Food Ordering Application: A Step-by-Step Development Guide

This guide will walk you through the process of creating a full-stack food ordering application.

Core Technologies:

- **Backend:** Java 11/17+ with Spring Boot (for REST APIs, Security, etc.)
- **Database:** MongoDB (NoSQL document database)
- **Frontend:** React (JavaScript library for UI)
- **Build Tools:** Maven or Gradle (for Java), npm or yarn (for React)
- **Version Control:** Git

Phase 1: Project Setup & Planning

Step 1: Environment Setup

- **Java Development Kit (JDK):** Install JDK 11 or 17 (or newer). Set up JAVA_HOME.
- **IDE for Java:** IntelliJ IDEA (recommended for Spring Boot) or Eclipse.
- **Maven/Gradle:** Ensure it's installed and configured (usually comes with IDEs or can be installed separately).
- **MongoDB:**
  - Install MongoDB Community Server locally or use a cloud service like MongoDB Atlas (offers a free tier).
  - Install MongoDB Compass or Robo 3T (Studio 3T) for easier database management and visualization.
- **Node.js and npm/yarn:** Install Node.js (which includes npm). Yarn is an alternative package manager you can install if preferred.
- **React Developer Tools:** Install the browser extension for your preferred browser (Chrome, Firefox) for easier debugging of React applications.
- **Git:** Install Git for version control.

Step 2: Project Structure & Initialization

- **Backend (Spring Boot):**
  - Use **Spring Initializr** (start.spring.io) to generate your project skeleton.
  - **Dependencies to include:**
    - Spring Web: For building RESTful APIs.
    - Spring Data MongoDB: For MongoDB integration.
    - Spring Security: For handling authentication and authorization.
    - Lombok: To reduce boilerplate code (e.g., getters, setters, constructors). (Optional but highly recommended)
    - Spring Boot DevTools: For automatic application restarts/reloads during

        development. (Optional)
        - Validation: For request payload validation (Java Bean Validation API).
    - Choose your build tool (Maven or Gradle).
    - Define a clear package structure, for example:
        - com.yourdomain.foodorderingapp.controller
        - com.yourdomain.foodorderingapp.service
        - com.yourdomain.foodorderingapp.repository
        - com.yourdomain.foodorderingapp.model (or entity)
        - com.yourdomain.foodorderingapp.dto (Data Transfer Objects)
        - com.yourdomain.foodorderingapp.config
        - com.yourdomain.foodorderingapp.exception
        - com.yourdomain.foodorderingapp.security
- **Frontend (React):**
    - Open your terminal or command prompt.
    - Use Create React App (CRA) or Vite to initialize your React project:
      ```
      # Using Create React App
      npx create-react-app food-ordering-frontend
      cd food-ordering-frontend

      # OR Using Vite (recommended for faster setup and builds)
      npm create vite@latest food-ordering-frontend -- --template react
      cd food-ordering-frontend
      npm install # or yarn
      ```
    - Establish a sensible folder structure, for example:
        - src/components (reusable UI components)
        - src/pages (page-level components)
        - src/services (API call functions)
        - src/contexts or src/store (for state management)
        - src/assets (images, fonts, etc.)
        - src/utils (helper functions)
        - src/routes (routing configuration)
        - src/hooks (custom React hooks)
- **Version Control:**
    - Initialize a Git repository in your main project directory (which might contain backend and frontend subdirectories).
    - Make initial commits for both backend and frontend projects.

Step 3: Define Core Features & User Stories
Outline the essential functionalities. Examples:

- **User Authentication:**
  - As a user, I want to register for an account.
  - As a user, I want to log in to my account.
  - As a user, I want to log out.
  - As a user, I want to manage my profile.
- **Restaurant Browsing & Searching:**
  - As a user, I want to view a list of available restaurants.
  - As a user, I want to search for restaurants by name, cuisine, or location.
  - As a user, I want to view details of a specific restaurant (menu, address, hours, reviews).
- **Menu Interaction:**
  - As a user, I want to view the menu items of a selected restaurant.
  - As a user, I want to see details of a menu item (description, price, image).
- **Ordering System:**
  - As a user, I want to add menu items to my cart.
  - As a user, I want to view and modify my cart (change quantities, remove items).
  - As a user, I want to proceed to checkout.
  - As a user, I want to place an order.
- **Order Management:**
  - As a user, I want to view my order history.
  - As a user, I want to see the status of my current order (e.g., "Pending", "Confirmed", "Preparing", "Out for Delivery", "Delivered").
- **Admin/Restaurant Owner Features (Phase 2 or later):**
  - As an admin, I want to add/update/delete restaurants.
  - As a restaurant owner, I want to add/update/delete menu items for my restaurant.
  - As a restaurant owner, I want to view and manage incoming orders for my restaurant.

Phase 2: Database Design (MongoDB)

Step 4: Define MongoDB Schemas (Collections)
MongoDB is schema-less, but for application development, you should have a clear structure in mind.

- **users Collection:**
  - _id: ObjectId (Primary Key, automatically generated)
  - username: String (unique, indexed)
  - email: String (unique, indexed)
  - password: String (hashed)

    - roles: Array of Strings (e.g., ["ROLE_USER", "ROLE_ADMIN", "ROLE_RESTAURANT_OWNER"])
    - firstName: String
    - lastName: String
    - addresses: Array of Objects (e.g., { street: String, city: String, state: String, zipCode: String, country: String, type: String })
    - phoneNumber: String
    - createdAt: ISODate
    - updatedAt: ISODate
- **restaurants Collection:**
    - _id: ObjectId
    - name: String (indexed)
    - description: String
    - cuisineTypes: Array of Strings (indexed)
    - address: Object (street, city, state, zipCode, country)
    - location: Object (GeoJSON for geospatial queries, e.g., { type: "Point", coordinates: [longitude, latitude] })
    - phoneNumber: String
    - openingHours: String or Object (e.g., { mon: "9am-10pm", ... })
    - imageUrl: String
    - ownerId: ObjectId (references users collection, if applicable)
    - averageRating: Number (default: 0, calculated from reviews)
    - isActive: Boolean (default: true)
    - createdAt: ISODate
    - updatedAt: ISODate
- **menuItems Collection:**
    - _id: ObjectId
    - restaurantId: ObjectId (references restaurants collection, indexed)
    - name: String
    - description: String
    - price: Number
    - category: String (e.g., "Appetizer", "Main Course", "Dessert", "Beverage", indexed)
    - imageUrl: String
    - isAvailable: Boolean (default: true)
    - tags: Array of Strings (e.g., "Spicy", "Vegetarian")
    - createdAt: ISODate
    - updatedAt: ISODate
- **orders Collection:**

    - _id: ObjectId
    - userId: ObjectId (references users collection, indexed)
    - restaurantId: ObjectId (references restaurants collection, indexed)
    - items: Array of Objects:
        - menuItemId: ObjectId (references menuItems collection)
        - name: String (denormalized for quick display)
        - quantity: Number
        - pricePerItem: Number (at the time of order)
        - totalPriceForItem: Number
    - totalAmount: Number
    - orderStatus: String (e.g., "PENDING_PAYMENT", "RECEIVED", "PREPARING", "READY_FOR_PICKUP", "OUT_FOR_DELIVERY", "DELIVERED", "CANCELLED", indexed)
    - deliveryAddress: Object (copied from user's selected address at time of order)
    - paymentDetails: Object (e.g., { paymentMethod: String, transactionId: String, paymentStatus: String })
    - specialInstructions: String
    - createdAt: ISODate
    - updatedAt: ISODate
    - estimatedDeliveryTime: ISODate (optional)
- **reviews Collection (Optional):**
    - _id: ObjectId
    - userId: ObjectId (references users collection)
    - restaurantId: ObjectId (references restaurants collection, indexed)
    - menuItemId: ObjectId (optional, if reviewing a specific item)
    - rating: Number (1-5)
    - comment: String
    - createdAt: ISODate

Phase 3: Backend Development (Java - Spring Boot)

