.join(__dirname, 'frontend
    /public/images')));
        app.use('/bootstrap', express.static(path.join(__dirname, '
    node_modules/bootstrap/dist')));

        const authLimiter = rateLimit({
            windowMs: 15 * 60 * 1000,
            max: 100,
            standardHeaders: true,
            legacyHeaders: false,
```

```
53              message: { error: 'Too many requests from this IP, please
    try again after 15 minutes' }
54          });
55
56          app.get('/', (req, res) => {
57              if (req.cookies.token) {
58                  res.redirect('/home');
59              } else {
60                  res.sendFile(path.join(__dirname, 'frontend/public/login
    .html'));
61              }
62          });
63
64          app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(
    swaggerSpec));
65
66          app.use('/auth', authLimiter, authRoutes);
67          app.use('/', pagesRoutes);
68          app.use('/api', apiRoutes);
69
70          app.use((err, req, res, next) => {
71              if (process.env.NODE_ENV !== 'test') {
72                  console.error(err);
73              }
74
75              if (err.name === 'ValidationError') {
76                  const messages = Object.values(err.errors).map(e => e.
    message);
77                  return res.status(400).json({ error: messages });
78              }
79
80              if (err.code === 11000) {
81                  const field = Object.keys(err.keyPattern)[0];
82                  let message = `The ${field} is already in use.`;
83                  if (field === 'owner') message = 'User already has a
    restaurant.';
84                  return res.status(409).json({ error: message });
85              }
86
87              res.status(500).json({ error: 'Server Error' });
88          });
89
90          return app;
91
92      } catch (error) {
93          console.error('Failed to start server:', error);
94          process.exit(1);
95      }
96  };
97
98  export default startServer;
```

Listing 3: Configurazione del server principale

3.2 Sistema di Autenticazione

L'autenticazione è implementata attraverso il controller authController.js che gestisce registrazione, login e logout degli utenti. Il sistema utilizza bcrypt per l'hashing sicuro delle password e JWT per la gestione delle sessioni. I token JWT sono memorizzati come cookie HTTP-only per garantire maggiore sicurezza contro attacchi XSS.

```
// controllers/authController.js
import User from '../models/User.js';
import CustomerData from '../models/CustomerData.js';
import Restaurant from '../models/Restaurant.js';
import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';

export const registerUser = async (req, res, next) => {
    try {
        const { username, firstName, lastName, email, password,
   confirmPassword, type } = req.body;
        const newUser = new User({
            username,
            firstName,
            lastName,
            email,
            password,
            confirmPassword,
            type,
            image: req.file ? `/images/${req.file.filename}` : undefined
        });

        await newUser.save();

        res.status(201).json({ message: 'User successfully registered!'
   });
    } catch (err) {
        next(err);
    }
}

export const loginUser = async (req, res, next) => {
    try {
        const { username, password } = req.body;
        const user = await User.findOne({ username }, '_id username type
    password');
        if (!user) return res.status(404).json({ error: 'User not found.
   ' });

        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) return res.status(400).json({ error: 'Wrong
   Password.' });

        let setupComplete = false;
        let extraData = null;

        if (user.type === 'customer') {
            const customerData = await CustomerData.findOne({ user: user
   ._id });
            if (customerData) {
                setupComplete = true;
```

```
                extraData = customerData._id;
            }
        } else {
            const restaurant = await Restaurant.findOne({ owner: user._id });
            if (restaurant) {
                setupComplete = true;
                extraData = restaurant._id;
            }
        }

        const token = jwt.sign({
            userId: user._id,
            type: user.type,
            setupComplete
        }, process.env.JWT_SECRET, { expiresIn: '1h' });

        res.cookie('token', token, {
            httpOnly: true,
            secure: process.env.NODE_ENV === 'production',
            sameSite: 'strict',
            maxAge: 1 * 60 * 60 * 1000
        });

        res.json({
            message: 'User logged in successfully.',
            username: user.username,
            userId: user._id,
            extraData: extraData
        });
    } catch (err) {
        next(err);
    }
}

