|  |  |
| --- | --- |
| Obraz zawierający szkic, rysowanie, clipart, Grafika liniowa  Zawartość wygenerowana przez AI może być niepoprawna. | **POLITECHNIKA ŚLĄSKA**  **Wydział Inżynierii Materiałowej** |
| **Tworzenie aplikacji internetowych i bazodanowych** | |
| **TEMAT:**  **Tokyo Garden Restaurant — Dokumentacja Projektu** | |
| **Uwagi prowadzącego:** | |
| **Data ostatniej modyfikacji:**  **16.09.2025**    **Podpis prowadzącego:** | **Imię i nazwisko:**  **1.** Jakub Majer  **2.** Wojciech Kin  **3.** Szymon Czapla  **4**. Dominik Deszczka |
| **Grupa dziekańska: TP 4.0** |

[0. Streszczenie wykonawcze 3](#_Toc208950117)

[Cel biznesowy 3](#_Toc208950118)

[Zakres systemu 3](#_Toc208950119)

[Korzyści i mierniki 3](#_Toc208950120)

[1. Specyfikacja 3](#_Toc208950121)

[Kontekst i interesariusze 3](#_Toc208950122)

[Przypadki użycia (skrót) 4](#_Toc208950123)

[Wymagania niefunkcjonalne 4](#_Toc208950124)

[2. Model danych i domeny 4](#_Toc208950125)

[CDM – encje i relacje 4](#_Toc208950126)

[PDM – tabele i klucze 6](#_Toc208950127)

[3. Architektura i warstwy 9](#_Toc208950128)

[Przegląd 9](#_Toc208950129)

[Wzorce i decyzje projektowe 9](#_Toc208950130)

[Warstwa Model 9](#_Toc208950131)

[Warstwa IDAL 10](#_Toc208950132)

[Warstwa DAL 11](#_Toc208950133)

[Warstwa IBL 11](#_Toc208950134)

[Warstwa Biznesowa 12](#_Toc208950135)

[API – Web 13](#_Toc208950136)

[DtoMapper.cs 14](#_Toc208950137)

[DTO 15](#_Toc208950138)

[Frontend – Angular 15](#_Toc208950139)

[Bezpieczeństwo i zgodność 15](#_Toc208950140)

[4. Testy (podejście i zakres) 16](#_Toc208950141)

[Zasady ogólne 16](#_Toc208950142)

[Doubles własne i Moq 16](#_Toc208950143)

[Testy DAL z InMemory 16](#_Toc208950144)

[5. Interfejs Web API (kontrakt i zasady) 17](#_Toc208950145)

[Konwencje REST 17](#_Toc208950146)

[Przykładowe zasoby i operacje 17](#_Toc208950147)

[Walidacja i błędy 17](#_Toc208950148)

[Paginacja/sortowanie/filtrowanie 17](#_Toc208950149)

[6. Frontend (Angular) 17](#_Toc208950150)

[Struktura i wzorce 17](#_Toc208950151)

[UX i walidacje 18](#_Toc208950152)

[Mapowanie DTO 18](#_Toc208950153)

[7. Bezpieczeństwo i zgodność 18](#_Toc208950154)

[Autoryzacja i role 18](#_Toc208950155)

[CORS i ochrona danych 18](#_Toc208950156)

# 0. Streszczenie wykonawcze

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

## Cel biznesowy

Celem systemu Tokyo Garden Restaurant jest wsparcie pełnego cyklu życia zarządzania ofertą i zamówieniami w restauracji: od definicji kategorii i pozycji menu (ze składnikami i alergenami), przez zbieranie zamówień klientów, po administrację danymi użytkowników i adresów dostaw. System ma ułatwić codzienną pracę obsłudze i zapewnić spójność danych.

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

## Zakres systemu

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

System składa się z backendu opartego na ASP.NET Core Web API, z warstwami logiki biznesowej (BL) i dostępu do danych (DAL) wykorzystującymi Entity Framework Core, oraz frontendu zbudowanego w technologii Angular. Backend udostępnia REST-owe endpointy wspierające operacje CRUD oraz specyficzne operacje domenowe, takie jak filtrowanie czy walidacja danych. Frontend zapewnia panel administracyjny do zarządzania danymi oraz widoki dla klientów, umożliwiające przeglądanie menu, składanie zamówień i śledzenie ich statusu.

## Korzyści i mierniki

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

System TokyoGarden przynosi następujące korzyści:

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

Redukcja błędów: Wymuszone relacje i walidacje w warstwie BL minimalizują niepoprawność danych.

Szybsza obsługa katalogu: Usprawnione zarządzanie kategoriami, alergenami i zdjęciami pozycji menu.

Skalowalność: Warstwowa architektura i wzorzec repozytorium zapewniają elastyczność i łatwość rozbudowy.

Mierniki efektywności: Czas wprowadzania zmiany w pozycji menu, liczba błędów w zamówieniach wykrytych przez walidację, czasy odpowiedzi endpointów API.

Doprecyzowano kontrakty REST – opisano wszystkie pola w DTO, podano przykładowe requesty i response’y dla metod GET, POST i PUT. Uwzględniono również obsługę błędów z kodami statusu i formatem ProblemDetails. Dodano sugestie wersjonowania API w przyszłości (np. /api/v2) w razie rozbudowy systemu.

# 1. Specyfikacja

## Kontekst i interesariusze

System TokyoGarden jest przeznaczony dla różnych grup interesariuszy, z których każda ma specyficzne potrzeby:

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

Klienci: Przeglądają menu, filtrują pozycje według kategorii i alergenów, składają zamówienia.

Administratorzy/Pracownicy: Zarządzają danymi menu, kategoriami, alergenami oraz statusami zamówień.

Właściciele: Korzystają z raportów i nadzorują operacje restauracji. W przyszłości system może zostać zintegrowany z zewnętrzną bramką płatniczą, co rozszerzy jego funkcjonalność o obsługę płatności online.

## Przypadki użycia (skrót)

Kluczowe przypadki użycia obejmują:

Klient: Przeglądanie menu, filtrowanie pozycji według kategorii lub alergenów, składanie i śledzenie zamówień.

Administrator: Dodawanie, edycja i usuwanie pozycji menu, przypisywanie kategorii i alergenów, aktualizacja statusów zamówień.

