

Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

Streszczenie

Operacje na bazie danych za pomocą Entity Framework Core

asp.net Core

Operacja na bazie danych

**Operacje na bazie danych**

Będziemy pracować na ApplicationDbContex, ale w wersji core będzie to trochę inaczej wyglądać.  
Tutaj mamy już zaimplementowany mechanizm dependency injection.  
Dzięki temu nie będziemy w tym miejscu w klasie TaskRepository nowej instancji ApplicationDbContex, zostanie ona nam przekazana poprzez konstruktor w naszym request.

W startup.cs mamy zdefiniowane miejsce gdzie dodajemy DbContext, który używa  
connectionstring’a o kluczu DefaultConnection

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddDbContext<ApplicationDbContext>(options =>  options.UseSqlServer(  Configuration.GetConnectionString("DefaultConnection")));  services.AddDatabaseDeveloperPageExceptionFilter();  services.AddDefaultIdentity<ApplicationUser>(options => options.SignIn.RequireConfirmedAccount = true)  .AddEntityFrameworkStores<ApplicationDbContext>();  services.AddControllersWithViews();  } |

Dzięki temu do naszego repozytorium może być przekazany ten kontekst w request’cie.

|  |
| --- |
| namespace MyTasks.Controllers  {  [Authorize]  public class TaskController : Controller  {  private TaskRepository \_taskRepository = new TaskRepository();  private CategoryRepository \_categoryRepository = new CategoryRepository(); |

W **Kontrolerze** tworzymy konstruktor, który będzie oczekiwał parametru typu ApplicationDbContext

|  |
| --- |
| namespace MyTasks.Controllers  {  [Authorize]  public class TaskController : Controller  {  private TaskRepository \_taskRepository;  private CategoryRepository \_categoryRepository;  public TaskController(ApplicationDbContext context)  {  \_taskRepository = new TaskRepository(context);  \_categoryRepository = new CategoryRepository(context);  } |

Podobnie w repozytorium tworzymy pole prywatne \_context typu ApplicationDbContext.  
Pole to inicjujemy w konstruktorze

|  |
| --- |
| namespace MyTasks.Persistence.Repositories  {  public class TaskRepository  {  private ApplicationDbContext \_context;    public TaskRepository(ApplicationDbContext context)  {  \_context = context;  } |

Teraz możemy pracować na tym polu \_context, i dzięki temu, że mamy zastosowany mechanizm DI, nie musimy się nawet martwić o wywołanie metody Dispose, nie musimy wywoływać metod na kontekście używając składni using.

Na ten moment musimy wiedzieć, że nowa instancja ApplicationDbContext zostanie przekazana do konstruktora w kontrolerze, a następnie do TaksRepository i nie musimy wywoływać metody Dispose na tym obiekcie.

Metoda Get

|  |
| --- |
| public IEnumerable<Task> Get(string userId,  bool isExecuted = false,  int categoryId = 0,  string title = null)  {  // task jest typu **IQuerable**  var tasks = \_context.Tasks  .Include(x => x.Category)  .Where(x => x.UserId == userId && x.IsExecuted == isExecuted);  if (categoryId != 0)  tasks = tasks.Where(x => x.CategoryId == categoryId);  if (!string.IsNullOrWhiteSpace(title))  tasks = tasks.Where(x => x.Title.Contains(title));  return tasks.OrderBy(x => x.Term) .ToList();;  } |

Jak najedziemy na słowo var przed tasks, to zobaczymy, że zapytanie zwróci nam typ IQuerable.  
IQuerable mówi nam, że to zapytanie w tym miejscu jeszcze nie zostanie wywołane.  
Zostanie wykonane dopiero t momencie zamiany na liste przez metodę ToList().  
Dopiero gdy dodamy sobie wszystkie filtrowania w momencie wywołania ToList() całe to zapytanie zostanie zamienione na SQL’a