Step 5: Setup Spring Data MongoDB

- Configure MongoDB connection details in src/main/resources/application.properties or application.yml:
  ```
  # application.properties
  spring.application.name=food-ordering-app
  server.port=8080

  spring.data.mongodb.uri=mongodb://localhost:27017/food_ordering_db
  ```

```
# For MongoDB Atlas, use the connection string provided by Atlas:
#
spring.data.mongodb.uri=mongodb+srv://<username>:<password>@<cluster-uri>/
food_ordering_db?retryWrites=true&w=majority
```

- Create Java model classes (POJOs) in your model package, annotated with @Document from Spring Data MongoDB. Use annotations like @Id, @Field, @Indexed.

```
// Example: User.java
package com.yourdomain.foodorderingapp.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import java.util.Set;
// Add Lombok annotations if using
import lombok.Data; // or @Getter, @Setter, @NoArgsConstructor,
@AllArgsConstructor

@Data
@Document(collection = "users")
public class User {
   @Id
   private String id; // String for MongoDB ObjectId
   private String username;
   private String email;
   private String password;
   private Set<String> roles;
   // ... other fields, getters, setters
}
```

Create similar model classes for Restaurant, MenuItem, Order, etc.

Step 6: Create Repositories

- Create interfaces in your repository package that extend MongoRepository<EntityType, IDType>. Spring Data will automatically provide implementations for basic CRUD operations and allow you to define custom query methods.

```
// Example: UserRepository.java
package com.yourdomain.foodorderingapp.repository;
```

```
import com.yourdomain.foodorderingapp.model.User;
import org.springframework.data.mongodb.repository.MongoRepository;
import java.util.Optional;

public interface UserRepository extends MongoRepository<User, String> {
    Optional<User> findByUsername(String username);
    Optional<User> findByEmail(String email);
    Boolean existsByUsername(String username);
    Boolean existsByEmail(String email);
}
```

Step 7: Implement Services (Business Logic)

- Create service classes (annotated with @Service) in your service package to encapsulate business logic.
- Inject repositories into services using @Autowired or constructor injection.

```
// Example: UserServiceImpl.java
package com.yourdomain.foodorderingapp.service;

import com.yourdomain.foodorderingapp.model.User;
import com.yourdomain.foodorderingapp.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder; // Will
be added later
import org.springframework.stereotype.Service;
import java.util.HashSet; // For roles
import java.util.Set;

public interface UserService { // Define an interface first
    User registerUser(User user, Set<String> roles);
    Optional<User> findByUsername(String username);
    // ... other methods
}

@Service
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder; // Will be injected later
    @Autowired
    public UserServiceImpl(UserRepository userRepository, PasswordEncoder
passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public User registerUser(User user, Set<String> strRoles) {
        if (userRepository.existsByUsername(user.getUsername())) {
            throw new RuntimeException("Error: Username is already taken!"); //
Replace with custom exception
        }
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new RuntimeException("Error: Email is already in use!"); // Replace
with custom exception
        }
        user.setPassword(passwordEncoder.encode(user.getPassword()));

        Set<String> roles = new HashSet<>();
        if (strRoles == null || strRoles.isEmpty()) {
             roles.add("ROLE_USER"); // Default role
        } else {
            // Logic to assign roles based on strRoles (e.g. "admin", "user")
            // For now, let's assume strRoles contains valid role names like
"ROLE_ADMIN"
            roles.addAll(strRoles);
        }
        user.setRoles(roles);
        return userRepository.save(user);
    }

    @Override
    public Optional<User> findByUsername(String username) {
        return userRepository.findByUsername(username);
    }
    // ...
}
```

Step 8: Implement REST Controllers (@RestController)

