

**AKADEMIA HANDLOWA
NAUK STOSOWANYCH W RADOMIU**



**RADOM
ACADEMY OF ECONOMICS**

Wydział Studiów Strategicznych i Technicznych

Kierunek: Informatyka, rok II, semestr III (2021/2022)

LABORATORIUM NR 2 ZAAWANSOWANE METODY PROGRAMOWANIA OBIEKTOWEGO

Prowadzący: dr Piotr Dobosz

Zespół laboratoryjny:

Magdalena Szafrńska, nr albumu: 18345

Spis treści

Cel ćwiczenia	3
Informacje wstępne	3
Aplikacja internetowa (webowa)	3
Backend	4
ASP.NET Core MVC	4
Model-View-Controller (MVC)	5
Frontend	5
HTML	5
CSS	5
JavaScript	6
React	6
RAZOR	6
HTML5	6
Stos technologiczny (użyte narzędzia)	7
Przebieg laboratorium	8
Kod źródłowy	8
Założenia aplikacji	8
Budowa projektu	9
Widok Index	11
Widok Dodaj zadanie	14
Widok Szczegóły	17
Widok Edytuj	18
Widok Usuń	21
Baza danych	25
Łączenie z bazą danych	25
Tabela z zadaniami	26
Łączenie aplikacji z bazą danych (Repositories)	27
Poprawa wyglądu (Bootstrap)	38
Funkcjonalność	39
Dodanie nowego zadania	39
Odznaczenie zadania jako wykonanego	40
Szczegóły zadania	41
Edycję istniejącego zadania	41
Usunięcie wybranego zadania	42
Wnioski	43

Cel ćwiczenia

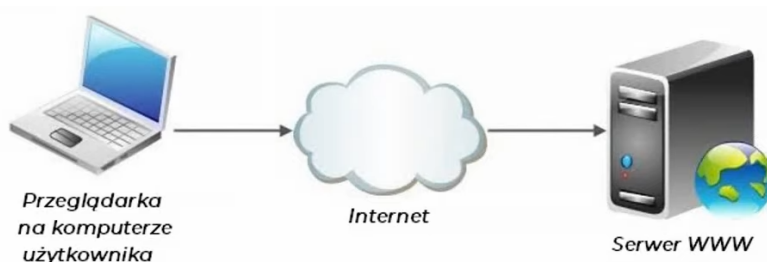
Niniejsze laboratorium numer 2 ma za zadanie pokazać, że każda aplikacja może posiadać interfejs napisany w języku HTML. Jest on wykorzystywany przy tworzeniu interfejsu użytkownika (tzw. front-end) w projektach aplikacji.

Informacje wstępne

W programowaniu od dłuższego czasu utrzymuje się trend modularnej budowy aplikacji. Ideowo każdy z modułów tworzonej aplikacji powinien być niezależny i możliwy do zastąpienia/poprawy. Potrzeba takiego rozwiązania rodzi się z tego powodu, że projekty programistyczne są coraz bardziej rozbudowane i dawno już nie są tworzone przez jedną osobę. Dodatkowo, wraz z rozwojem technologii tworzy się coraz większy podział pomiędzy osoby tworzące logikę aplikacji (back-end), interfejs użytkownika (front-end) oraz projektantów rozwiązań.

Aplikacja internetowa (webowa)

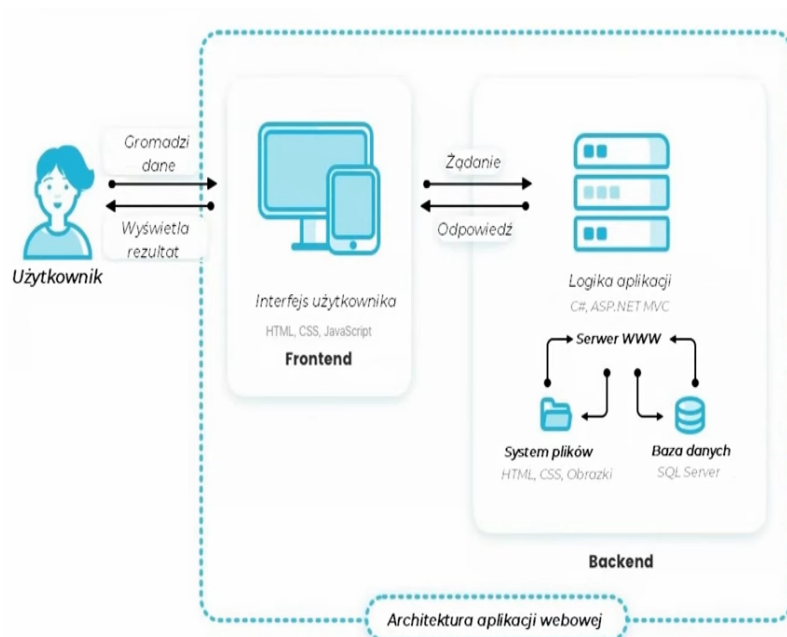
Aplikacja internetowa (ang. web application), zwana również aplikacją webową, to program komputerowy, który pracuje na serwerze i komunikuje się poprzez sieć komputerową z hostem użytkownika komputera z wykorzystaniem przeglądarki internetowej użytkownika.



Aplikacje webowe działają w oparciu o architekturę klient - serwer. Serwer www to program działający na serwerze internetowym, obsługujący żądania protokołu komunikacyjnego HTTP. Klient za to to program komputerowy, komputer, lub host, na którym działa program w roli klienta. Przeglądarka internetowa łączy się poprzez sieć komputerową z serwerem www, aby pobrać wskazaną stronę www. Przeglądarka internetowa pełni w tym wypadku rolę klienta.

Architektura klient - serwer umożliwia podział zadań. Polega to na ustalaniu, że serwer zapewnia usługi dla klientów zgłaszających do serwera żądania obsługi.

Każda aplikacja webowa składa się z frontendu i backendu. W największym uproszczeniu, aplikacja frontend wysyła żądanie do aplikacji backendu, która z kolei wysyła je do bazy danych i wykonuje logikę biznesową, po czym wraca z odpowiedzią do frontendu.



Backend

Backend, lub inaczej zaplecze, aplikacja po stronie serwera. To system połączonych komputerów, które komunikują się ze sobą. Zajmuje się on logiką i integracją funkcji po stronie serwera. Mamy tam aplikację lub zestaw aplikacji podłączonych do Internetu lub chmury. Dwa komponenty, które definiują zaplecze to serwery www i bazy danych. Proces serwera www to wszystko, co odpowiada na żądanie HTTP i zwraca do niego plik. Druga część zaplecza to baza danych. Nie każda baza danych potrzebuje serwera www, ale 99% projektów z serwerem www wymaga bazy danych. W większości przypadków serwer i baza danych znajdują się na różnych komputerach, ale mają własne połączenie, co zapewnia wyższą wydajność i optymalne wykorzystanie przestrzeni.

ASP.NET Core MVC

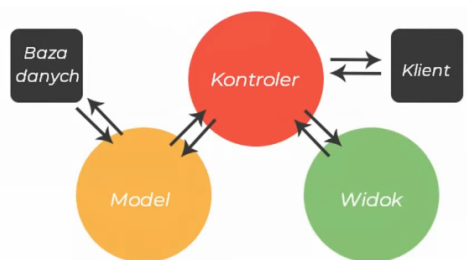
Platforma aplikacyjna do budowy aplikacji internetowych opartych na wzorcu Model-View-Controller (MVC). Do najważniejszych cech platformy należą:

1. Wykorzystanie silnika ASP.NET Core pozwalającego na wykorzystanie wielu komponentów infrastrukturalnych oferowanych przez tę platformę, takich jak mechanizmy zarządzania stanem aplikacji mechanizmy uwierzytelniania i autoryzacji, profile, cache itp.
2. Model programistyczny (API) platformy jest mocno oparty na interfejsach, pozwalając na łatwą rozbudowę, dorabianie i testowanie poszczególnych komponentów.
3. Elastyczny mechanizm mapowania adresów Uniform Resource Locator pozwalających na łatwą budowę aplikacji według wzorca Representational State Transfer (REST), wykorzystujących czytelną dla użytkowników strukturę adresów URL.

Model-View-Controller (MVC)

Wprowadza on podział w kodzie źródłowym na trzy sekcje (model, view, controller), dzięki czemu kod ten staje się czytelniejszy i łatwiejszy w późniejszym rozwijaniu projektu.

- **Model** - określa jakie operacje będziemy wykonywać po stronie tzw. backendu
- **View** - to, co użytkownik będzie widział, wykonywane tylko po stronie widoku (frontend)
- **Controller** - spina oba powyższe, czyli widok odwołuje się do funkcji w kontrolerze, a kontroler wykonuje funkcję lub sekwencję funkcji z modelu, żeby uzyskać pożądane efekty i zwrócić je do widoku.



Przykładem może być moduł produktu w sklepie internetowym gdzie w modelu będziemy przechowywać wszystkie właściwości produktu, w widoku znajdzie się cała otoczka wizualna a kontroler będzie odpowiadał za reakcję na zdarzenia wywoływane np. przez użytkownika.

Frontend

Frontend nazywamy aplikacją po stronie klienta. Można powiedzieć, że frontend jest warstwą prezentacyjną aplikacji i stron internetowych. Sekcja ta obejmuje wszystko, co może zobaczyć docelowy użytkownik. W bardziej fachowym języku jest to kod odpowiedzialny za przedstawianie pożądanych informacji odbiorcy. Front-end development obejmuje zatem tworzenie intuicyjnych i przyjaznych użytkownikowi interfejsów, jak również efektywne przechowywanie, prezentowanie i aktualizowanie danych otrzymywanych z backendu lub API.

Frontend development wykorzystuje takie technologie jak:

HTML

To język znaczników opisujących strukturę witryny, podstawowy budulec strony internetowej. Za pomocą tego języka programowania określa się to, co ma zawierać witryna

CSS

Język opisujący wygląd strony. Służy do formatowania elementów wyglądu opisanych za pomocą HTML, dzięki czemu nadaje on ostateczny wygląd strony internetowej.

JavaScript

Służy do zakodowania interaktywnych funkcji strony internetowej, które muszą reagować na działania użytkownika.

React

Ułatwia tworzenie interaktywnych interfejsów użytkownika. Świetnie sprawdza się w przypadku native developmentu, czyli tworzenia aplikacji z myślą o konkretnych urządzeniach i platformach.

RAZOR

To połączenie kodu HTML z kodem aplikacji, np. C#. RAZOR to silnik renderujący pozwalający na bardzo łatwe oddzielenie kodu HTML od kodu aplikacji. Posiada prostą składnię. W porównaniu do starszych silników renderujących, aby uzyskać taki sam efekt, wymaga napisania mniejszej ilości kodu. Aby wyświetlić wartość zmiennej, wystarczy postawić przed nią znak „@”, analogicznie postępujemy w przypadku pętli oraz innych elementów nie należących do składni języka HTML czy JavaScript. Dodatkowo RAZOR obsługuje klamry, które są bardzo pomocne, gdy mamy więcej niż jedną linię kodu wymagającego użycia RAZORA.

HTML5

Od blisko dwudziestu lat, od czasu zaprezentowania HTML5, powstaje coraz więcej projektów, które wykorzystują ten właśnie język projektowania do tworzenia GUI (Graphic User Interface). Dzięki temu programy mogą działać po stronie serwera, nierzadko bez instalacji po stronie klienta.

Rozwiązanie to posiada szereg zalet:

- Licencjonowanie zgodne z deklaracją
- Łatwiejsze wdrażanie aktualizacji
- Łatwiejsze udzielanie pomocy
- Lepsze zabezpieczenie danych aplikacji (brak ewentualnego czynnika ryzyka po stronie klienta)
- Zwiększenie poziomu bezpieczeństwa poprzez personalizację dostępu

Posiada także wady, m.in.:

- Mniejsze bezpieczeństwo oraz poufność danych (włamanie na serwer aplikacji może spowodować niewyobrażalne szkody)
- Brak sieci powoduje utratę przez klienta dostępu do aplikacji (aczkolwiek są opcje na zniwelowanie tego problemu)
- Podśluch aplikacji (niewystarczające zabezpieczenie może spowodować, że dane z aplikacji wyciekną podczas transferu pomiędzy programem a serwerem)
- Duża liczba klientów może doprowadzić do spowolnienia działania aplikacji po stronie serwera.

Pomimo wad interfejs użytkownika stale kieruje się jednak w stronę HTML i/lub technologii podobnych. Nierzadko aplikacje offline posiadają interfejs obsługi HTML (wykorzystywany jest do tego komponent WebView). Dzięki temu osoby zajmujące się projektowaniem UX/UI mają ujednoliconą pracę, zaś programiści mogą zająć się jedynie usprawnianiem projektu.

Stos technologiczny (użyte narzędzia)

- **Windows 10 Home** (wersja 21H1)
- **Microsoft Visual Studio Community 2019** (wersja 16.11.1)
- **GIT** - repozytorium kodu
- **Github** - zdalne repozytorium Gita (link do repozytorium niniejszego projektu: [znajduje się tutaj](#))
- **.NET** w wersji 5.0. - platforma uruchomieniowa
- **IIS Express** - Internet Information Services Express, to deweloperski serwer www, który umożliwia użytkownikom obsługę własnej witryny WWW oraz FTP w sieci
- Backend:
 - **C#** - jako główny język programowania użyty w aplikacji
 - **ASP.NET Core MVC** - platforma do budowy aplikacji internetowych opartych na wzorcu MVC
 - **Microsoft® SQL Server® 2019 Express** - baza danych
 - **Microsoft SQL Server Management Studio** (wersja 18.10) - zintegrowane środowisko do zarządzania wszystkimi komponentami (baza danych, usługi analityczne, usługi raportowe itd.), wchodzącymi w skład Microsoft SQL Server.
 - **Entity Framework Core** - do mapowania obiektowo relacyjnego
- Frontend
 - **HTML, CSS, JavaScript**
 - **json** - format wymiany danych używany do przesyłania informacji z serwera do aplikacji klienta.
 - **Bootstrap** - biblioteka do poprawienia wyglądu w końcowej fazie tworzenia projektu

Przebieg laboratorium

Laboratorium polega na utworzeniu dowolnego projektu oprogramowania z wykorzystaniem wszystkich opisanych wyżej rozwiązań i narzędzi. W moim projekcie stworzyłam program, który pozwala tworzyć tzw. listy TODO, czyli tworzy i zarządza listą rzeczy do zrobienia. Nazwałam go “Menadżerem zadań”.

Do głównych funkcjonalności programu należy zaliczyć integrację z bazą danych i możliwość przechowywania w niej zadań, dostępnych również po zamknięciu aplikacji. Dane te zapisywane są w sposób dynamiczny, bez ingerencji użytkownika. Po ponownym uruchomieniu aplikacji dane są dopisywane do istniejącego pliku - o ile takowy już się znajduje.

Kod źródłowy

Kod źródłowy całego projektu wraz z plikiem bazy danych umieściłam w repozytorium zdalnym na GitHubie. Link do niego: [znajduje się tutaj](#).

Uwaga! W repozytorium znajdują się również pliki innego projektu “Kalkulator”, który miał być pierwotnie moim projektem zaliczeniowym - nie należy tego folderu brać pod uwagę. Dzięki poszerzeniu wiedzy postanowiłam stworzyć odwzorowanie projektu z pierwszych laboratoriów, czyli listy TODO i to folder “MagdaSzafranska_TaskManager” jest projektem zaliczeniowym.

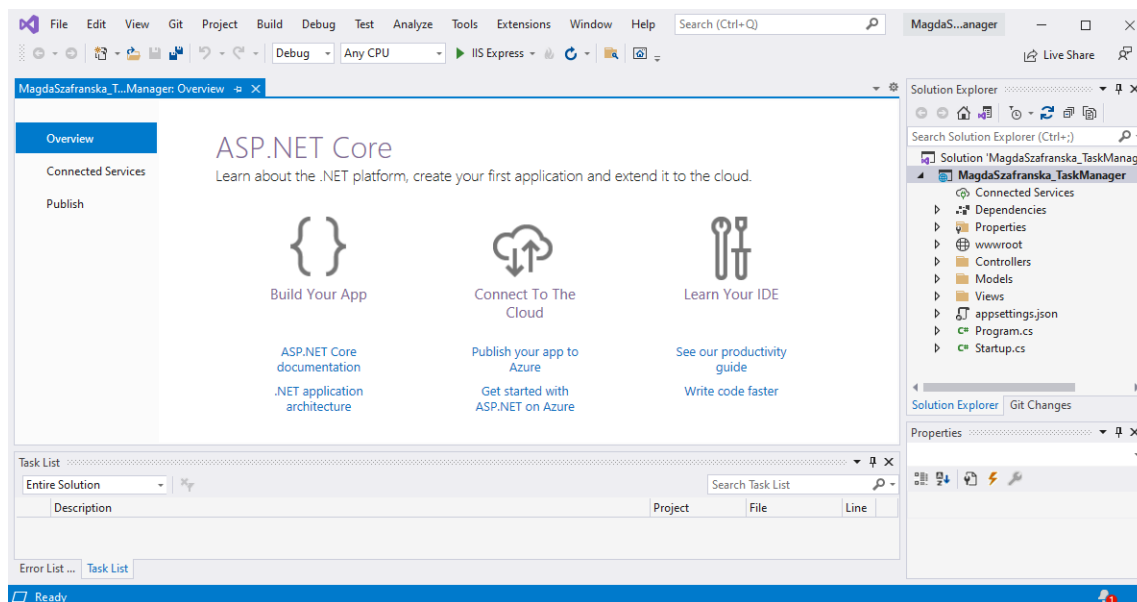
Założenia aplikacji

Aplikacja “*Menadżer zadań*” w ramach laboratorium będzie zawierała następujące funkcje:

1. Dodanie nowego zadania
2. Odznaczenie zadania jako wykonanego
3. Szczegóły zadania
4. Edycję istniejącego zadania
5. Usunięcie wybranego zadania

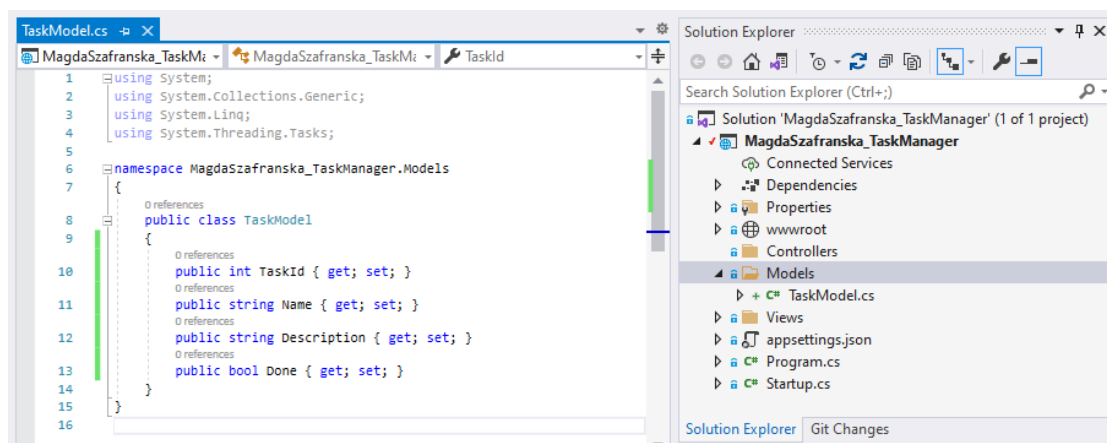
Budowa projektu

Tworzę szablon dla aplikacji w technologii ASP.NET MVC w środowisku programistycznym Microsoft Visual Studio.