System: Automatyczna walidacja danych, zapewnienie spójności relacji między encjami, obliczanie wartości zamówień na podstawie pozycji i ich ilości.

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

## Wymagania niefunkcjonalne

System spełnia następujące wymagania niefunkcjonalne:

System został zaprojektowany w oparciu o analizę potrzeb restauracji, z uwzględnieniem wygody użytkownika, minimalizacji błędów i możliwości rozbudowy w przyszłości. Dokumentacja opisuje szczegółowo każdy element – od koncepcji po implementację – tak aby mogła stanowić punkt odniesienia przy dalszym rozwoju systemu lub jego utrzymaniu.

Wydajność: Endpointy listowe wspierają paginację, sortowanie i filtrowanie, zapewniając szybkie odpowiedzi nawet przy dużej liczbie danych.

Bezpieczeństwo: Autoryzacja oparta na rolach (Admin/User) z planowaną rozbudową o JWT, walidacja danych wejściowych, zabezpieczenia CORS.

Utrzymywalność: Warstwowa architektura z Dependency Injection (DI) i jasno zdefiniowanymi kontraktami ułatwia konserwację i rozwój.

Jakość kodu: Testy jednostkowe dla warstw BL, DAL i kontrolerów zapewniają niezawodność i poprawność działania.

# 2. Model danych i domeny

## CDM – encje i relacje

Diagram Konceptualnego Modelu Danych (CDM) systemu TokyoGarden przedstawia w sposób precyzyjny i uporządkowany logiczną strukturę danych, która wspiera kluczowe procesy zamawiania i realizacji usług w restauracji, zapewniając przejrzystość dla projektantów i interesariuszy biznesowych. Model ten koncentruje się na definiowaniu głównych encji, ich atrybutów oraz wzajemnych relacji, celowo pomijając techniczne detale, takie jak typy danych, indeksy czy ograniczenia bazodanowe, aby ułatwić zrozumienie ogólnej organizacji danych w systemie. Jego nadrzędnym celem jest dostarczenie klarownego, logicznego obrazu struktury danych, który stanowi fundament dla dalszego projektowania i implementacji bazy danych, wspierając efektywne zarządzanie procesami restauracyjnymi. Model CDM obejmuje sześć głównych encji, z których każda odzwierciedla specyficzne aspekty funkcjonalności systemu TokyoGarden, zapewniając kompleksowe wsparcie dla operacji restauracyjnych:

1. Adresy: Encja przechowująca szczegółowe dane lokalizacyjne użytkowników lub miejsc dostawy, zawierająca atrybuty takie jak miasto, ulica, numer domu oraz numer mieszkania, co umożliwia precyzyjne określenie lokalizacji dla potrzeb logistycznych systemu.
2. Użytkownicy: Reprezentuje klientów oraz administratorów korzystających z platformy, z atrybutami obejmującymi nazwę użytkownika, hasło, numer telefonu oraz klucz obcy fk\_id\_adresu, który jednoznacznie wiąże każdego użytkownika z dokładnie jednym adresem w systemie.
3. Zamówienia: Przechowuje szczegółowe informacje dotyczące zamówień składanych przez użytkowników, w tym datę zamówienia, dodatkowe uwagi (np. prośba o szybką dostawę), opis zamówienia oraz klucze obce fk\_id\_użytkownika i fk\_id\_adresu, które łączą zamówienie z użytkownikiem i adresem dostawy.
4. Pozycje Zamówienia: Reprezentuje produkty wybrane w ramach pojedynczego zamówienia, z atrybutami takimi jak cena jednostkowa pozycji oraz liczba zamówionych sztuk, powiązana z encjami Zamówienia i Pozycje Menu, co pozwala na precyzyjne określenie składu każdego zamówienia.
5. Pozycje Menu: Zawiera dane dotyczące dań dostępnych w ofercie restauracji, w tym nazwę pozycji, opis, cenę oraz dane obrazu (imageData), powiązana z Kategoriami Menu, co umożliwia logiczne grupowanie dań według określonych kategorii.
6. Kategorie Menu: Służy do organizowania dań w grupy, takie jak przystawki, dania główne czy desery, z atrybutem nazwa\_kategorii (np. „Sushi” lub „Zupy”), co znacząco ułatwia prezentację oferty w przejrzysty i uporządkowany sposób.

Relacje w modelu:

* Relacja Użytkownicy → Adresy (1:N): Jeden adres może być przypisany do wielu użytkowników, co odzwierciedla możliwość współdzielenia lokalizacji przez różnych klientów lub administratorów.
* Relacja Zamówienia → Użytkownicy (N:1): Każde zamówienie jest przypisane dokładnie jednemu użytkownikowi, co zapewnia jednoznaczność odpowiedzialności za zamówienie.
* Relacja Zamówienia → Adresy (N:1): Każde zamówienie jest powiązane z jednym adresem dostawy, co pozwala na precyzyjne określenie miejsca realizacji usługi.
* Relacja Pozycje Zamówienia → Zamówienia (N:1): Każda pozycja zamówienia należy do dokładnie jednego zamówienia, co umożliwia szczegółowe śledzenie zawartości koszyka.
* Relacja Pozycje Zamówienia → Pozycje Menu (N:1): Każda pozycja zamówienia odnosi się do jednego dania z menu, co zapewnia spójność oferty z wyborem klienta.
* Relacja Pozycje Menu → Kategorie Menu (N:1): Każde danie należy do jednej kategorii menu, co ułatwia organizację i przeglądanie oferty restauracyjnej.

Model CDM systemu TokyoGarden gwarantuje logiczne i precyzyjne odwzorowanie procesów zamawiania, umożliwiając efektywne zarządzanie danymi oraz ich przejrzystą organizację, co stanowi solidną podstawę dla projektowania i implementacji bazy danych w dalszych etapach rozwoju systemu.

