

Projekt z rozproszonych i obiektowych systemów baz
danych

Projekt aplikacji wykorzystującej repliki MongoDB

Autorzy:

Maciej Kiedrowski, nr indeksu: 200105

Joanna Piątek, nr indeksu: 199966

Grupa: Poniedziałek 16:10

Data oddania: 23.01.2017

Prowadzący zajęcia: Dr inż. Robert Wójcik, W4/K-9

Ocena pracy:

Spis treści

1	Wstęp	4
1.1	Cele projektu	4
1.2	Założenia projektowe	4
2	Replikacja	5
2.1	Replikacja w MongoDB	5
2.1.1	Zestaw replik	5
2.1.2	Replikacja	6
2.1.3	Wysoka dostępność	6
2.1.4	Dostępność	7
3	Implementacja bazy danych w środowisku MongoDB	8
3.1	Infrastruktura	8
3.1.1	Instancje	8
3.1.2	Sieć wewnętrzna	8
3.1.3	Sieć publiczna	8
3.2	Konfiguracja MongoDB	8
3.3	Schemat bazy danych	10
3.4	Weryfikacja	10
4	Projekt i implementacja aplikacji	11
4.1	Diagram przypadków użycia	11
4.2	Realizacja funkcjonalności	11

Spis rysunków

1	Struktura systemu i schemat komunikacji	4
2	Zestaw replik	5
3	Schemat działania zestawu replik	6
4	Wysoka dostępność z wykorzystaniem procesu <i>failover</i>	7
5	Monitorowanie działania węzłów	7
6	Sieć wirtualna	8
7	Konfiguracja zestawu replik	9
8	Interfejs REST	9
9	Robomongo	10
10	Diagram przypadków użycia	11
11	Model na przykładzie <i>RecipeDto</i>	12
12	Widok edycji dla <i>IngredientsDto</i>	13
13	Kontroler dla kolekcji <i>Ingredients</i> - metody edycji	14
14	Klasa <i>DbContext</i> - bezpośrednie połączenie z bazą danych	15
15	Klasa <i>GenericDataService</i> - operacje <i>dodaj</i> , <i>zmodyfikuj</i> i <i>pobierz kolekcję</i> . . .	16

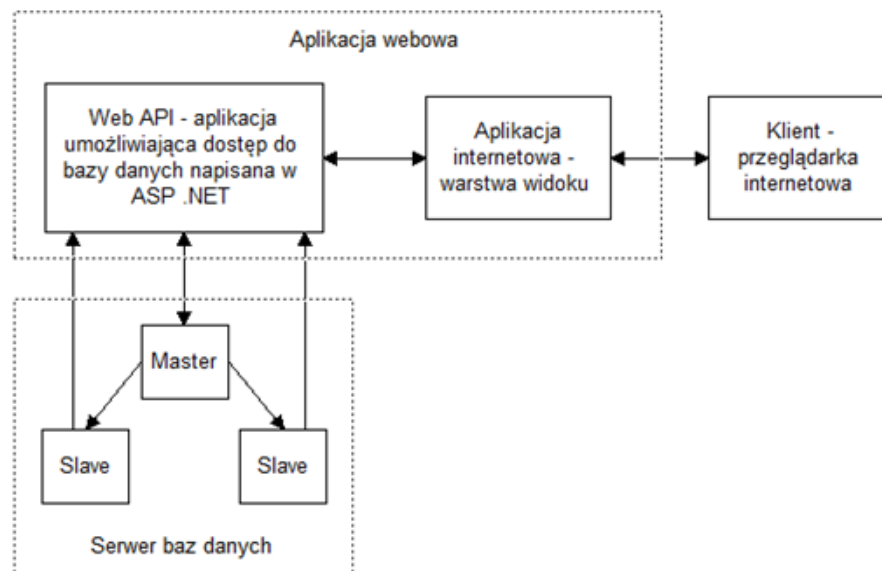
1 Wstęp

Niniejszy projekt realizowany w ramach kursu "Rozproszone i obiektowe systemy baz danych" ma na celu zaprojektowanie oraz implementację rozproszonej aplikacji, odpornej na awarię baz danych, pozwalającej zarządzać stanem magazynowym oraz sprzedażą w sieci restauracji.

