

Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

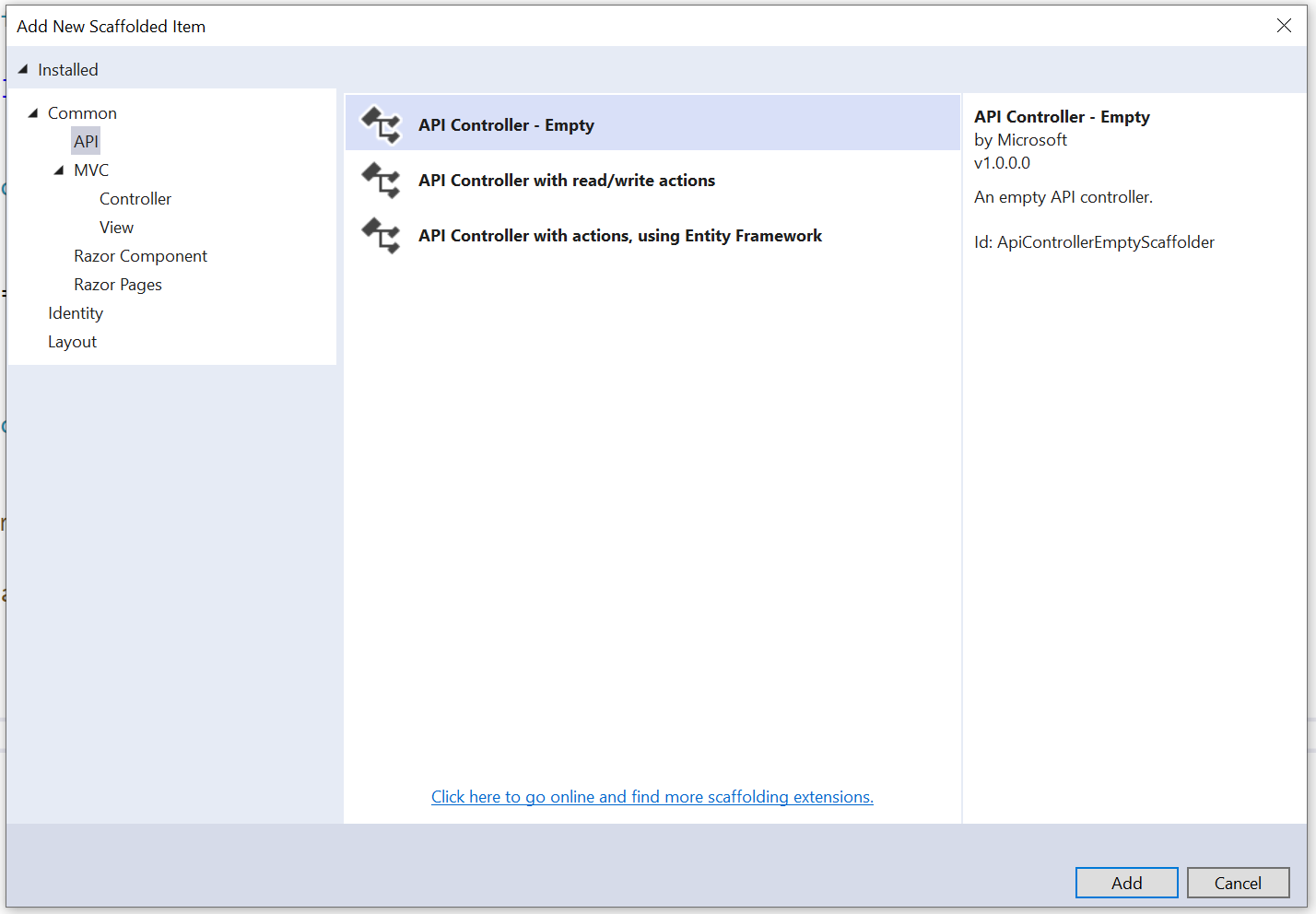
Streszczenie

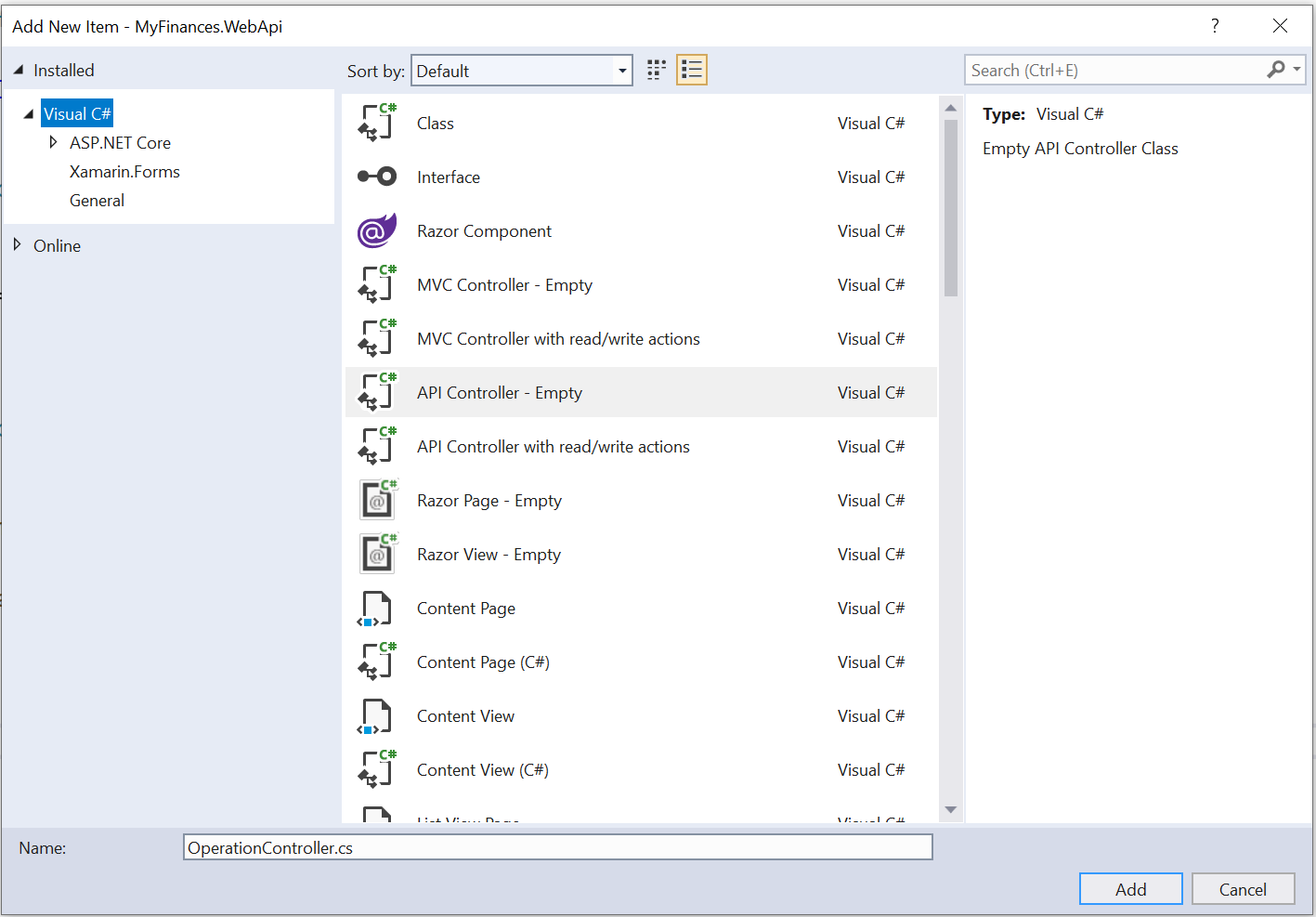
Jak utworzyć kontroler Api  
Metody zwracania wyniku w kontrolerze  
Typy akcji: HttpGet, HttpPut, HttpPost, HttpDelete

WEB API

Tworzenie kontrolerów

**Tworzenie kontrolera**

Ustawiamy się na folderze Controllers i dodajemy nowy kontroler. Należy zwrócić uwagę, aby był to kontroler API a nie MVC.   