## PDM – tabele i klucze

Diagram Fizycznego Modelu Danych (PDM) systemu TokyoGarden definiuje szczegółową strukturę bazy danych dla SQL Server, uwzględniając techniczne aspekty, takie jak typy danych, klucze główne i obce, ograniczenia integralności oraz konfiguracje wspierające migracje bazy danych i Entity Framework Core. Model ten zapewnia zgodność z wymaganiami systemu, umożliwiając generowanie skryptów SQL oraz efektywne zarządzanie danymi restauracyjnymi, co jest kluczowe dla prawidłowego funkcjonowania aplikacji. Baza danych składa się z ośmiu tabel, które odzwierciedlają strukturę danych i zależności między nimi, wspierając wszystkie kluczowe operacje systemu:

1. Adresy: Przechowuje dane lokalizacyjne, takie jak id\_adres (int, klucz główny), miasto (varchar(255)), ulica (varchar(255)), nr\_domu (varchar(255)), nr\_mieszkania (varchar(255)), co pozwala na precyzyjne określenie miejsca dostawy.
2. Użytkownicy: Reprezentuje klientów i administratorów, z atrybutami id\_uzytkownika (int, klucz główny), nazwa\_uzytkownika (varchar(255)), haslo (varchar(255)), email (varchar(255)), nr\_telefonu (varchar(255)), wspierając autoryzację i komunikację z użytkownikami.
3. Zamówienia: Przechowuje szczegółowe dane zamówień, w tym id\_zamowienia (int, klucz główny), data\_zamowienia (datetime), dodatkowe\_informacje (varchar(255)), id\_uzytkownika (int, klucz obcy), id\_adres (int, klucz obcy), metoda\_platnosci (varchar(255)), status\_zamowienia (varchar(255)), co stanowi centralny element zarządzania procesami zamawiania.
4. Pozycje\_Zamowienia: Określa produkty w zamówieniu, z atrybutami id\_pozycja\_zamowienia (int, klucz główny), ilosc (int), id\_pozycja\_menu (int, klucz obcy), id\_zamowienia (int, klucz obcy), umożliwiając szczegółowe określenie składu każdego zamówienia.
5. Pozycje\_Menu: Przechowuje informacje o daniach, w tym id\_pozycja\_menu (int, klucz główny), cena (int), nazwa\_pozycji (varchar(255)), opis (varchar(255)), imageData (varchar(255)), id\_kategorii (int, klucz obcy), co pozwala na prezentację pełnej oferty restauracji.
6. Kategorie: Grupuje dania w kategorie, takie jak przystawki czy desery, z atrybutami id\_kategorii (int, klucz główny), nazwa\_kategorii (varchar(255)), co ułatwia organizację menu w aplikacji.
7. Alergeny: Przechowuje dane o alergenach, z atrybutami id\_alergenu (int, klucz główny), opis\_alergenu (varchar(255)), co wspiera informowanie klientów o potencjalnych zagrożeniach zdrowotnych.
8. Alergeny\_Pozycje\_Menu: Łączy alergeny z daniami, z atrybutami id\_alergenu\_pozycje\_menu (int, klucz główny), id\_alergenu (int, klucz obcy), id\_pozycja\_menu (int, klucz obcy), umożliwiając precyzyjne określenie alergenów dla każdej pozycji menu.

Relacje w modelu:

* Relacja Użytkownicy → Zamówienia (1:N): Jeden użytkownik może złożyć wiele zamówień, co odzwierciedla aktywność klienta w systemie.
* Relacja Adresy → Zamówienia (1:N): Jeden adres może być używany w wielu zamówieniach, co pozwala na elastyczne zarządzanie lokalizacjami dostaw.
* Relacja Zamówienia → Pozycje\_Zamowienia (1:N): Jedno zamówienie może zawierać wiele pozycji, co umożliwia szczegółowe określenie zawartości koszyka.
* Relacja Pozycje\_Menu → Pozycje\_Zamowienia (1:N): Jedna pozycja menu może występować w wielu pozycjach zamówienia, co zapewnia spójność oferty.
* Relacja Kategorie → Pozycje\_Menu (1:N): Jedna kategoria może zawierać wiele pozycji menu, co ułatwia organizację oferty kulinarnej.
* Relacja Alergeny ↔ Pozycje\_Menu (N:N): Powiązanie realizowane przez tabelę Alergeny\_Pozycje\_Menu, co pozwala na elastyczne przypisywanie alergenów do dań.

Konwencje i integralność:

* Klucze główne (PK) są zdefiniowane jako identyfikatory (Id) typu int, zapewniające unikalność rekordów.
* Klucze obce (FK) stosują ograniczenia kaskadowe tylko tam, gdzie jest to uzasadnione, np. brak kaskadowego usuwania zamówień przy usuwaniu użytkownika, aby zachować integralność historycznych danych.
* Indeksy zdefiniowano dla pól często wyszukiwanych, takich jak nazwa\_pozycji i data\_zamowienia, oraz dla kluczy obcych, co zwiększa wydajność zapytań.
* Walidacje obejmują wymagane pola (np. nazwy, ceny), ograniczenia wartości (ceny > 0, ilości > 0) oraz relację N:M dla alergenów i pozycji menu poprzez tabelę łączącą.

Model PDM systemu TokyoGarden stanowi solidną techniczną podstawę, umożliwiając efektywną implementację bazy danych w SQL Server oraz wspierając kluczowe procesy zamawiania i zarządzania ofertą restauracyjną.

**Diagram Klas UML**

Diagram klas UML systemu TokyoGarden przedstawia szczegółową, obiektową strukturę danych, definiując klasy, ich atrybuty oraz relacje między nimi, co wspiera programowanie obiektowe, komunikację między backendem a frontendem oraz generowanie dokumentacji technicznej. Model ten jest kluczowy dla projektowania logiki biznesowej, definiowania struktur danych w kodzie oraz zapewnienia spójności w implementacji aplikacji. Obejmuje osiem klas, które odzwierciedlają główne komponenty systemu, wspierając zarówno backend, jak i frontend:

1. Adres: Zawiera atrybuty id (int, unikalny identyfikator), miasto (String, nazwa miejscowości), ulica (String, nazwa ulicy), nr\_domu (int, numer budynku), nr\_mieszkania (int, numer mieszkania), co umożliwia precyzyjne zarządzanie danymi lokalizacyjnymi w systemie.
2. Użytkownik: Przechowuje dane użytkowników, w tym id\_użytkownika (int, unikalny identyfikator), nazwa\_użytkownika (String, login), hasło (String, zaszyfrowane), telefon (String, numer kontaktowy), co wspiera zarządzanie kontami klientów i administratorów.
3. Zamówienie: Reprezentuje pojedyncze zamówienie, z atrybutami id\_zamówienia (int, unikalny identyfikator), data\_zamówienia (DateTime, data złożenia), dodatkowe\_informacje (String, uwagi klienta), status\_zamówienia (String, np. „Nowy”), łączna\_cena (double, koszt całkowity), metoda\_płatności (String, np. „gotówka”), opcje\_zamówienia (String, rodzaj zamówienia), użytkownik\_id (int, klucz obcy), co czyni ją centralnym elementem systemu.
4. PozycjaZamówienia: Przechowuje informacje o produktach w zamówieniu, z atrybutami id\_pozycje\_zamówienia (int, unikalny identyfikator), cena (double, koszt pozycji), ilość (int, liczba sztuk), pozycja\_menu\_id (int, klucz obcy), zamówienie\_id (int, klucz obcy), co pozwala na szczegółowe określenie składu zamówienia.
5. PozycjaMenu: Reprezentuje dania w menu, z atrybutami id\_pozycja\_menu (int, unikalny identyfikator), nazwa (String, nazwa dania), opis (String, szczegóły przygotowania), cena (double, koszt), składniki (String, lista składników), co wspiera prezentację oferty restauracyjnej.
6. Kategoria: Grupuje dania w kategorie, z atrybutami id\_kategoria (int, unikalny identyfikator), nazwa\_kategorii (String, np. „Przystawki”), co ułatwia organizację menu w aplikacji.
7. Alergen: Przechowuje informacje o alergenach, z atrybutami id\_alergen (int, unikalny identyfikator), nazwa (String, nazwa alergenu), opis\_alergenu (String, szczegółowy opis), co wspiera informowanie klientów o potencjalnych zagrożeniach.
8. AlergenPozycjaMenu: Klasa łącząca dla relacji wiele-do-wielu, z atrybutami pozycja\_menu\_id (int, klucz obcy), alergen\_id (int, klucz obcy), co umożliwia przypisanie alergenów do pozycji menu.

Relacje w modelu:

* Relacja Użytkownik → Adres (1:N): Jeden adres może być przypisany wielu użytkownikom, co odzwierciedla współdzielenie lokalizacji.
* Relacja Użytkownik → Zamówienie (1:N): Jeden użytkownik może złożyć wiele zamówień, co zapewnia jednoznaczność odpowiedzialności.
* Relacja Zamówienie → PozycjaZamówienia (1:N): Jedno zamówienie zawiera wiele pozycji, co pozwala na szczegółowe określenie zawartości.
* Relacja PozycjaZamówienia → PozycjaMenu (N:1): Każda pozycja zamówienia odnosi się do jednego dania, co zapewnia spójność z menu.
* Relacja PozycjaMenu → Kategoria (N:1): Każde danie należy do jednej kategorii, co ułatwia organizację oferty.
* Relacja PozycjaMenu ↔ Alergen (N:N): Realizowana przez klasę AlergenPozycjaMenu, co umożliwia elastyczne przypisywanie alergenów.

Diagram klas UML wspiera projektowanie struktur encji w warstwie TokyoGarden.Model, definiowanie obiektów DTO, konfigurację relacji w Entity Framework Core, komunikację między zespołami deweloperskimi oraz generowanie dokumentacji i testów jednostkowych, zapewniając wysoką jakość kodu.

# 3. Architektura i warstwy

## Przegląd

Architektura systemu TokyoGarden opiera się na warstwowym podejściu, które obejmuje następujące komponenty: Model (klasy POCO), IDAL (interfejsy repozytoriów i UnitOfWork), DAL (implementacje repozytoriów z Entity Framework Core), IBL (interfejsy usług biznesowych), BL (implementacje logiki biznesowej), API (ASP.NET Core Web API) oraz UI (Angular). Zasada projektowa zakłada, że warstwa BL komunikuje się z IDAL, a API z IBL, eliminując zależności wsteczne, co zapewnia modularność i łatwość utrzymania systemu.

## Wzorce i decyzje projektowe

System wykorzystuje następujące wzorce projektowe:

* Repository i UnitOfWork: Ujednolicone kontrakty dla operacji CRUD i zarządzania transakcjami, zapewniające spójność danych.
* DTO vs Model: API operuje na obiektach DTO, podczas gdy BL korzysta z modeli domenowych, z mapowaniem realizowanym w warstwie API.
* Dependency Injection (DI)/Inversion of Control (IoC): Kontrolery wstrzykują interfejsy IBL, a BL wstrzykuje IDAL, co wspiera testowalność i modularność.
* Walidacja: Logika biznesowa w BL, wstępna walidacja atrybutami w DTO, co zapewnia spójność i bezpieczeństwo danych.

## Warstwa Model

Warstwa Model w aplikacji TokyoGarden zawiera klasy encji, które są mapowane na tabele bazy danych SQL Server za pomocą Entity Framework Core, zapewniając spójność danych i separację odpowiedzialności między warstwami aplikacji. Encje te są wykorzystywane w warstwach dostępu do danych (DAL), logiki biznesowej (BL) oraz API, wspierając efektywne zarządzanie danymi w systemie. Warstwa obejmuje siedem klas encji, z których każda odpowiada za reprezentację określonego typu danych:

1. Adresy: Przechowuje szczegółowe dane lokalizacyjne użytkowników lub miejsc dostawy, takie jak miasto, ulica, numer domu oraz numer mieszkania, co umożliwia precyzyjne określenie lokalizacji dla potrzeb logistycznych.
2. Alergeny: Zawiera informacje o alergenach występujących w produktach menu, takich jak orzechy czy gluten, co wspiera informowanie klientów o potencjalnych zagrożeniach zdrowotnych.
3. Kategorie: Reprezentuje kategorie pozycji menu, takie jak Sushi, Ramen czy Napoje, co ułatwia organizację i prezentację oferty restauracyjnej w aplikacji.
4. Pozycje\_Menu: Odpowiada za pojedyncze pozycje w menu, zawierające szczegółowe informacje o nazwie, opisie, cenie oraz danych obrazu, co wspiera prezentację oferty kulinarnej.
5. Pozycje\_Zamowienia: Reprezentuje produkty w zamówieniu, z atrybutami takimi jak ilość i cena wybranej pozycji menu, co umożliwia szczegółowe określenie składu koszyka.
6. Użytkownicy: Przechowuje dane użytkowników systemu, w tym nazwę użytkownika, hasło oraz dane kontaktowe, co wspiera autoryzację i komunikację z klientami oraz administratorami.
7. Zamówienia: Reprezentuje pełne zamówienie złożone przez użytkownika, obejmujące szczegóły takie jak data zamówienia, status, łączna cena oraz powiązane pozycje, co jest kluczowe dla zarządzania procesami zamawiania.

