Rzeszów, 2016

Dokumentacja projektu z przedmiotu

Aplikacje internetowe

Perykasza Sylwia

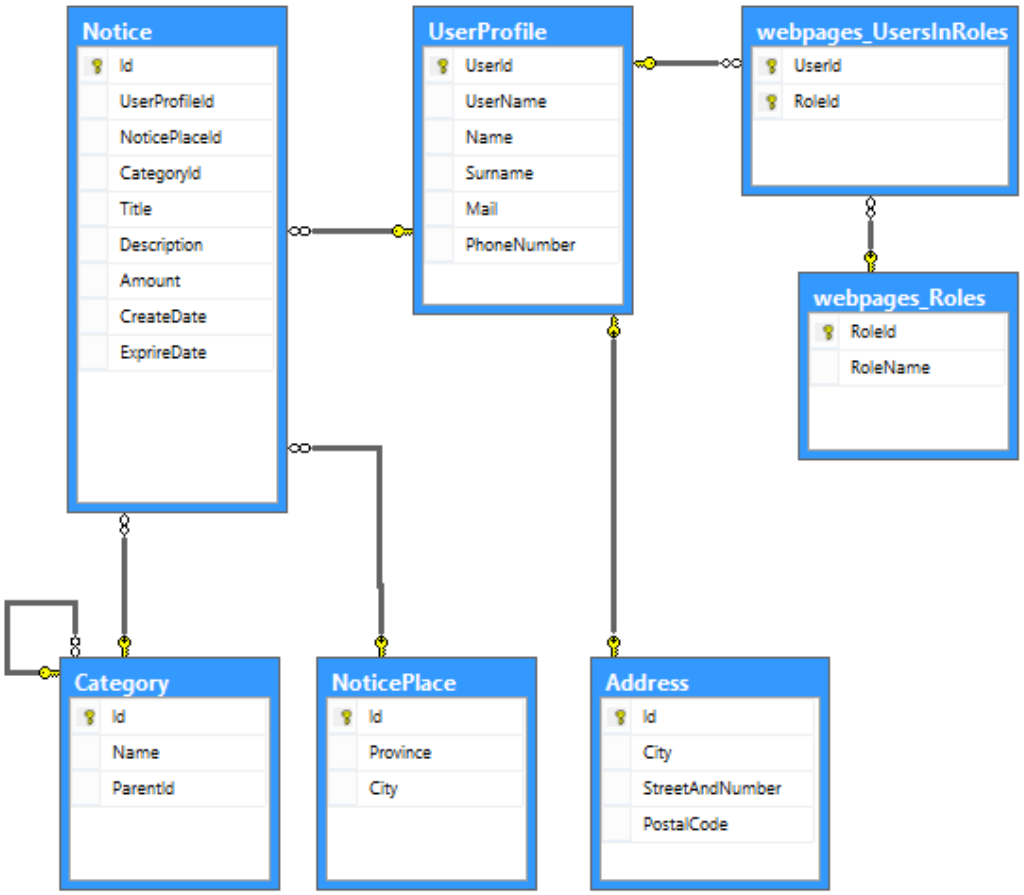
L 03

1. Tworzenie bazy danych metodą *code first*.

Metoda *code first* polega na generowaniu bazy danych za pomocą kodu C# i *Entity Framework*. W projekcie zastosowano tą metodę ze względu na prostotę tworzenia bazy poprzez kod.

1.1 Modele bazodanowe

Model bazodanowy jest to klasa, która odzwierciedla strukturę tabeli. Na poniższym rysunku przedstawiono strukturę bazy danych. Dla przykładu zostaną opisane tabele wraz z relacjami typu: jeden do wielu (UserProfile do Notice), wiele do jeden (Notice do UserProfile), jeden do jeden (UserProfile do Address).