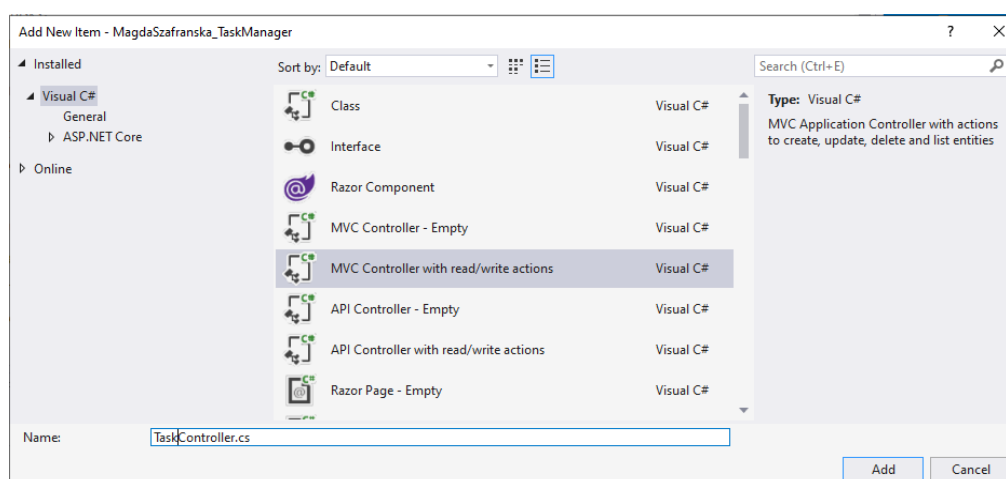
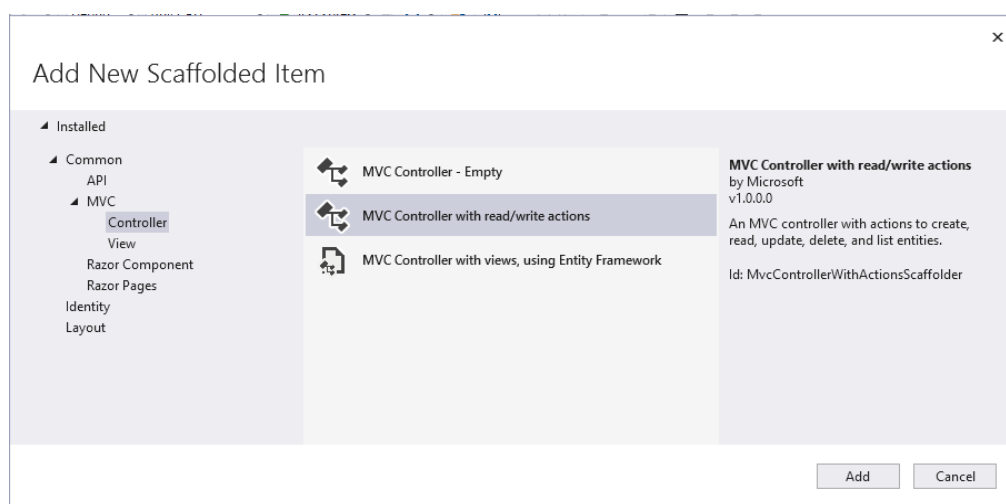
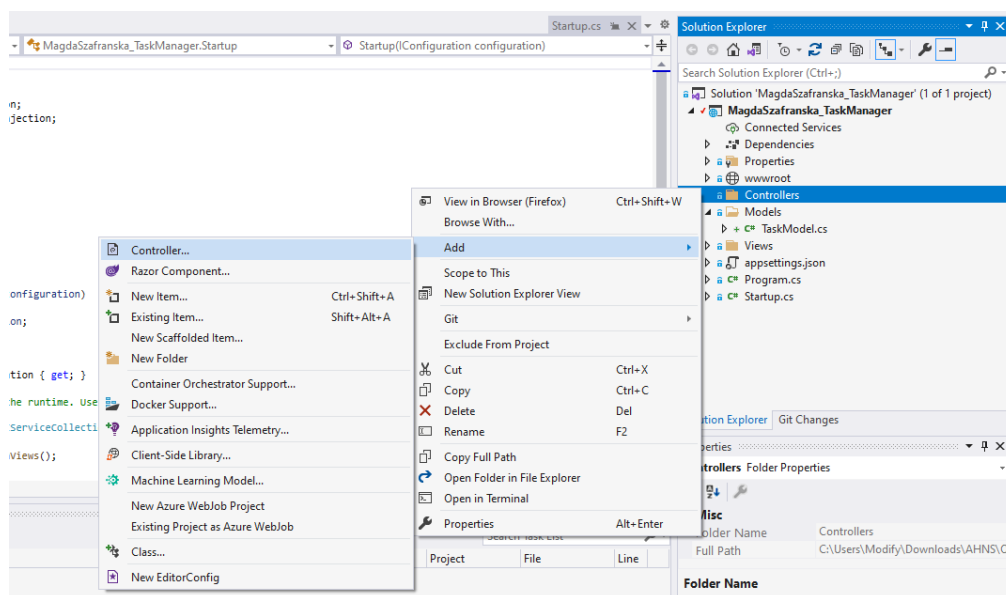


Na początku usuwam wszystkie pliki, które zostały dodane automatycznie, a których nie będę wykorzystywać.

Do folderu *Models* dodaję klasę *TaskModel.cs* - będzie ona przechowywać podstawowe właściwości zadań dodawanych do listy zadań.



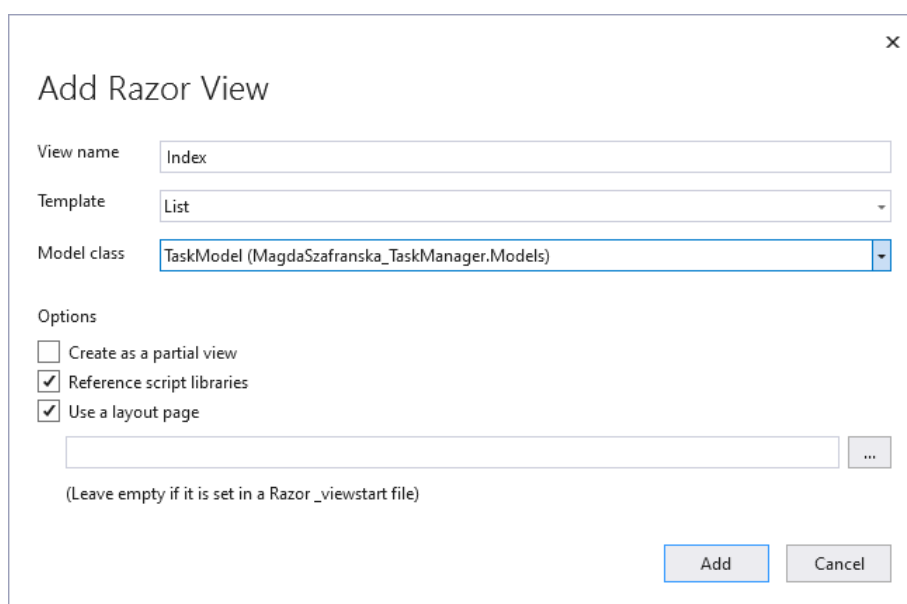
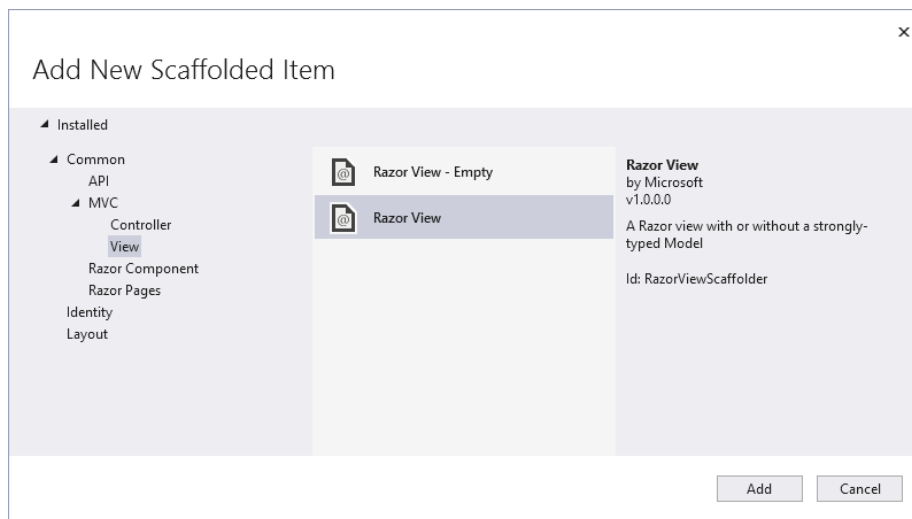
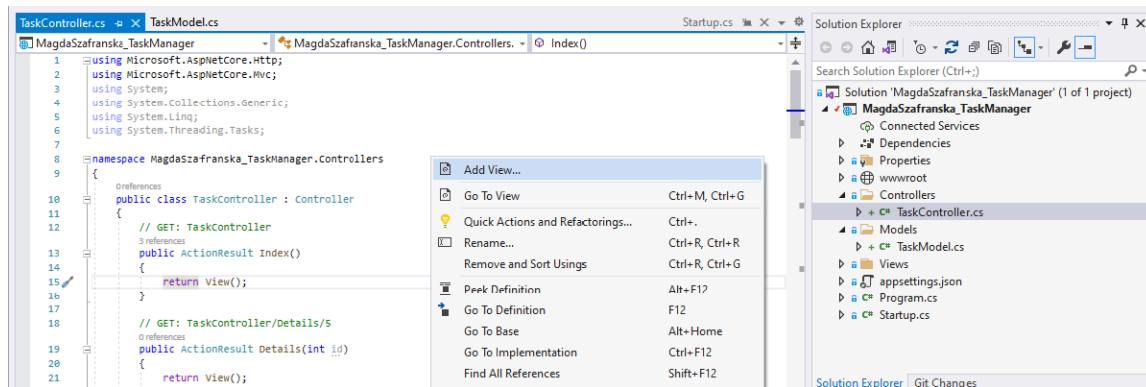
Do folderu *Controllers* dodaję nową klasę kontrolera.



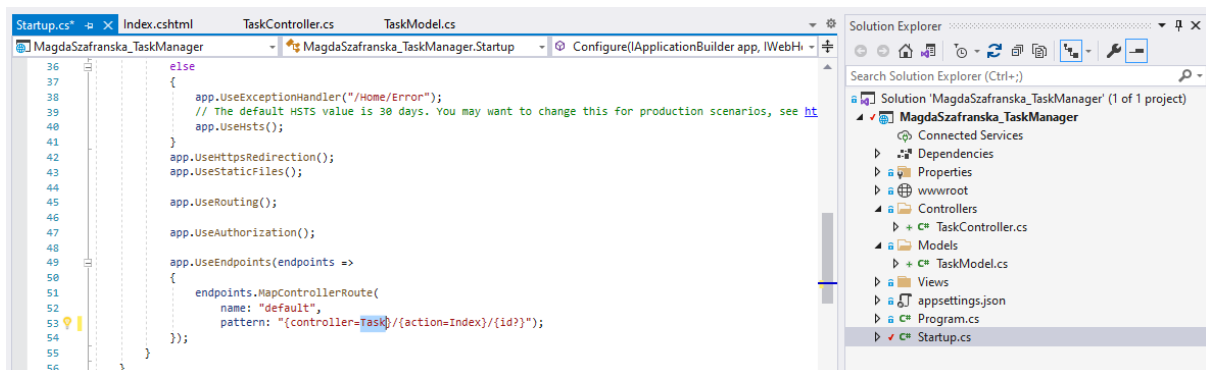
Dzięki temu Visual Studio wygenerował mi automatycznie akcje wyświetlania listy zadań, szczegółów zadania, dodawania, edycji a także ich usuwania.

Widok Index

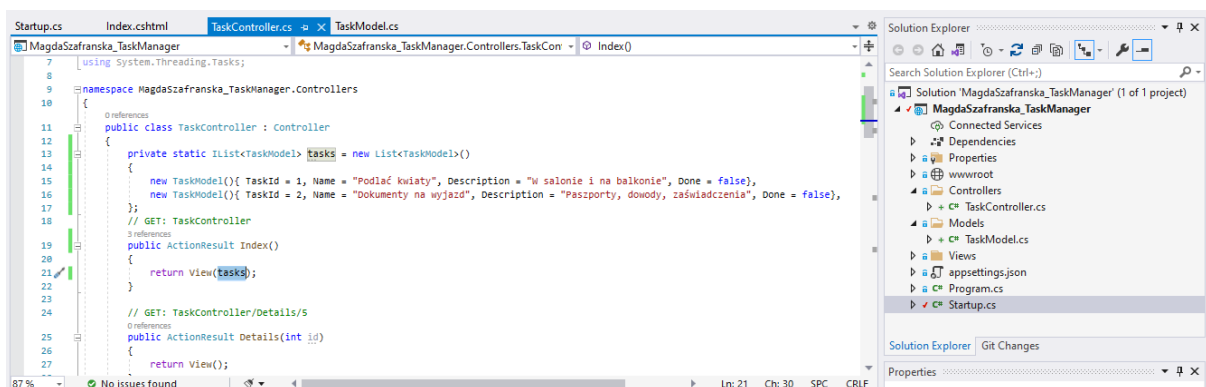
Tworzę pierwszy widok: widok do akcji *Index*. Klikam PPM w miejscu, w którym ma zostać utworzony widok.



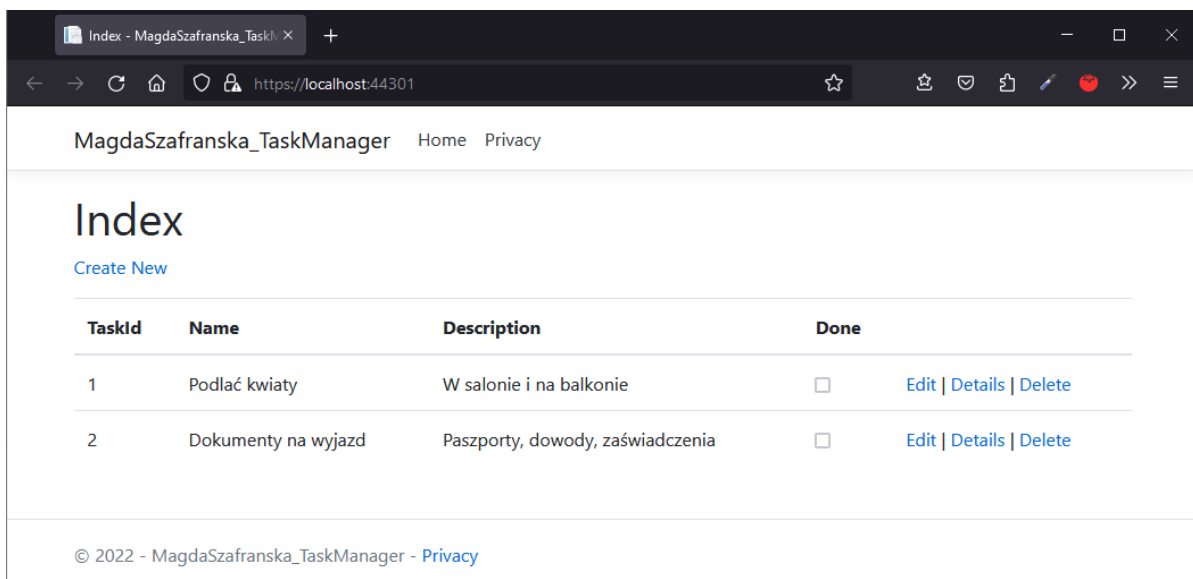
Zostaje przeniesiona do nowo utworzonego widoku *Index*. W klasie *Startup.cs* przechodzę do miejsca w kodzie odpowiedzialnej za domyślny routing url. Zmieniam nazwę kontrolera na *Task*.



W kontrolerze w klasie *TaskController.cs* deklaruję i inicjalizuję listę obiektów typu *TaskModel* o nazwie *tasks*. Następnie przekazuję listę *tasks* jako argument widoku w akcji *Index*.



Po zbudowaniu i uruchomieniu aplikacji otrzymuję oczekiwane efekty w oknie przeglądarki.

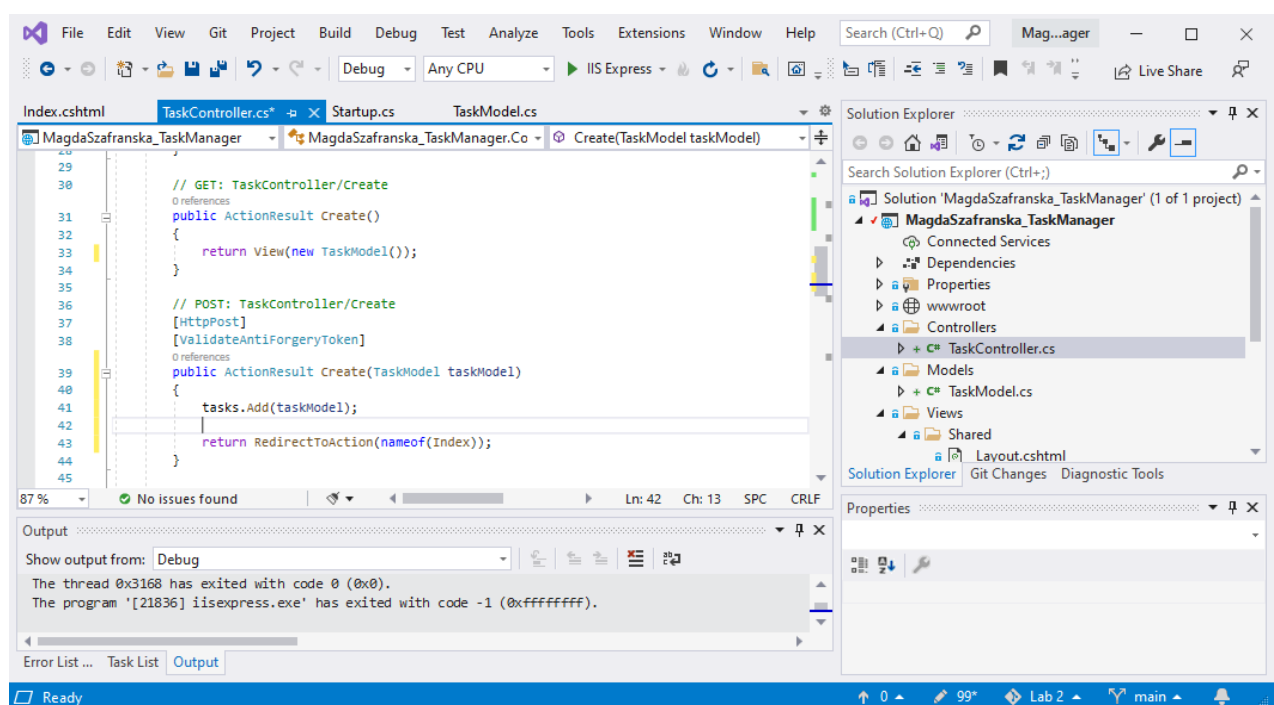


W adresie url nie ma parametrów wskazujących na kontroler, akcję oraz identyfikator - dlatego trasa przyjmuje wartości domyślne, czyli kontroler: *tasks* oraz akcja: *Index*. Spowodowało to wywołanie metody *Index* z klasy *TaskController*. Metoda *Index* zwraca widok, czyli plik zawierający kod HTML oraz C#, który to widok następnie zostaje wyświetlony w oknie przeglądarki. Model danych w postaci listy zadań zostaje przekazany do widoku bezpośrednio w akcji kontrolera.

Z widoku w pliku *Index.cshtml* usuwam dane, których nie chcę wyświetlać, a więc identyfikator, opis i znacznik 'gotowe' - usuwam zarówno nagłówki jak i dane.

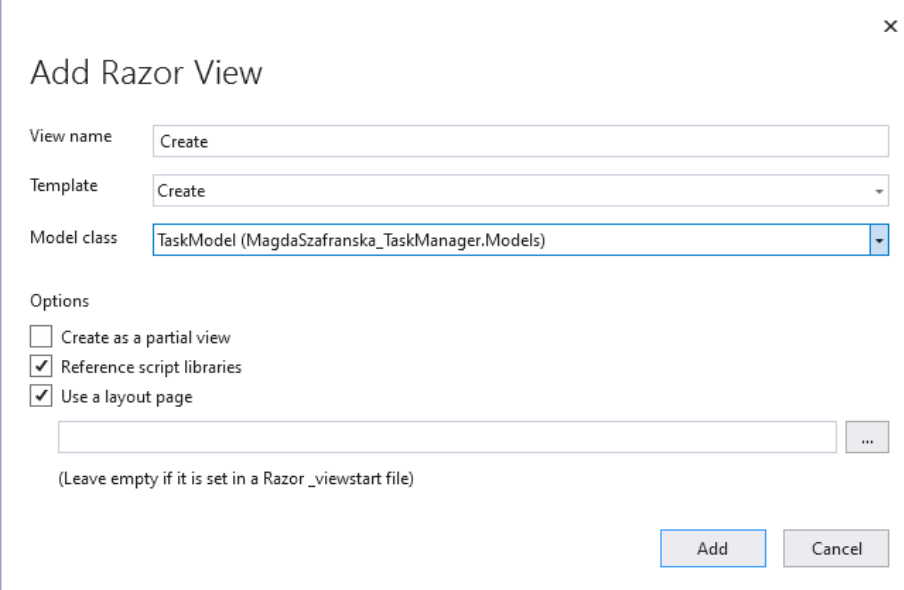
W klasach kontrolera (*TaskController.cs*) znajdują się dwie metody o nazwie *Create*.

- Pierwsza z nich - bezparametrowa - uruchamiana jest w chwili wyświetlenia pustego formularza 'Dodaj zadanie'. Jest to metoda typu GET, służąca do pobierania danych z określonego zasobu. Przekazuję jej do widoku nowy obiekt *TaskModel()*.
- Druga z metod, zawierająca parametr *collection* uruchamiana jest w momencie kliknięcia wykonania akcji na uzupełnionym formularzu 'Dodaj zadanie'. Jest to metoda typu POST, czyli służy do wysyłania danych do serwera w celu utworzenia bądź aktualizacji zasobów. W tej metodzie zmieniam typ oraz nazwę parametru. Dodaję do listy *tasks* obiekt przesłany w parametrze *taskModel*. Metoda *RedirectToAction()* przekierowuje do określonej akcji przy użyciu nazwy akcji.