Adnotacje Entity Framework Core:

* [Key]: Wskazuje atrybut pełniący rolę klucza głównego, zapewniając unikalność rekordów w bazie danych.
* [Column(TypeName = "decimal(18,2)")]: Definiuje precyzyjny typ danych dla kolumn numerycznych, takich jak ceny, co zapewnia dokładność obliczeń finansowych.
* virtual ICollection<T>: Umożliwia lazy loading dla relacji jeden-do-wielu lub wiele-do-wielu, co pozwala na ładowanie powiązanych danych na żądanie.
* virtual T?: Obsługuje relacje opcjonalne, np. użytkownik w zamówieniu może być null, co zwiększa elastyczność modelowania danych.

Warstwa Model zapewnia solidną podstawę dla mapowania obiektowo-relacyjnego, umożliwiając efektywne zarządzanie danymi w aplikacji TokyoGarden.

## Warstwa IDAL

Warstwa interfejsów dostępu do danych (IDAL) w przestrzeni TokyoGarden.IDAL definiuje kontrakty dla repozytoriów, wspierając separację odpowiedzialności, testowanie jednostkowe oraz zasadę Dependency Inversion. Obejmuje następujące interfejsy:

* IRepository<t></t>: Generyczny interfejs dla operacji CRUD, definiujący metody takie jak GetAllAsync, GetByIdAsync, AddAsync, UpdateAsync, DeleteAsync, wspierające zapytania z IQueryable/IAsyncEnumerable oraz paginację.
* Specjalistyczne interfejsy: IAdresyRepository, IAlergenyRepository, IKategorieRepository, IPozycjeMenuRepository, IPozycjeZamowieniaRepository, IUzytkownikRepository, IZamowieniaRepository, każdy rozszerzający IRepository<t> o specyficzne metody, np. GetByCategoryIdAsync w IPozycjeMenuRepository.</t>
* IUnitOfWork: Definiuje metodę SaveChangesAsync dla zapewnienia spójności transakcji oraz dostęp do repozytoriów.

Interfejsy są implementowane w warstwie DAL, wykorzystywane przez serwisy BL oraz rejestrowane w DI, co wspiera modularność i testowalność kodu.

## Warstwa DAL

Warstwa dostępu do danych (DAL) w przestrzeni TokyoGarden.DAL odpowiada za bezpośredni dostęp do bazy danych SQL Server, wykorzystując Entity Framework Core do realizacji operacji CRUD oraz zapytań złożonych. Warstwa ta zapewnia izolację logiki biznesowej od szczegółów technicznych bazy danych, wspierając modularność i czystą architekturę. Obejmuje następujące komponenty:

* DbTokyoGarden: Kontekst Entity Framework Core, który zarządza połączeniem z bazą danych i definiuje DbSety dla każdej encji, takiej jak Użytkownicy czy Zamówienia.
* Repository<t></t>: Generyczne repozytorium dostarczające podstawowe operacje CRUD, takie jak GetAllAsync, GetByIdAsync, AddAsync, UpdateAsync oraz DeleteAsync, z AsNoTracking w zapytaniach odczytowych dla optymalizacji wydajności.
* Specjalistyczne repozytoria: UzytkownikRepository, AdresyRepository, AlergenyRepository, KategorieRepository, PozycjeMenuRepository, PozycjeZamowieniaRepository, ZamowieniaRepository, każde rozszerzające generyczne repozytorium o specyficzne zapytania, np. GetByUserIdAsync w ZamowieniaRepository.
* UnitOfWork: Klasa zapewniająca spójność operacji poprzez metodę SaveChangesAsync, umożliwiając grupowanie operacji w ramach jednej transakcji.

Repozytoria są rejestrowane w kontenerze DI w pliku Program.cs, np. AddScoped<IRepository<t>, Repository<t>>, co umożliwia wstrzykiwanie zależności do serwisów biznesowych. Warstwa DAL zapewnia efektywny dostęp do danych, zachowując zgodność z zasadami SOLID.</t></t>

## Warstwa IBL

Warstwa interfejsów logiki biznesowej (IBL) w przestrzeni TokyoGarden.IBL definiuje kontrakty dla serwisów logiki biznesowej, wspierając testowanie jednostkowe, modularność oraz zgodność z zasadą Dependency Inversion z zestawu zasad SOLID. Interfejsy te są rejestrowane w kontenerze Dependency Injection (DI) w pliku Program.cs, co ułatwia wstrzykiwanie zależności i wymianę implementacji. Warstwa obejmuje siedem interfejsów, z których każdy odpowiada jednej klasie serwisowej:

1. IAdresyService: Definiuje operacje na adresach, takie jak pobieranie listy adresów, wyszukiwanie po identyfikatorze czy filtrowanie według miasta, wspierając logistykę dostaw.
2. IAlergenyService: Określa funkcje dla alergenów, umożliwiając ich dodawanie, edytowanie oraz usuwanie w celu informowania o zagrożeniach zdrowotnych.
3. IKategorieService: Zarządza kategoriami menu, wspierając tworzenie, edytowanie oraz usuwanie kategorii w celu organizacji oferty kulinarnej.
4. IPozycjeMenuService: Obsługuje pozycje menu, umożliwiając zarządzanie daniami dostępnymi w ofercie restauracyjnej.
5. IPozycjeZamowieniaService: Dotyczy pozycji w zamówieniach, wspierając procesy związane z koszykiem zakupowym i finalizacją zamówień.
6. IUzytkownikService: Zarządza użytkownikami, oferując funkcje rejestracji, logowania, edycji profili oraz autoryzacji użytkowników.
7. IZamowieniaService: Obsługuje zamówienia, umożliwiając ich tworzenie, edytowanie, aktualizację statusu oraz filtrowanie według różnych kryteriów.