Rys. Diagram ERD

```
10      [Table("Notice")]
        14 references
11      public class NoticeEntity
12      {
13          [Key]
            8 references
14          public int Id { get; set; }
            4 references
15          public int UserProfileId { get; set; }
            6 references
16          public int NoticePlaceId { get; set; }
            7 references
17          public int CategoryId { get; set; }
            8 references
18          public string Title { get; set; }
            12 references
19          public string Description { get; set; }
            8 references
20          public decimal Amount { get; set; }
            6 references
21          public DateTime CreateDate { get; set; }
            5 references
22          public DateTime ExprireDate { get; set; }
23
            1 reference
24          public virtual UserProfileEntity UserProfile { get; set; }
            2 references
25          public virtual CategoryEntity Category { get; set; }
            2 references
26          public virtual NoticePlaceEntity NoticePlace { get; set; }
27      }
```

Klasa NoticeEntity jest odpowiednikiem tabeli Notice. Informuje nas o tym adnotacja klasy `[Table("Notice")]`. Za pomocą tej adnotacji wskazujemy nazwę tabeli, która ma być powiązana z tą klasą. Za pomocą atrybutu [Key] określamy, która właściwość klasy jest kluczem głównym – 13, 14 linia kodu.

Do tworzenia relacji pomiędzy modelami stosuje się pola wirtualne. Linia 24 informuje nas o tym, że klasa NoticeEntity jest powiązana z klasą UserProfile.

```
55      [Table("UserProfile")]
        8 references
56      public class UserProfileEntity
57      {
            1 reference
58          public UserProfileEntity()
59          {
60              Notices = new List<NoticeEntity>();
61          }
62          [Key]
63          [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]
            0 references
64          public int UserId { get; set; }
            2 references
65          public string UserName { get; set; }
            0 references
66          public string Name { get; set; }
            0 references
67          public string Surname { get; set; }
            0 references
68          public string Mail { get; set; }
            0 references
69          public string PhoneNumber { get; set; }
70
            2 references
71          public virtual ICollection<NoticeEntity> Notices { get; set; }
            1 reference
72          public virtual AddressEntity Address { get; set; }
73      }
```

Relacja jeden do wielu pomiędzy tabelą *UserProfile* a *Notice* – w klasie *UserProfileEntity* mamy pole wirtualne *Notice*, które mówi nam o powiązaniu jeden do wielu z klasą *NoticeEntity*, ponieważ jest typu `ICollection<NoticeEntity>`. W klasie *NoticeEntity* jest również powiązanie z klasą *UserProfileEntity*.

```
10      [Table("Address")]
        3 references
11      public class AddressEntity
12      {
13          [Key]
            0 references
14          public int Id { get; set; }
            0 references
15          public string City { get; set; }
            0 references
16          public string StreetAndNumber { get; set; }
            0 references
17          public string PostalCode { get; set; }
18
            1 reference
19          public virtual UserProfileEntity UserProfile { get; set; }
20      }
```

Relacja jeden do jeden pomiędzy tabelą *UserProfile* a *Address* – w klasie *UserProfileEntity* mamy pole wirtualne *Address*, które mówi nam o powiązaniu jeden do jeden z klasą *AddressEntity*, które jest typu *AdressEntity*. Klasa *AddressEntity* posiada pole wirtualne *UserProfile* typu *UserProfileEntity*, które mówi nam o powiązaniu w drugą stronę (adresu do profilu użytkownika).

1.2 Relacje pomiędzy tabelami – mapping

Mapping jest to mechanizm do wskazywania powiązań (relacji) pomiędzy tabelami, za pomocą klas, które są odpowiednikami tabel w bazie danych(opisane w poprzednim podpunkcie). Klasa *UsersContext* służy do komunikacji z bazą danych. W bezparametrowym konstruktorze klasy *UsersContext* przekazywany jest parametr do konstruktora klasy *DbContext*. Parametr *DefaultConnection* jest nazwą *ConnectionString'a*, który znajduje się w pliku Web.config w sekcji *configuration*.

ConnectionString użyty w projekcie:

```
<configuration>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=SYLWIA\sqlexpress2012;Initial Catalog=NoticeBoard;Integrated Security=True"
providerName="System.Data.SqlClient" />
```

```
12  public class UsersContext : DbContext
13  {
        10 references
14      public UsersContext()
15          : base("DefaultConnection")
16      {
17      }
18
        2 references
19      public DbSet<UserProfileEntity> UserProfiles { get; set; }
        1 reference
20      public DbSet<CategoryEntity> Categories { get; set; }
        0 references
21      public DbSet<AddressEntity> Addresses { get; set; }
        6 references
22      public DbSet<NoticeEntity> Notices { get; set; }
        1 reference
23      public DbSet<NoticePlaceEntity> NoticePlaces { get; set; }
24
```