Widok Dodaj zadanie

Dla drugiej akcji *Create* z parametrem tworzę widok.



View name: Create

Template: Create

Model class: TaskModel (MagdaSzafranska_TaskManager.Models)

Options

- ☐ Create as a partial view
- ☒ Reference script libraries
- ☒ Use a layout page

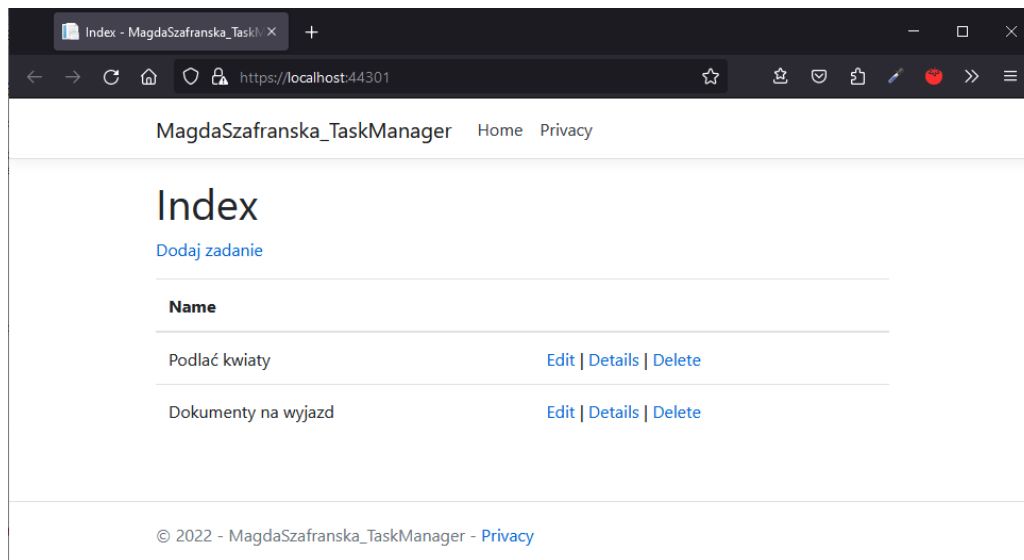
Layout: ...

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

W nowo utworzonym widoku *Create* zmieniam nagłówek na 'Dodaj zadanie' i usuwam zbędne dane tworzone automatycznie.

W *TaskController.cs* dodaję inicjalizację właściwości *TaskId* obiektu *taskModel*. Przebudowuję i uruchamiam program.



Po kliknięciu '*Dodaj zadanie*' zgodnie z oczekiwaniami na ekranie wyświetla się formularz do dodawania zadania. W adresie url jest wyraźnie widoczny kontroler: *Task* oraz akcja: *Create*.

Dodaję nowe zadanie.

Dodaj zadanie - MagdaSzafranska_TaskManager

Home Privacy

Create

Name

Zadzwoń do urzędu

Description

Jakie wymogi do wyjazdu do Grecji

Dodaj

[Back to List](#)

© 2022 - MagdaSzafranska_TaskManager - [Privacy](#)

Nowo wprowadzone zadanie pojawiło się na liście zadań.

Index - MagdaSzafranska_TaskManager

Home Privacy

Index

[Dodaj zadanie](#)

Name	
Podlać kwiaty	Edit Details Delete
Dokumenty na wyjazd	Edit Details Delete
Zadzwoń do urzędu	Edit Details Delete

© 2022 - MagdaSzafranska_TaskManager - [Privacy](#)

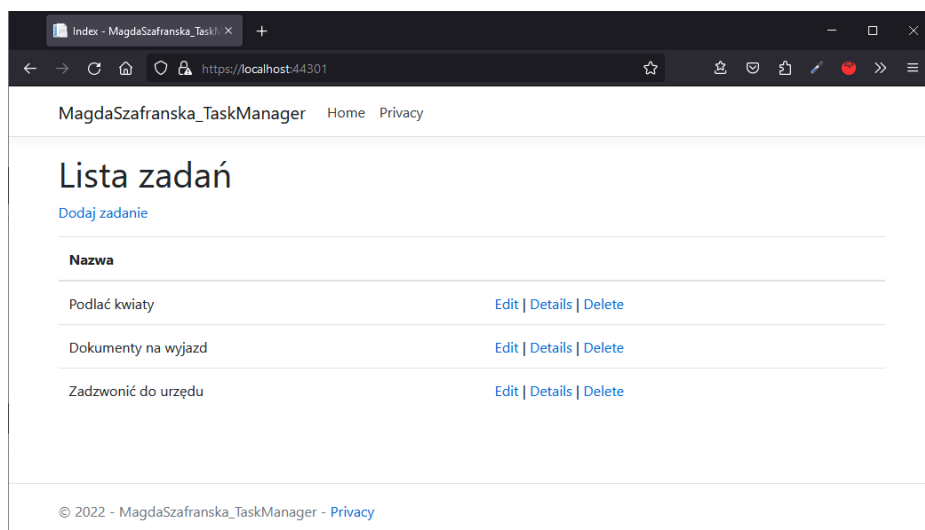
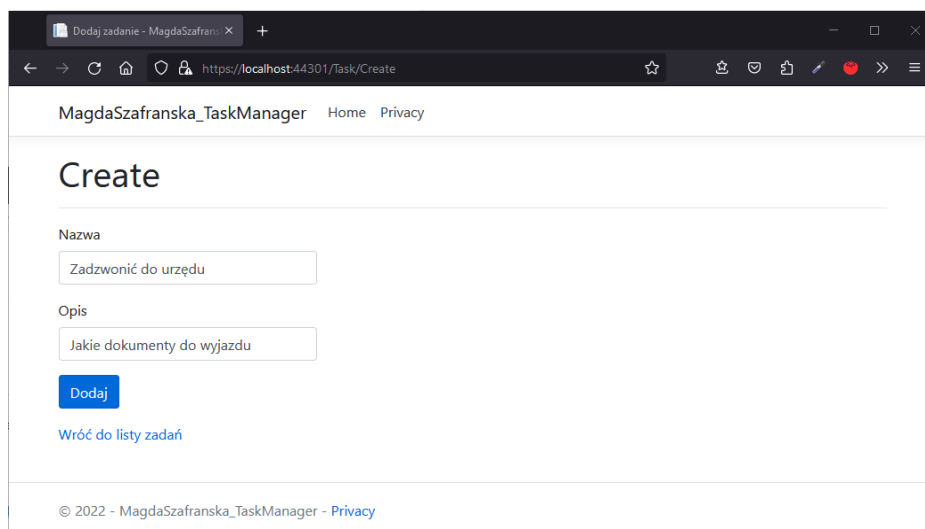
W widoku Create (*Create.cshtml*) zmieniam nazwę etykiety akcji na 'Wróć do listy zadań'. W klasie Modelu *TaskModel.cs* dodaję do właściwości *Name* atrybut *DisplayName* z argumentem 'nazwa'. Dzięki temu na formularzach zamiast etykiety *Name* będzie wyświetlała się etykieta *Nazwa*. Analogicznie postępuję z właściwością *Description* nadając jej etykieta *Opis*.

```

Create.cshtml  Index.cshtml  TaskModel.cs*  Startup.cs
MagdaSzafranska_TaskM  MagdaSzafranska_TaskM  Description
4  using System.Linq;
5  using System.Threading.Tasks;
6
7  namespace MagdaSzafranska_TaskManager.Models
8  {
9      12 references
10     public class TaskModel
11     {
12         3 references
13         public int TaskId { get; set; }
14         [DisplayName("Nazwa")]
15         7 references
16         public string Name { get; set; }
17         [DisplayName("Opis")]
18         5 references
19         public string Description { get; set; }
20         2 references
21         public bool Done { get; set; }
22     }
23 }

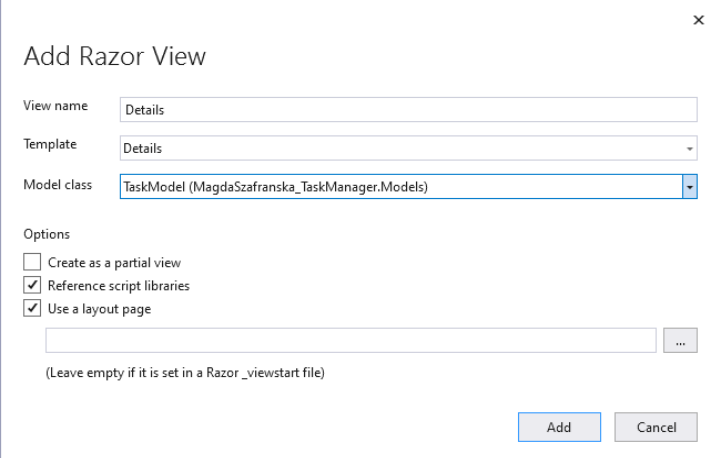
```

W klasie widoku *Index* zmieniam nagłówkę na *Lista zadań*. Po tych kosmetycznych poprawkach aplikacja prezentuje się jak poniżej. W formularzu do dodawania zadania widać nowe etykiety *Nazwa* oraz *Opis*.



Widok Szczegóły

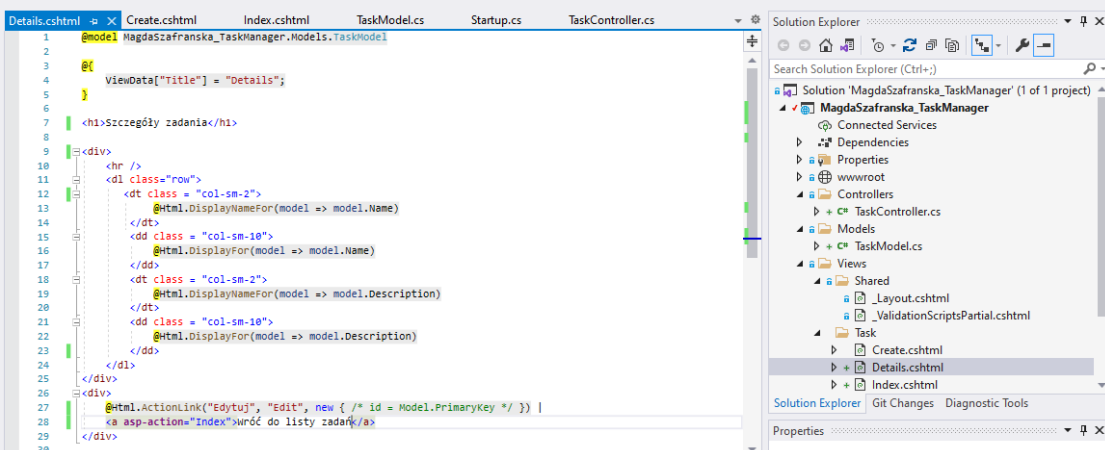
W kontrolerze w akcji *Details* dodaję widok.



The 'Add Razor View' dialog box is shown. It has the following fields and options:

- View name: Details
- Template: Details
- Model class: TaskModel (MagdaSzafranska_TaskManager.Models)
- Options:
 - ☐ Create as a partial view
 - ☒ Reference script libraries
 - ☒ Use a layout page
- Buttons: Add, Cancel

Zmieniam nagłówek na *Szczegóły zadania*, etykietę linku na *Edytuj* oraz etykietę akcji na *Wróć do listy zadań*. Ustawiam nagłówek podrzędny oraz zbędne dane, których nie chcę wyświetlać.



The screenshot shows the 'Details.cshtml' file in Visual Studio. The code is as follows:

```
1 @model MagdaSzafranska_TaskManager.Models.TaskModel
2
3 @{
4     ViewData["Title"] = "Details";
5 }
6
7 <h1>Szczegóły zadania</h1>
8
9 <div>
10     <hr />
11     <dl class="row">
12         <dt class="col-sm-2">
13             @Html.DisplayNameFor(model => model.Name)
14         </dt>
15         <dd class="col-sm-10">
16             @Html.DisplayFor(model => model.Name)
17         </dd>
18         <dt class="col-sm-2">
19             @Html.DisplayNameFor(model => model.Description)
20         </dt>
21         <dd class="col-sm-10">
22             @Html.DisplayFor(model => model.Description)
23         </dd>
24     </dl>
25 </div>
26 <div>
27     @Html.ActionLink("Edytuj", "Edit", new { /* id = Model.PrimaryKey */ }) |
28     <a asp-action="Index">Wróć do listy zadań</a>
29 </div>
30
```

W widoku *Index* zmieniam etykietę linku na *Szczegóły* oraz odkomentuję fragment kodu odpowiedzialny za przesłanie parametru *id* do metody *Details()* i przekazuję *TaskId*.



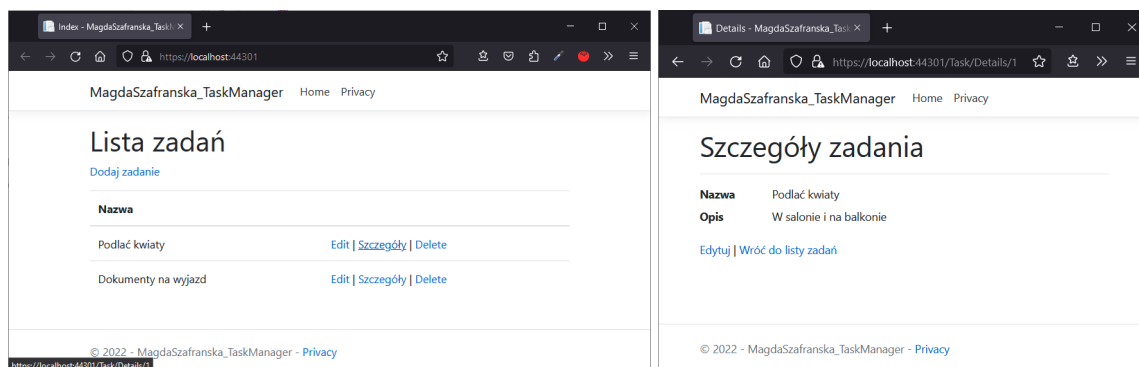
The screenshot shows the 'Index.cshtml' file in Visual Studio. The code is as follows:

```
22 @foreach (var item in Model) {
23     <tr>
24         <td>
25             @Html.DisplayFor(modelItem => item.Name)
26         </td>
27         <td>
28             @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
29             @Html.ActionLink("Szczegóły", "Details", new { id=item.TaskId }) |
30             @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
31         </td>
32     </tr>
33 }
34 </tbody>
35 </table>
36
```

W kontrolerze w akcji *Details* przekazuję obiekt *tasks* jako argument widoku w akcji *Details*. Za pomocą metody *FirstOrDefault()* wyszukuję na liście *tasks* element o podanym identyfikatorze.

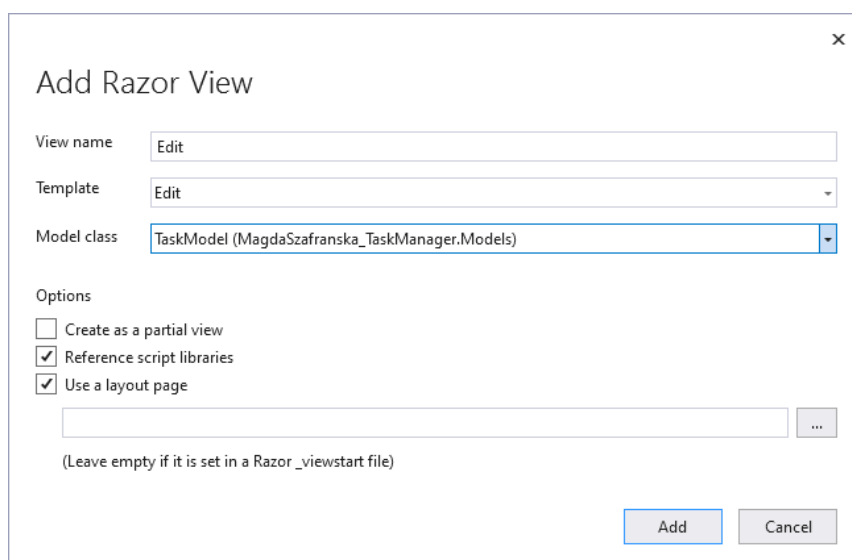
```
Details.cshtml | Create.cshtml | Index.cshtml* | TaskModel.cs | Startup.cs | TaskController.cs* | Details(int id)
// GET: TaskController/Details/5
0 references
public ActionResult Details(int id)
{
    return View(tasks.FirstOrDefault(x => x.TaskId == id));
}
```

Po uruchomieniu aplikacji klikam na *Szczegóły* pierwszego zadania. Zgodnie z oczekiwaniami, na ekranie wyświetla się formularz ze szczegółami zadania.

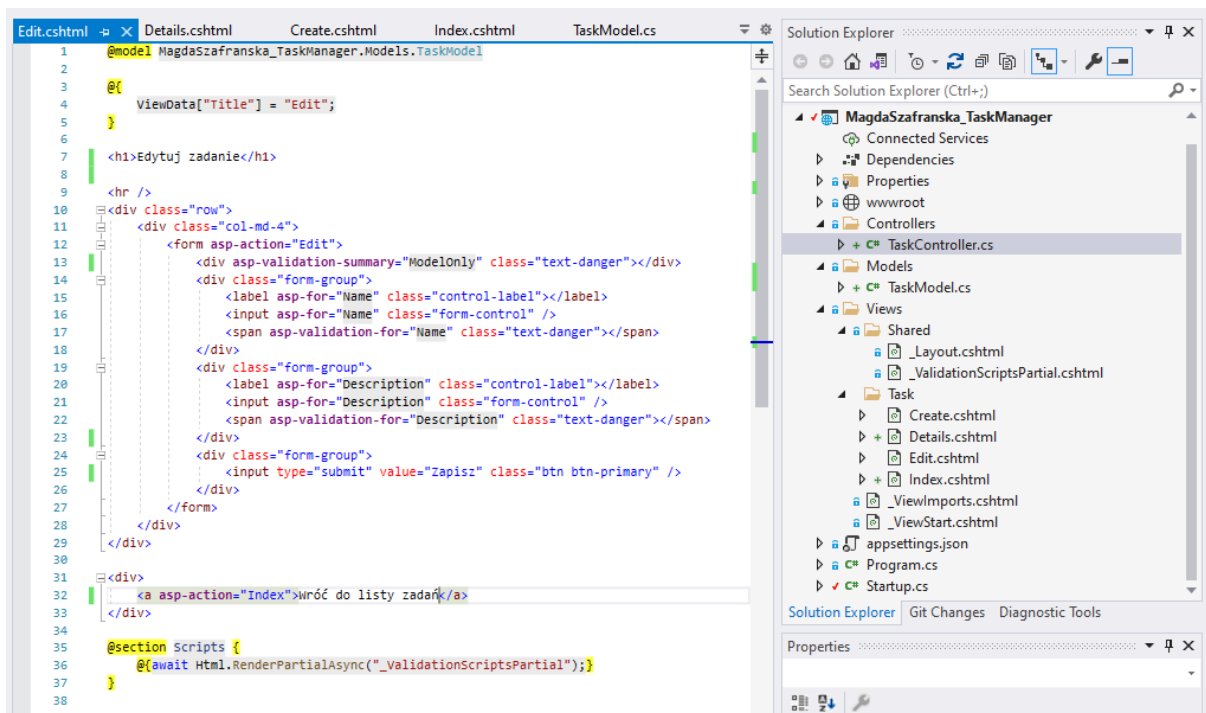


Widok Edytuj

W kontrolerze w akcji *Edit* dodaję widok.

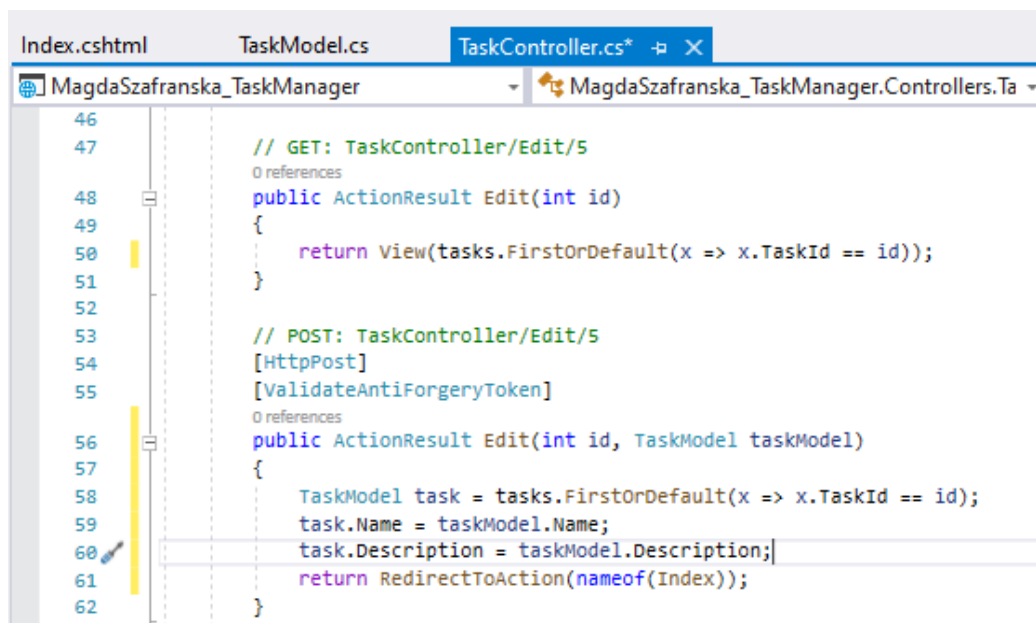


Zmieniam nagłówek na *Edytuj zadanie*, usuwam podrzędny nagłówek oraz dane, których nie chcę wyświetlać, a więc identyfikator i znacznik gotowy. Zmieniam etykietę przycisku na *Zapisz* a także zmieniam nazwę etykiety akcji na *Wróć do listy zadań*.