- Define API endpoints in your controller package.
- Use annotations like @GetMapping, @PostMapping, @PutMapping, @DeleteMapping.
- Inject services into controllers.
- Use **Data Transfer Objects (DTOs)** for request and response payloads to decouple your API from your internal domain models and for validation.

```
// Example: AuthController.java (for registration and login)
package com.yourdomain.foodorderingapp.controller;

import com.yourdomain.foodorderingapp.dto.LoginRequest;
import com.yourdomain.foodorderingapp.dto.RegisterRequest;
import com.yourdomain.foodorderingapp.dto.JwtResponse; // You'll create this
import com.yourdomain.foodorderingapp.dto.MessageResponse; // You'll create
this
import com.yourdomain.foodorderingapp.model.User;
import com.yourdomain.foodorderingapp.service.UserService;
// ... other imports for Spring Security, JWT
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationTo
ken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
// import com.yourdomain.foodorderingapp.security.jwt.JwtUtils; // You'll create
this
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;
import java.util.HashSet;
import java.util.Set;


@RestController
@RequestMapping("/api/auth")
public class AuthController {
   @Autowired
   AuthenticationManager authenticationManager; // Configure this in
SecurityConfig

   @Autowired
   UserService userService;

   // @Autowired
   // JwtUtils jwtUtils; // You'll create this

   @PostMapping("/register")
   public ResponseEntity<?> registerUser(@Valid @RequestBody RegisterRequest
registerRequest) {
      // Create new user's account
      User user = new User();
      user.setUsername(registerRequest.getUsername());
      user.setEmail(registerRequest.getEmail());
      user.setPassword(registerRequest.getPassword()); // Password will be
encoded in service

      Set<String> strRoles = registerRequest.getRoles();

      try {
         userService.registerUser(user, strRoles);
         return ResponseEntity.ok(new MessageResponse("User registered
successfully!"));
      } catch (RuntimeException e) {
          return ResponseEntity.badRequest().body(new
MessageResponse(e.getMessage()));
      }
   }

   @PostMapping("/login")
   public ResponseEntity<?> authenticateUser(@Valid @RequestBody
LoginRequest loginRequest) {
      Authentication authentication = authenticationManager.authenticate(
            new
UsernamePasswordAuthenticationToken(loginRequest.getUsername(),
loginRequest.getPassword()));
```

```
        SecurityContextHolder.getContext().setAuthentication(authentication);
        // String jwt = jwtUtils.generateJwtToken(authentication);

        // UserDetailsImpl userDetails = (UserDetailsImpl)
authentication.getPrincipal(); // You'll create UserDetailsImpl
        // List<String> roles = userDetails.getAuthorities().stream()
        //        .map(item -> item.getAuthority())
        //        .collect(Collectors.toList());

        // For now, returning a simple message. Replace with JwtResponse later.
        // return ResponseEntity.ok(new JwtResponse(jwt, userDetails.getId(),
userDetails.getUsername(), userDetails.getEmail(), roles));
        return ResponseEntity.ok(new MessageResponse("User logged in
successfully! (JWT generation pending)"));
    }
}
```

- **Example DTOs:** RegisterRequest.java, LoginRequest.java, RestaurantDto.java, OrderRequest.java.

Step 9: Implement Security (Spring Security + JWT)
This is a crucial and complex part.