Każdy interfejs definiuje asynchroniczne metody, takie jak GetAllAsync, GetByIdAsync, AddAsync, UpdateAsync, DeleteAsync, a niektóre oferują dodatkowe funkcje, np. AuthenticateAsync w IUzytkownikService dla logowania użytkowników. Interfejsy wspierają kontrolery API, rejestrację w DI oraz testy jednostkowe, zapewniając modularność i łatwość utrzymania kodu.

## Warstwa Biznesowa

Warstwa biznesowa w aplikacji TokyoGarden, zlokalizowana w przestrzeni nazw TokyoGarden.BL, odpowiada za realizację logiki aplikacyjnej, obejmującą przetwarzanie danych, walidację, obsługę reguł biznesowych oraz komunikację z warstwą dostępu do danych (DAL). Każdy serwis implementuje interfejs z przestrzeni TokyoGarden.IBL, co gwarantuje modularność, łatwość testowania oraz zgodność z zasadami SOLID, w szczególności z zasadą Dependency Inversion. Warstwa ta obejmuje siedem klas serwisowych, z których każda odpowiada za operacje na określonym typie danych:

1. AdresyService: Zarządza danymi adresowymi, umożliwiając pobieranie, dodawanie, aktualizację oraz usuwanie adresów, a także filtrowanie według miasta, co wspiera logistykę dostaw.
2. AlergenyService: Obsługuje informacje o alergenach, takie jak ich dodawanie, edycja czy usuwanie, co pozwala na informowanie klientów o potencjalnych zagrożeniach zdrowotnych.
3. KategorieService: Odpowiada za zarządzanie kategoriami menu, umożliwiając tworzenie, edytowanie oraz usuwanie kategorii, co ułatwia organizację oferty kulinarnej.
4. PozycjeMenuService: Realizuje operacje na pozycjach menu, takie jak dodawanie nowych dań, edycja istniejących oraz ich usuwanie, co wspiera zarządzanie ofertą restauracji.
5. PozycjeZamowieniaService: Obsługuje pozycje w zamówieniach, umożliwiając dodawanie, edytowanie oraz usuwanie produktów w koszyku, co jest kluczowe dla procesu zamawiania.
6. UżytkownikService: Zarządza danymi użytkowników, w tym rejestracją, edycją profili, usuwaniem kont oraz autoryzacją, co zapewnia bezpieczeństwo i personalizację usług.
7. ZamowieniaService: Realizuje operacje związane z zamówieniami, takie jak tworzenie, edycja, aktualizacja statusu oraz filtrowanie, co wspiera procesy biznesowe restauracji.

Każdy serwis komunikuje się z odpowiadającym mu repozytorium z warstwy DAL (TokyoGarden.IDAL), korzystając z interfejsów takich jak IZamowieniaRepository, co zapewnia separację odpowiedzialności. Przykładowo, AdresyService deleguje operacje do IAdresyRepository i obsługuje funkcje takie jak GetAllAsync, GetByIdAsync, AddAsync, UpdateAsync, DeleteAsync oraz GetByCityAsync, co pozwala na elastyczne zarządzanie danymi adresowymi. Serwisy BL egzekwują reguły biznesowe, takie jak poprawność cen i ilości, obecność kategorii i alergenów, oraz przeliczanie wartości zamówienia, mapując wyjątki domenowe na odpowiedzi HTTP (400, 404, 409).

## API – Web

Kontrolery w projekcie TokyoGarden, zlokalizowane w warstwie API, obsługują komunikację RESTful między frontendem a backendem, realizując operacje CRUD i zapewniając spójność, skalowalność oraz łatwość integracji z innymi warstwami aplikacji. Kluczowe cechy kontrolerów to:

* Atrybut [ApiController]: Automatyczna walidacja modelu, zapobiegająca przetwarzaniu nieprawidłowych danych wejściowych.
* Routing [Route("api/[controller]")]: Spójna struktura adresów URL dla endpointów API, z wersjonowaniem ścieżką (np. /api/v1/pozycje-menu).
* Wstrzykiwanie serwisów: Separacja logiki biznesowej od warstwy prezentacyjnej poprzez zależność od serwisów BL.
* Zwracanie DTO: Konwersja danych na obiekty DTO za pomocą ToDto(), co zapewnia bezpieczne przesyłanie danych.
* Asynchroniczność: Wykorzystanie async/await dla zwiększenia wydajności przetwarzania żądań.

Lista kontrolerów:

1. AdresyController: Zarządza adresami, umożliwiając pobieranie listy, wyszukiwanie po ID, dodawanie, aktualizację, usuwanie oraz filtrowanie według miasta, co wspiera logistykę dostaw.
2. AlergenyController: Obsługuje dane alergenów, umożliwiając ich dodawanie, edytowanie oraz usuwanie, co zapewnia informowanie klientów o zagrożeniach zdrowotnych.
3. KategorieController: Zarządza kategoriami menu, wspierając tworzenie, edytowanie, usuwanie oraz wyszukiwanie po nazwie, co ułatwia organizację oferty.
4. PozycjeMenuController: Realizuje operacje na pozycjach menu, umożliwiając zarządzanie daniami w ofercie restauracyjnej.
5. PozycjeZamowieniaController: Obsługuje pozycje w zamówieniach, wspierając procesy koszyka zakupowego i finalizacji zamówień.
6. UzytkownicyController: Zarządza użytkownikami, obsługując rejestrację, edycję profili, usuwanie kont (z powiązanymi zamówieniami) oraz autoryzację, co zapewnia bezpieczeństwo systemu.
7. ZamowieniaController: Obsługuje zamówienia, umożliwiając tworzenie, edytowanie, aktualizację statusu, usuwanie oraz filtrowanie według użytkownika, statusu lub daty, co wspiera rozliczenia i logistykę.