W kontrolerze w bezparametrowej metodzie `Edit()` przekazuję do widoku obiekt klasy `TaskModel` o podanym identyfikatorze (`id`).

W drugiej metodzie `Edit` zmieniam typ oraz nazwę parametru. Tworzę nowy obiekt `task` typu `TaskModel` i przypisuję do niego obiekt z listy `tasks` o podanym identyfikatorze. Następnie przypisuję do właściwości `Name` oraz `Description` obiektu `task` właściwości z parametru `taskModel`.



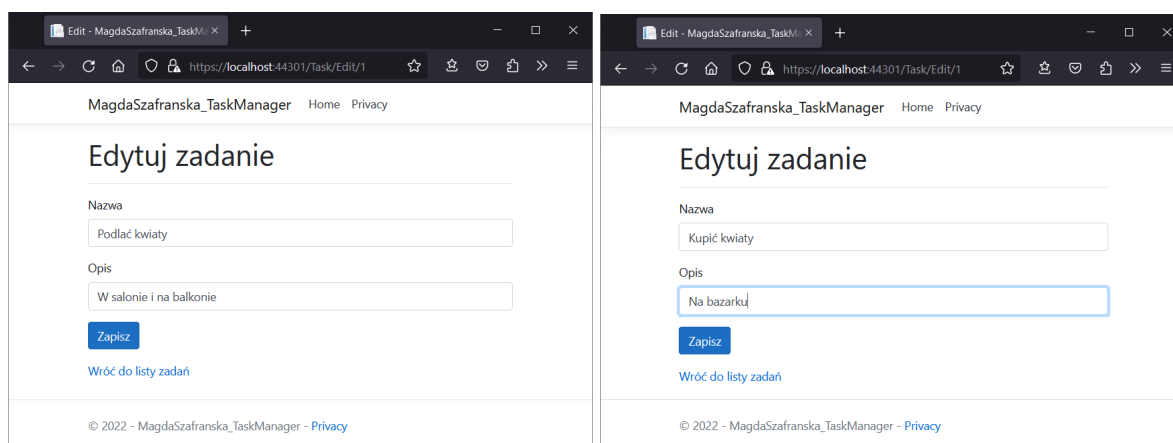
Przechodzę do widoku `Index` (`Index.cshtml`). Zmieniam etykietę linku na `Edytuj` oraz odkomentowuję fragment kodu odpowiedzialny za przesłanie parametru `id` do metody `Details()` i przekazuję `TaskId`.

```
Index.cshtml TaskModel.cs TaskController.cs
22 @foreach (var item in Model) {
23     <tr>
24         <td>
25             @Html.DisplayFor(modelItem => item.Name)
26         </td>
27         <td>
28             @Html.ActionLink("Edytuj", "Edit", new { id=item.TaskId }) |
29             @Html.ActionLink("Szczegóły", "Details", new { id=item.TaskId }) |
30             @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
31         </td>
32     </tr>
33 }
34 </tbody>
35 </table>
```

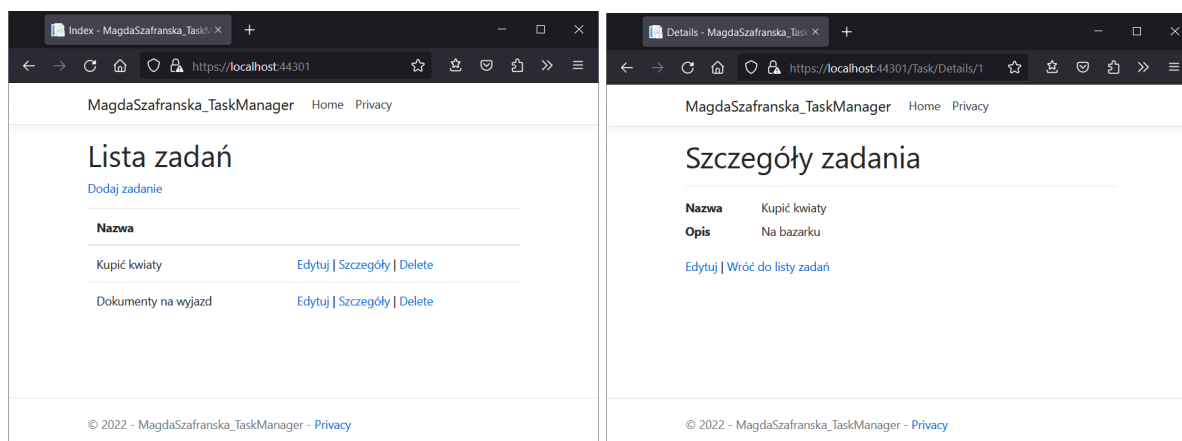
W widoku *Details* odkomentowuję fragment kodu odpowiedzialny za przesłanie parametru *id* do metody *Edit()* i przekazuję *TaskId*.

```
Details.cshtml Index.cshtml TaskModel.cs TaskController.cs
25 </div>
26 <div>
27     @Html.ActionLink("Edytuj", "Edit", new { id = Model.TaskId }) |
28     <a asp-action="Index">Wróć do listy zadań</a>
29 </div>
30
```

Uruchamiam aplikację i klikam *Edytuj* przy pierwszym zadaniu. Zgodnie z oczekiwaniami na ekranie wyświetla się formularz do edycji zadania. Zmieniam nazwę zadania oraz jego opis i zapisuję.



Jak widać poniżej, oba pola zostały zmienione. Po wejściu w szczegóły widać również zmieniony opis. Zadanie zostało w pełni zaktualizowane.



Widok Usun

W kontrolerze w akcji *Delete* dodaję widok.

×

Add Razor View

View name:

Template:

Model class:

Options

☐ Create as a partial view
☒ Reference script libraries
☒ Use a layout page

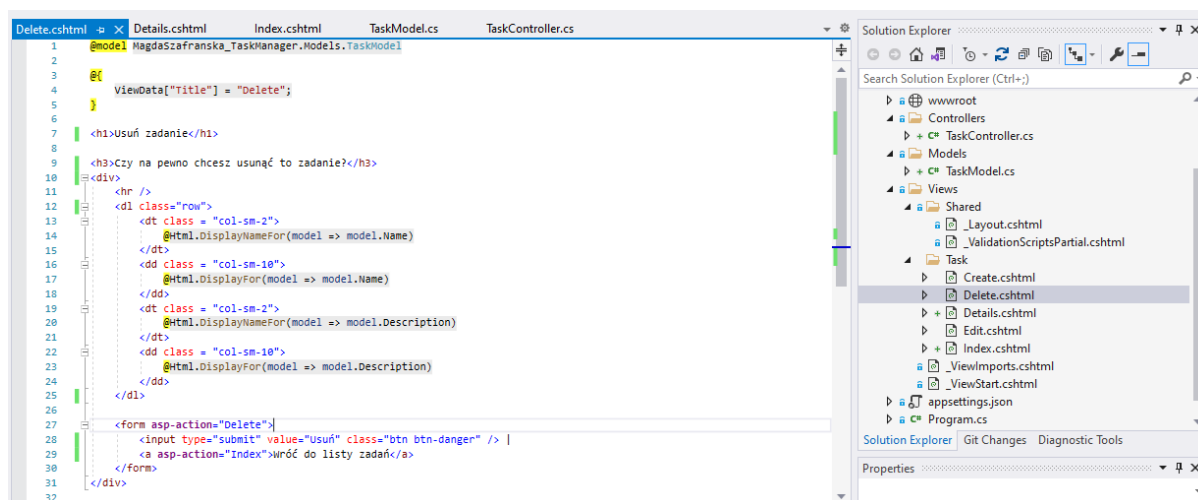
...

(Leave empty if it is set in a Razor _viewstart file)

Add

Cancel

Zmieniam nagłówek na *Usun zadanie*, usuwam podrzędny nagłówek oraz dane, których nie chcę wyświetlać, a więc identyfikator i znacznik gotowy. Zmieniam etykietę przycisku na *Usun* a także zmieniam nazwę etykiety akcji na *Wróć do listy zadań*.



Przechodzę do widoku *Index (Index.cshtml)*. Zmieniam etykietę linku na *Usuń* oraz odkomentowuję fragment kodu odpowiedzialny za przesłanie parametru *id* do metody *Delete()* i przekazuję *TaskId*.

```

Delete.cshtml      Details.cshtml      Index.cshtml*  TaskModel.cs      TaskController.cs
22  @foreach (var item in Model) {
23      <tr>
24          <td>
25              @Html.DisplayFor(modelItem => item.Name)
26          </td>
27          <td>
28              @Html.ActionLink("Edytuj", "Edit", new { id=item.TaskId }) |
29              @Html.ActionLink("Szczegóły", "Details", new { id=item.TaskId }) |
30              @Html.ActionLink("Usuń", "Delete", new { id = item.TaskId })
31          </td>
32      </tr>
33  }
34  </tbody>
35  </table>

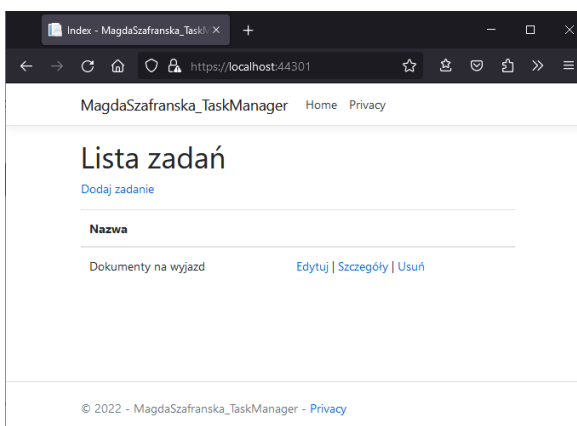
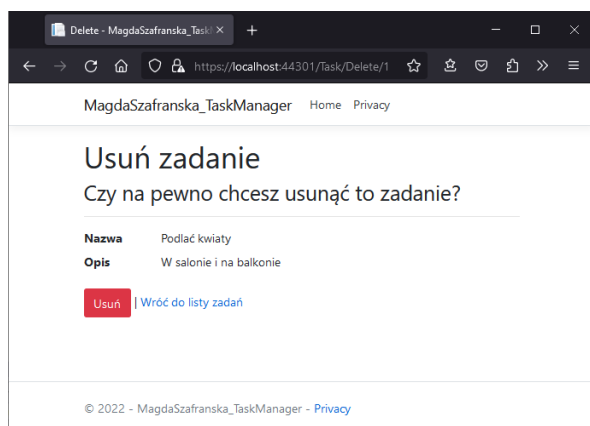
```

Przechodzę do kontrolera. Bezparametrowej metodzie *Delete* przekazuję do widoku obiekt *TaskModel* o podanym identyfikatorze. W drugiej metodzie *Delete* zmieniam typ oraz nazwę parametru. Tworzę nowy obiekt *task* typu *TaskModel* i przypisuję do niego obiekt z listy *tasks* o podanym identyfikatorze. Następnie usuwam z listy *tasks* obiekt *task*.

```
Delete.cshtml    Details.cshtml    Index.cshtml    TaskModel.cs    TaskController.cs  + X
MagdaSzafranska_TaskManager  MagdaSzafranska_TaskManager.Controllers.TaskController  Delete(int id, TaskModel taskModel)

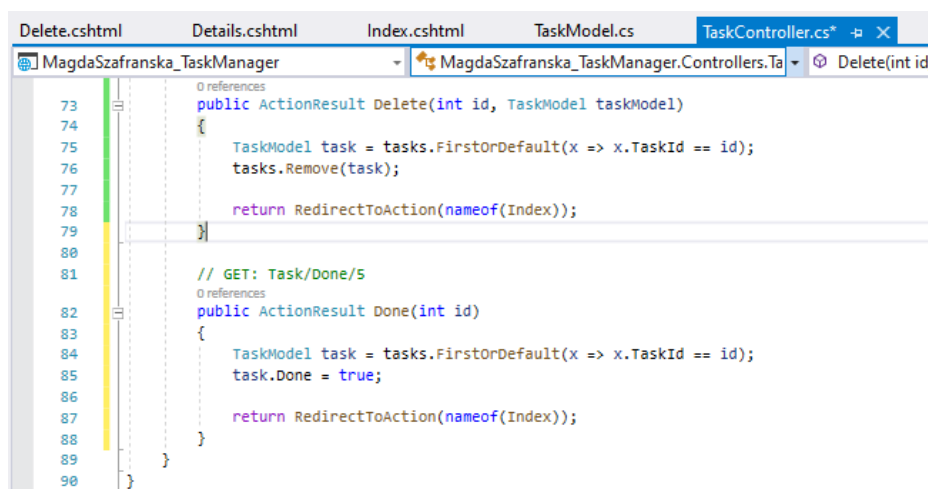
63
64 // GET: TaskController/Delete/5
65 0 references
66 public ActionResult Delete(int id)
67 {
68     return View(tasks.FirstOrDefault(x => x.TaskId == id));
69 }
70
71 // POST: TaskController/Delete/5
72 [HttpPost]
73 [ValidateAntiForgeryToken]
74 0 references
75 public ActionResult Delete(int id, TaskModel taskModel)
76 {
77     TaskModel task = tasks.FirstOrDefault(x => x.TaskId == id);
78     tasks.Remove(task);
79     return RedirectToAction(nameof(Index));
80 }
81 }
```

Uruchamiam aplikację. Klikam na *Usuń* przy wybranym zadaniu. Zgodnie z oczekiwaniami na ekranie wyświetla się formularz do usuwania zadania. Po kliknięciu *Usuń* zadanie zostało usunięte z listy zadań.

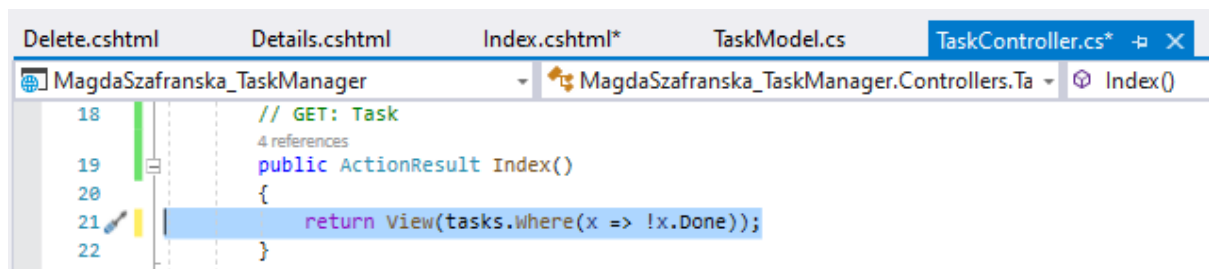


W kontrolerze dodaję akcję *Done* z parametrem *id*. Trasa do akcji będzie wyglądać następująco: *Task/Done/5*. Przy okazji poprawiam trasy dla pozostałych akcji.

Metoda *Done* będzie zmieniała znacznik *done* na *true* co oznacza, że zadanie zostało zakończone i zniknie z listy zadań. Tworzę nowy obiekt *task* typu *TaskModel* i przypisuję do niego obiekt z listy *tasks* o podanym identyfikatorze. Następnie przypisuję do właściwości *Done* wartość *true*. Na koniec zwracam rezultat metody *RedirectToAction*, która przekierowuje do akcji *Index*.

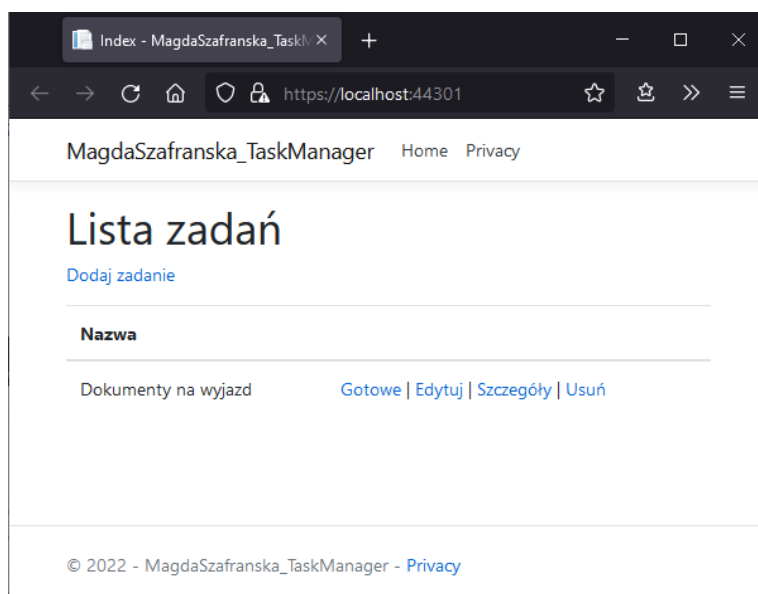
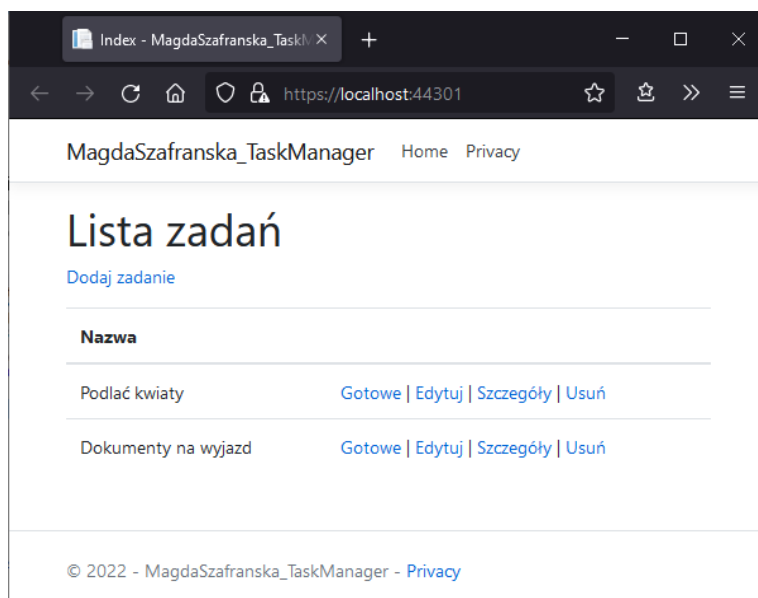


W kontrolerze w akcji *Index* edytuję zwracany parametr widoku tak, aby zwrócić tylko aktywne zadania czyli takie ze znacznikiem *done* ustawionym na wartość *false*. Aby to zrobić dodaję odpowiedni warunek *Where*<>



```
18 // GET: Task
19 4 references
20 public ActionResult Index()
21 {
22     return View(tasks.Where(x => !x.Done));
23 }
```

Uruchamiam aplikację. Klikam przy pierwszym zadaniu na *Gotowe* i zadanie zniknęło z listy zadań.

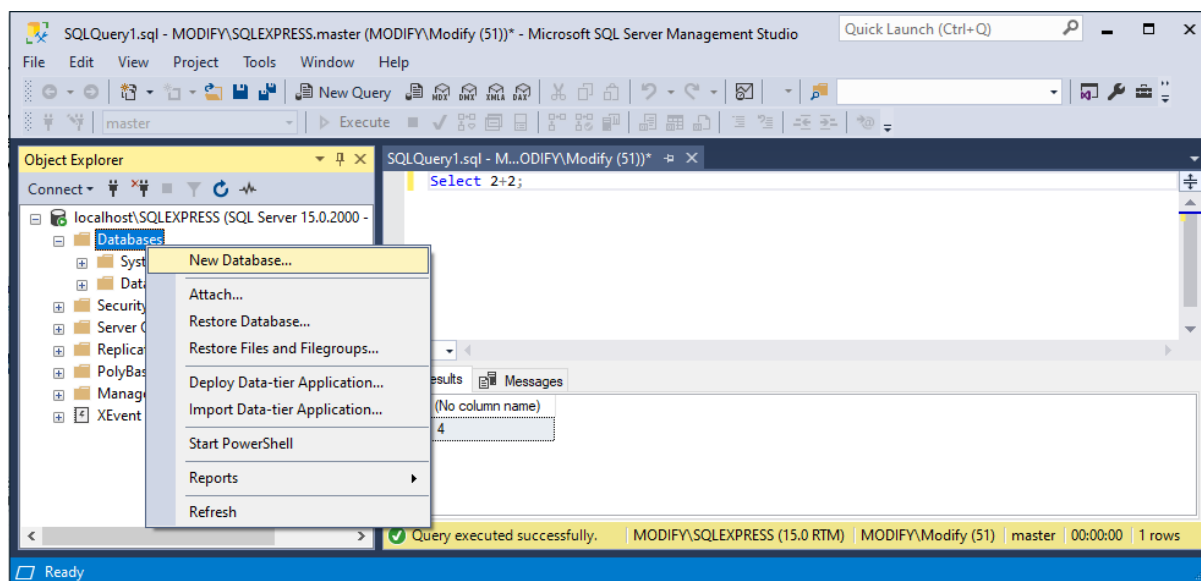


Baza danych

Do tej pory wszystkie wykonywane przeze mnie operacje wykonywały się w pamięci. Docelowo chcę, aby dane przechowywane były w bazie danych. Utworzę teraz bazę danych a następnie zaimplementuję warstwę dostępu do danych.