- **Configuration:** Create a security configuration class (e.g., SecurityConfig.java) that extends WebSecurityConfigurerAdapter (older style) or defines a SecurityFilterChain bean (newer style).

```
// Example: SecurityConfig.java (Conceptual - newer style)
package com.yourdomain.foodorderingapp.config;

import com.yourdomain.foodorderingapp.security.jwt.AuthEntryPointJwt; // You'll
create
import com.yourdomain.foodorderingapp.security.jwt.AuthTokenFilter; // You'll
create
import
com.yourdomain.foodorderingapp.security.services.UserDetailsServiceImpl; //
You'll create
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
```

```
import
org.springframework.security.config.annotation.authentication.configuration.Auth
enticationConfiguration;
import
org.springframework.security.config.annotation.method.configuration.EnableGlob
alMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSec
urity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticati
onFilter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true) // For @PreAuthorize
public class SecurityConfig {

  @Autowired
  private UserDetailsServiceImpl userDetailsService; // Your implementation of
UserDetailsService

  @Autowired
  private AuthEntryPointJwt unauthorizedHandler; // Handles unauthorized
attempts

  @Bean
  public AuthTokenFilter authenticationJwtTokenFilter() {
    return new AuthTokenFilter(); // Your JWT filter
  }

  @Bean
  public AuthenticationManager
authenticationManager(AuthenticationConfiguration authConfig) throws
Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable()
            .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).
and()
            .authorizeRequests()
            .antMatchers("/api/auth/**").permitAll()
            .antMatchers("/api/restaurants/**").permitAll() // Example: public restaurant
listings
            .antMatchers("/api/test/**").permitAll() // Example for testing
            .anyRequest().authenticated();

        http.addFilterBefore(authenticationJwtTokenFilter(),
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

- **Password Encoding:** Use a PasswordEncoder bean (e.g., BCryptPasswordEncoder).
- **UserDetailsService:** Implement this interface (e.g., UserDetailsServiceImpl.java) to load user-specific data from your MongoDB users collection. It will return a UserDetails object (you'll create UserDetailsImpl.java).
- **JWT (JSON Web Tokens):**
    - Add JWT library dependency (e.g., io.jsonwebtoken:jjwt-api, jjwt-impl, jjwt-jackson).
    - Create JwtUtils.java to generate, validate, and parse JWTs.
    - Create AuthTokenFilter.java (a OncePerRequestFilter) to intercept requests,

extract JWT from the Authorization header, validate it, and set the Authentication in Spring Security's SecurityContextHolder.
    - Create AuthEntryPointJwt.java to handle authentication errors (commence method).
- **Secure Endpoints:** Configure which endpoints are public (e.g., /api/auth/**, GET /api/restaurants) and which require authentication/specific roles using http.authorizeRequests() and antMatchers() or method-level security with @PreAuthorize("hasRole('ADMIN')").

Step 10: Implement Exception Handling

- **Custom Exception Classes:** Create specific exceptions (e.g., ResourceNotFoundException.java, BadRequestException.java, AuthenticationException.java) extending RuntimeException.
- **Global Exception Handler (@ControllerAdvice):**
    - Create a class annotated with @ControllerAdvice (e.g., GlobalExceptionHandler.java).
    - Define methods annotated with @ExceptionHandler(SpecificException.class) to catch specific exceptions and return standardized error responses (e.g., a JSON object with timestamp, status, error, message, path).

```
// Example: GlobalExceptionHandler.java
package com.yourdomain.foodorderingapp.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import java.util.Date;
import java.util.stream.Collectors;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorDetails>
handleResourceNotFoundException(ResourceNotFoundException ex,
WebRequest request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
HttpStatus.NOT_FOUND.value(), "Not Found", ex.getMessage(),
request.getDescription(false));
        return new ResponseEntity<>(errorDetails, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(BadRequestException.class)
    public ResponseEntity<ErrorDetails>
handleBadRequestException(BadRequestException ex, WebRequest request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
HttpStatus.BAD_REQUEST.value(), "Bad Request", ex.getMessage(),
request.getDescription(false));
        return new ResponseEntity<>(errorDetails, HttpStatus.BAD_REQUEST);
    }

    // Handle validation errors from @Valid
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorDetails>
handleValidationExceptions(MethodArgumentNotValidException ex, WebRequest
request) {
        String errors = ex.getBindingResult().getFieldErrors().stream()
                        .map(error -> error.getField() + ": " + error.getDefaultMessage())
                        .collect(Collectors.joining(", "));
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
HttpStatus.BAD_REQUEST.value(), "Validation Error", errors,
request.getDescription(false));
        return new ResponseEntity<>(errorDetails, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(Exception.class) // Generic fallback
    public ResponseEntity<ErrorDetails> handleGlobalException(Exception ex,
WebRequest request) {
        ErrorDetails errorDetails = new ErrorDetails(new Date(),
HttpStatus.INTERNAL_SERVER_ERROR.value(), "Internal Server Error",
ex.getMessage(), request.getDescription(false));
        // Log the full exception for debugging
        // logger.error("Unhandled exception: ", ex);
        return new ResponseEntity<>(errorDetails,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
// ErrorDetails.java (a simple DTO for error responses)
// package com.yourdomain.foodorderingapp.exception;
// import lombok.AllArgsConstructor;
// import lombok.Data;
// import java.util.Date;
// @Data @AllArgsConstructor
// public class ErrorDetails {
//    private Date timestamp;
//    private int status;
//    private String error;
//    private String message;
//    private String path;
// }
```

- **Validation:** Use Java Bean Validation annotations (@NotNull, @Size, @Email, etc.) on your DTOs. Spring Boot automatically validates request bodies annotated with @Valid. MethodArgumentNotValidException will be thrown for validation failures, which you can handle in your GlobalExceptionHandler.

Step 11: Implement Logging

- Spring Boot uses SLF4J with Logback by default, which is a powerful combination.
- Configure logging levels in application.properties or for more advanced configuration, use a logback-spring.xml file in src/main/resources.