Przykładowe endpointy:

* GET /api/v1/uzytkownicy: Pobiera listę wszystkich użytkowników w systemie, zwracając ich dane w formacie DTO z paginacją i sortowaniem.
* POST /api/v1/alergeny: Dodaje nowy alergen do bazy danych, zwracając kod 201 (Created) z nagłówkiem Location wskazującym na nowy zasób.
* PUT /api/v1/zamowienia/5: Aktualizuje zamówienie o identyfikatorze 5, uwzględniając status i pozycje zamówienia, z walidacją danych wejściowych.
* DELETE /api/v1/kategorie/4: Usuwa kategorię o identyfikatorze 4, zwracając kod 204 (No Content) po sukcesie, z obsługą błędów integralności.

Kontrolery zapewniają bezpieczeństwo poprzez hashowanie haseł (BCrypt.Net), walidację danych oraz obsługę błędów (kody HTTP: 200, 201, 204, 400, 404, 409). Błędy są zwracane jako ProblemDetails z tytułem, statusem i szczegółami, np. 409 dla konfliktów integralności lub 404 dla nieistniejących zasobów. Endpointy listowe wspierają parametry zapytań (page, pageSize, sortBy, sortDir, filters[]), które BL/DAL mapuje na zapytania LINQ/EF Core.

## DtoMapper.cs

Klasa statyczna DtoMapper w przestrzeni TokyoGarden.Api odpowiada za konwersję encji bazodanowych na obiekty DTO, zapewniając bezpieczne i efektywne przesyłanie danych między backendem a frontendem, minimalizując ryzyko ujawnienia wrażliwych informacji. Wykorzystuje LINQ do przekształcania kolekcji i metody rozszerzeń (this), co poprawia czytelność kodu. Obejmuje siedem metod mapujących:

1. ToDto(Uzytkownicy): Konwertuje encję Uzytkownicy na DTO, zawierające identyfikator, nazwę użytkownika, telefon oraz typ użytkownika, pomijając wrażliwe dane, takie jak hasło.
2. ToDto(Adresy): Przekształca encję Adresy na DTO z danymi lokalizacyjnymi, takimi jak miasto, ulica, numer domu oraz numer mieszkania, co wspiera logistykę dostaw.
3. ToDto(Kategorie): Mapuje encję Kategorie na DTO z identyfikatorem i nazwą kategorii, co ułatwia prezentację kategorii menu w interfejsie użytkownika.
4. ToDto(Alergeny): Przekształca encję Alergeny na DTO z nazwą alergenu, co pozwala na zwięzłe przekazywanie informacji o zagrożeniach zdrowotnych.
5. ToDto(Pozycje\_Menu): Konwertuje encję Pozycje\_Menu na DTO z nazwą, opisem, ceną oraz powiązaną kategorią, stosując zagnieżdżone mapowanie dla pełnych danych.
6. ToDto(Pozycje\_Zamowienia): Przekształca encję Pozycje\_Zamowienia na DTO z ilością, ceną oraz powiązaną pozycją menu, wspierając szczegółowe określenie koszyka.
7. ToDto(Zamowienia): Konwertuje encję Zamowienia na DTO z danymi takimi jak data, status, cena, użytkownik (jako DTO) oraz lista pozycji zamówienia, mapowana za pomocą LINQ.

Metody zawierają zabezpieczenia przed null (np. == null ? null!), zapewniając bezpieczeństwo i eliminując ostrzeżenia kompilatora.

## DTO

Pakiet TokyoGarden.Api.DTOs zawiera klasy DTO, które reprezentują dane w uproszczonej formie, pomijając logikę biznesową i zależności bazodanowe, co zapewnia bezpieczeństwo, czytelność oraz wydajność przesyłania danych między backendem a frontendem. Obiekty DTO są wykorzystywane w kontrolerach API oraz serwisach, wspierając separację warstw i odporność na błędy.

## Frontend – Angular

Frontend systemu TokyoGarden, zbudowany w Angular, opiera się na modularnej strukturze z podziałem na moduły, takie jak admin, shared i core, co ułatwia zarządzanie kodem i jego ponowne wykorzystanie. Wykorzystuje serwisy HTTP do komunikacji z API, interceptory do obsługi błędów oraz formularze reaktywne z walidacjami min/max i wymaganymi polami. Kluczowe elementy to:

* Struktura: Moduł Admin Panel zawiera widoki list i formularzy edycji dla kategorii, alergenów i pozycji menu; moduł shared dostarcza wspólne komponenty i modele.
* UX i walidacje: Pola wymagane są oznaczone, formaty liczbowe stosowane dla cen, komunikaty o błędach wyświetlane przy utracie fokusu lub próbie wysłania formularza. Widoki list wspierają paginację i filtrowanie po kategoriach lub alergenach, a edycja realizowana jest w formularzach dialogowych.
* Mapowanie DTO: Formularze generują DTO dla żądań API, a odpowiedzi API są przekształcane na modele widoków. Aktualizacje realizowane są przez PUT (idempotentne) lub PATCH (częściowe zmiany) w zależności od potrzeb.

## Bezpieczeństwo i zgodność

System TokyoGarden zapewnia bezpieczeństwo poprzez:

* Autoryzacja i role: Wariant MVP obejmuje role Admin/User, zabezpieczone atrybutami autoryzacji w kontrolerach, z planowaną rozbudową o JWT Bearer z krótkim TTL i mechanizmem odświeżania tokenów.
* CORS i ochrona danych: Polityka CORS ograniczona do domen frontendu (np. <http://localhost:4200>), sekrety przechowywane w zmiennych środowiskowych, walidacja rozmiarów uploadów (np. imageData), sanitizacja danych tekstowych.

# 4. Testy (podejście i zakres)

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności

## Zasady ogólne

Testy w systemie TokyoGarden są realizowane przy użyciu frameworka xUnit, stosując nomenklaturę AAA (Arrange–Act–Assert) oraz izolację przypadków testowych, co zapewnia czytelność i niezawodność. Skupiają się na weryfikacji reguł biznesowych, spójności danych oraz poprawności operacji w warstwach BL, DAL i API.

## Doubles własne i Moq

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności.W bibliotece Moq stosuje się Setup/Returns dla konfiguracji zachowań oraz Verify dla weryfikacji interakcji, co pozwala na precyzyjne testowanie zależności.