Łączenie z bazą danych

Uruchamiam Microsoft SQL Server Management Studio (SSMS) i łączę się do serwera localhost. Tworzę nową bazę danych.



Nazywam moją bazę *TaskManagerDB*.

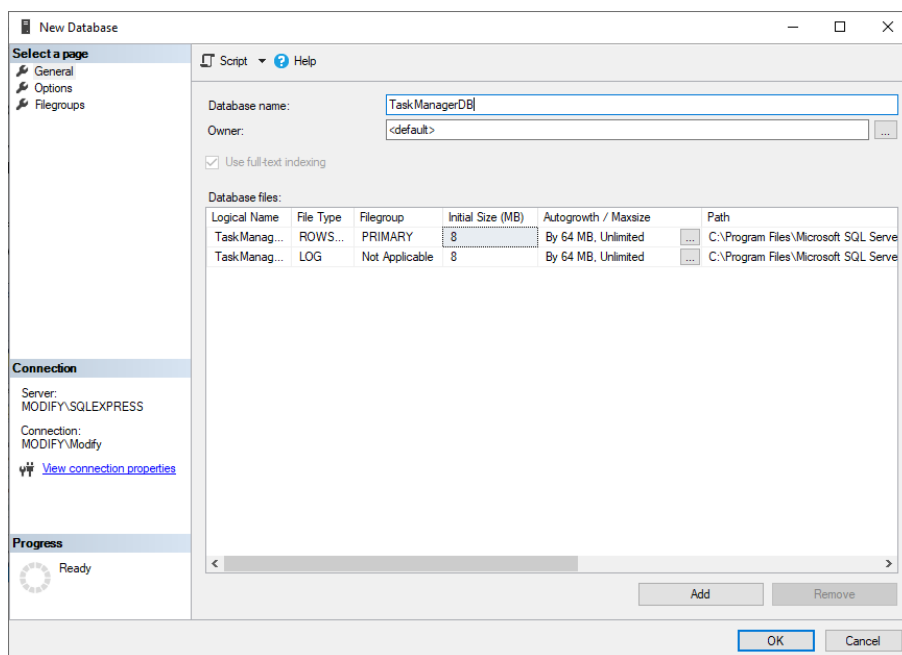
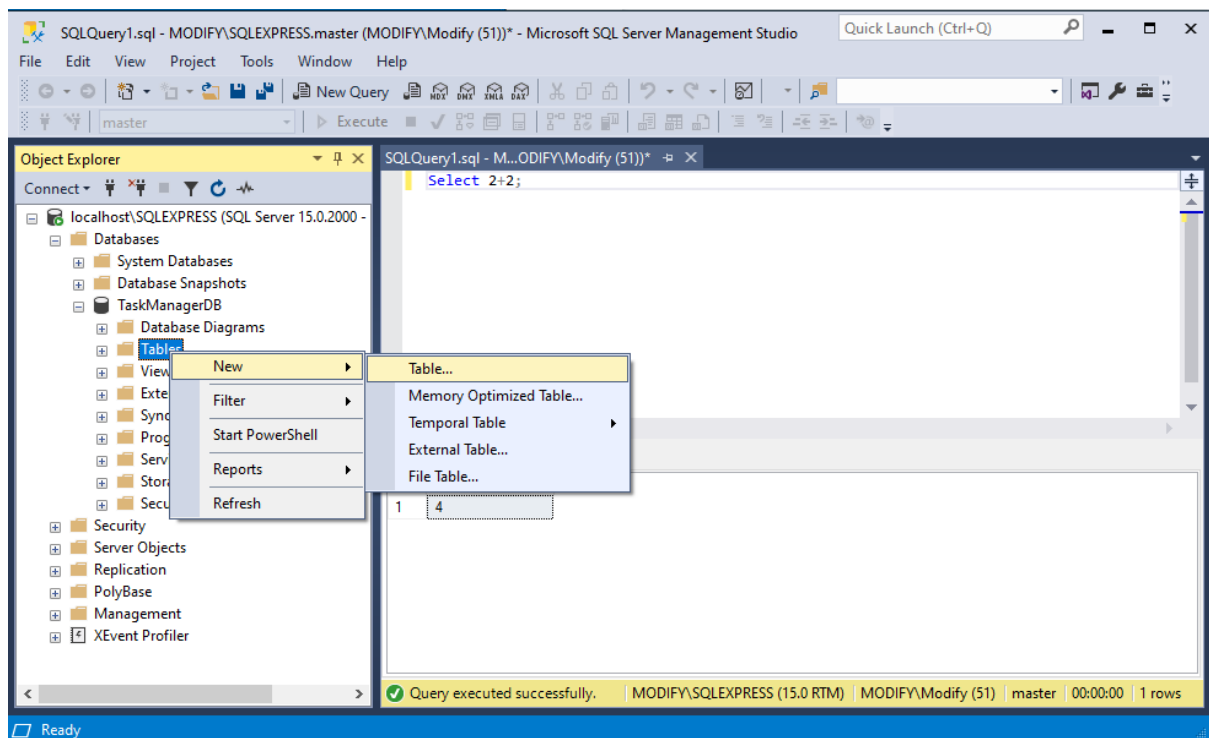
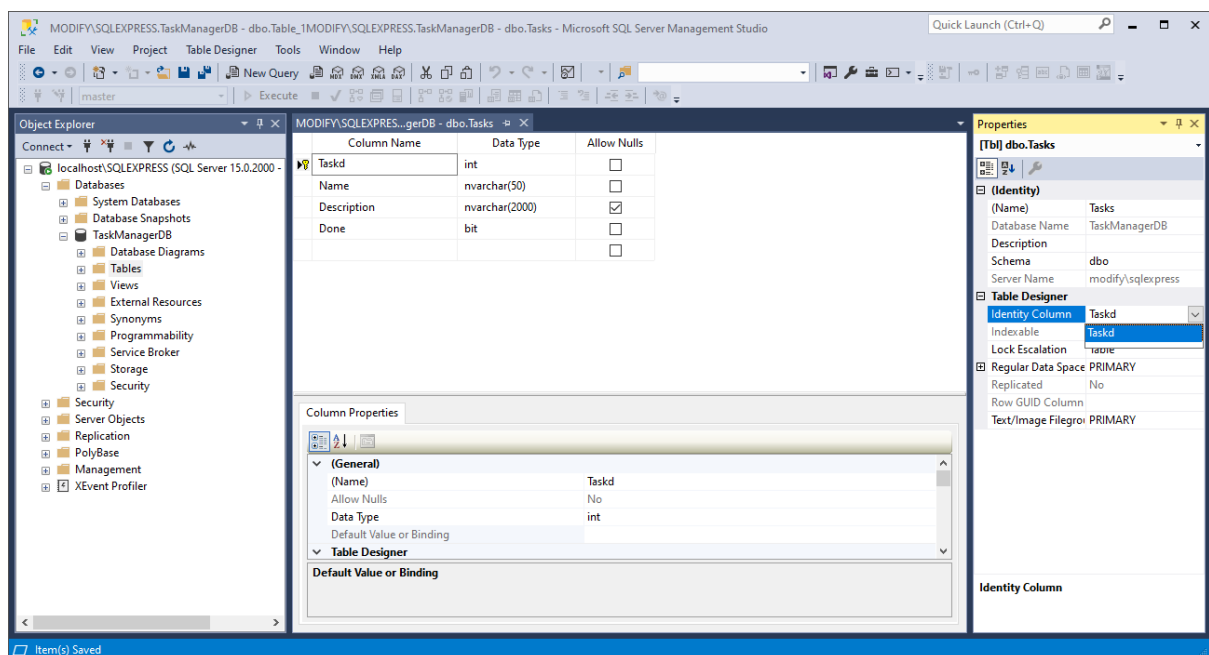


Tabela z zadaniami

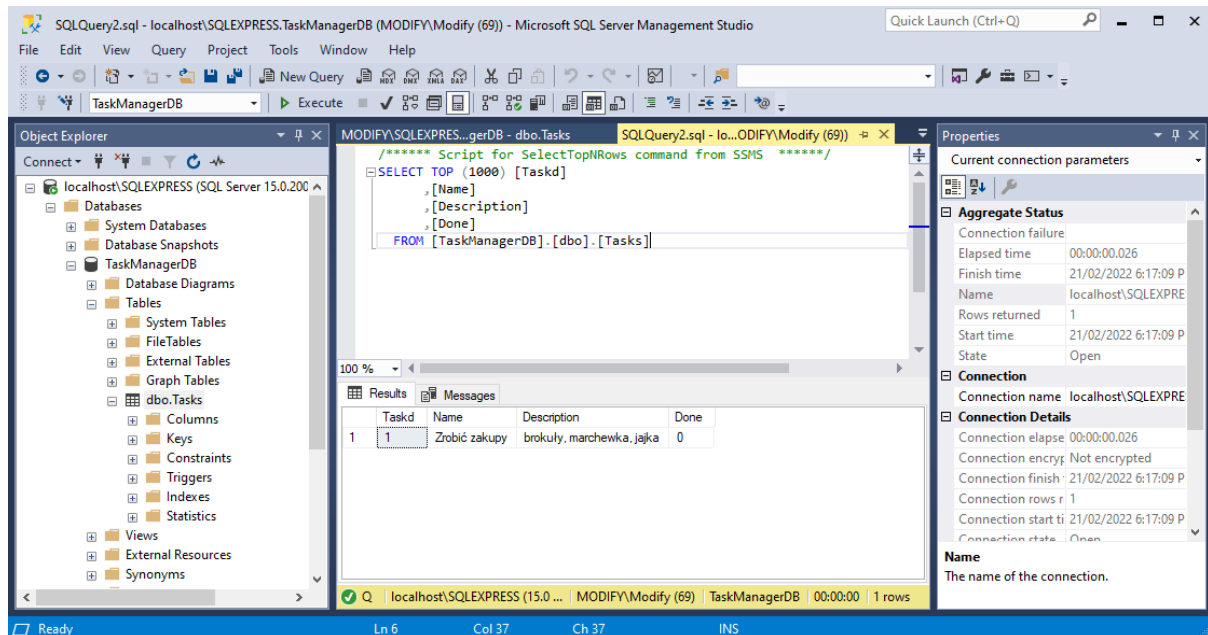
W bazie danych tworzę nową tabelę.



Nazywam tabelę *Tasks* oraz wprowadzam nazwy kolumn. Pierwszą kolumnę (*TaskId*) oznaczam jako klucz główny a także zaznaczam jej auto inkrementację w polu właściwości za pomocą właściwości *Identity Column*.



Sprawdzam poprawność wprowadzonych danych: w tabeli *Tasks* wykonuję zapytanie *Select*. Tabela jest pusta, zatem dodaję jeden przykładowy rekord. Po dodaniu zadania i ponownym wykonaniu *Selecta* widać, że w tabeli znajduje się jeden rekord. Widać też, że do kolumny *Done* wstawiła się domyślna wartość 0.

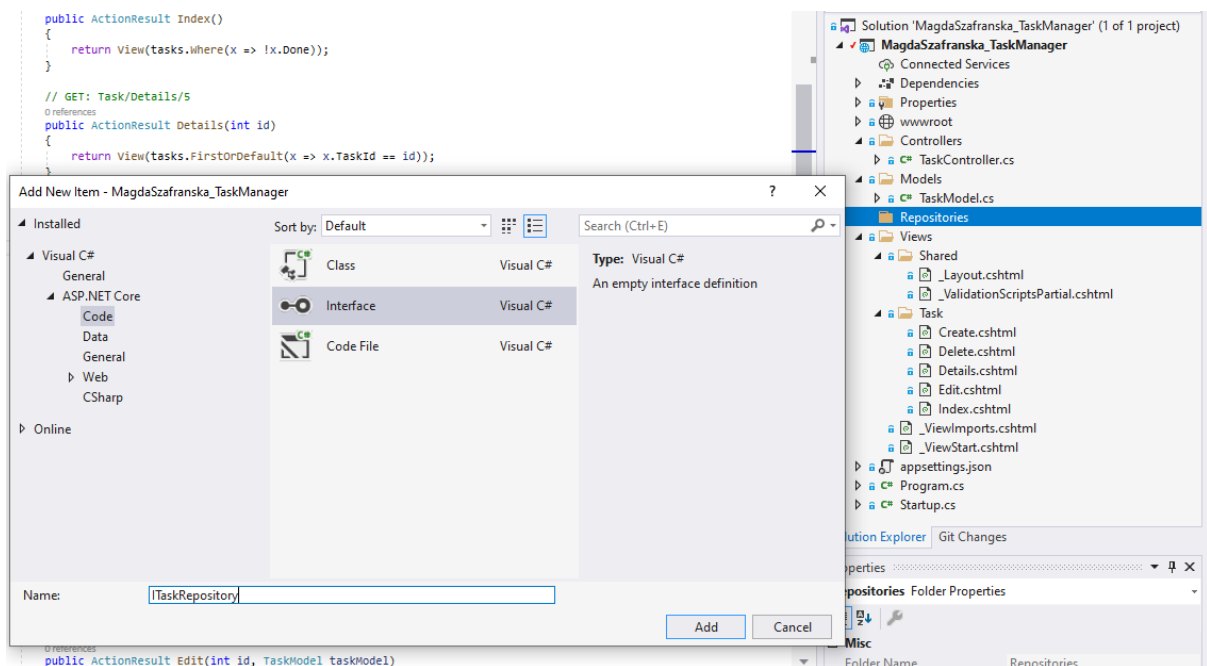


Łączenie aplikacji z bazą danych (Repositories)

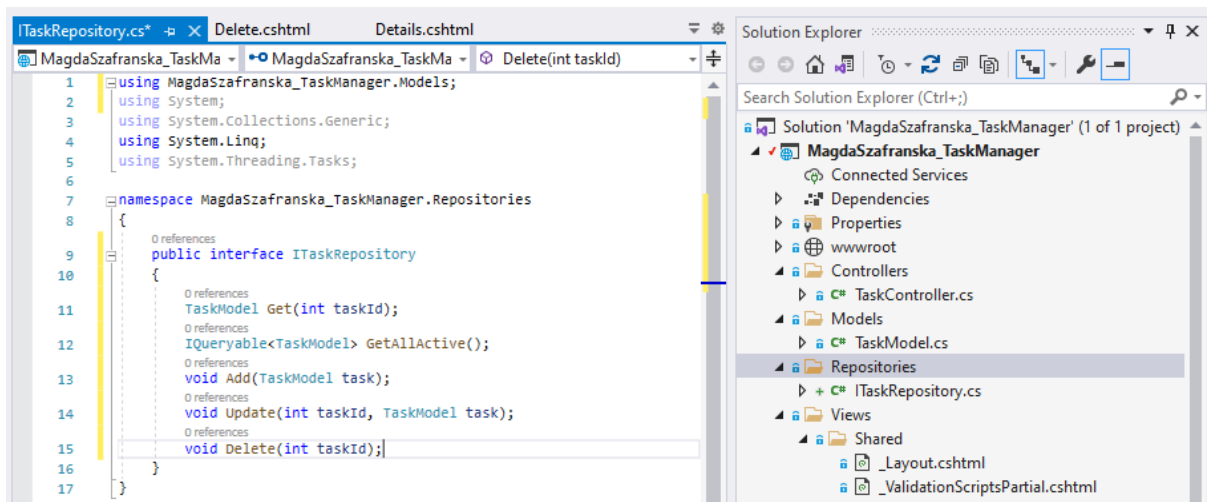
W standardowym podejściu zapytania do bazy danych znajdują się bezpośrednio w kontrolerze lub w klasach z Modelem. Ja jednak zastosuję w projekcie wzorzec repozytorium, który każe przenieść wszystkie operacje korzystające z bazy danych do innej warstwy w aplikacji.

W kontrolerze wywołuję tylko metody z repozytorium, które zwracają dane do kontrolera. Kontroler nie bierze odpowiedzialności za stworzenie kontekstu, czyli obiektu dostępu do danych i nie ma pojęcia o tym skąd pochodzą dane ani jaką strukturę ma baza danych. Kontroler otrzymuje tylko to, co z jego punktu widzenia jest istotne, czyli dane.

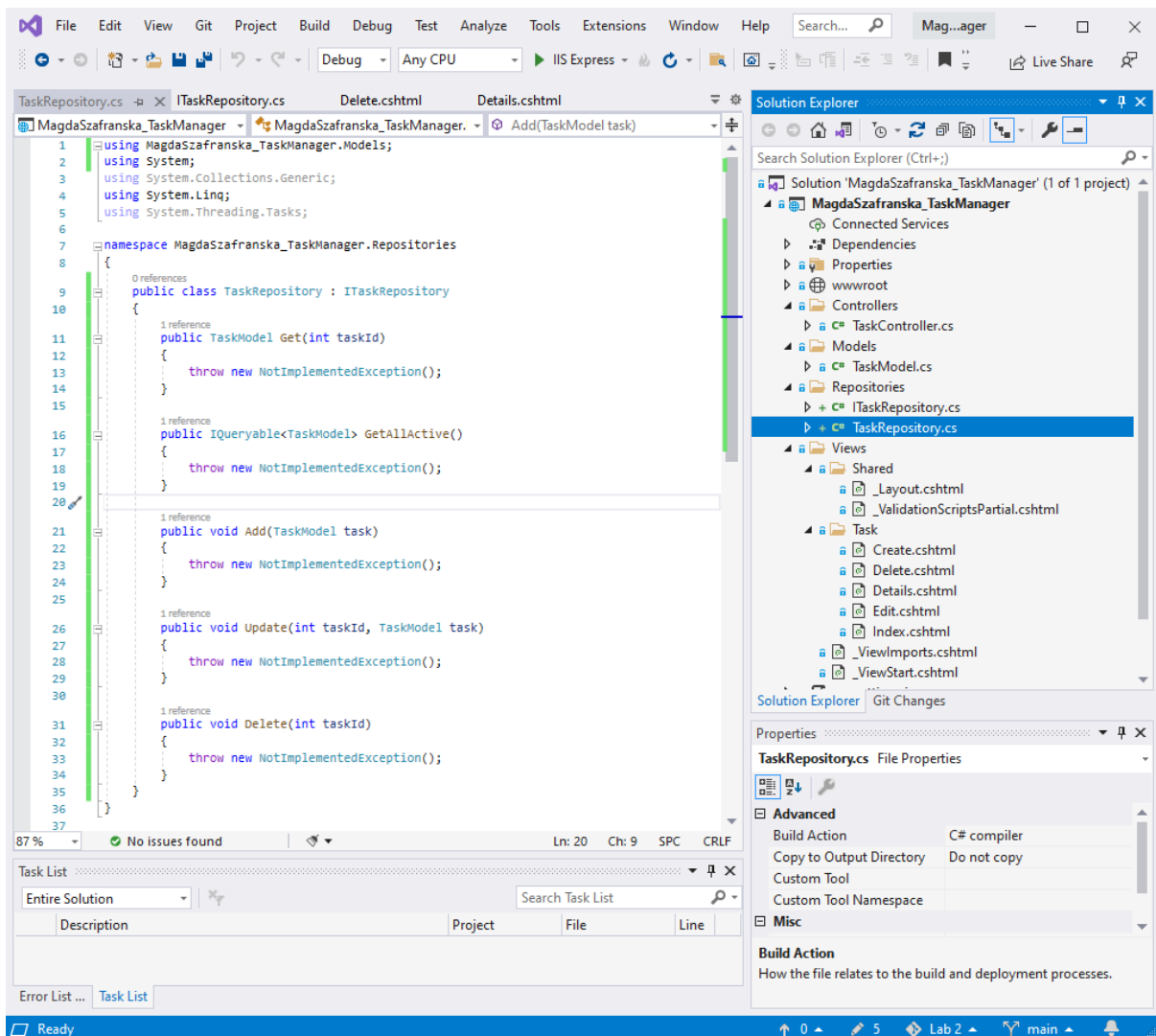
W projekcie tworzę nowy folder *Repositories* i dodaję w nim nowy interfejs *ITaskRepository*.