1.1 Cele projektu

Celem projektu jest stworzenie systemu pozwalającego zarządzać siecią restauracji. System pozwala na utworzenie nowej restauracji w bazie, utworzenie menu wspólnego dla całej sieci, zarządzania stanem magazynowym poszczególnych restauracji, sprzedaż produktów wpływający na stan magazynowy. Dane o wszystkich restauracjach przechowywane są we wspólnej, centralnej bazie danych - wymaga ona maksymalnej dostępności oraz ochrony danych przed ich utratą.

Dostęp do systemu możliwy jest poprzez dowolną przeglądarkę internetową - użytkownik za pomocą interfejsu webowego posiada możliwość wykonywania operacji na bazie danych za pośrednictwem aplikacji serwerowej. strukturaSystemu



Rysunek 1: Struktura systemu i schemat komunikacji

1.2 Założenia projektowe

Jako silnik bazy danych wybrany został system MongoDB ze względu na skalowalność, wydajność oraz architekturę zaprojektowaną z myślą o łatwej replikacji. Utworzone zostały trzy serwery bazy danych tworzące zestaw replik MongoDB. Webowa aplikacja kliencka został wykonana w oparciu o platformę ASP .NET MVC.

2 Replikacja

Proces replikacji polega na synchronizacji danych pomiędzy serwerami. Pozwala osiągnąć następujące korzyści:

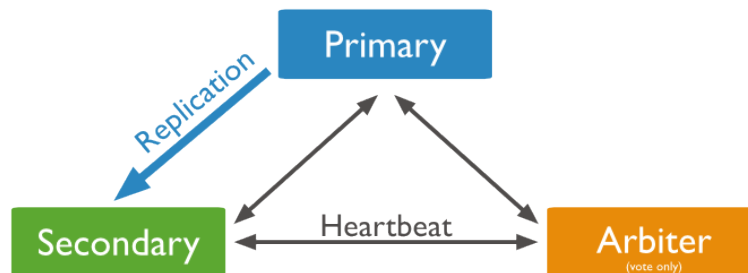
- Bezpieczeństwo danych poprzez redundancję
- Wysoką dostępność
- Skalowanie wydajności

2.1 Replikacja w MongoDB

Replikacja wbudowana w platformę *MongoDB* opiera się na *replica set* - zestawie replik.

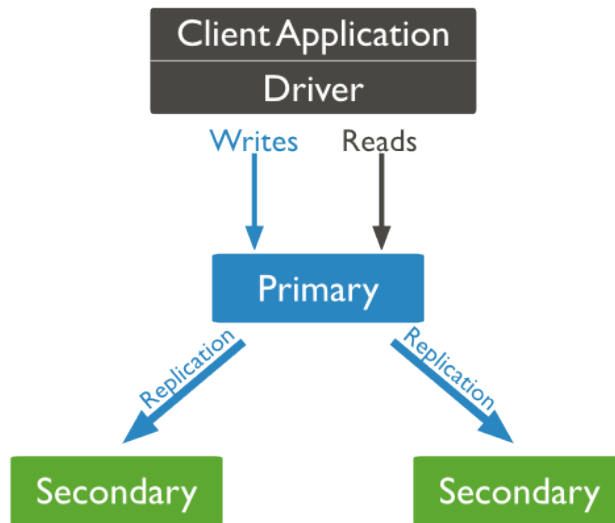
2.1.1 Zestaw replik

Zestaw replik grupa procesów *mongod* utrzymujących ten sam zestaw danych. Zestaw replik w ramach MongoDB składa się z węzłów zawierających dane oraz ewentualnego węzła wspomagającego arbitraż.



Rysunek 2: Zestaw replik

W każdym zestawie replik jeden z węzłów pełni rolę *Primary*. Węzeł ten jest jedynym, który akceptuje operacje zapisu, jest również domyślnym węzłem dla wszystkich operacji odczytu z zestawu replik. Pozostałe węzły wchodzące w skład zestawu, a niebędące węzłem arbitrażowym działają w trybie *Secondary*. Minimalna ilość węzłów zapewniająca poprawną pracę zestawu to 3, ilość węzłów powinna być nieparzysta.