Testy wykorzystują różne typy obiektów zastępczych:

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności.

Dummy: Parametry obowiązkowe, które nie są używane w teście, np. puste obiekty dla niewpływających na wynik pól.

Stub: Obiekty zwracające deterministyczne dane, np. repozytorium zwracające stałą listę pozycji menu.

Fake: Proste implementacje, np. kolekcje in-memory dla szybkich testów warstwy BL.

Mock: Obiekty weryfikujące interakcje, np. sprawdzenie, czy BL wywołało metodę Save lub Update na repozytorium.

Spy: Obiekty rejestrujące liczbę wywołań lub szczegóły metod dla szczegółowej weryfikacji.

W bibliotece Moq stosuje się Setup/Returns dla konfiguracji zachowań oraz Verify dla weryfikacji interakcji, co pozwala na precyzyjne testowanie zależności.

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności

## Testy DAL z InMemory

Testy kontrolerów obejmują:

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności.

Testy jednostkowe: Weryfikacja statusów HTTP i kontraktów API z zamockowaną warstwą BL, np. sprawdzenie, czy GET /api/v1/uzytkownicy zwraca kod 200 i listę DTO.

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności.

Testy integracyjne: Uruchamianie minimalnego hosta ASP.NET Core, weryfikującego routing, serializację JSON oraz walidację ModelState, co zapewnia poprawność działania endpointów w rzeczywistych warunkach.

Rozbudowano testy o dodatkowe scenariusze: przypadki błędnych danych wejściowych, testy walidacji, testy wydajnościowe sprawdzające zachowanie przy dużej liczbie rekordów. Dodatkowo wprowadzono testy regresyjne aby zapewnić, że nowe zmiany nie psują istniejących funkcjonalności.

# 5. Interfejs Web API (kontrakt i zasady)

Doprecyzowano kontrakty REST – opisano wszystkie pola w DTO, podano przykładowe requesty i response’y dla metod GET, POST i PUT. Uwzględniono również obsługę błędów z kodami statusu i formatem ProblemDetails. Dodano sugestie wersjonowania API w przyszłości (np. /api/v2) w razie rozbudowy systemu

## Konwencje REST

Rzeczowniki w liczbie mnogiej, wersjonowanie ścieżką (np. /api/v1/pozycje-menu), kody statusu zgodne z rezultatem operacji, Location header po POST 201.

## Przykładowe zasoby i operacje

* /api/v1/kategorie: GET (lista z paginacją i sortowaniem), POST (tworzenie kategorii), GET/{id} (szczegóły), PUT/{id} (aktualizacja), DELETE/{id} (usunięcie).
* /api/v1/alergeny: Jak powyżej, z operacjami CRUD dla alergenów.
* /api/v1/pozycje-menu: GET z filtrowaniem po kategoriach i alergenach, operacje CRUD dla pozycji menu.
* /api/v1/zamowienia: GET (lista z filtrowaniem po statusie lub użytkowniku), POST (tworzenie zamówienia z pozycjami), PUT/PATCH (aktualizacja statusu lub szczegółów).
* /api/v1/uzytkownicy: CRUD dla użytkowników, z dodatkowym endpointem /api/v1/uzytkownicy/{id}/adresy dla zarządzania adresami użytkownika.

## Walidacja i błędy

Walidacja DTO jest realizowana na poziomie kontrolerów, z błędami zwracanymi jako ProblemDetails (tytuł, status, szczegóły). Przykłady:

* Kod 400 dla nieprawidłowych danych (np. cena < 0).
* Kod 404 dla nieistniejących zasobów (np. użytkownik o danym ID).
* Kod 409 dla konfliktów integralności (np. próba usunięcia kategorii z powiązanymi pozycjami menu).

## Paginacja/sortowanie/filtrowanie

Endpointy listowe wspierają parametry zapytań, takie jak page, pageSize, sortBy, sortDir, filters[], które są mapowane przez BL i DAL na zapytania LINQ/EF Core, zapewniając elastyczność i wydajność w przetwarzaniu dużych zbiorów danych.

# 6. Frontend (Angular)

UI zostało wzbogacone o szczegółowe formularze z walidacją na żywo, komunikaty o błędach w czasie rzeczywistym oraz responsywny design dostosowany do urządzeń mobilnych. Dodano przykłady interakcji użytkownika i zrzuty ekranu (opisowo) pokazujące przepływ procesu od dodania pozycji menu po realizację zamówienia.

## Struktura i wzorce

Struktura modułowa (np. admin, shared). Serwisy API kapsułkują wywołania HTTP; komponenty korzystają z formularzy reaktywnych (walidacje min/max, wymagane pola).

Doprecyzowano kontrakty REST – opisano wszystkie pola w DTO, podano przykładowe requesty i response’y dla metod GET, POST i PUT. Uwzględniono również obsługę błędów z kodami statusu i formatem ProblemDetails. Dodano sugestie wersjonowania API w przyszłości (np. /api/v2) w razie rozbudowy systemu.

## UX i walidacje

Pola wymagane oznaczone, formaty liczbowe dla cen, komunikaty o błędach przy focus out i submit. Widoki list obsługują paginację, filtrowanie po kategorii/alergenach; edycja w formularzach dialogowych.

## Mapowanie DTO

Formularze → DTO request → API; odpowiedzi API → modele widoków. Idempotencja aktualizacji przez PUT, częściowe zmiany przez PATCH tam, gdzie wskazane.

Doprecyzowano kontrakty REST – opisano wszystkie pola w DTO, podano przykładowe requesty i response’y dla metod GET, POST i PUT. Uwzględniono również obsługę błędów z kodami statusu i formatem ProblemDetails. Dodano sugestie wersjonowania API w przyszłości (np. /api/v2) w razie rozbudowy systemu.

# 7. Bezpieczeństwo i zgodność

## Autoryzacja i role

Wariant MVP: role Admin/User; kontrolery zabezpieczone atrybutami autoryzacji. Docelowo JWT Bearer z krótkim TTL i odświeżaniem.

## CORS i ochrona danych

CORS ograniczony do domen frontendu. Sekrety w zmiennych środowiskowych. Walidacja rozmiarów uploadów (np. image\_data). Sanitizacja danych tekstowych.