

Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

Streszczenie

UnitOfWork – poprawna implementacja wzorca  
Przerobienie klasy repozytorium na UnitOfWork

asp.net Core

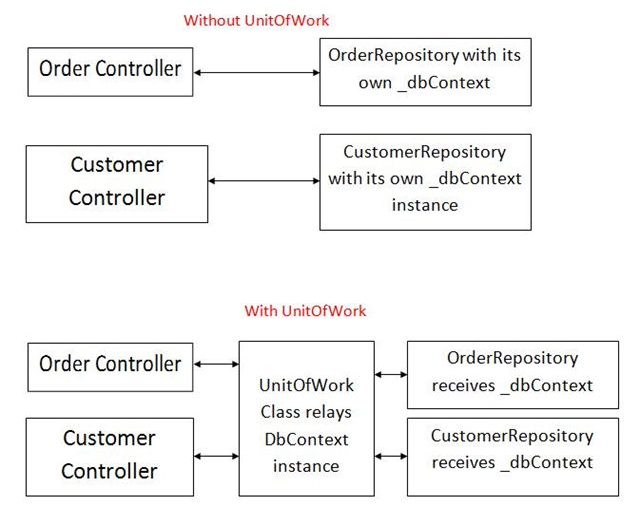
UnitOfWork and Repository

**Tworzymy Modele**

W poprzednich tygodniach wszystkie zapytania w **Entity Framework** pisaliśmy w osobnych klasach, w tzw. repozytoriach. Każdy taki byt miał swoje repozytorium, wszystkie komendy, które wykonujemy na bazie danych są w tej klasie w jednym miejscu w aplikacji.  
Dzięki temu mamy odseparowaną logikę na bazie od logiki aplikacji, minimalizujemy także potrzebę duplikowania kodu. Łatwo też przeprowadzić testy jednostkowe, ponieważ za nasze repozytorium możemy w łatwy sposób postawić **mocka**.

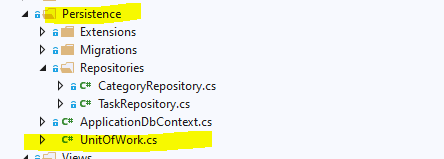
Z naszym repozytorium jest jednak jeden problem. Repozytorium powinno działać jak jakoś kolekcja, czyli powinna być możliwość dodawania, edycji i usuwania danych, ale nie powinno być możliwości zapisywania.

Tutaj z pomocą przychodzi wzorzec **UnitOfWork**, który powinien być taką warstwą nad repozytorium, powinien zarządzać tymi repozytoriami oraz zapisywać wszystkie zmiany. Dzięki UnitOfWork możemy wykonać wiele metod z różnych repozytoriów, następnie po wykonaniu wszystkich zmian możemy je zapisać do bazy danych, uzyskujemy przez to transakcyjność, czyli albo zapiszą się wszystkie zmiany albo żadna. Ten wzorzec również zwiększa elastyczność i czytelność kodu i jeszcze dodatkowo ułatwia pisanie testów jednostkowych.  
Dodatkowo taki wzorzec uniezależnia nas od źródła danych, wszelakich zmian dokonujemy tylko w repozytorium, możemy wymieniać frameworki, możemy nawet całkowicie zrezygnować z zapisywania danych do bazy i pobierać je z pliku czy też jakiegoś web serwisu.



Implementacja wzorca UnitOfWork

Tworzymy klasę UnitOfWork w katalogu Persistence



|  |
| --- |
| namespace MyTasks.Persistence  {  public class UnitOfWork  {  // readonly przy polu oznacza, że jego wartość  // możemy zmienić tylko w konstruktorze  private readonly ApplicationDbContext \_context;  public UnitOfWork(ApplicationDbContext context)  {  \_context = context;  }  // obiekty repozytoryjne  public TaskRepository Task { get; set; }  public CategoryRepository Category { get; set; }  // na koniec metoda zapisująca zmiany  public void Complete()  {  \_context.SaveChanges();  }  }  } |

W tej chwili wykonujemy różne metody z Repozytoriów w kontrolerach, możemy zmienić te wywołania na właśnie stworzony UnitOfWork.

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

Wywołania akcji

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

Założeniem wzorca UnitOfWork jest również zapisywanie danych, dla tego dla każdej metodzie modyfikującej dane dopisujemy:

|  |
| --- |
| [HttpPost]  public IActionResult Finish(int id)  {  try  {  var userId = User.GetUserId();  \_unitOfWork.Task.Finish(id, userId);  \_unitOfWork.Complete();  }  catch (Exception ex)  {  // logowanie do pliku  return Json(new { success = false, message = ex.Message });  }  return Json(new { success = true });  } |

Z repozytoriów natomiast wyrzucamy metodę context.SaveChanges().  
Od tego momentu za zapisywanie danych jest odpowiedzialny UnitOfWork

|  |
| --- |
| public void Finish(int id, string userId)  {  var taskToUpdate = \_context.Tasks.Single(x => x.Id == id);  taskToUpdate.IsExecuted = true;  \_context.SaveChanges();  } |