Rysunek 3: Schemat działania zestawu replik

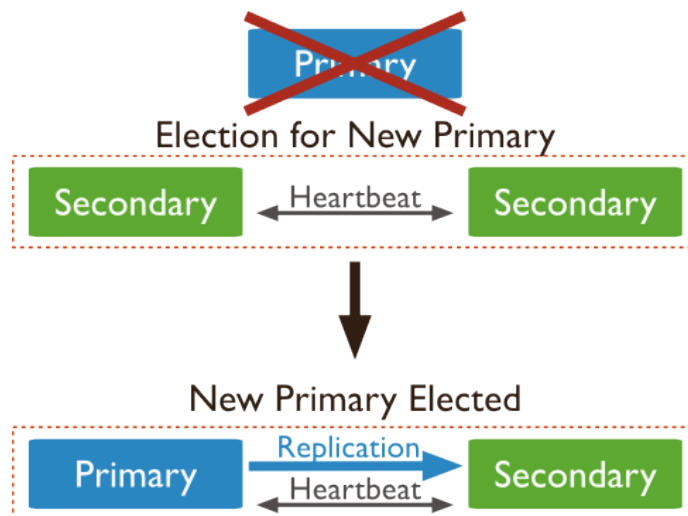
2.1.2 Replikacja

Replikacja w *MongoDB* jest replikacją asynchroniczną. Węzły *Secondary* replikują dziennik operacji (*oplog*) węzła *Primary* i wykonują operacje w nim zawarte na przechowywanym zbiorze danych.

2.1.3 Wysoka dostępność

Zestawie replik gwarantuje wysoką dostępność danych. W przypadku awarii węzła, w szczególności węzła *Primary* uruchamiany jest proces *failover*. Ma on na celu zapewnienie płynności dostępu do danych. Pozostałe działające węzły w ramach repliki przeprowadzają głosowanie nad wyborem nowego *Primary*, po zakończeniu głosowania zestaw odzyskuje pełną sprawność. Proces *failover* trwa przeważnie poniżej minuty, z czego 10-30 sekund to wykrycie awarii węzła *Primary*, następne 10-30 sekund proces głosowania.

W trakcie głosowania zestaw pracuje w trybie *read-only* - żaden z węzłów nie posiada statusu *Primary* a zatem wszystkie żądania zapisu do bazy są odrzucane.



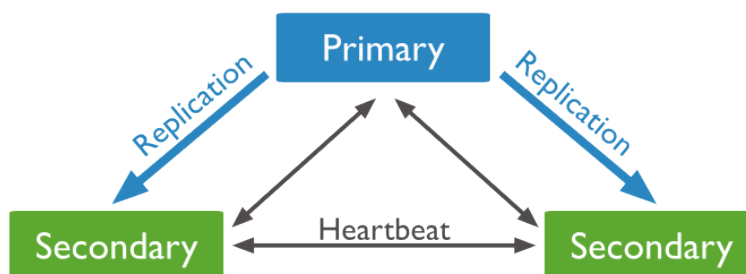
Rysunek 4: Wysoka dostępność z wykorzystaniem procesu *failover*

Warunkiem koniecznym do istnienia węzła *Primary* w zestawie, jest dostępność ponad 50% węzłów. Oznacza to, że dla zestawu składającego się z 3 instancji, jest on w pełni funkcjonalny przy awarii 1 węzła, natomiast zestaw składający się z 5 węzłów pozwala na awarię 2 węzłów.

W przypadku dostępności mniejszej liczby węzłów w zestawie, wszystkie pozostałe przechodzą w tryb *Secondary* - zestaw działa w trybie *read-only*. Mechanizm ten służy zabezpieczeniu danych przed brakiem lub niewystarczającą replikacją w systemie.