Powstał kontroler zawiera on klasę dziedziczącą po ControllerBase, został on oznaczony atrybutem **ApiControler**, jest również wyznaczona ścieżka pod którą będą dostępne akcje z tego kontrolera.  
Będzie tam adres URL, następnie api, później nazwa tego kontrolera.

|  |
| --- |
| namespace MyFinances.WebApi.Controllers  {  **[Route("api/[controller]")]**  **[ApiController]**  public class **OperationController** : **ControllerBase**  {  }  } |

Będziemy tutaj potrzebowali nowej instancji UnitOfWork, wstrzykniemy więc ją za pomocą mechanizmu Dependency Injection, tzn. oczekujemy, że w kontrolerze zostanie wstrzyknięty nowy obiekt klasy UnitOfWork.  
Tworzymy konstruktor, jako parametr podajemy obiekt typu UnitOfWork, potem CTRL+. I wybieramy opcję create and assign field.

|  |
| --- |
| namespace MyFinances.WebApi.Controllers  {  [Route("api/[controller]")]  [ApiController]  public class OperationController : ControllerBase  {  private readonly UnitOfWork \_unitOfWork;  public OperationController(UnitOfWork unitOfWork)  {  \_unitOfWork = unitOfWork;  }  }  } |

W klasie **Startup.cs** w metodzie **ConfigureServices**, dodamy deklaracje Dependency Injection.  
Tzn chcemy aby w każdym nowym request’cie została dodana nowa instancja klasy UnitOfWork.

|  |
| --- |
| public void **ConfigureServices**(IServiceCollection services)  {  services.**AddScoped**<**UnitOfWork**, **UnitOfWork**>();  services.AddDbContext<MyFinancesContext>(options =>  options.UseSqlServer(Configuration.GetConnectionString("MyFinancesContext")));  services.AddControllers();  services.AddSwaggerGen(c =>  {  c.SwaggerDoc("v1", new OpenApiInfo { Title = "MyFinances.WebApi", Version = "v1" });  });  } |

Wracamy do kontrolera. Oczekujemy aby akcje z kontrolera zwracały Json’y.

|  |
| --- |
| **[HttpGet]**  public IEnumerable<Operation> Get()  {  return \_unitOfWork.Operation.Get();  } |

Takie rozwiązanie byłoby dobre, ale zastosujemy trochę inne. Gdyby tutaj powstał jakiś błąd, to użytkownik nie zostanie o tym powiadomiony.  
Tworzymy nowy model, który zawiera w sobie informację czy na serwerze, czyli w naszym API wystąpił jakiś błąd, jeżeli tak, to te informacje umieszczamy w właściwości errors naszego modelu.

Rozwiążemy to w ten sposób. Najpierw tworzymy nowy folder o nazwie Response a w nim klasę Response oraz Error

|  |
| --- |
| public class Response  {  public Response()  {  Errors = new List<Error>();  }  public List<Error> Errors { get; set; }  public bool IsSuccess  {  get  {  return Errors == null || !Errors.Any();  }  }  } |

Mamy właściwość isSuccess aby za każdym razem jej nie ustawiać

|  |
| --- |
| public bool **isSuccess** { get; set; } |

Zapisujemy ją w ten sposób, aby sama się ustawiała na podstawie listy błędów

|  |
| --- |
| public bool **IsSuccess**  {  get  {  return Errors == null || !Errors.Any();  }  } |

Ten drugi zapis jest jednoznaczny z zapisem poniżej.

|  |
| --- |
| public bool **isSuccess** => Errors == null || !Errors.Any(); |

Możemy go zastosować, jeżeli właściwość ma tylko jedną linię.

**Klasa z Error**

Dwie właściwości – źródło błędu i treść błędu, aby sobie ułatwić te dwie właściwości inicjalizujemy w konstruktorze.

|  |
| --- |
| namespace MyFinances.WebApi.Response  {  public class Error  {  public Error(string source, string message)  {  Source = source;  Message = message;  }  public string Source { get; set; }  public string Message { get; set; }  }  } |

Aby móc przekazać jakiś model, tworzymy dodatkowo jeszcze jedną klasę:  
DataResponse dziedziczy po klasie Response więc ma już te same właściwości. Dodatkowo dodajemy właściwość, która będzie zawierać jakiś obiekt, obiekt ten może być różnych typów dlatego zrobimy tę klasę jako generyczną.