Aby odnieść się do danych z bazy musimy w kontekście dodać pola, które będą wskazywać na tabele w bazie danych. Pola określające tabele muszą być typu *DbSet<NazwaKlasy>*.

```
25      protected override void OnModelCreating(DbModelBuilder modelBuilder)
26      {
27
28          modelBuilder.Entity<NoticeEntity>()
29              .HasRequired<UserProfileEntity>(x => x.UserProfile)
30              .WithMany(x => x.Notices)
31              .HasForeignKey(x => x.UserProfileId);
32
33          modelBuilder.Entity<NoticeEntity>()
34              .HasRequired<NoticePlaceEntity>(x => x.NoticePlace)
35              .WithMany(x => x.Notices)
36              .HasForeignKey(x => x.NoticePlaceId);
37
38          modelBuilder.Entity<NoticeEntity>()
39              .HasRequired<CategoryEntity>(x => x.Category)
40              .WithMany(x => x.Notices)
41              .HasForeignKey(x => x.CategoryId);
42
43          modelBuilder.Entity<CategoryEntity>()
44              .HasOptional<CategoryEntity>(x => x.ParentCategory)
45              .WithMany(x => x.CategoryItem)
46              .HasForeignKey(x => x.ParentId);
47
48          modelBuilder.Entity<AddressEntity>()
49              .HasRequired<UserProfileEntity>(x => x.UserProfile)
50              .WithOptional(x => x.Address);
51
52      }
53  }
```

W metodzie *OnModelCreating* tworzymy *mapping*, to znaczy określamy relacje pomiędzy tabelami. W liniach 38-41 określona została relacja jeden do wielu pomiędzy tabelą *Category* a *Notice*. W 38 linii wskazujemy na encję, dla której będziemy budować powiązanie. W 39 linii określamy jakiego typu ma być dana relacja (*Notice* do *Category*) – w tym przypadku obowiązkowa, ponieważ została użyta metoda *HasRequired*, w której parametrze wskazujemy

na pole, dzięki któremu będziemy mogli odwołać się do powiązanej tabeli (powiązanych danych). W 40 linii określamy jakiego typu ma być dana relacja (*Category* do *Notice*) – w tym przypadku jedna kategoria może mieć powiązanie do wielu ogłoszeń. W 41 linii wskazujemy, które pole ma być kluczem obcym.

2. Opis architektury programu na podstawie funkcjonalności Dodawanie ogłoszenia

2.1 Komunikacja pomiędzy controllerem i modelem – pobieranie danych z bazy

W projekcie stworzono dodatkową warstwę pośrednią pomiędzy controllerem a modelem – repozytorium. Pobieranie danych w controllerze odbywa się poprzez wywołanie odpowiedniej metody z repozytorium – controller nie ma bezpośredniego dostępu do bazy i nie buduje modelu.

W pierwszej kolejności tworzymy obiekt typu *CategoryRepository* i pobieramy listę wszystkich kategorii wywołując metodę *GetAllCategory()*. Na tej samej zasadzie, za pomocą odpowiedniej metody pobieramy listę miejscowości. Tworzenie instancji repozytorium znajduje się w liniach 29 i 31, natomiast obieranie danych w liniach 30 i 32. W liniach 36 – 51 odbywa się mapowanie listy kategorii i miejscowości na obiekty listy rozwijalnej, akceptowanej na widoku przez silnik Razor.

```
27          public ActionResult AddNotice()
28          {
29              var categoryRepo = new CategoryRepository();
30              var allCategory = categoryRepo.GetAllCategory();
31              var noticePlaceRepo = new NoticePlaceRepository();
32              var allNoticePlace = noticePlaceRepo.GetAllNoticePlace();
33
34              var result = new NewNoticeViewModel();
35
36              var categoryListTemporary = new List<SelectListItem>();
37              foreach (var category in allCategory)
38              {
39                  categoryListTemporary.AddRange(category.SubCategory.Select(x => new SelectListItem
40                  {
41                      Text = string.Format("{0}, {1}", category.Name, x.Name),
42                      Value = x.Id.ToString()
43                  }));
44              }
45              result.CategoryList = categoryListTemporary;
46
47              result.NoticePlaceList = allNoticePlace.Select(x => new SelectListItem
48              {
49                  Text = string.Format("{0}, {1}", x.Province, x.City),
50                  Value = x.Id.ToString()
51              }).OrderBy(x => x.Text);
52              return View(result);
53          }
```

Wszystkie metody repozytorium komunikują się z bazą danych, nie zwracają one modeli bazy danych, tylko przemapowane obiekty. Na przykładzie metody *GetAllCategory()* opisana zostanie komunikacja z bazą danych.