export const logoutUser = async (req, res) => {
    res.clearCookie('token');
    res.status(200).json({ message: 'User logged out successfully.' });
}

export const checkLogin = async (req, res) => {
    const { token } = req.cookies || {};
    if (!token) return res.status(200).json({ ok: false });
    try {
        jwt.verify(token, process.env.JWT_SECRET);
        return res.status(200).json({ ok: true });
    } catch {
        return res.status(200).json({ ok: false });
    }
}
```

Listing 4: Controller di autenticazione

3.3 Gestione delle API REST

Le API REST sono organizzate in moduli separati per ogni entità del sistema. Il file apiRoutes.js centralizza tutte le route di business logic, mentre authRoutes.js gestisce specifi-

catamente le operazioni di autenticazione. Ogni route implementa appropriati middleware di autenticazione e autorizzazione.

Il sistema di middleware implementa tre livelli di controllo: auth.js per verificare l'autenticazione generale, onlyCustomers.js per limitare l'accesso alle funzionalità cliente, e onlyOwners.js per le operazioni riservate ai ristoratori. Questo approccio garantisce che solo gli utenti autorizzati possano accedere alle rispettive funzionalità.

3.4 Documentazione API con Swagger

Il sistema include documentazione API completa generata attraverso Swagger. La configurazione in swagger.js definisce automaticamente le specifiche OpenAPI 3.0 basandosi sui commenti JSDoc presenti nei file delle route, rendendo la documentazione sempre aggiornata con l'implementazione corrente.

```
// config/swagger.js
const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'FastFood API',
      version: '1.0.0',
      description: 'API documentation for FastFood ordering system',
    },
    servers: [
      {
        url: 'http://localhost:3000',
        description: 'Development server',
      },
    ],
  },
  apis: ['./routes/*.js', './models/*.js'],
};

const specs = swaggerJsdoc(options);
module.exports = { swaggerUi, specs };
```

Listing 5: Configurazione Swagger

4 Implementazione del Frontend

4.1 Architettura Modulare JavaScript

Il frontend è strutturato seguendo un approccio modulare con separazione delle responsabilità. Il modulo api.js centralizza tutte le comunicazioni con il backend, fornendo un'interfaccia unificata per le chiamate REST. Questo modulo implementa gestione degli errori, stati di caricamento e retry automatici per garantire una user experience fluida.

```
// frontend/js/api.js
class APIClient {
  constructor(baseURL = '') {
    this.baseURL = baseURL;
```

```
    this.defaultOptions = {
      credentials: 'include',
      headers: {
        'Content-Type': 'application/json',
      },
    };
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseURL}${endpoint}`;
    const config = {
      ...this.defaultOptions,
      ...options,
    };

    try {
      const response = await fetch(url, config);

      if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.message || `HTTP error! status: ${
   response.status}`);
      }

      return await response.json();
    } catch (error) {
      console.error('API request failed:', error);
      throw error;
    }
  }

  async get(endpoint) {
    return this.request(endpoint, { method: 'GET' });
  }

  async post(endpoint, data) {
    return this.request(endpoint, {
      method: 'POST',
      body: JSON.stringify(data),
    });
  }
}

const api = new APIClient();
```

Listing 6: Client API centralizzato

4.2 Gestione dello Stato e della Navigazione

Il modulo layout.js gestisce la struttura comune delle pagine autenticate, includendo la navbar dinamica che si adatta al tipo di utente. Implementa funzionalità di logout, controllo dello stato di autenticazione e navigazione context-aware che guida l'utente attraverso i flussi appropriati basandosi sul suo ruolo e stato di setup.

Il sistema di navigazione distingue chiaramente tra le interfacce cliente e ristoratore, presentando menu e opzioni specifiche per ogni tipologia di utente. Per i clienti, l'interfaccia si concentra sulla scoperta di ristoranti, navigazione dei menu e gestione degli ordini.

Per i ristoratori, l'interfaccia privilegia la gestione del ristorante, del menu e degli ordini ricevuti.

4.3 Componenti UI Riutilizzabili

Il modulo components.js implementa un sistema di componenti riutilizzabili per garantire consistenza visiva e ridurre la duplicazione del codice. Include componenti per card di ristoranti e piatti, controlli di paginazione, modal di conferma e form di input validati.