2.1.4 Dostępność

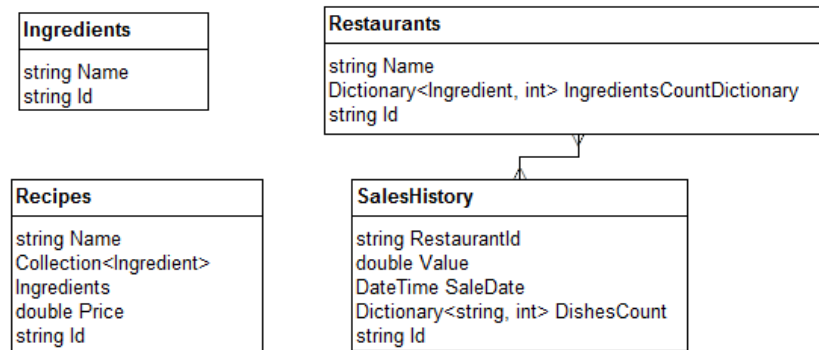
Poprawne działanie węzłów monitorowane jest za pomocą *Heartbeats*. Instancje co 2 sekundy wzajemnie informują się o poprawnym działaniu, brak informacji przez 10 sekund traktowany jest jako awaria węzła.



Rysunek 5: Monitorowanie działania węzłów

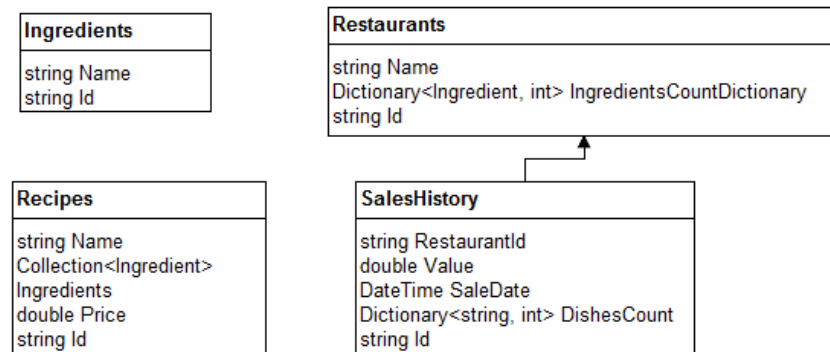
3 Analiza wymagań

3.1 Model konceptualny



Rysunek 6: Model konceptualny bazy danych

3.2 Model fizyczny



Rysunek 7: Model konceptualny bazy danych

4 Implementacja bazy danych w środowisku MongoDB

Celem implementacji było stworzenie zestawu replik składającego się z 3 procesów *mongod* działających na niezależnych instancjach.

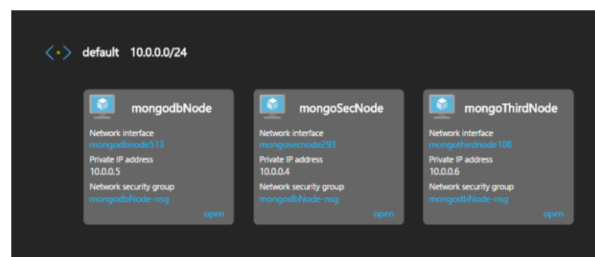
4.1 Infrastruktura

4.1.1 Instancje

Baza danych oparta została o chmurę *Azure* w modelu IaaS - *infrastruktura jako usługa*. Utworzone zostały 3 maszyny wirtualne z wykorzystaniem obrazów dostarczanych przez *Bitnami*. Systemem operacyjnym działającym na maszynach wirtualnych jest *Ubuntu 14.04*, natomiast wersja MongoDB to 3.2.9.

4.1.2 Sieć wewnętrzna

Maszyny wirtualne zostały połączone przez prywatną sieć wirtualną, zilustrowaną poniżej.