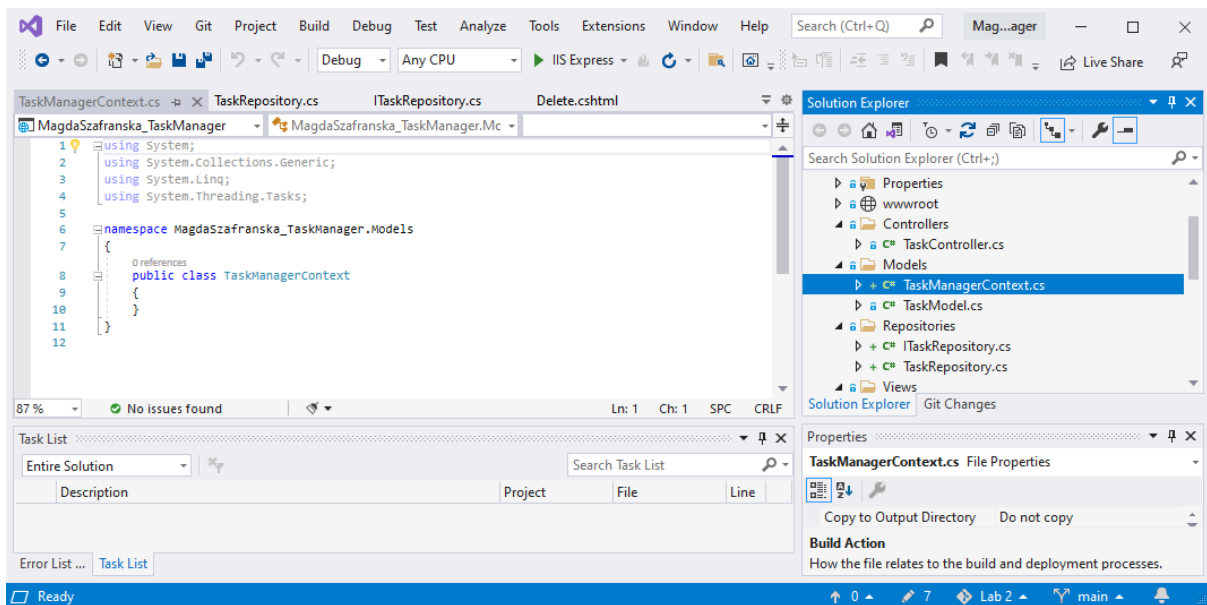
Definiuję sygnatury metod repozytorium.



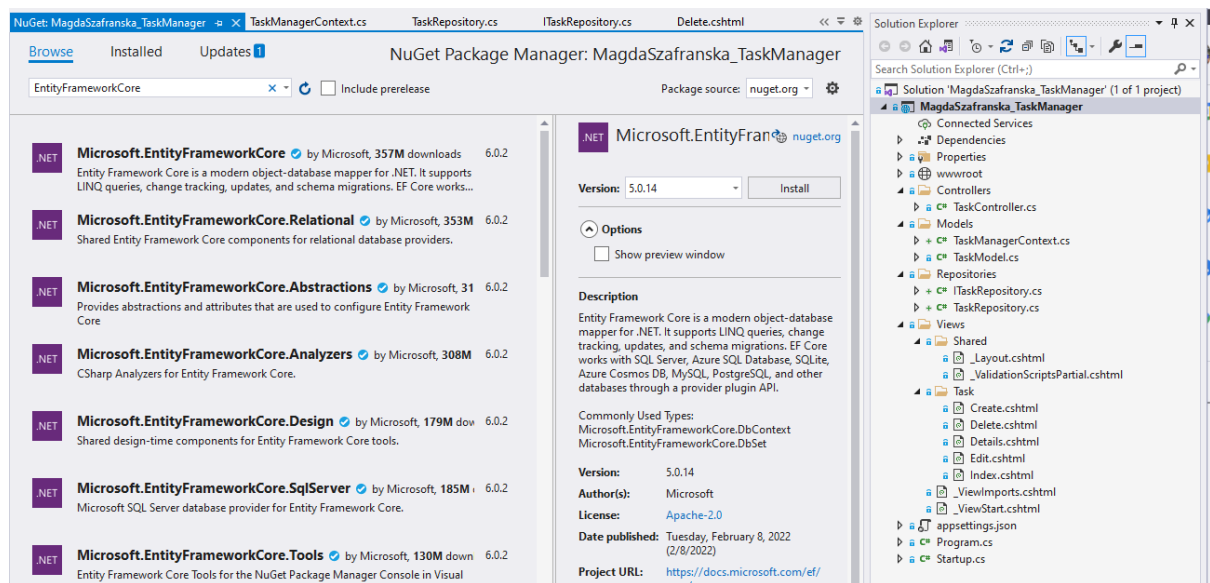
W tym samym folderze *Repositories* tworzę klasę *TaskRepository*, która będzie implementować interfejs *ITaskRepository*.



Tworzę w Modelu klasę kontekstu (*TaskManagerContext*), która będzie reprezentowała połączenie z bazą danych. Dzięki klasie kontekstu będzie można wykonać operacje takie jak tworzenie, odczytywanie, aktualizację oraz usuwanie danych.



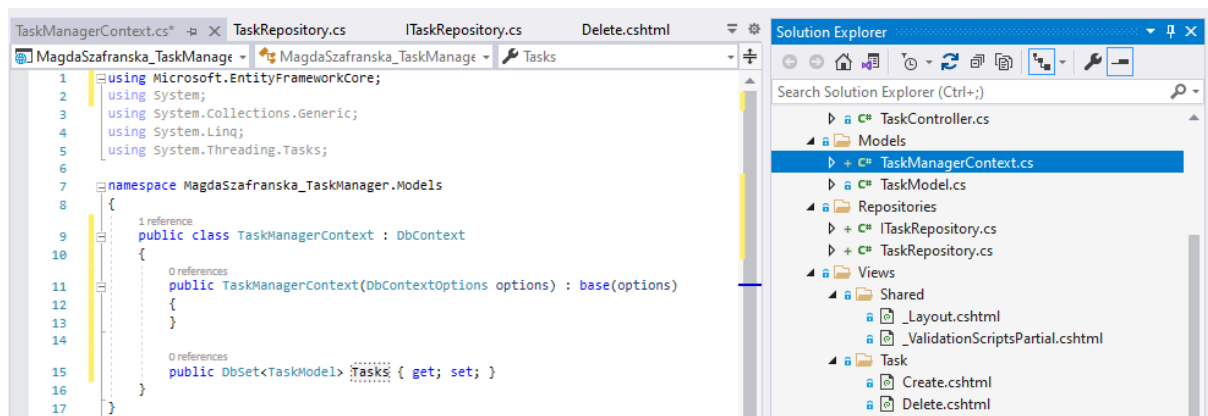
Za pomocą menedżera pakietów NuGet dołączę do projektu biblioteki zapewniające dostęp do danych Entity Framework Core.



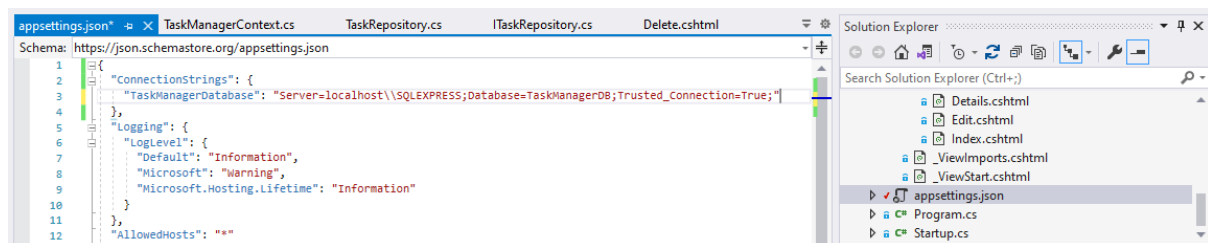
Instaluję również Microsoft.EntityFrameworkCore.SqlServer.

Dodaję dziedziczenie do klasy *TaskManagerContext* po klasie *DbContext* a także konstruktor.

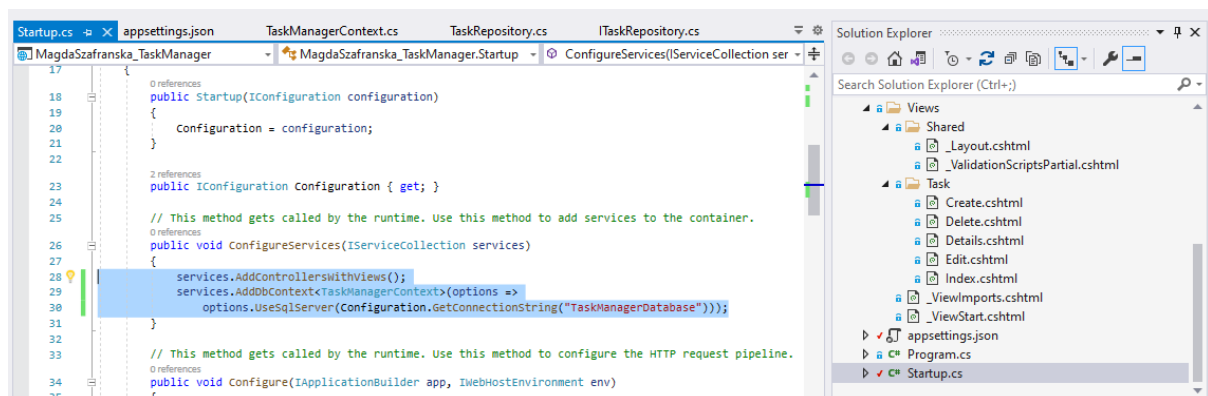
Następnie dodaję właściwość *Tasks* klasy *DbSet*, która reprezentuje zestaw encji, którego z kolei można używać do operacji tworzenia, odczytu, aktualizacji i usuwania. Klasa kontekstu musi zawierać właściwości *DbSet* dla encji, które są mapowane na tabele i widoki bazy danych.



Deklaruję ciąg połączenia do bazy danych (tzw. connection string). W tym celu w pliku *appsettings.json* dodaję nową sekcję *ConnectionStrings*.

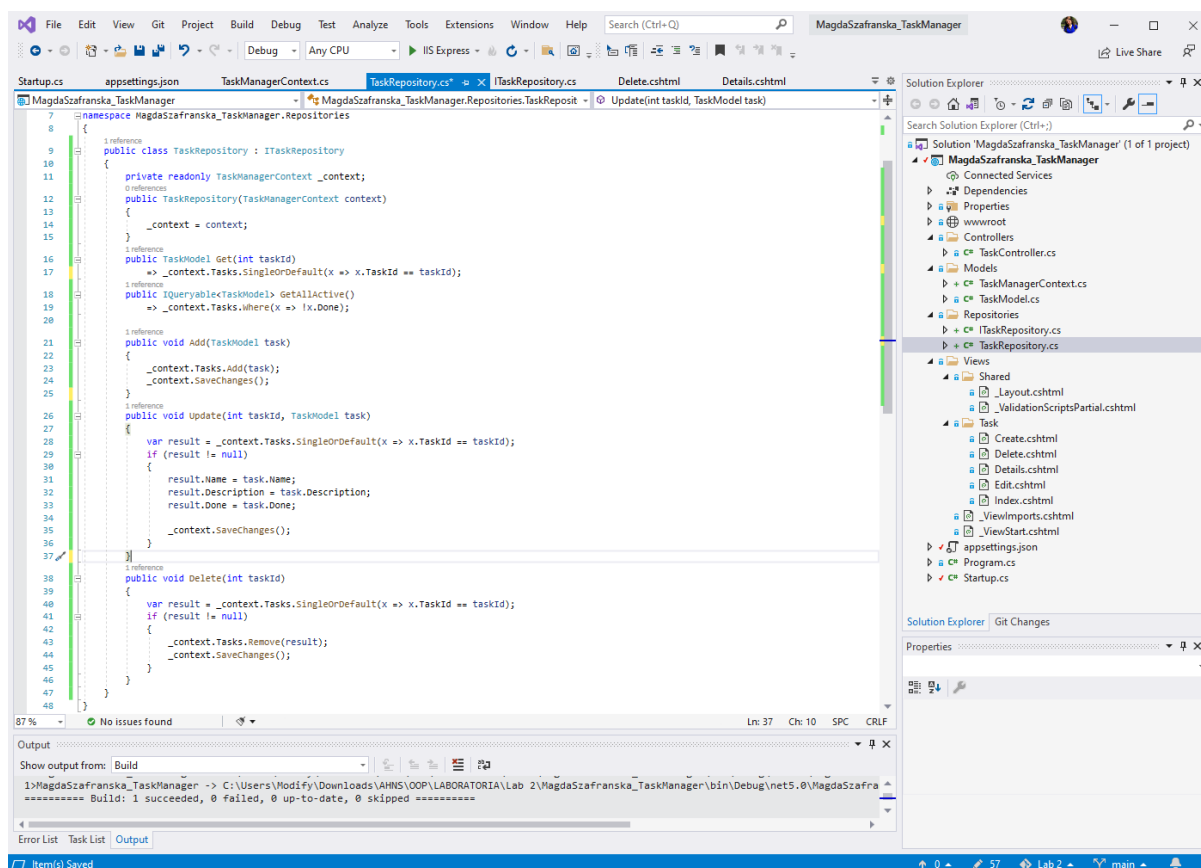


W klasie *Startup.cs* w metodzie *ConfigureServices* konfiguruję kontekst połączenia z bazą. Dodaję odpowiedni kontekst, czyli *TaskManagerContext*, następnie wywołuję metodę *UseSqlServer* wskazującej na dostawcę bazy danych czyli SQL server i jako parametr metody podaję ciąg połączenia do bazy danych. *ConnectionString* z pliku *appsettings.json* mogę pobrać wywołując metodę *GetConnectionString* z klasy *Configuration* podając jako argument nazwę connection stringa.



Teraz mogę przejść do właściwej implementacji repozytorium.

W klasie *TaskRepository* dodaję konstruktor a także prywatne pole tylko do odczytu *_context* typu *TaskManagerContext*. Jako parametr konstruktora dodaję kontekst połączenia do bazy. Modyfikuję pozostałe metody klasy *TaskRepository*.

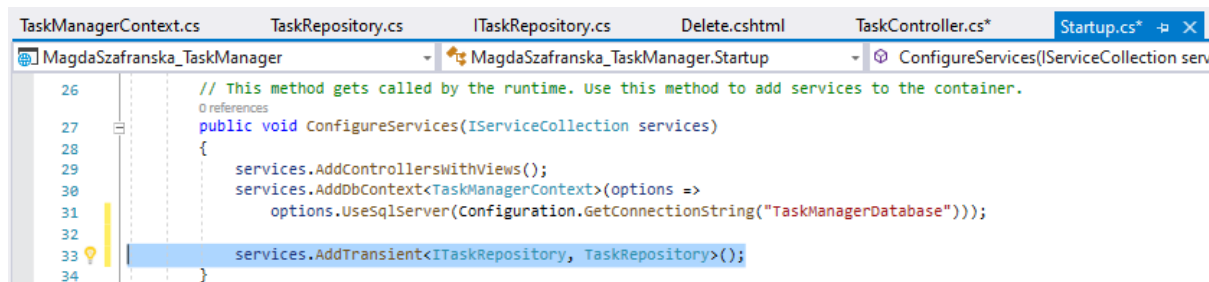


Repozytorium jest w tym momencie gotowe. Przechodzę do kontrolera. Operacje wykonywane w pamięci zamienię na te wykonywane w bazie danych.

Usuam inicjalizację listy z zadaniami. Dodaję prywatne pole tylko do odczytu *taskRepository*. Dodaję także konstruktor z parametrem typu *ITaskRepository*. Następnie wstrzykuję zależności do konstruktora kontrolera.

Dependency Injection, czyli wstrzykiwanie zależności, polega na przekazywaniu gotowych utworzonych instancji obiektów udostępniających swoje metody i właściwości obiektom, które z nich korzystają, np. jako parametry konstruktora.

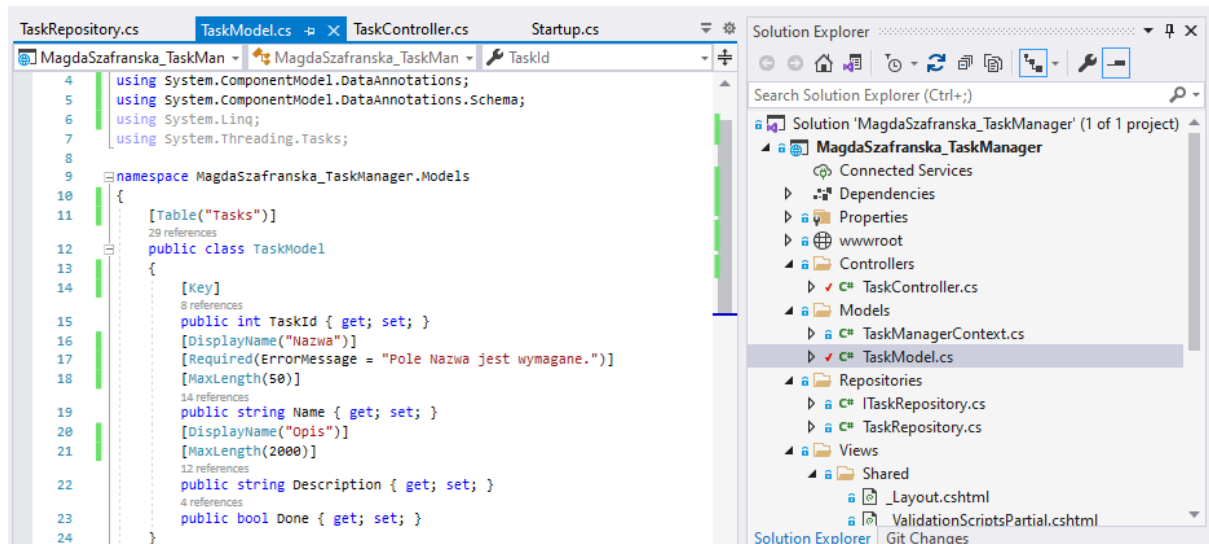
ASP.NET Core oferuje własny kontener do wstrzykiwania zależności, który odpowiedzialny jest za tworzenie i używanie zależności z dowolnym obiektem, który jest wymagany przez inny obiekt. W moim przypadku klasa *TaskController.cs* zależy od interfejsu *ITaskRepository*, który jest parametrem konstruktora. Jeśli chcę wstrzyknąć ten interfejs do kontrolera to muszę zarejestrować odpowiednią usługę w kontenerze usług.



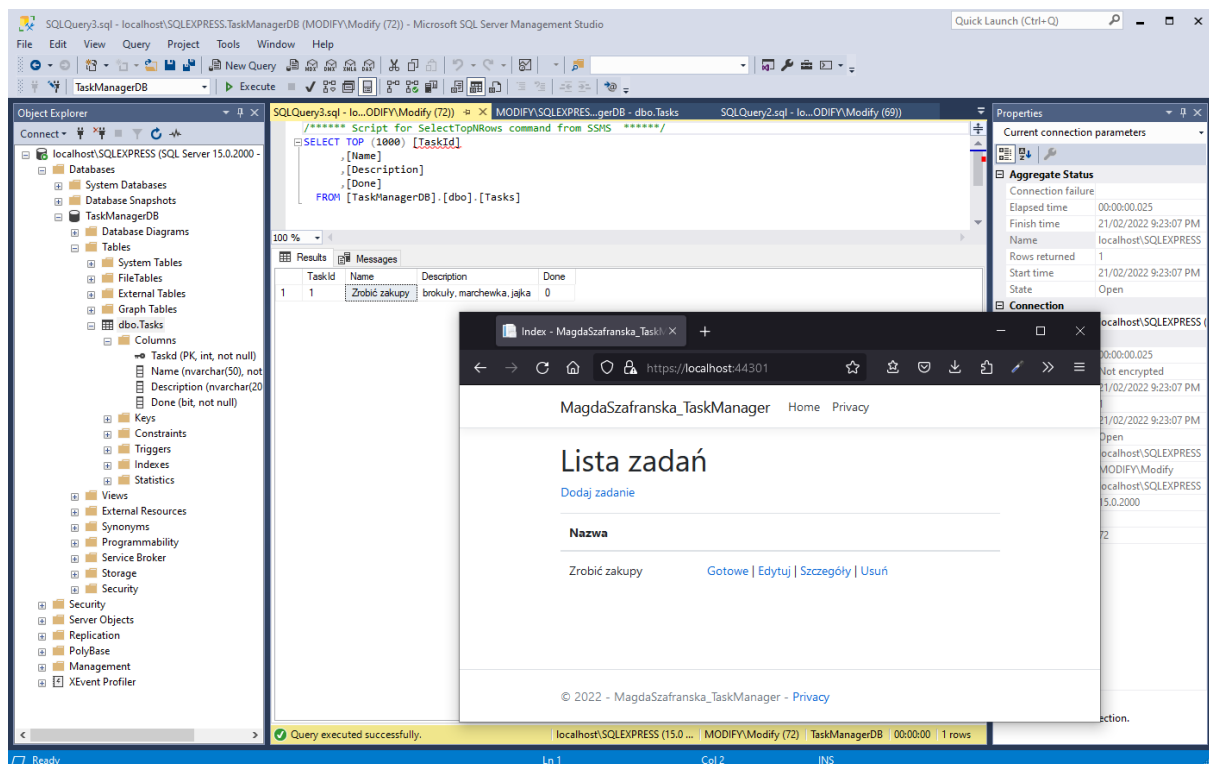
Teraz za każdym razem, kiedy będzie wywoływany konstruktor kontrolera, zostanie do niego wstrzyknięty obiekt *taskRepository*, który implementuje interfejs *ITaskRepository*.