|  |
| --- |
| namespace MyFinances.WebApi.Response  {  public class DataResponse<T> : Response  {  public T Data { get; set; }  }  } |

Ponownie wracając do kontrolera

Pierwotnie napisana Akcja

|  |
| --- |
| **[HttpGet]**  public IEnumerable<Operation> Get()  {  return \_unitOfWork.Operation.Get();  } |

Jednym z możliwości uniknięcia wad powyższego rozwiązania jest zapis jak poniżej.  
Akcja ma zwracać **IActionResult**, dzięki temu możemy zwrócić wiele statusów operacji tak jak poniżej: BadRequest(), NotFound()  
a w przypadku powodzenia Ok() z wynikiem zapytania jako argument.

|  |
| --- |
| [HttpGet]  public IActionResult Get3()  {  if (true)  {  return BadRequest();  }  if (true)  {  return NotFound();  }  return Ok(\_unitOfWork.Operation.Get());  } |

My jednak wykorzystamy nasze klasy do opakowania zwracanej wartości. Jest to rozwiązanie, które daje większą elestyczność

|  |
| --- |
| **[HttpGet]**  public DataResponse<IEnumerable<Operation>> Get()  {  var responce = new DataResponse<IEnumerable<Operation>>();  try  {  responce.Data = \_unitOfWork.Operation.Get();  }  catch (Exception ex)  {  // logowanie do pliku ...  responce.Errors.Add( new Error(ex.Source,ex.Message));  }    return responce;  } |

|  |
| --- |
| **Uwaga**  Nazwa metody nie ma znaczenia do działania kontrolera. Zamiast Get możemy wpisać dowolną nazwę. Jeżeli w przeglądarce wpiszemy api/[controller] to zostaniemy przekierowani właśnie do tej metody. Liczy się to co wpiszemy w atrybucie np. HttpGet. |

W akcja poniżej zostanie wywołana jeżeli w przeglądarce wpiszemy api/[controller]/id .

|  |
| --- |
| **[HttpGet("{id}")]**  public DataResponse<Operation> Get(int id)  {  var responce = new DataResponse<Operation>();  try  {  responce.Data = \_unitOfWork.Operation.Get(id);  }  catch (Exception ex)  {  // logowanie do pliku ...  responce.Errors.Add(new Error(ex.Source, ex.Message));  }  return responce;  } |

Kolejna akcja to dodawanie rekordu, tutaj nazwa też nie ma znaczenia, ważne aby była typu HttpPost, nazwiemy ją Add, aby nazwa nam coś mówiła.

Tutaj zwrócimy jako dane nie cały obiekt lecz tylko id, jest to operacja dodania rekordu do bazy, więc id zostanie dopiero dodane, a użytkownik może sobie odczytać ten obiekt podając id.

|  |
| --- |
| **[HttpPost]**  public DataResponse<int> Add(Operation operation)  {  var responce = new DataResponse<int>();  try  {  \_unitOfWork.Operation.Add(operation);  \_unitOfWork.Complete();  // tutaj jest już znane id dodanego rekordu  responce.Data = operation.Id;  }  catch (Exception ex)  {  // logowanie do pliku ...  responce.Errors.Add(new Error(ex.Source, ex.Message));  }  return responce;  } |

Teraz dodamy akcję, która będzie aktualizowała rekord. Akcja taka musi być typu HttpPut, nazwa może być dowolna, może być np. Update.

|  |
| --- |
| **[HttpPut]**  public Response Update(Operation operation)  {  var responce = new Response();  try  {  \_unitOfWork.Operation.Update(operation);  \_unitOfWork.Complete();  }  catch (Exception ex)  {  // logowanie do pliku ...  responce.Errors.Add(new Error(ex.Source, ex.Message));  }  return responce;  } |

Podobnie z Delete

|  |
| --- |
| **[HttpDelete("{id}")]**  public Response Delete(int id)  {  var responce = new Response();  try  {  \_unitOfWork.Operation.Delete(id);  \_unitOfWork.Complete();  }  catch (Exception ex)  {  // logowanie do pliku ...  responce.Errors.Add(new Error(ex.Source, ex.Message));  }  return responce;  } |