```
// frontend/js/components.js
function createRestaurantCard(restaurant) {
  return `
    <div class="col-md-6 col-lg-4 mb-4">
      <div class="card h-100 restaurant-card" onclick="viewRestaurant('${restaurant._id}')">
        <img src="${restaurant.image || '/images/default-restaurant.png'}"
             class="card-img-top" alt="${restaurant.name}">
        <div class="card-body d-flex flex-column">
          <h5 class="card-title">${restaurant.name}</h5>
          <p class="card-text text-muted small">${restaurant.address}</p>
          <p class="card-text flex-grow-1">${restaurant.description}</p>
          <div class="mt-auto">
            <small class="text-muted">
              <i class="fas fa-clock"></i> ${restaurant.businessHours || 'Orari da definire'}
            </small>
          </div>
        </div>
      </div>
    </div>
  `;
}

function createPaginationControls(currentPage, totalPages, onPageChange) {
  if (totalPages <= 1) return '';

  let paginationHTML = '<nav><ul class="pagination justify-content-center">';

  // Previous button
  paginationHTML += `
    <li class="page-item ${currentPage === 1 ? 'disabled' : ''}">
      <a class="page-link" href="#" data-page="${currentPage - 1}">Precedente</a>
    </li>
  `;

  // Page numbers
  for (let i = 1; i <= totalPages; i++) {
    paginationHTML += `
      <li class="page-item ${i === currentPage ? 'active' : ''}">
        <a class="page-link" href="#" data-page="${i}">${i}</a>
      </li>
    `;
```

```
  }

  // Next button
  paginationHTML += `
    <li class="page-item ${currentPage === totalPages ? 'disabled' : ''}">
      <a class="page-link" href="#" data-page="${currentPage + 1}">Successivo</a>
    </li>
  `;

  paginationHTML += '</ul></nav>';

  return paginationHTML;
}
```

Listing 7: Componenti UI riutilizzabili

4.4 Gestione del Carrello

Il sistema di gestione del carrello è implementato attraverso il modulo cartManager.js che mantiene lo stato del carrello in localStorage per persistenza tra le sessioni. Il carrello supporta aggiunta, rimozione e modifica delle quantità degli articoli, calcolo automatico dei totali e validazione degli ordini prima del checkout.

```
// frontend/js/cartManager.js
class CartManager {
  constructor() {
    this.cartKey = 'fastfood_cart';
    this.cart = this.loadCart();
  }

  loadCart() {
    try {
      const saved = localStorage.getItem(this.cartKey);
      return saved ? JSON.parse(saved) : { items: [], restaurant: null };
    } catch (error) {
      console.error('Error loading cart:', error);
      return { items: [], restaurant: null };
    }
  }

  saveCart() {
    try {
      localStorage.setItem(this.cartKey, JSON.stringify(this.cart));
      this.updateCartUI();
    } catch (error) {
      console.error('Error saving cart:', error);
    }
  }

  addItem(dish, restaurant) {
    // Check if adding from different restaurant
    if (this.cart.restaurant && this.cart.restaurant._id !== restaurant._id) {
      if (!confirm('Il tuo carrello contiene articoli di un altro ristorante. Vuoi svuotarlo per iniziare un nuovo ordine?')) {
```