Zmieniam implementację wykonywanych operacji we wszystkich metodach kontrolera na taką, która wykorzystuje *Repository*.

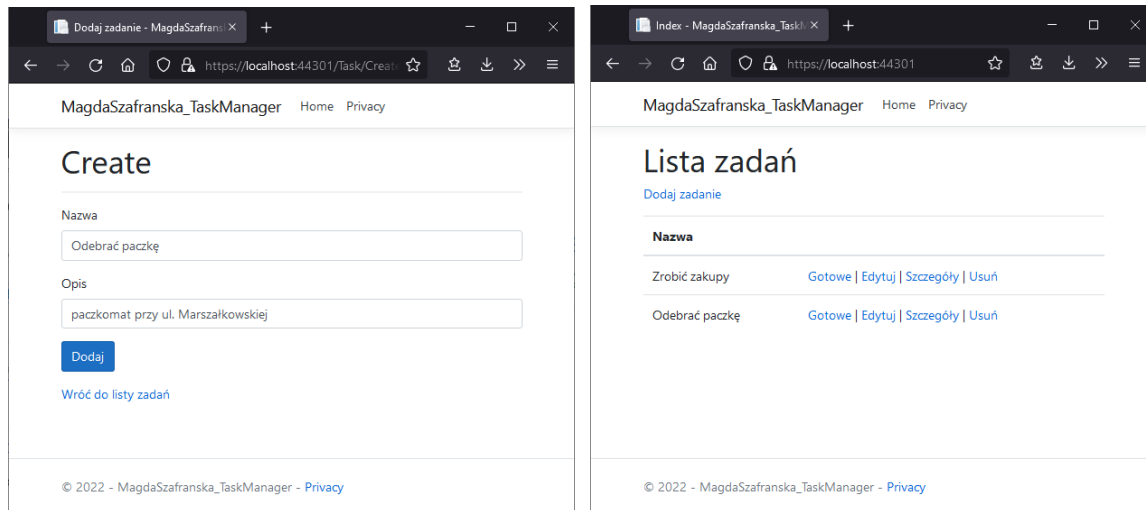
Dodaję w klasie *TaskModel* kilka atrybutów jak na screenie poniżej.



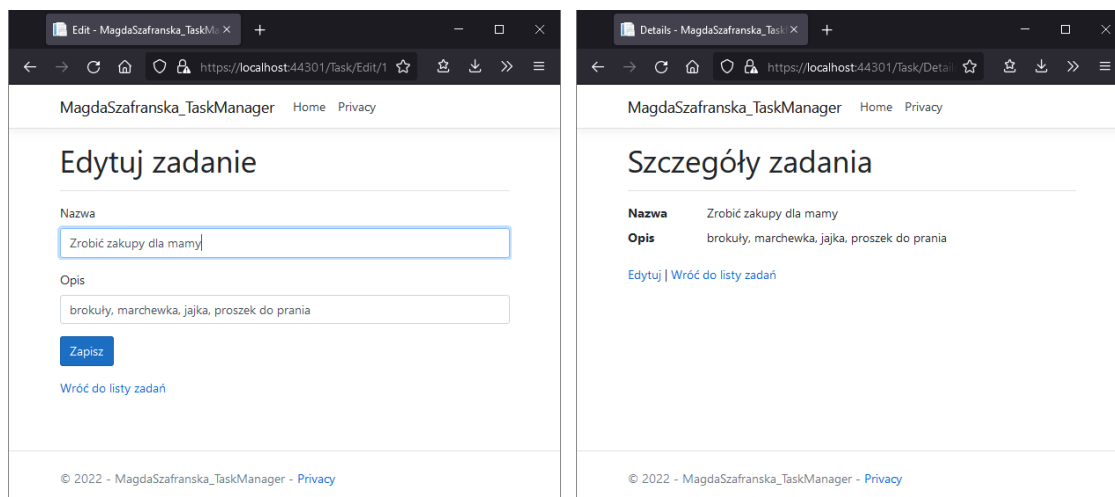
Uruchamiam aplikację. Na liście zadań widzę zadanie, które ręcznie dodałam do bazy danych.



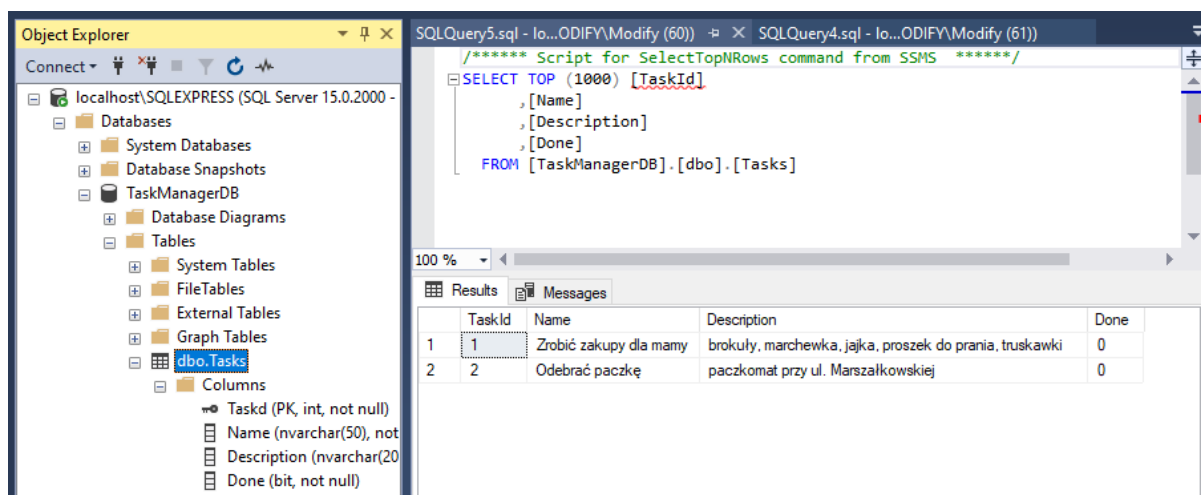
Teraz dodam kolejne zadanie już w mojej aplikacji. Zadanie pojawiło się na liście w aplikacji.



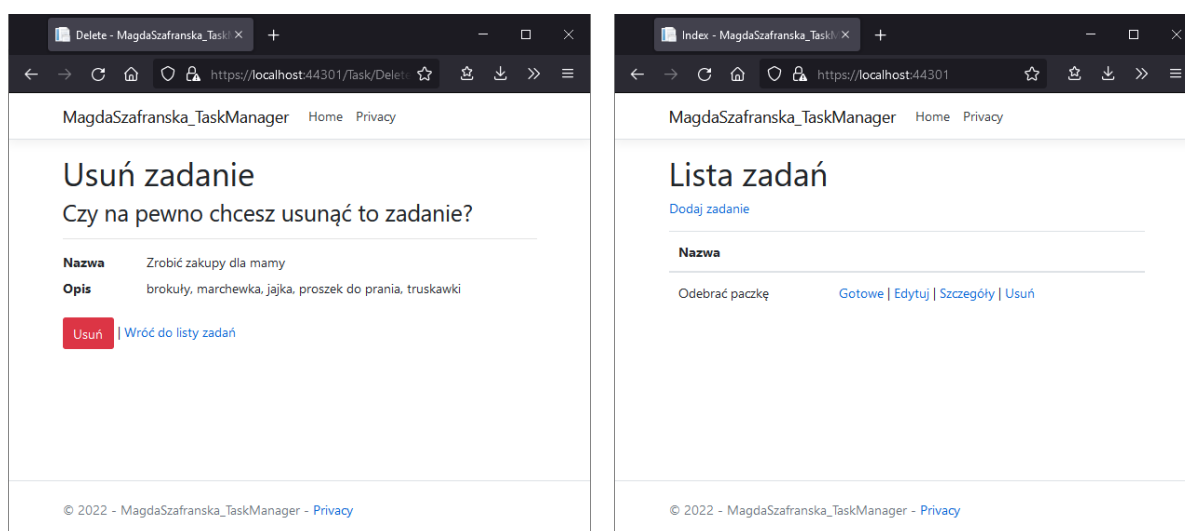
Edytuję teraz pierwsze na liście zadanie, zapisuję, po czym przechodzę do szczegółów tego zadania. Zarówno nazwa jak i opis zadania zostały zaktualizowane.



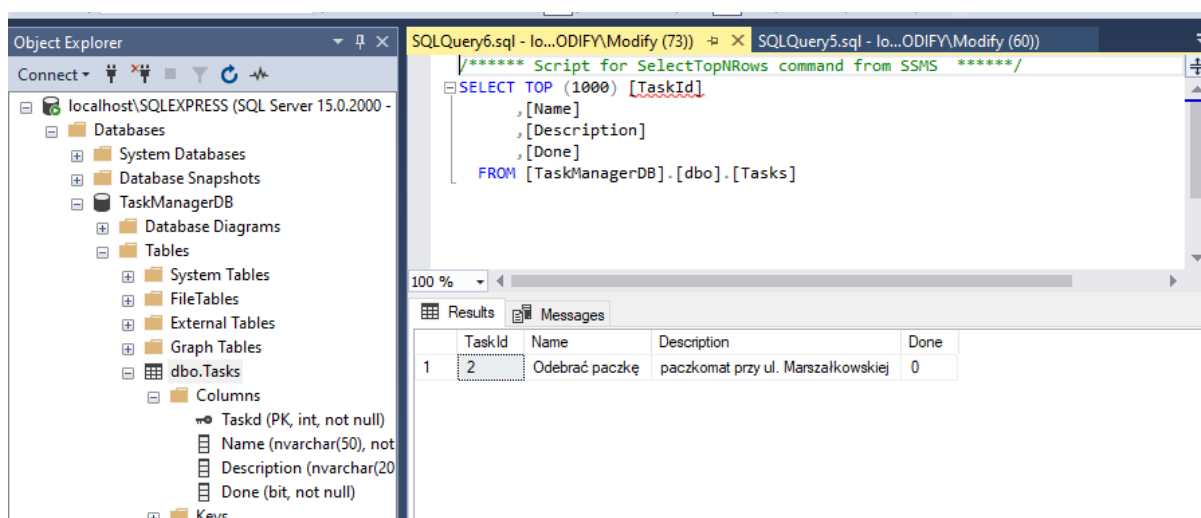
Jeszcze raz edytuję zadanie z poziomu *Szczegółów zadania* dopisując coś do opisu. Tymczasem, w bazie danych widoczna już jest dokonana przed chwilą zmiana.



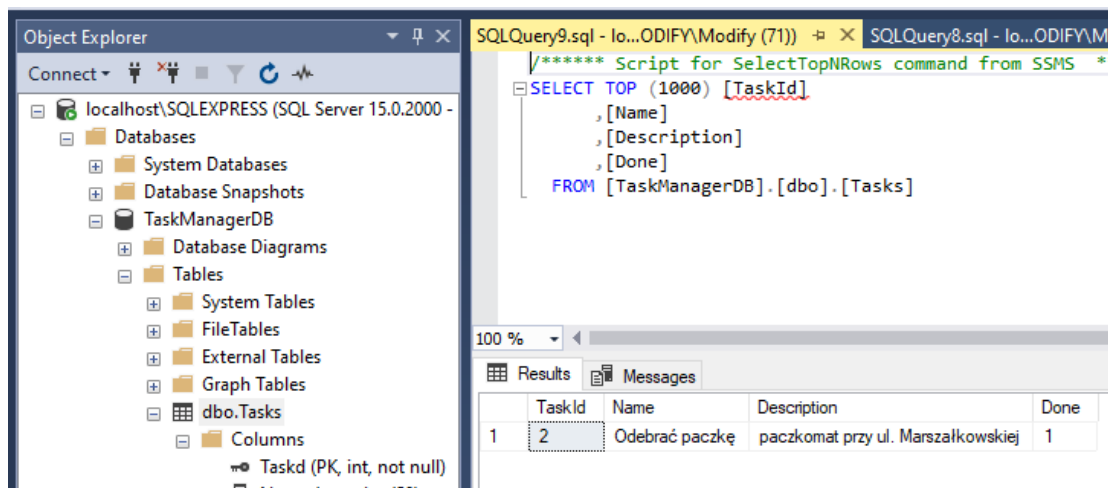
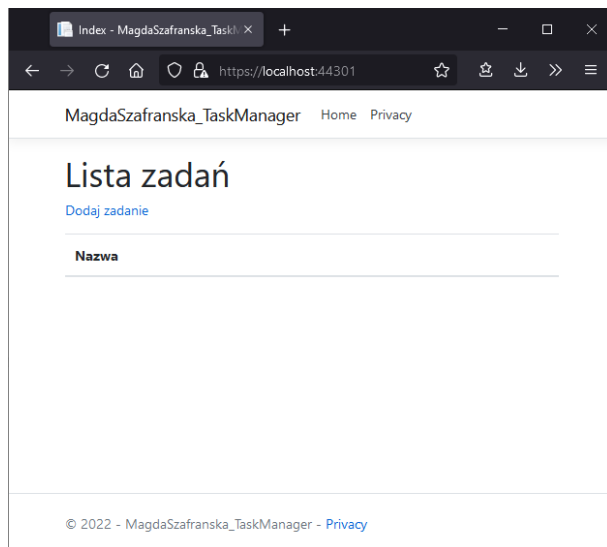
Usuwa zadanie z listy w aplikacji.



Zadanie zostało usunięte z listy oraz z bazy.

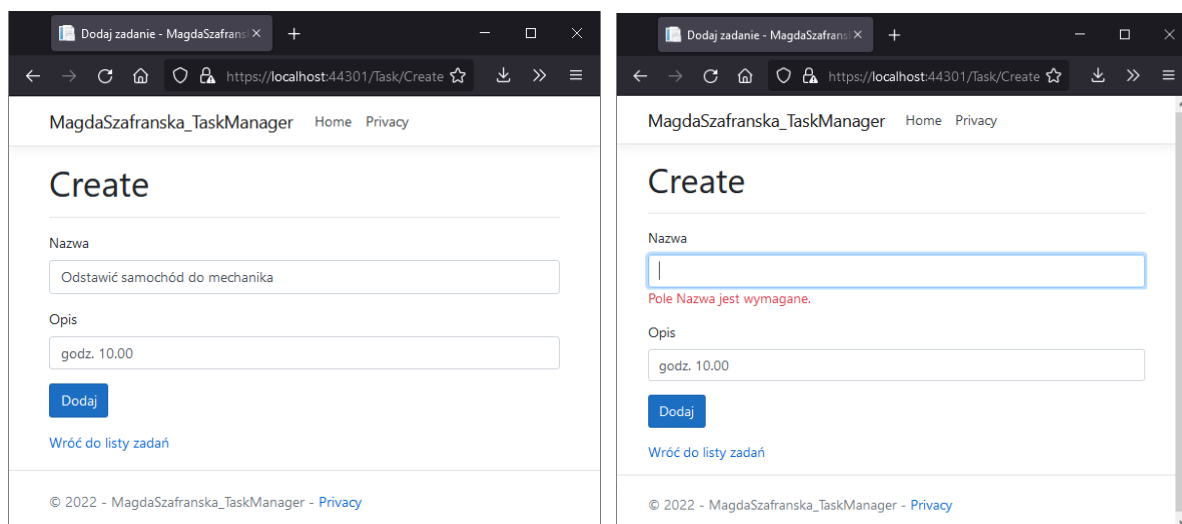


Przy zadaniu, które pozostało w aplikacji klikam przycisk *Gotowe*. Zadanie zniknęło z listy, a w bazie danych zostało zaznaczone jako zakończone (w kolumnie *Done* ma wartość 1).

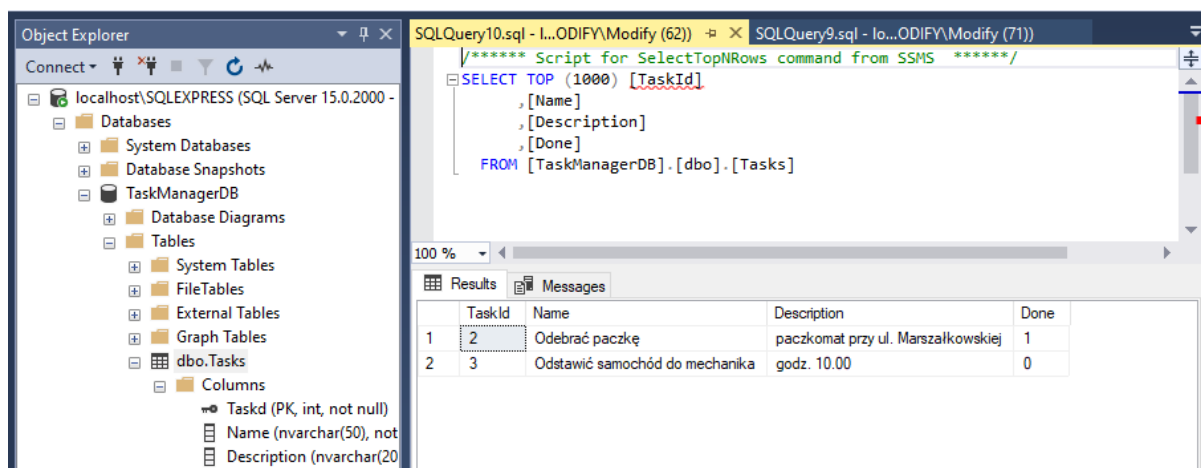
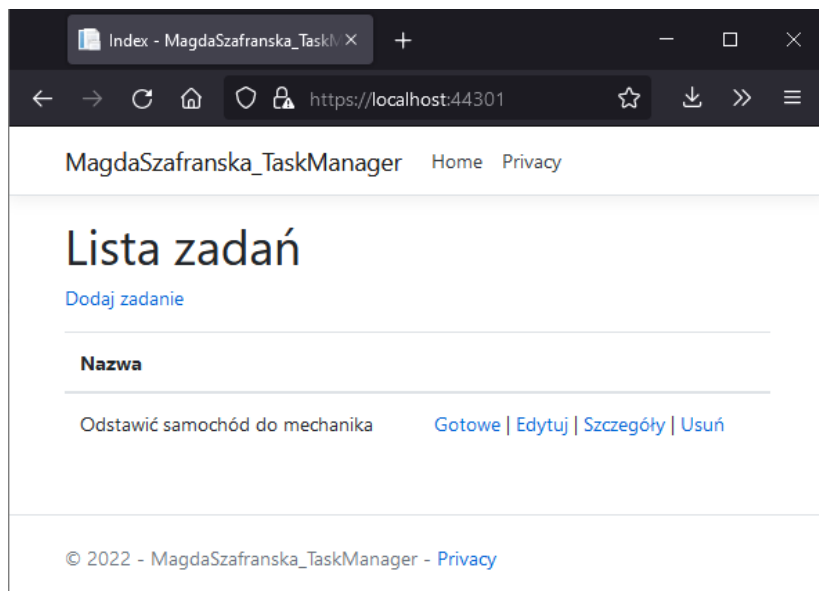


Jak widać wszystkie operacje działają prawidłowo.

Sprawdzam teraz dodatkowo walidację. Dodaję zadanie: wprowadzam nazwę, która jest wymagana oraz dodaję opis. Następnie usuwam nazwę. Zgodnie z oczekiwaniem wyświetlił się komunikat błędu *"Pole Nazwa jest wymagane."* Bez uzupełnienia tego pola dodanie zadania jest niemożliwe.



Po ponownym dodaniu nazwy zadania, nowe zadanie zostało stworzone a także dodane do bazy danych.