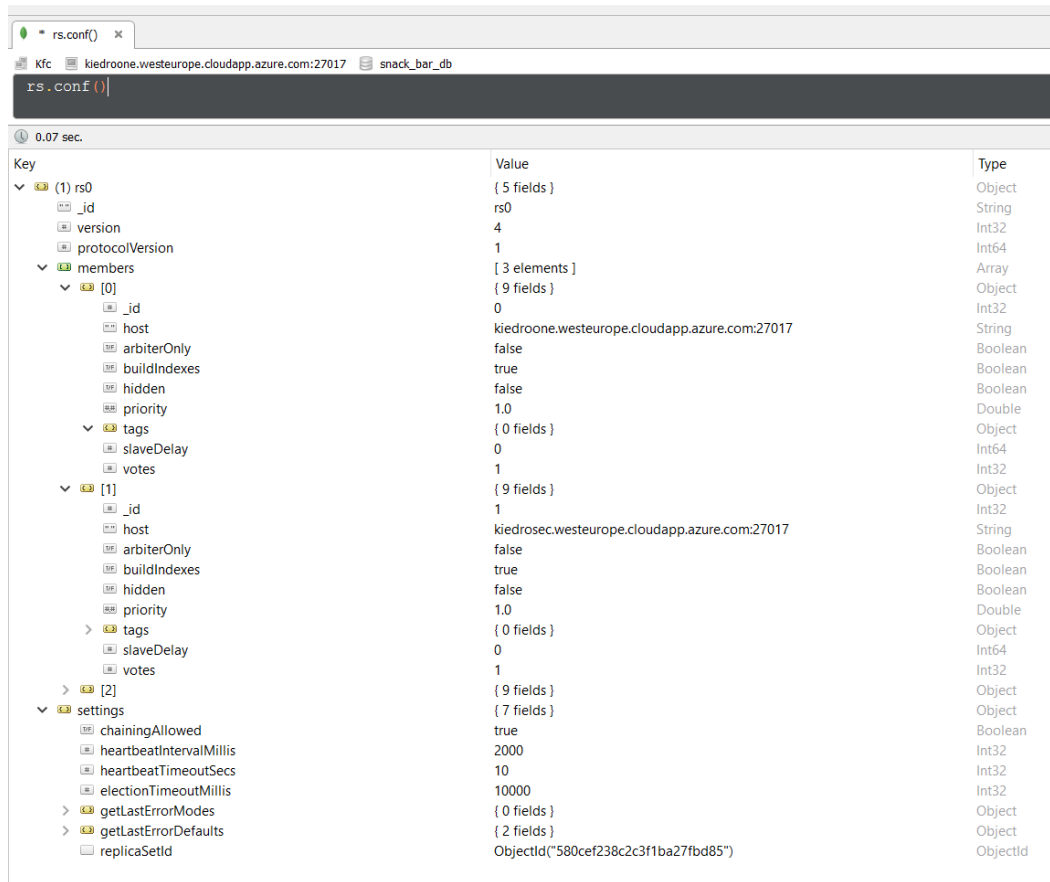
Rysunek 8: Sieć wirtualna

4.1.3 Sieć publiczna

Aby umożliwić połączenie z zestawem replik od dowolnego klienta, bez wymogu jego wdrożenia w chmurze *Azure*, każdej z instancji maszyn wirtualnych został przydzielony publiczny adres IP, oraz nazwa DNS umożliwiająca dostęp do maszyn.

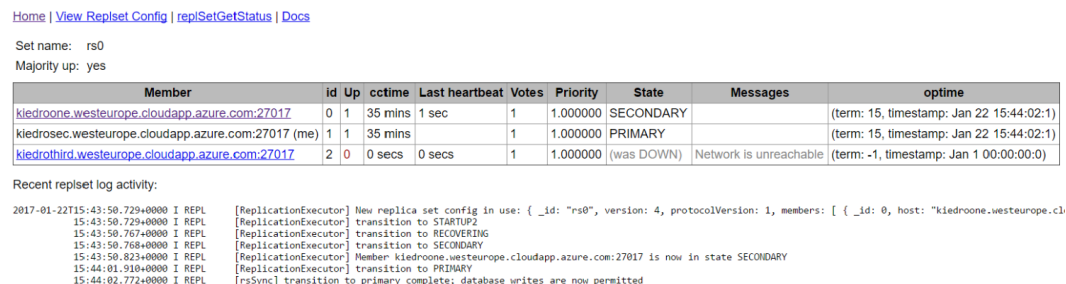
4.2 Konfiguracja MongoDB

Korzystając z *mongo shell* utworzony został zestaw replik oraz jego konfiguracja.