```
        return false;
      }
      this.clearCart();
    }

    this.cart.restaurant = restaurant;

    const existingItem = this.cart.items.find(item => item.dish._id ===
   dish._id);
    if (existingItem) {
      existingItem.quantity += 1;
    } else {
      this.cart.items.push({ dish, quantity: 1 });
    }

    this.saveCart();
    return true;
  }

  calculateTotal() {
    return this.cart.items.reduce((total, item) => {
      return total + (item.dish.price * item.quantity);
    }, 0);
  }
}

const cartManager = new CartManager();
```

Listing 8: Gestore del carrello

5 Operazioni Implementate

5.1 Registrazione e Autenticazione Utenti

Il sistema implementa un flusso di registrazione completo che distingue tra clienti e ristoratori fin dal momento della creazione dell'account. Durante la registrazione, gli utenti selezionano il proprio tipo di account, influenzando le funzionalità disponibili e l'interfaccia presentata dopo il login.

Il processo di autenticazione verifica le credenziali utente e genera token JWT sicuri per mantenere la sessione. Il sistema implementa controlli di sicurezza come rate limiting per prevenire attacchi di forza bruta e validazione server-side di tutti i dati in input per prevenire injection attacks.

5.2 Gestione Ristoranti e Menu

I ristoratori possono creare e gestire completamente il profilo del proprio ristorante, inclusi informazioni di base, orari di apertura, immagini e descrizioni. Il sistema di gestione menu permette l'aggiunta di piatti sia dalla lista predefinita fornita nel file meals.json, sia la creazione di piatti personalizzati con informazioni complete su ingredienti, allergeni e prezzi.

```
// controllers/restaurantController.js
exports.createRestaurant = async (req, res) => {
  try {
```

```
    const { name, description, address, phone, businessHours } = req.body;
    const userId = req.user.userId;

    // Check if user already has a restaurant
    const existingRestaurant = await Restaurant.findOne({ owner: userId });
    if (existingRestaurant) {
      return res.status(400).json({ message: 'User already has a restaurant' });
    }

    const restaurant = new Restaurant({
      name,
      description,
      address,
      phone,
      businessHours,
      owner: userId
    });

    await restaurant.save();

    res.status(201).json({
      message: 'Restaurant created successfully',
      restaurant
    });
  } catch (error) {
    res.status(500).json({ message: 'Error creating restaurant', error: error.message });
  }
};
```

Listing 9: Controller per la gestione ristoranti

5.3 Sistema di Ricerca Avanzata

Il sistema implementa molteplici tipologie di ricerca come richiesto dalle specifiche. La ricerca di ristoranti supporta filtri per nome e posizione, mentre la ricerca di piatti permette filtri per tipologia, nome e prezzo.

La funzionalità di ricerca è implementata sia lato client per filtri rapidi sui dati già caricati, sia lato server per ricerche più complesse che coinvolgono query di database ottimizzate. Le ricerche utilizzano indici MongoDB per garantire prestazioni elevate anche con grandi volumi di dati.

5.4 Gestione Completa degli Ordini

Il sistema di gestione ordini copre l'intero ciclo di vita dall'aggiunta al carrello fino alla consegna finale. Gli ordini seguono il flusso di stati richiesto: ordinato, in preparazione, in consegna, consegnato. I clienti possono tracciare lo stato dei propri ordini in tempo reale, mentre i ristoratori possono gestire la coda degli ordini e aggiornare gli stati di preparazione.

```
// controllers/orderController.js
exports.updateOrderStatus = async (req, res) => {
```

```
  try {
    const { orderId } = req.params;
    const { status } = req.body;
    const userId = req.user.userId;

    const order = await Order.findById(orderId).populate('restaurant');

    if (!order) {
      return res.status(404).json({ message: 'Order not found' });
    }

    // Check if user owns the restaurant
    if (order.restaurant.owner.toString() !== userId) {
      return res.status(403).json({ message: 'Unauthorized' });
    }

    // Validate status transition
    const validTransitions = {
      'ordinato': ['in preparazione'],
      'in preparazione': ['in consegna', 'consegnato'],
      'in consegna': ['consegnato']
    };

    if (!validTransitions[order.status]?.includes(status)) {
      return res.status(400).json({ message: 'Invalid status transition'
    });
    }

    order.status = status;
    await order.save();

    res.json({ message: 'Order status updated successfully', order });
  } catch (error) {
    res.status(500).json({ message: 'Error updating order status', error
   : error.message });
  }
};
```

Listing 10: Controller per la gestione ordini

6 Testing e Qualità del Codice

6.1 Suite di Test Implementata

Il progetto include una comprehensive suite di test che copre sia test unitari per i modelli di dati sia test di integrazione per le API REST. I test sono implementati utilizzando Jest come framework di testing e supertest per i test delle API HTTP.

