Dokumentacja aplikacji sieciowej PomoToDoro

Spis treści

Artur Kotyra

Opis projektu

Celem aplikacji sieciowej PomoToDoro jest połączenie funkcjonalności popularnych aplikacji typu ToDo oraz Pomodoro. Aplikacje typu ToDo są cyfrowym odpowiednikiem robienia na papierze list rzeczy, które musimy zrobić. Takie aplikacje pozwalają użytkownikowi zdefiniować zadania, które chce wykonać oraz usunąć je po ich wykonaniu. Pozwala to na dokładniejsze zorganizowanie swojej pracy. Powszechnie stosowaną metodą zwiększania produktywności jest również technika Pomodoro, opracowana przez Francesco Cirillo jeszcze w latach 80., polegająca na podziale swojej pracy na bloki po 25 minut i krótkie przerwy między nimi. Nazwa techniki pochodzi od kuchennego minutnika w kształcie pomidora. Obie wspomniane funkcjonalności stanowią samoistną, przydatną całość, ale na rynku brakuje projektów, które łączyły by je w jedną aplikację.

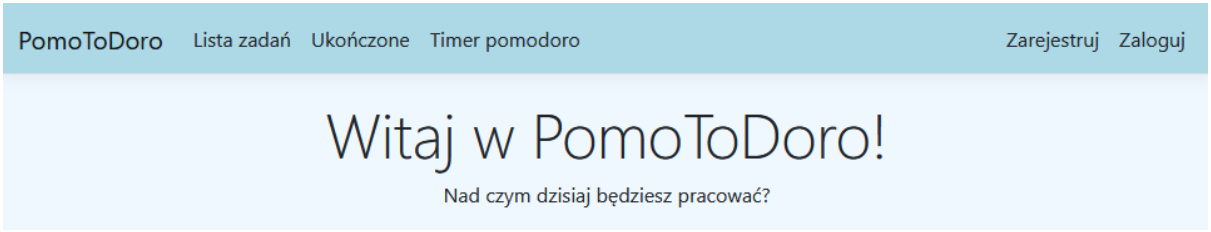
Wykorzystane technologie

Aplikacja została stworzona we frameworku ASP.NET z wykorzystaniem wzorca MVC, cechującego się podziałem na modele, widoki i kontrolery. Aplikacja została zaprogramowana głównie w języku C#. Do prawidłowego działania funkcji timera, został użyty również język JavaSript. Program korzysta z bazy danych w systemie SQLite oraz realizuje podstawowe funkcje CRUD. Do odpowiedniego działania niektórych funkcji aplikacji oraz do elementów kosmetycznych wykorzystana została biblioteka Bootstrap. Całość została zrealizowana w środowisku JetBrains Rider.

Architektura aplikacji

Aplikacja posiada cztery główne zakładki. Użytkownik jako pierwszą zobaczy stronę główną z prostym powitalnym komunikatem, z której może przejść do pozostałych trzech zakładek. Odnośniki do wszystkich zakładek są zdefiniowane w pliku _Layout.cshtml i są wspólne dla wszystkich widoków aplikacji. Oprócz odnośników do zakładek spełniających główne funkcjonalności programu, użytkownik ma do dyspozycji również odnośniki do rejestracji i logowania, generowane automatycznie przez środowisko.

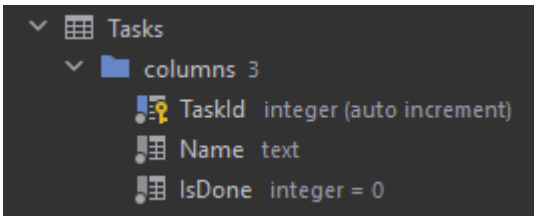
Strona główna - widok Index kontrolera HomeController:



Odnośniki do zakładek w pliku _Layout.cshtml, widoczne powyżej:

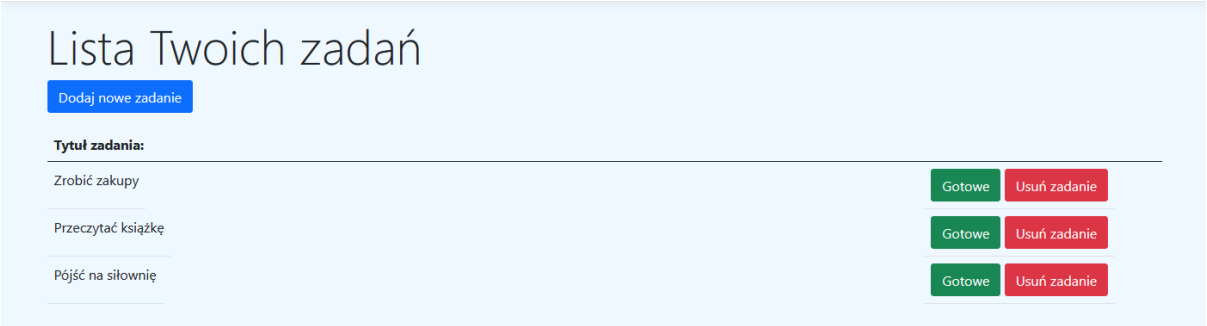
```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Task" asp-action="Index">Lista zadań</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Task" asp-action="IndexDone">Ukończone</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Timer" asp-action="Index">Timer pomodoro</a>
        </li>
    </ul>
    <partial name="_LoginPartial"/>
</div>
```

Po przejściu do zakładki ‘Lista zadań’, użytkownik dostaje do dyspozycji listę swoich zadań, umieszczoną w bazie danych aplikacji. Model danych zadania TaskModel posiada trzy pola: TaskId, Name oraz IsDone, z których pierwsze to niedostępny dla użytkownika klucz główny w bazie danych, drugie to nazwa zadania, które wpisuje użytkownik, a trzecie to stan gotowości zadania, który użytkownik może zmienić odpowiednim przyciskiem.



W widoku listy zadań użytkownik za pomocą odpowiednich przycisków jest w stanie dodawać nowe zadania do swojej listy, usuwać je oraz zmieniać ich stan na ‘gotowe’. Przyciski realizują odpowiednie funkcje kontrolera TaskController: Create, Delete oraz Done, z których dwie pierwsze posiadają swoje własne widoki.

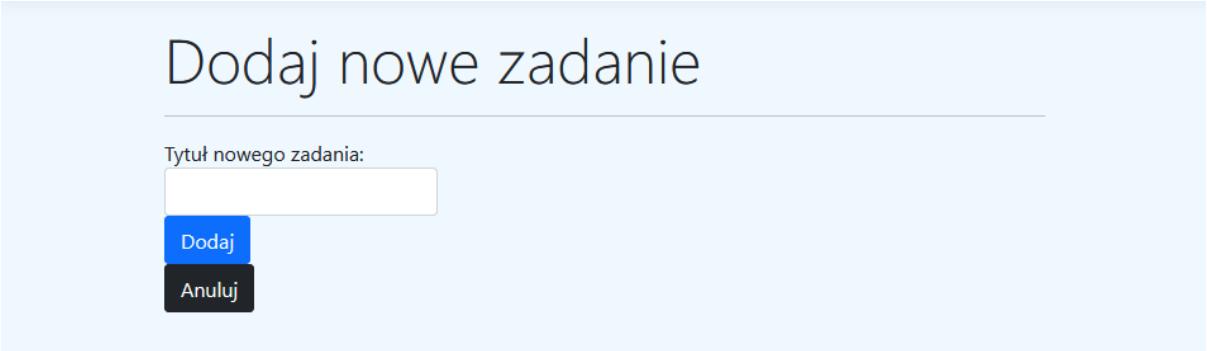
Lista zadań - widok Index kontrolera TaskController:



Funkcja Create kontrolera TaskController, dodająca nowe zadanie do listy:

```
public ActionResult Create()
{
    return View(new TaskModel());
}
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(TaskModel taskModel){
    using (SqliteConnection con = new SqliteConnection("Data Source=PomoToDoroDB.sqlite")){
        using (var tableCmd :SqliteCommand = con.CreateCommand()){
            con.Open();
            tableCmd.CommandText = $"INSERT INTO Tasks (name) VALUES ('{taskModel.Name}')";
            try{
                tableCmd.ExecuteNonQuery();
            }
            catch (Exception ex){
                Console.WriteLine(ex.Message);
            }
        }
    }
    return RedirectToAction(nameof(Index));
}
```