```
# application.properties
logging.level.root=INFO
logging.level.com.yourdomain.foodorderingapp=DEBUG # Your base package
logging.level.org.springframework.web=INFO
logging.level.org.springframework.security=INFO
logging.level.org.mongodb.driver=INFO

# To log to a file
# logging.file.name=logs/food-ordering-app.log
# logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
%logger{36} - %msg%n
```

- Use Logger instances in your classes:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class SomeService {
   private static final Logger logger =
LoggerFactory.getLogger(SomeService.class);

   public void performAction(String input) {
      logger.debug("Performing action with input: {}", input);
      try {
         // ... business logic ...
         if (input.equals("test")) {
            logger.info("Test action performed successfully.");
         } else {
            logger.warn("Unexpected input received: {}", input);
         }
      } catch (Exception e) {
         logger.error("Error performing action with input {}: ", input, e);
         // throw custom exception
      }
   }
}
```

- Log important events: method entries/exits (using AOP or manually), errors, business milestones (user registration, order placement), security events.

Phase 4: Frontend Development (React)

Step 12: Install Frontend Dependencies

- **React Router:** For client-side navigation (react-router-dom).
- **Axios:** For making HTTP requests to your backend (or use the built-in fetch API).
- **State Management Library (choose one based on complexity):**
  - **Context API + useReducer:** Built into React, good for simpler to medium global state.
  - **Redux Toolkit:** Industry standard for complex state management, provides a structured way.
  - **Zustand or Jotai:** Simpler, more modern alternatives to Redux for global state.
- **UI Library (Optional but recommended for faster development):**
  - Material-UI (MUI)
  - Ant Design
  - Chakra UI
  - Tailwind CSS (utility-first CSS framework, often used with Headless UI)
- **Form Handling (Optional):** Formik, React Hook Form.
- **Utility Libraries:** date-fns or moment (for date formatting), classnames (for conditional CSS classes).

```
# Navigate to your frontend directory
cd food-ordering-frontend
```

```
npm install react-router-dom axios
# Example for Redux Toolkit
npm install @reduxjs/toolkit react-redux
# Example for Material-UI (MUI)
npm install @mui/material @emotion/react @emotion/styled @mui/icons-material
# Example for Tailwind CSS (requires post-css setup)
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Step 13: Setup Routing

- In src/App.js or a dedicated src/routes/AppRoutes.js file, configure your application's routes using BrowserRouter, Routes, and Route from react-router-dom.
- Create PrivateRoute or ProtectedRoute components that check for user authentication (e.g., presence of a JWT in local storage or global state) before rendering a protected route. If not authenticated, redirect to the login page.

Step 14: Create Components & Pages

- **Pages (Containers/Views):** These are top-level components for each route.
  - HomePage.js
  - LoginPage.js, RegisterPage.js
  - RestaurantListPage.js, RestaurantDetailPage.js (shows menu)
  - CartPage.js, CheckoutPage.js
  - OrderHistoryPage.js, OrderConfirmationPage.js
  - UserProfilePage.js
  - AdminDashboardPage.js (if applicable)
- **Reusable Components:** Smaller, focused UI pieces.
  - Navbar.js, Footer.js
  - RestaurantCard.js, MenuItemCard.js
  - CartItem.js, OrderSummary.js
  - SearchBar.js, FilterDropdown.js
  - ReviewForm.js, StarRating.js
  - Button.js, Input.js, Modal.js (if not using a UI library)
  - AlertMessage.js (for success/error notifications)

Step 15: State Management

- Decide on your state management strategy.
- **Global State:** Typically includes user authentication status (isLoggedIn, user

object, token), shopping cart contents, site-wide notifications.
- **Local Component State (useState, useReducer):** Used for form inputs, UI toggles (e.g., modal visibility), component-specific data.
- If using Redux Toolkit, define "slices" for different parts of your state (e.g., authSlice.js, cartSlice.js, restaurantSlice.js).

Step 16: API Service Layer

- Create a dedicated folder (e.g., src/services) for functions that make API calls to your backend.
- Use Axios (or fetch). Configure an Axios instance with a base URL for your backend API.
- Implement Axios interceptors to:
  - Automatically add the JWT token to the Authorization header for authenticated requests.
  - Handle global API error responses (e.g., redirect to login on 401 Unauthorized).