Co innego gdybyśmy zapisali to tak

|  |
| --- |
| public IEnumerable<Task> Get(string userId,  bool isExecuted = false,  int categoryId = 0,  string title = null)  {  // task jest typu **IEnumerable**  var tasks = \_context.Tasks  .Include(x => x.Category)  .Where(x => x.UserId == userId && x.IsExecuted == isExecuted).AsEnumerable();  if (categoryId != 0)  tasks = tasks.Where(x => x.CategoryId == categoryId);  if (!string.IsNullOrWhiteSpace(title))  tasks = tasks.Where(x => x.Title.Contains(title));  return tasks.OrderBy(x => x.Term) .ToList();;  } |

Wówczas zapytanie do bazy zostało by już wykonane przy pierwszym zapytaniu LINQ.  
Potem było by również wykonane przy drugim i trzecim wyrażeniu.

Kod repozytorium: TaskRepository

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using MyTasks.Core.Models.Domains;  using System;  using System.Collections.Generic;  using System.Linq;  namespace MyTasks.Persistence.Repositories  {  public class TaskRepository  {  private ApplicationDbContext \_context;  public TaskRepository(ApplicationDbContext context)  {  \_context = context;  }  public IEnumerable<Task> Get(string userId,  bool isExecuted = false,  int categoryId = 0,  string title = null)  {  var tasks = \_context.Tasks  .Include(x => x.Category)  .Where(x => x.UserId == userId && x.IsExecuted == isExecuted);  if (categoryId != 0)  tasks = tasks.Where(x => x.CategoryId == categoryId);  if (!string.IsNullOrWhiteSpace(title))  tasks = tasks.Where(x => x.Title.Contains(title));  return tasks.OrderBy(x => x.Term).ToList();  }  // jak użyjemy Single() to gdy zostanie zwrócona inna liczba rekordów niż 1  // to zostanie rzucony wyjątek  // Jak użyjemy First lub FirstOrDefault, to jeżeli nie było by rekordu to zostanie zwrócony null  public Task Get(int id, string userId)  {  var task = \_context.Tasks.Single(x => x.Id == id && x.UserId == userId);  return task;  }  public void Add(Task task)  {  \_context.Add(task);  \_context.SaveChanges();  }  public void Update(Task task)  {  var taskToUpdate = \_context.Tasks.Single(x => x.Id == task.Id);  taskToUpdate.Title = task.Title;  taskToUpdate.Description = task.Description;  taskToUpdate.Term = task.Term;  taskToUpdate.IsExecuted = task.IsExecuted;  taskToUpdate.CategoryId = task.CategoryId;  \_context.SaveChanges();  }  public void Finish(int id, string userId)  {  var taskToUpdate = \_context.Tasks.Single(x => x.Id == id);  taskToUpdate.IsExecuted = true;  \_context.SaveChanges();  }  public void Delete(int id, string userId)  {  var taskToDelete = \_context.Tasks.Single(x => x.Id == id);  \_context.Remove(taskToDelete);  \_context.SaveChanges();  }  }  } |

Kod repozytorium: TaskRepository

|  |
| --- |
| using MyTasks.Core.Models.Domains;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  namespace MyTasks.Persistence.Repositories  {  public class CategoryRepository  {  private ApplicationDbContext \_context;  public CategoryRepository(ApplicationDbContext context)  {  \_context = context;  }  public IEnumerable<Category> GetCategories()  {  return \_context.Categories.OrderBy(x => x.Name).ToList();  }  }  } |

Kontroler  
Tworzymy pola prywatne dla każdego repozytorium i inicjujemy je w konstruktorze.

|  |
| --- |
| namespace MyTasks.Controllers  {  [Authorize]  public class TaskController : Controller  {  private **TaskRepository** \_taskRepository;  private **CategoryRepository** \_categoryRepository;  public TaskController(ApplicationDbContext context)  {  \_taskRepository = new TaskRepository(context);  \_categoryRepository = new CategoryRepository(context);  } |

Użycie repozytorium w Akcji kontrolera

|  |
| --- |
| public IActionResult Tasks()  {  var userId = User.GetUserId();  var vm = new TasksViewModel()  {  FilterTasks = new FilterTasks(),  Tasks = \_taskRepository.Get(userId),  Categories = \_categoryRepository.GetCategories()  };  return View(vm);  } |