Rysunek 9: Konfiguracja zestawu replik

Korzystając z pliku konfiguracyjnego uaktywiono prosty interfejs REST, pozwalający na łatwą diagnozę zestawu replik.



Rysunek 10: Interfejs REST

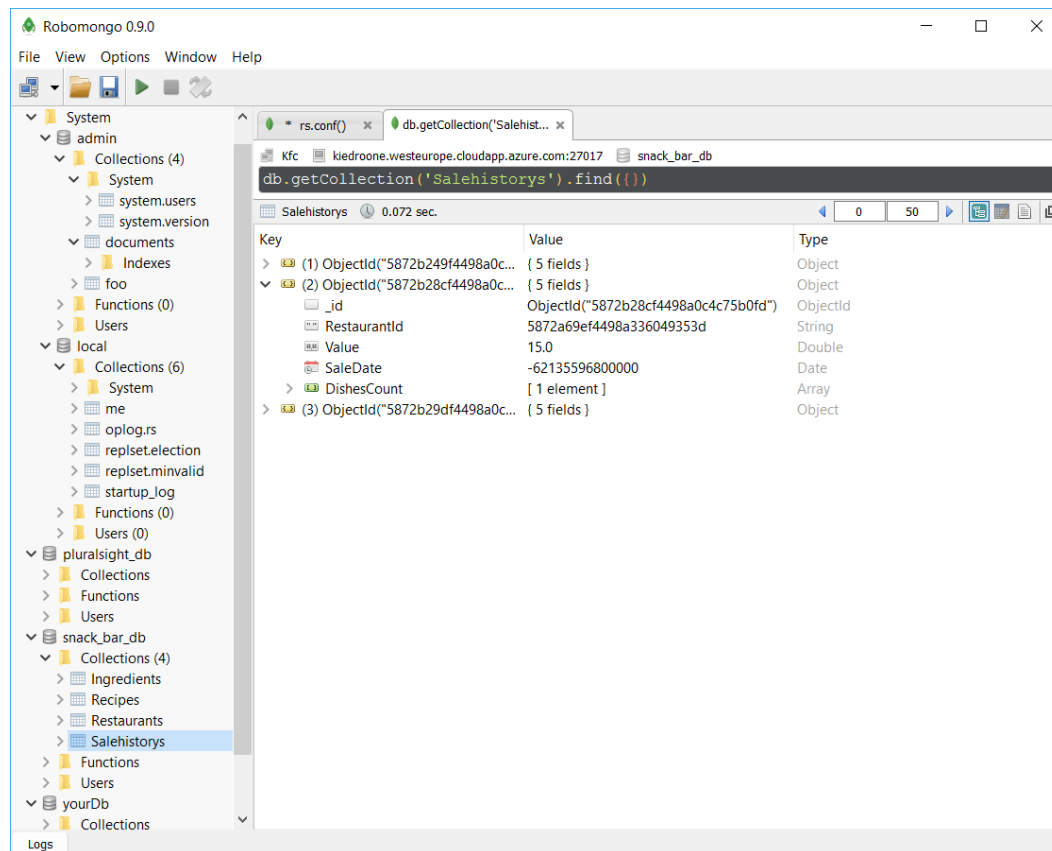
4.3 Schemat bazy danych

MongoDB należy do baz NoSQL typu dokumentowego, jedną z cech charakterystycznych baz tego typu jest brak ustalonego schematu bazy danych. Kolekcje tworzone dynamicznie w trakcie działania aplikacji pozwalają na przechowywanie dowolnych danych - jedynym warunkiem jest możliwość ich serializacji w formacie JSON.

Ta cecha baz NoSQL upraszcza wstępną konfigurację bazy danych.

4.4 Weryfikacja

Do weryfikacji poprawnego działania bazy danych - możliwości połączenia ze wszystkimi węzłami, odczytu, zapisu, działania mechanizmu replikacji posłużyło narzędzie *Robomongo* w wersji 0.9.