```
// src/services/api.js (Example with Axios)
import axios from 'axios';

const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:8080/api'; // Set
in .env file

const apiClient = axios.create({
  baseURL: API_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});

// Request interceptor to add JWT token
apiClient.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('authToken'); // Or get from Redux
store/Context
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response interceptor for global error handling (optional)
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response && error.response.status === 401) {
      // Handle unauthorized access, e.g., logout user, redirect to login
      // localStorage.removeItem('authToken');
      // window.location.href = '/login';
      console.error("Unauthorized access - 401");
    }
    return Promise.reject(error);
  }
);

export default apiClient;

// src/services/authService.js
// import apiClient from './api';
//
// export const loginUser = (credentials) => apiClient.post('/auth/login', credentials);
// export const registerUser = (userData) => apiClient.post('/auth/register', userData);
// export const getCurrentUser = () => apiClient.get('/users/me'); // Example
```

Step 17: Implement User Interface and Core Features

- **Authentication Forms:** Create Login and Registration forms. On successful login from the backend, store the received JWT (e.g., in localStorage, sessionStorage, or HttpOnly cookie managed by backend) and user information in your global state.
- **Display Data:** Fetch and display restaurants, menu items. Implement search, filtering, and pagination.
- **Shopping Cart:** Implement functionality to add items to the cart, remove items, and update quantities. Persist cart data (e.g., in local storage or sync with backend if user is logged in).
- **Checkout Process:** Create a multi-step checkout form to collect delivery address, and payment information (initially, you might simulate payment or integrate a test mode of a payment gateway).
- **Error Handling:** Display user-friendly error messages received from API responses or from client-side validation. Use your AlertMessage.js component.
- **Loading States:** Show loading indicators (spinners, skeletons) while data is being

fetched.

Step 18: Styling

- Choose your styling approach:
  - **CSS Modules:** Scoped CSS for each component.
  - **Styled-Components or Emotion:** CSS-in-JS libraries.
  - **UI Library (MUI, Ant Design):** Comes with pre-styled components, customize as needed.
  - **Tailwind CSS:** Apply utility classes directly in your JSX.
- Ensure your application is responsive and looks good on different screen sizes.

Phase 5: Integration, Testing & Deployment

Step 19: CORS Configuration (Backend)

- Your React app (e.g., running on http://localhost:3000) will make requests to your Spring Boot backend (e.g., http://localhost:8080). Browsers enforce the Same-Origin Policy, so you'll need to enable Cross-Origin Resource Sharing (CORS) on the backend.
- Configure CORS in your Spring Boot SecurityConfig or by creating a WebMvcConfigurer bean:

```
// In SecurityConfig.java, within the http.cors() chain or as a separate bean:
// @Bean
// public WebMvcConfigurer corsConfigurer() {
//    return new WebMvcConfigurer() {
//        @Override
//        public void addCorsMappings(CorsRegistry registry) {
//            registry.addMapping("/api/**") // Or "/**" for all endpoints
//                    .allowedOrigins("http://localhost:3000") // Your React app's origin
//                    .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
//                    .allowedHeaders("*")
//                    .allowCredentials(true); // If you need to send cookies/auth headers
//        }
//    };
// }
// Or use @CrossOrigin annotation on controllers/methods for finer-grained control.
```

  Ensure your SecurityConfig calls .cors() in the HttpSecurity chain.

Step 20: Testing

- **Backend (Java/Spring Boot):**
  - **Unit Tests (JUnit 5, Mockito):** Test individual service methods, utility classes, and components in isolation. Mock dependencies.
  - **Integration Tests (@SpringBootTest, Testcontainers for MongoDB):** Test interactions between components, including the service layer, repository layer, and database. Testcontainers can spin up a real MongoDB instance for tests.
  - **API/Controller Tests (MockMvc or RestAssured):** Test your REST API endpoints, request/response formats, and status codes.
- **Frontend (React):**
  - **Unit Tests (Jest, React Testing Library):** Test individual React components, custom hooks, and utility functions.
  - **Integration Tests (React Testing Library):** Test interactions between multiple components, user flows within a part of the application (e.g., adding an item to the cart and seeing the cart update).
  - **End-to-End (E2E) Tests (Cypress, Playwright):** Test complete user flows through the entire application, interacting with the UI as a user would. These are slower but crucial for verifying overall functionality.

Step 21: Build & Deployment (Conceptual)

- **Backend (Spring Boot):**
  - Build an executable JAR or WAR file: mvn clean package or gradle clean build.
  - **Dockerize:** Create a Dockerfile to containerize your Spring Boot application. This makes deployment consistent across environments.
  - Deploy to a cloud platform (AWS EC2, AWS Elastic Beanstalk, Heroku, Google Cloud App Engine/Kubernetes Engine, Azure App Service) or a self-managed server.
- **Frontend (React):**
  - Build static assets: npm run build or yarn build. This creates an optimized build folder.
  - Deploy these static files to a web server (Nginx, Apache) or static hosting services (Netlify, Vercel, AWS S3 + CloudFront, GitHub Pages, Firebase Hosting).
- **Database (MongoDB):**
  - Use a managed MongoDB service like MongoDB Atlas in production for reliability, backups, and scaling.
- **Environment Variables:**
  - Manage configuration (database URLs, API keys, JWT secrets, frontend API URLs) using environment variables.
  - Spring Boot: Use application-{profile}.properties (e.g.,

    application-prod.properties) and OS environment variables.
  - React: Use .env files (.env.development, .env.production) and access variables via process.env.REACT_APP_YOUR_VARIABLE.

Phase 6: Further Enhancements & Considerations (Optional)

- **Real-time Order Updates:** Use WebSockets (e.g., Spring WebSocket, STOMP) or Server-Sent Events (SSE) for live order tracking for users and new order notifications for restaurants.
- **Payment Gateway Integration:** Integrate with Stripe, PayPal, or other local payment gateways.
- **Image Uploads:** Allow restaurants to upload images for their venue and menu items. Store these on a cloud storage service (AWS S3, Google Cloud Storage, Cloudinary).
- **Geospatial Queries:** Implement "find restaurants near me" functionality using MongoDB's geospatial indexing and query capabilities.
- **Caching:** Implement caching for frequently accessed data (e.g., restaurant lists, popular menu items) using Redis or Ehcache (Spring Cache abstraction).
- **API Documentation:** Use Swagger/OpenAPI (Springdoc OpenAPI UI is excellent for Spring Boot) to generate interactive API documentation.
- **Advanced Search & Filtering:** Implement more complex search capabilities (e.g., full-text search with MongoDB Atlas Search).
- **Notifications:** Email or push notifications for order confirmations, status updates, etc.
- **CI/CD Pipelines:** Set up Continuous Integration/Continuous Deployment pipelines (e.g., using GitHub Actions, Jenkins, GitLab CI) to automate testing and deployment.
- **Scalability & Performance Optimization:** Database indexing, query optimization, load balancing if needed.
- **Accessibility (a11y):** Ensure your frontend is accessible to users with disabilities by following WCAG guidelines.
- **Code Quality Tools:** Use linters (ESLint for React, Checkstyle/PMD/SpotBugs for Java) and code formatters (Prettier).

This step-by-step guide provides a solid foundation for building your food ordering application. Remember to break down tasks into smaller, manageable pieces, test frequently, and commit your code regularly. Good luck!