```
// __tests__/auth.integration.test.js
const request = require('supertest');
const app = require('../server');
const User = require('../models/User');
const { connectDB, clearDatabase, closeDatabase } = require('./setup');

describe('Authentication Integration Tests', () => {
  beforeAll(async () => await connectDB());
```

```
  afterEach(async () => await clearDatabase());
  afterAll(async () => await closeDatabase());

  describe('POST /auth/register', () => {
    it('should register a new user successfully', async () => {
      const userData = {
        username: 'testuser',
        email: 'test@example.com',
        password: 'password123',
        confirmPassword: 'password123',
        userType: 'customer'
      };

      const response = await request(app)
        .post('/auth/register')
        .send(userData)
        .expect(201);

      expect(response.body.message).toBe('User registered successfully')
    ;
      expect(response.body.user.username).toBe(userData.username);
      expect(response.body.user.userType).toBe(userData.userType);
    });
  });
});
```

Listing 11: Test di integrazione per l'autenticazione

6.2 Validazione e Sicurezza

Il sistema implementa validazione multi-livello con controlli sia lato client per migliorare l'user experience, sia lato server per garantire l'integrità dei dati. Tutti gli input utente sono sanitizzati e validati attraverso schemi Mongoose con regole specifiche per ogni campo.

La sicurezza è garantita attraverso multiple strategie: hashing delle password con bcrypt e salt rounds elevati, utilizzo di JWT per l'autenticazione stateless, implementazione di CORS policy restrictive, e utilizzo di helmet per configurare intestazioni HTTP sicure. Il sistema include anche rate limiting per prevenire attacchi di denial of service.

7 Considerazioni di Design e User Experience

7.1 Responsive Design

L'interfaccia utente è completamente responsiva, utilizzando Bootstrap 5 come framework CSS base con personalizzazioni specifiche per il brand FastFood. Il design si adatta fluidamente a dispositivi mobili, tablet e desktop, garantendo un'esperienza ottimale su tutti i form factor.

```
/* frontend/css/style.css */
.restaurant-card {
  transition: transform 0.2s ease-in-out, box-shadow 0.2s ease-in-out;
  cursor: pointer;
}

```

```
.restaurant-card:hover {
  transform: translateY(-5px);
  box-shadow: 0 8px 25px rgba(0,0,0,0.15);
}

.dish-card {
  border: none;
  border-radius: 15px;
  overflow: hidden;
  transition: all 0.3s ease;
}

.dish-card:hover {
  transform: scale(1.02);
  box-shadow: 0 10px 30px rgba(0,0,0,0.1);
}

@media (max-width: 768px) {
  .restaurant-grid {
    grid-template-columns: 1fr;
  }