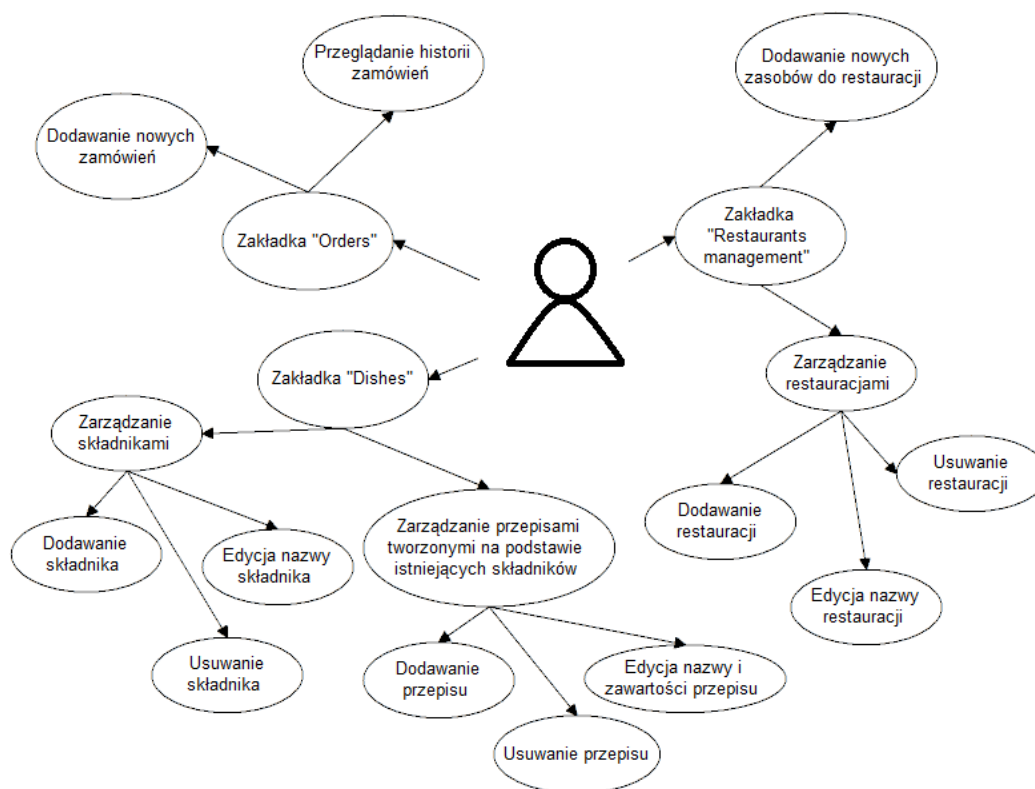


Rysunek 11: Robomongo

5 Projekt i implementacja aplikacji

5.1 Diagram przypadków użycia

Funkcje aplikacji, czyli diagram przypadków użycia przedstawia rysunek 10. Wszystkie z wymienionych funkcjonalności zostały zrealizowane.



Rysunek 12: Diagram przypadków użycia

5.2 Realizacja funkcjonalności

Aplikacja została zrealizowana w technologii ASP.NET MVC w środowisku Visual Studio 2015. Poniżej znajdują się przykłady kodu realizującego funkcje wzorca MVC: modelu, kontrolera i widoku, a także klasa DbContext, odpowiadająca za połączenie z bazą danych oraz klasa GenericDataService, która używa DbContext. Warto wspomnieć, że dwie ostatnie klasy zostały napisane w sposób generyczny, co oznacza, że nie trzeba było powielać implementacji dostępu do bazy danych dla każdej kolekcji z osobna.