W 14 linii tworzony jest kontekst bazy danych, za pomocą zmiennej *contex* będziemy komunikować się z bazą danych. W 16 linii pobieramy wszystkie kategorie poprzez odwołanie się do tabeli *Categories* w zmiennej *contex*. W zapytaniu została użyta klauzula *Where()*, w której poprzez wyrażenie *lambda* podajemy warunek, aby wyfiltrować żądane wyniki. W 17 linii pobrane wcześniej dane są mapowane za pomocą metody *Select()* na obiekty docelowe – *CategoryRepo*. W 21 linii również wywołana jest metoda *Select()* po to, aby przemapować podkategorie. W 28 linii zwracana jest lista wszystkich kategorii.

```
10  public class CategoryRepository
11  {
        4 references
12      public List<CategoryRepo> GetAllCategory()
13      {
14          using (var contex = new UsersContext())
15          {
16              var allCategory = contex.Categories.Where(x => x.ParentId == null).ToList();
17              var result = allCategory.Select(x => new CategoryRepo
18              {
19                  Id = x.Id,
20                  Name = x.Name,
21                  SubCategory = x.CategoryItem.Select(y => new CategoryRepo
22                  {
23                      Id = y.Id,
24                      Name = y.Name
25                  }).ToList()
26              }).ToList();
27
28              return result;
29          }
30      }
31  }
```

2.2 Komunikacja pomiędzy controllerem i widokiem – zwracanie modelu na widok, akcja zapisu ogłoszenia

Metoda *AddNotice()* zwraca obiekt typu *NewNoticeViewModel*, którego budowanie zostało opisane w poprzednim punkcie.

```
1   @model NoticeBoard.Models.ViewModels.NewNoticeViewModel
2
3   @{
4       ViewBag.Title = "AddNotice";
5   }
6
7   <h2>AddNotice</h2>
8   @using (Html.BeginForm("AddNoticeToDatabase", "UserNotices"))
9   {
10      @Html.Partial("_NoticeForm")
11  }
```

W pierwszej linii znajduje się deklaracja typu modelu, który będzie wykorzystywany na widoku. Dane do tego modelu budowane są w kontrolerze. W 8 linii znajduje się deklaracja formularza. Parametry, które podajemy podczas tworzenia formularza wskazują na kontroler i akcję kontrolera, która będzie wywołana po zatwierdzeniu formularza. W 10 linii renderujemy partiala (widok częściowy). Używając widoku częściowego uniknęliśmy powielenia kodu – ten partial używany jest również podczas edycji ogłoszenia.

```
1   @model NoticeBoard.Models.ViewModels.NewNoticeViewModel
2   @Html.HiddenFor(model => model.Id)
3
4   <div class="form-group">
5       <label for="exampleInputEmail1">Tytuł</label>
6       @Html.TextBoxFor(m => m.Title, new { @class = "form-control", placeholder = "Tytuł" })
7   </div>
```

W pierwszej linii znajduje się deklaracja modelu, który będzie używany na widoku częściowym. W drugiej linii znajduje się deklaracja ukrytego pola do przechowywania Id ogłoszenia – jest ono potrzebne podczas edycji ogłoszenia. W liniach 4-7 znajduje się przykładowe pole formularza – w tym przypadku jest to tytuł. Do stylizacji formularza zostały użyte klasy Bootstrap. W 6 linii renderowane jest pole formularza za pomocą silnika Razor. W pierwszym parametrze *TextBoxFor()* za pomocą wyrażenia lambda „podpinamy" się do konkretnej właściwości modelu `m => m.Title`. W drugim opcjonalnym parametrze możemy zdefiniować dodatkowe właściwości html, w tym przypadku jest to nadanie klasy *form-control* oraz dodanie atrybutu *placeholder*.

```
27 <div class="form-group">
28     <label for="exampleInputEmail1">Kategoria</label>
29     @Html.DropDownListFor(model => Model.SelectedCategoryId, Model.CategoryList, new { @class = "form-control" })
30
31 </div>
32 <div class="form-group">
33     <label for="exampleInputEmail1">Miejscowość</label>
34     @Html.DropDownListFor(model => Model.NoticePlaceId, Model.NoticePlaceList, new { @class = "form-control" })
35 </div>
36  <button type="submit" class="btn btn-default">Dodaj</button>
```