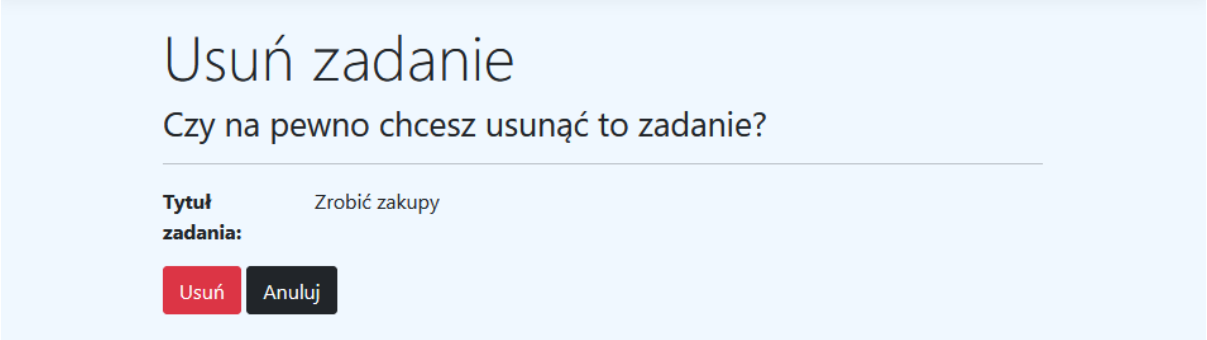
Widok Create kontrolera TaskController:



Funkcja Delete kontrolera TaskController, usuwająca zadanie z listy:

```
public ActionResult Delete(int id)
{
    TaskModel taskModel = GetTaskById(id);
    return View(taskModel);
}
[HttpPost]
[ValidateAntiForgeryToken]
1 usage
public ActionResult Delete(int id, TaskModel taskModel)
{
    using (SqliteConnection con =
           new SqliteConnection("Data Source=PomoToDoroDB.sqlite"))
    {
        using (var tableCmd :SqliteCommand = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = $"DELETE from Tasks WHERE TaskId = '{id}'";
            tableCmd.ExecuteNonQuery();
        }
    }
    return RedirectToAction(nameof(Index));
}
```

Widok Delete kontrolera TaskController:

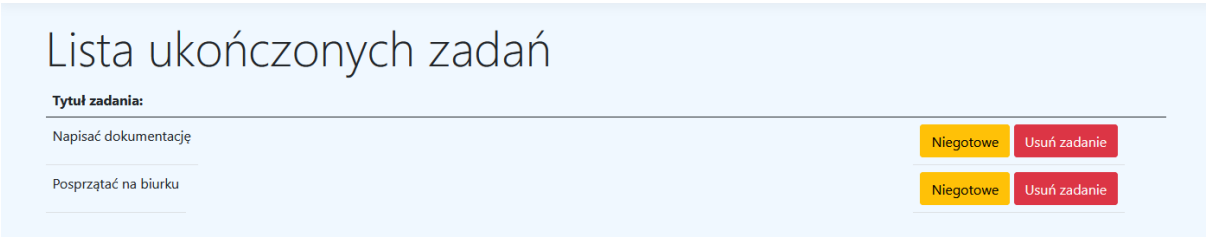


Funkcja Done kontrolera TaskController, zmieniająca stan zadania na gotowe:

```
public ActionResult Done(int id)
{
    using (SqliteConnection con =
           new SqliteConnection("Data Source=PomoToDoroDB.sqlite"))
    {
        using (var tableCmd :SqliteCommand = con.CreateCommand())
        {
            con.Open();
            tableCmd.CommandText = $"UPDATE Tasks SET IsDone = 1 WHERE TaskId = '{id}'";
            tableCmd.ExecuteNonQuery();
        }
    }
    return RedirectToAction(nameof(Index));
}
```