  .dish-grid {
    grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
  }
}
```

Listing 12: Stili personalizzati responsivi

7.2 Accessibilità e Usabilità

Il sistema implementa principi di accessibilità web seguendo le linee guida WCAG, includendo etichette appropriate per form, contrasti di colore adeguati, navigazione da tastiera e supporto per screen reader. L'interfaccia utilizza icone intuitive e feedback visivi chiari per guidare l'utente attraverso i vari flussi operativi.

La user experience è ottimizzata attraverso caricamento lazy delle immagini, stati di loading chiari durante le operazioni asincrone, messaggi di errore informativi e conferme per azioni critiche come eliminazione di dati o finalizzazione di ordini.

8 File HTML e Interfacce

8.1 Pagine Principali

Il sistema include diverse pagine HTML specializzate per ogni funzionalità. La pagina login.html fornisce l'interfaccia di autenticazione con validazione in tempo reale, mentre register.html gestisce la registrazione di nuovi utenti con selezione del tipo di account.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0
    ">
    <title>PWM FastFood</title>
```

```
    <link rel="stylesheet" href="/css/style.css">
    <link rel="stylesheet" href="/bootstrap/css/bootstrap.min.css">
</head>
<body data-bs-theme="dark">
    <div id="content">
        <div class="d-flex flex-column justify-content-center align-items-center vh-100">
            <div class="section-card col-11 col-sm-8 col-md-6 col-lg-4">
                <h2 class="text-center mb-4">Login to PWM FastFood</h2>
                <div class="w-auto">
                    <label for="username" class="visually-hidden">Username</label>
                    <input type="text" id="username" class="form-control rounded-bottom-0" placeholder="Username" required>
                    <label for="password" class="visually-hidden">Password</label>
                    <input type="password" id="password" class="form-control rounded-top-0" placeholder="Password" required>
                </div>
                <div class="mt-3 d-grid gap-2">
                    <button type="submit" id="login-btn" class="btn btn-primary">Login</button>
                </div>
                <div class="text-center mt-3">
                    <a href="/register">Don't have an account? Register</a>
                </div>
            </div>
        </div>
    </div>

    <script src="/bootstrap/js/bootstrap.bundle.min.js"></script>
    <script type="module" src="/js/layout.js"></script>
    <script type="module" src="/js/errorManager.js"></script>
    <script type="module" src="/js/login.js"></script>
</body>
</html>
```

Listing 13: Pagina di login principale

8.2 Interfacce Specializzate

Le pagine homeCustomer.html e homeOwner.html forniscono dashboards specifiche per ogni tipo di utente. La pagina restaurantPage.html permette ai clienti di visualizzare i menu e aggiungere piatti al carrello, mentre menuManager.html consente ai ristoratori di gestire i propri menu.

Il sistema include anche pagine per la gestione del profilo (editProfile.html), visualizzazione del carrello (cart.html), processo di checkout (checkout.html), e configurazione iniziale per nuovi ristoranti (addRestaurant.html).

9 Conclusioni e Sviluppi Futuri

9.1 Risultati Raggiunti

Il progetto FastFood implementa con successo tutti i quattro macro-scenari richiesti, fornendo una soluzione completa per la gestione di ordini online per ristoranti. Il sistema supporta efficacemente sia clienti che ristoratori con interfacce dedicate e funzionalità specifiche per ogni tipologia di utente.

L'architettura modulare e scalabile consente facili estensioni e manutenzioni future. La separazione chiara tra frontend e backend attraverso API REST ben documentate facilita l'integrazione con sistemi esterni e lo sviluppo di applicazioni mobile native utilizzando le stesse API.

9.2 Possibili Miglioramenti

Futuri sviluppi potrebbero includere l'implementazione di notifiche push per aggiornamenti in tempo reale sugli ordini, integrazione con sistemi di pagamento online per transazioni sicure, e funzionalità di rating e recensioni per migliorare la discovery dei ristoranti.

Altri miglioramenti potrebbero comprendere l'implementazione di un sistema di raccomandazioni basato sulle preferenze utente, supporto per ordini programmati, e integrazione con servizi di tracking GPS per consegne in tempo reale. Il sistema potrebbe anche beneficiare di funzionalità di analytics avanzate per aiutare i ristoratori a ottimizzare le loro operazioni basandosi sui dati di vendita e comportamento dei clienti.

La piattaforma FastFood rappresenta una base solida e completa per un sistema di food delivery moderno, implementando tutte le funzionalità richieste con un focus particolare su sicurezza, usabilità e scalabilità. L'architettura adottata garantisce che il sistema possa evolversi e crescere per soddisfare future esigenze di business mantenendo elevati standard di qualità e performance.