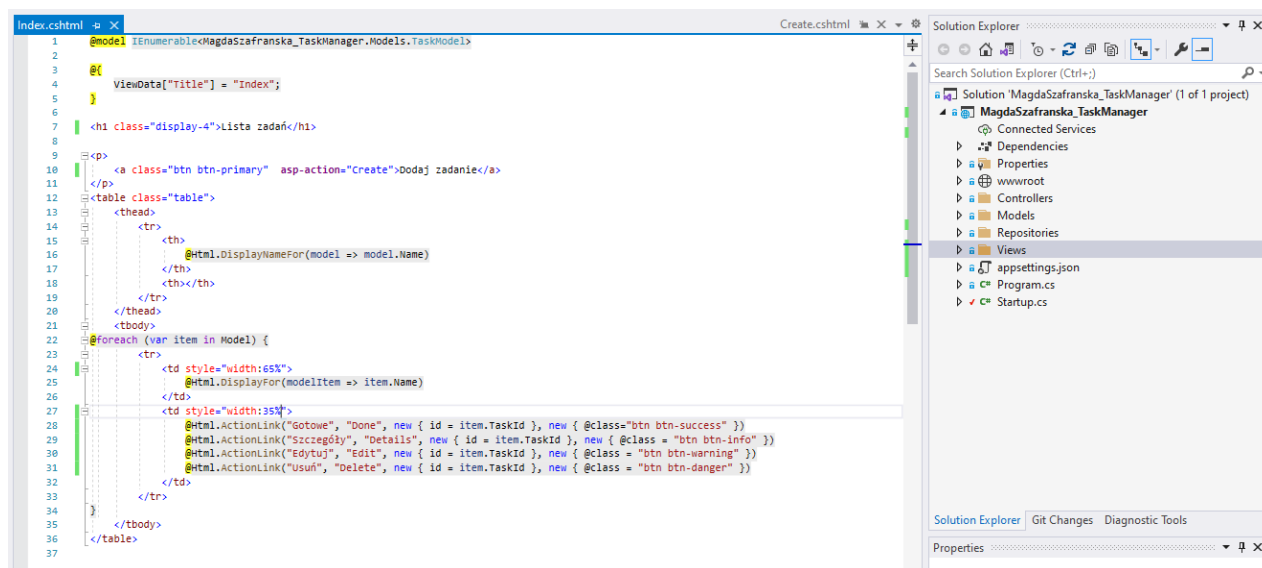


Poprawa wyglądu (Bootstrap)

Poprawiam nieco wygląd moich formularzy za pomocą klas Bootstrapa.

Bootstrap zawiera zestaw wielu przydatnych narzędzi ułatwiających tworzenie interfejsu graficznego stron oraz aplikacji internetowych. Bazuje głównie na gotowych rozwiązaniach HTML oraz CSS.

W klasie *Index.cshtml* wyróżniam nagłówki, zmieniam style przycisków jak poniżej.



The screenshot shows the Visual Studio IDE with the *Index.cshtml* file open in the editor. The code is as follows:

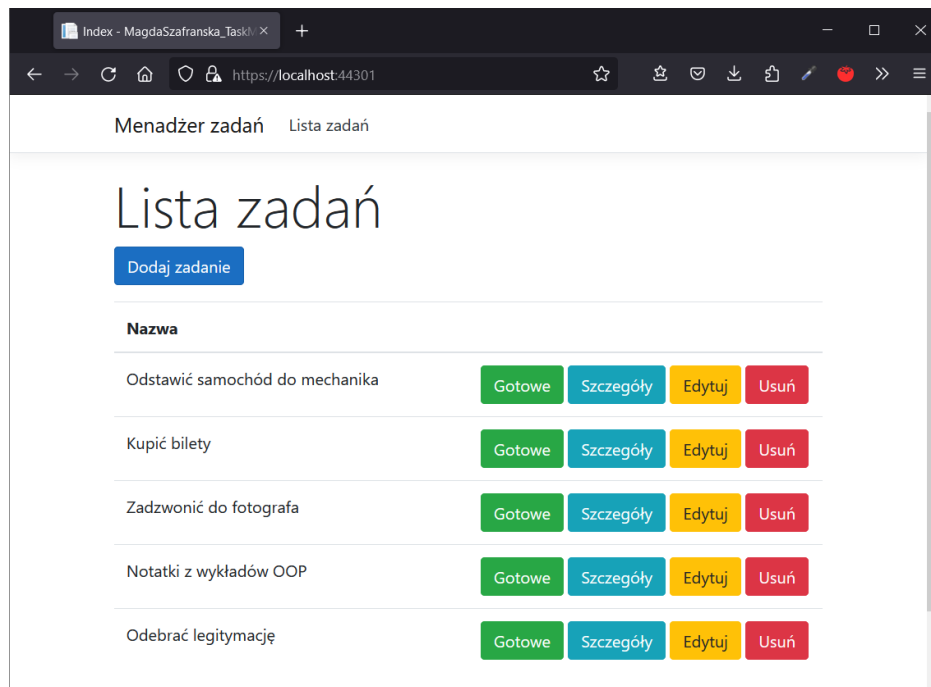
```
1 @model IEnumerable<MagdaSzafranska_TaskManager.Models.TaskModel>
2
3 @{
4     ViewData["Title"] = "Index";
5 }
6
7 <h1 class="display-4">Lista zadań</h1>
8
9 <p>
10     <a class="btn btn-primary" asp-action="Create">Dodaj zadanie</a>
11 </p>
12 <table class="table">
13     <thead>
14         <tr>
15             <th>
16                 @Html.DisplayNameFor(model => model.Name)
17             </th>
18             <th></th>
19         </tr>
20     </thead>
21     <tbody>
22         @foreach (var item in Model) {
23             <tr>
24                 <td style="width:65%">
25                     @Html.DisplayFor(modelItem => item.Name)
26                 </td>
27                 <td style="width:35%">
28                     @Html.ActionLink("Gotowe", "Done", new { id = item.TaskId }, new { @class="btn btn-success" })
29                     @Html.ActionLink("Szczegóły", "Details", new { id = item.TaskId }, new { @class="btn btn-info" })
30                     @Html.ActionLink("Edytuj", "Edit", new { id = item.TaskId }, new { @class="btn btn-warning" })
31                     @Html.ActionLink("Usuń", "Delete", new { id = item.TaskId }, new { @class="btn btn-danger" })
32                 </td>
33             </tr>
34         }
35     </tbody>
36 </table>
37
```

The Solution Explorer on the right shows the project structure for *MagdaSzafranska_TaskManager*, with the *Views* folder selected.

W widoku wprowadzam podobne zmiany w pozostałych klasach (*Create*, *Delete*, *Details*, *Edit* oraz *Layout*).

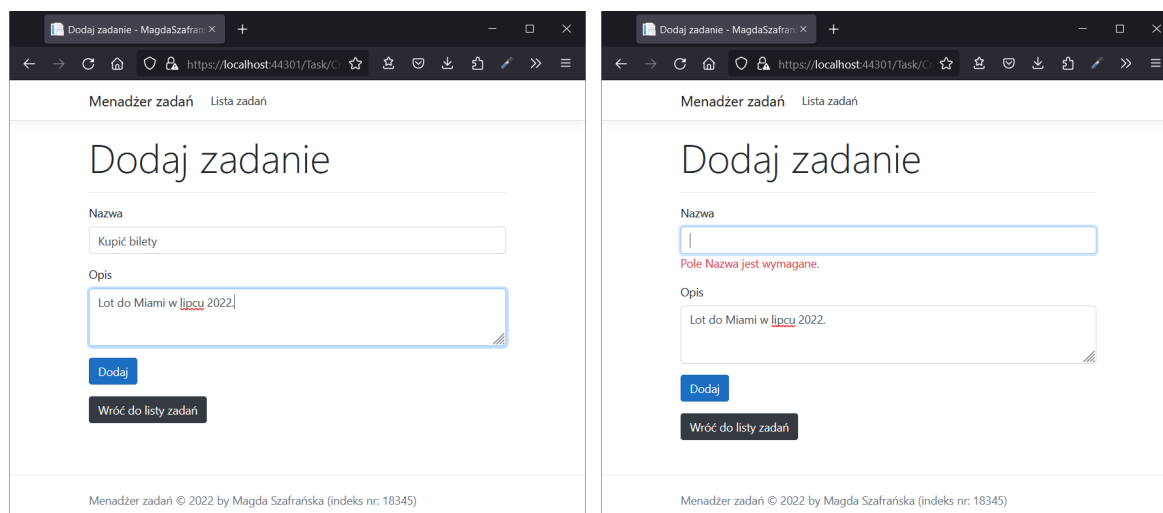
Funkcjonalność

Aplikacja webowa działa w pełni zgodnie z początkowymi założeniami. Posiada pełną współpracę z bazą danych, gdzie wszystkie zmiany pojawiają się w czasie rzeczywistym. Poniżej przedstawiam funkcje, jakie aplikacja obsługuje.



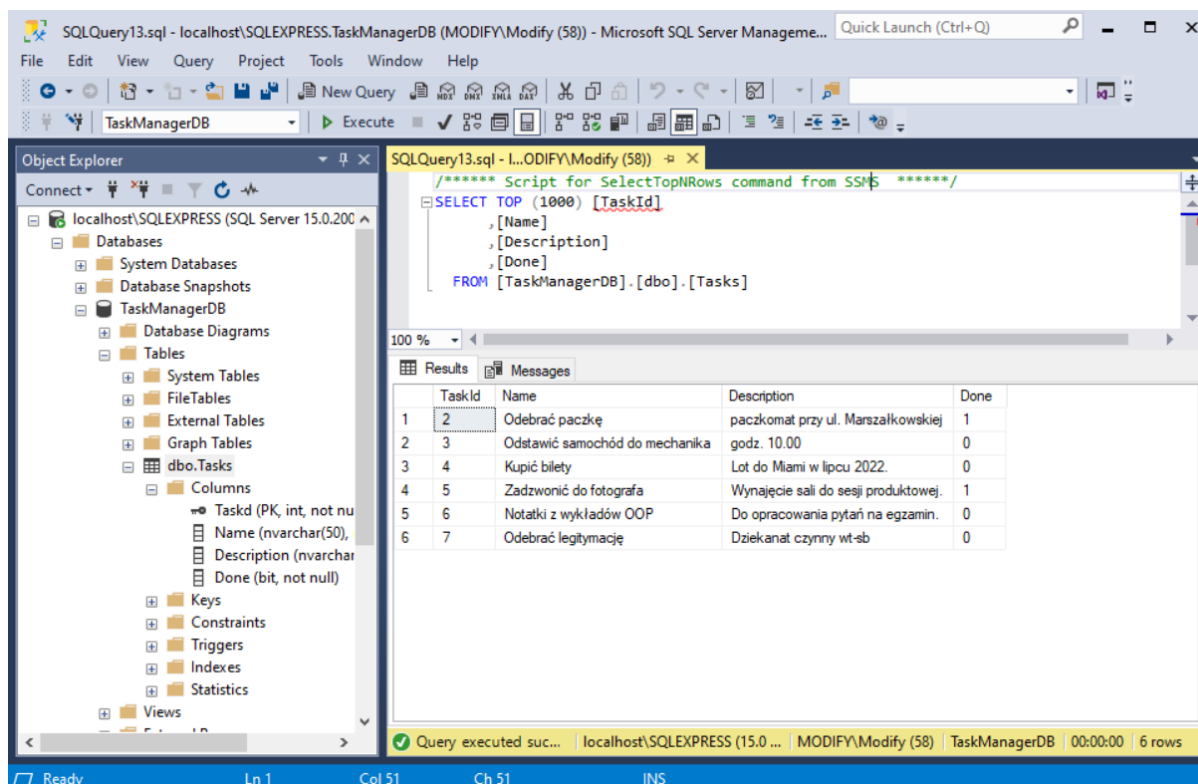
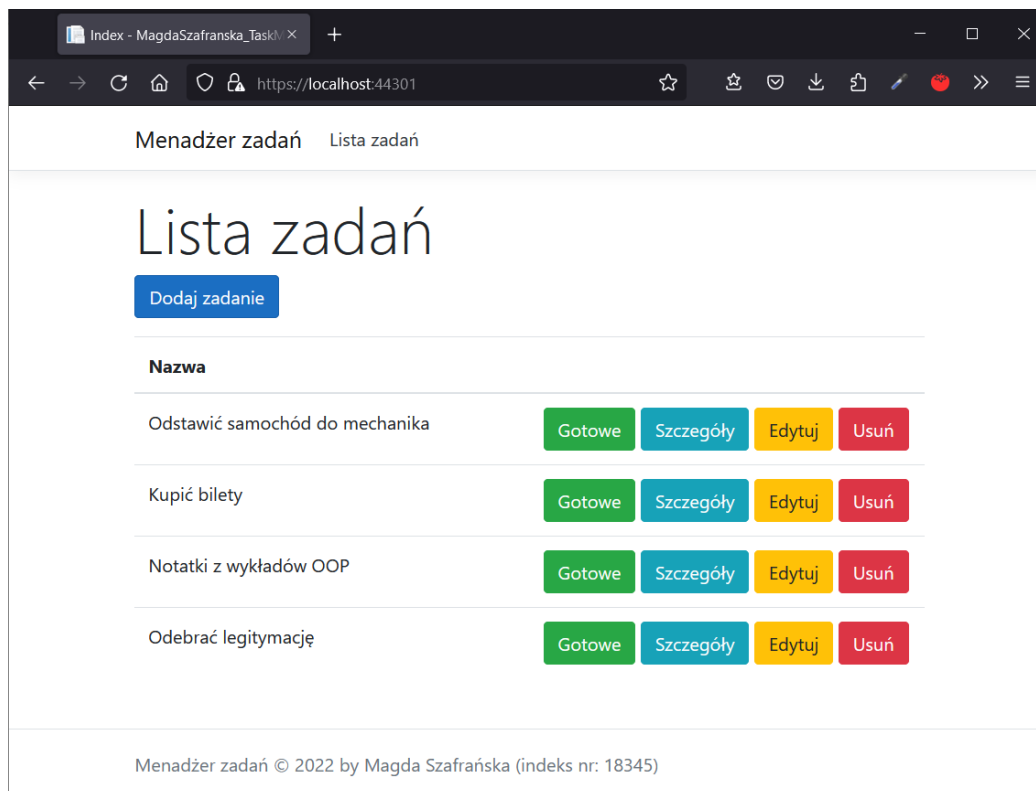
Dodanie nowego zadania

Walidacja na pole *Nazwa*, które jest obowiązkowe. Opis zadania jest opcjonalny.



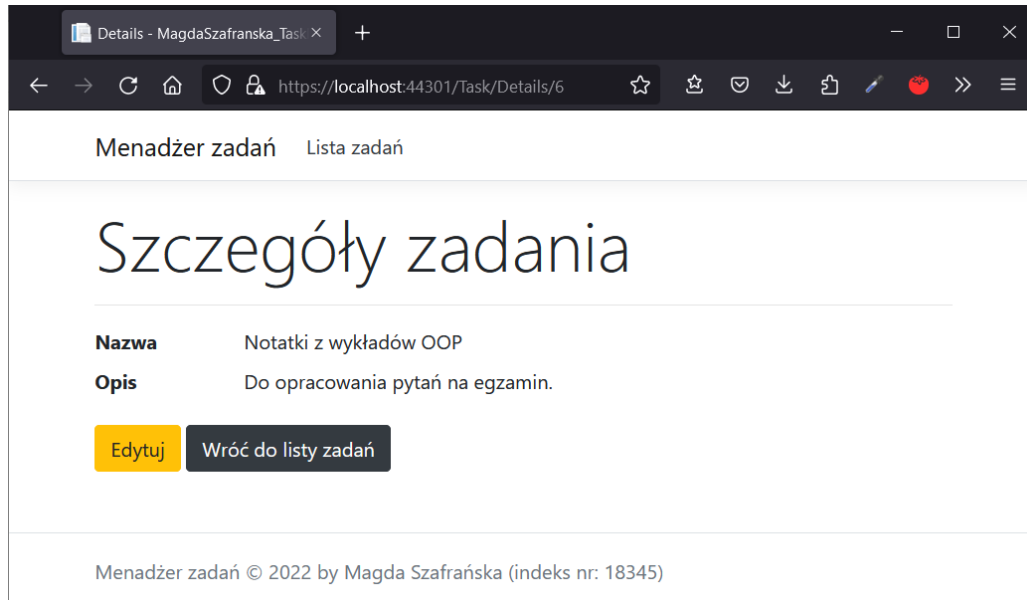
Odznaczenie zadania jako wykonanego

Powoduje to zmianę wartości *Done* w bazie danych na wartość 1 i zadanie nie wyświetla się już w aplikacji internetowej.



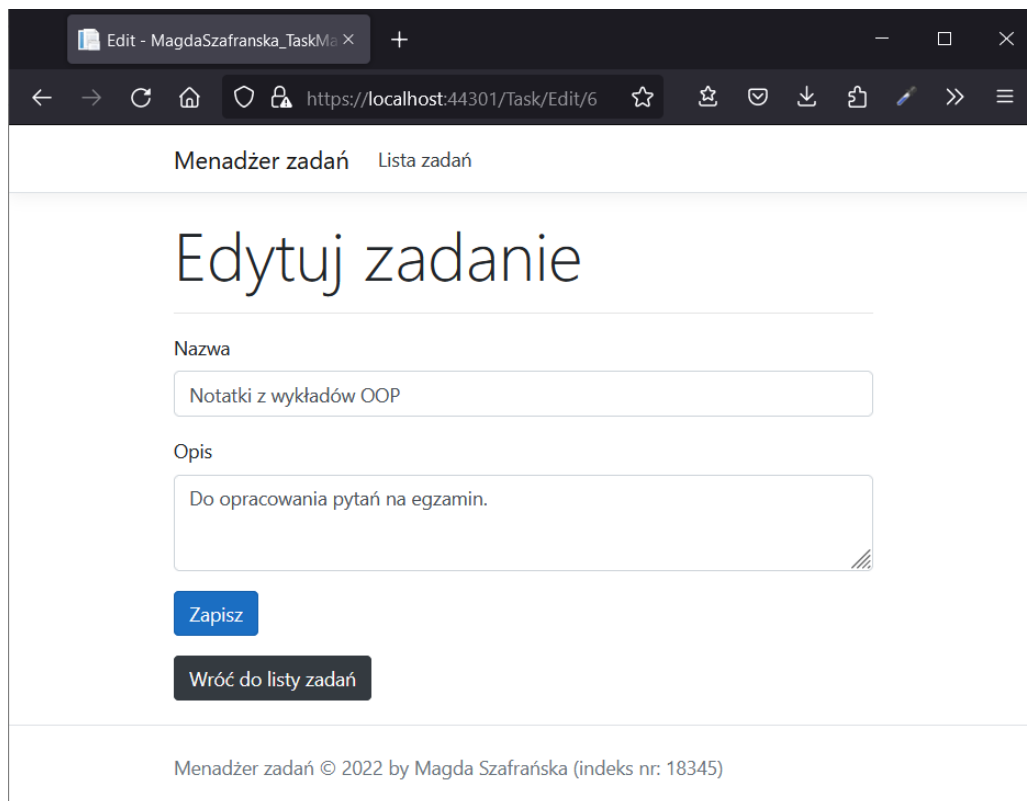
Szczegóły zadania

Widoczna jest wówczas nazwa zadania jak i jego opis. Z tego poziomu jesteśmy w stanie edytować oba te pola.



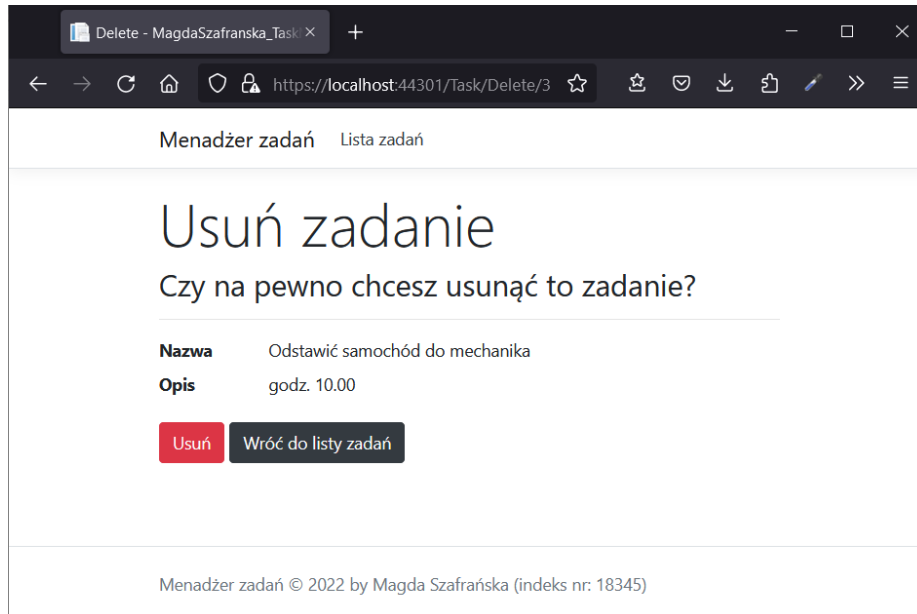
Edycję istniejącego zadania

Dostępna zarówno na stronie głównej jak i z poziomu szczegółów zadania.



Usunięcie wybranego zadania

Ochrona przed przypadkowym kliknięciem w przycisk: jesteśmy przeniesieni na stronę, która potwierdza chęć usunięcia zadania z listy i bazy danych.



Wnioski

Zastosowanie programowania obiektowego w projekcie udowodniło niesamowitą łatwość w potencjalnej dalszej rozbudowie. Wykorzystanie jego założeń bardzo ułatwi mi dalszą rozbudowę projektu o nowe funkcjonalności, jeśli zajdzie taka potrzeba w przyszłości.

Udało mi się zawrzeć wszystkie technologie i założenia planowanej funkcjonalności aplikacji. Podział na klasy idealnie odwzorowuje opisywaną we wstępie przydatność szczególnie w pracy zespołowej, kiedy członkowie zespołu mogą pracować równocześnie, niezależnie od siebie.