Należy zaznaczyć, że baza MongoDB, jako baza dokumentowa, zamiast klasycznych tabel posiada kolekcje dokumentów. W przypadku zapisywania elementu kolekcji, która nie istnieje, zostaje ona utworzona.

```

public class RecipeDto : DtoBase
{
    public string Name { get; set; }
    public ICollection<IngredientDto> Ingredients { get; set; }
    public double Price { get; set; }

    public RecipeDto() { }

    // Konstruktor dla ViewModel-u typu Recipe
    public RecipeDto(Recipe recipe)
    {
        Id = recipe.Id;
        Name = recipe.Name;
        Price = recipe.Price;
        Ingredients = recipe.IngredientsForRecipe;
    }

    // Konstruktor dla ViewModel-u typu RecipeSelect
    public RecipeDto(RecipeSelect select)
    {
        Id = select.Id;
        Name = select.Name;
        Price = select.Price;
        Ingredients = select.Ingredients ?? new List<IngredientDto>();
    }
}

```

Rysunek 13: Model na przykładzie *RecipeDto*

```

@using System.Web.Mvc.Html
@using System.Web.Optimization
@model Models.Dto.IngredientDto

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Ingredient</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.Id)

        <div class="form-group">
            @Html.LabelFor(model => model.Name, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Name, new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Name, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

Rysunek 14: Widok edycji dla *IngredientsDto*

```

public class IngredientsController : Controller
{
    private readonly IIngredientsService _ingredientsService;

    public async Task<ActionResult> Edit(string id)
    {
        if (string.IsNullOrEmpty(id) || id == 0.ToString())
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }

        var ingredient = await _ingredientsService.GetAsync(id);
        if (ingredient == null)
        {
            return HttpNotFound();
        }

        return View(ingredient);
    }

    [HttpPost]
    public async Task<ActionResult> Edit(IngredientDto ingredient)
    {
        if (ModelState.IsValid)
        {
            await _ingredientsService.UpdateAsync(ingredient);
            return RedirectToAction("Index");
        }

        return View(ingredient);
    }
    (...)
}

```

Rysunek 15: Kontroler dla kolekcji *Ingredients* - metody edycji

```

public class MongoClientContext
{
    protected static IMongoClient Client;
    protected static IMongoDatabase Database;
    private const int WriteConcernCount = 2; //Ilość potwierdzeń od baz danych,
                                              //na które aplikacja czeka przy zapisie

    public MongoClientContext()
    {
        // Pobranie connection stringa z ustawień aplikacji
        string mongoHost = ConfigurationManager.ConnectionStrings["Default"].ConnectionString;
        // Stworzenie ustawień dla bazy MongoDB na podstawie connection stringa
        MongoClientSettings settings = MongoClientSettings.FromUrl(new MongoClientUri(mongoHost));
        // Ustawienie ilości potwierdzeń od baz danych
        settings.WriteConcern = new WriteConcern(WriteConcernCount);
        // Utworzenie klienta i pobranie z niego bazy danych
        Client = new MongoClient(settings);
        Database = Client.GetDatabase(Settings.Default.MainDbName);
    }

    public IMongoCollection<T> GetCollection<T>()
    {
        // Przygotowanie nazwy dla kolekcji
        string collectionName = PrepareCollectionName(typeof(T).Name.ToLower());

        try
        {
            // Próba dostępu do danej kolekcji z preferowanym odczytem z bazy Secondary
            return Database.GetCollection<T>(collectionName)
                .WithReadPreference(ReadPreference.SecondaryPreferred) as IMongoCollection<T>;
        }
        catch (TimeoutException)
        {
            var message = "A Timeout Exception occurred./nCheck your database connection./nCheck if almost "
                + WriteConcernCount + " of your databases are working.";
            throw new TimeoutException(message);
        }
    }

    (...)
}

```

Rysunek 16: Klasa *DbContext* - bezpośrednie połączenie z bazą danych


```

public class GenericDataService<T> where T : DtoBase
{
    protected MongoDBContext MongoDBContext;

    public async Task AddAsync(T entity)
    {
        await MongoDBContext
            .GetCollection<T>()
            .InsertOneAsync(entity);
    }

    public async Task UpdateAsync(T entity)
    {
        await MongoDBContext
            .GetCollection<T>()
            .ReplaceOneAsync(i => i.Id == entity.Id, entity);
    }

    public async Task<List<T>> GetAllAsync()
    {
        var result = await MongoDBContext.GetCollection<T>()
            .FindAsync(_ => true);

        return result.ToList();
    }

    (...)
}

```

Rysunek 17: Klasa *GenericDataService* - operacje *dodaj*, *zmodyfikuj* i *pobierz kolekcję*