Do generowania listy rozwijalnej (29 i 34 linia) z kategoriami i miejscowościami użyto *DropDownListFor*. W pierwszym parametrze wskazujemy które pole w modelu ma przechowywać wybraną wartość, w drugim parametrze wskazujemy właściwość modelu, z którego mają być pobrane dane do wyświetlenia. Zmienne *Model.CategoryList* oraz *Model.NoticePlaceList*, które przechowują dane do wyświetlenia muszą być typu *List<SelectListItem>*. Przygotowanie tych obiektów zostało przedstawione w poprzednim podpunkcie.

```
54         public ActionResult AddNoticeToDatabase(NewNoticeViewModel model)
55         {
56             try
57             {
58                 var newNoticeToAdd = new NoticeEntity();
59                 newNoticeToAdd.Amount = decimal.Parse(model.Amount);
60                 newNoticeToAdd.CategoryId = int.Parse(model.SelectedCategoryId);
61                 newNoticeToAdd.CreateDate = DateTime.Now;
62                 newNoticeToAdd.Description = model.Description;
63                 newNoticeToAdd.ExprireDate = DateTime.Now.AddDays(int.Parse(model.ExpireTo));
64                 newNoticeToAdd.NoticePlaceId = int.Parse(model.NoticePlaceId);
65                 newNoticeToAdd.Title = model.Title;
66                 newNoticeToAdd.UserProfileId = WebMatrix.WebData.WebSecurity.CurrentUserId;
67
68                 var noticeRepo = new NoticeRepository();
69                 noticeRepo.AddNewNotice(newNoticeToAdd);
70                 ViewBag.Message = "Dodano ogłoszenie.";
71             }
72             catch (Exception ex)
73             {
74                 ViewBag.Message = "Ups coś poszło nie tak.";
75                 ViewBag.ExtendMessage = ex.Message;
76
77             }
78             return View();
79         }
```

3. Omówienie funkcjonalności: wyświetlanie, edycja, usuwanie ogłoszenia.

Aby wyświetlić wszystkie ogłoszenia zalogowanego użytkownika w controllerze tworzymy obiekt typu *NoticeRepository* i wywołujemy metodę *GetNoticeByUserID()*.

```
18          public ActionResult Index()
19          {
20              var noticeRepo = new NoticeRepository();
21              var noticesList = new List<NoticeRepo>();
22
23              noticesList = noticeRepo.GetNoticesByUserId();
24
25              return View(noticesList);
26          }
```

Samo pobieranie listy ogłoszeń odbywa się w podobny sposób jak opisany wcześniej przykład pobierania listy kategorii. Dodatkowo, jeśli chcemy wyświetlić listę ogłoszeń dodanych przez zalogowanego użytkownika musimy użyć odpowiednio skonstruowanego zapytania użytego w linii 39. Następnie mapujemy obiekty – podobnie, jak we wcześniej przedstawionym przykładzie.

```
34          public List<NoticeRepo> GetNoticesByUserId()
35          {
36              using (var context = new UsersContext())
37              {
38
39                  var allNotices = context.Notices.Where(x => x.UserProfileId == WebSecurity.CurrentUserId).ToList();
40                  var result = allNotices.Select(x => new NoticeRepo()
41                  {
42                      Amount = x.Amount,
43                      CreateDate = x.CreateDate,
44                      Description = x.Description,
45                      ExprireDate = x.ExprireDate,
46                      Id = x.Id,
47                      Title = x.Title
48
49                  }).ToList();
50
51                  return result;
52              }
53          }
54
```

Wyświetlanie ogłoszeń odbywa się w taki sam sposób jak wyżej:

```
1   @model List<NoticeBoard.Models.RepositoryModels.NoticeRepo>
2   @{
3       ViewBag.Title = "Twoje ogłoszenia";
4   }
5
6   <h2>Twoje ogłoszenia</h2>
7
8   <div class="col-sm-12">
9
10      @foreach (var notice in Model)
11      {
12          <div class="panel panel-default">
13              <div class="panel-heading">@notice.Title <span style="float: right;"> data dodania: @notice.CreateDate</span></div>
14              <div class="panel-body">
15                  @notice.Description
16
17                  <a href="@Url.Action("EditNotice", "UserNotices", new { noticeId = notice.Id }) ">
18                      <b>Edytuj</b>
19                  </a>
20              </div>
21          </div>
22      }
23  </div>
```