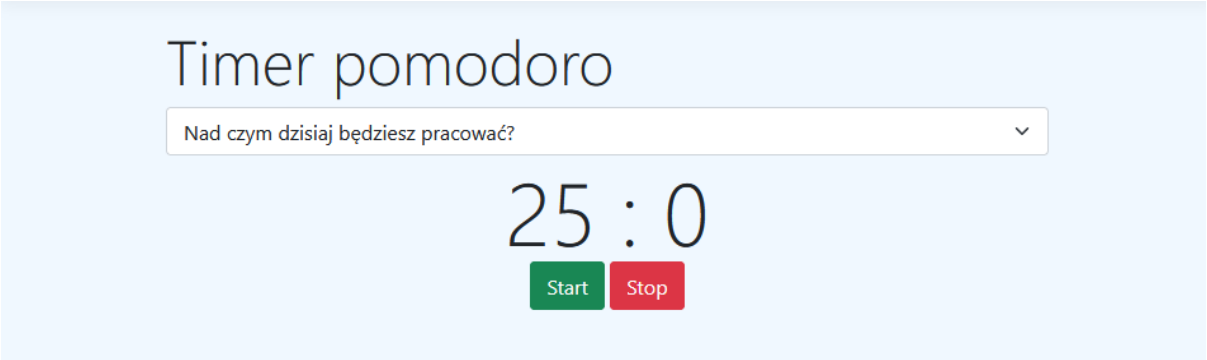
Po przejściu do zakładki ‘Ukończone’, użytkownik dostaje do dyspozycji listę zadań, których stan poprzednio zmienił na ‘gotowe’. Za pomocą odpowiednich przycisków użytkownik może usuwać zadania z listy oraz z powrotem zmieniać ich stan na ‘niegotowe’. Przyciski realizują odpowiednie funkcje kontrolera TaskController: DeleteDone oraz UnDone, analogiczne do fukcji pokazanych wyżej.

Lista ukończonych zadań - widok IndexDone kontrolera TaskController:

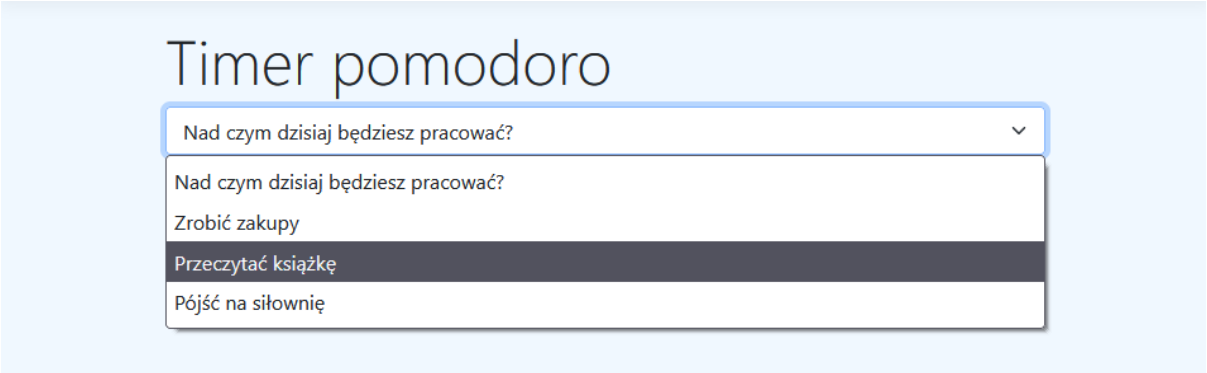


Po przejściu do zakładki ‘Timer pomodoro’ użytkownik dostaje do dyspozycji minutnik, dwa przyciski do jego kontroli oraz rozwijaną listę zadań, z których może wybrać to, nad którym aktualnie pracuje. Na rozwijanej liście użytkownik zobaczyć może jedynie zadania, których stan to ‘niegotowe’. Przycisk Start realizuje funkcję StartTimer, ustawiającą koniec odliczania na datę odległą od aktualnej o 15 minut i rozpoczynającą odliczanie, a przycisk Stop realizuje funkcję StopTimer, która zatrzymuje odliczanie i ustawia wyświetlany pozostały czas z powrotem na 25 minut. Minutnik po dotarciu do końca odliczania wyświetli odpowiedni komunikat.

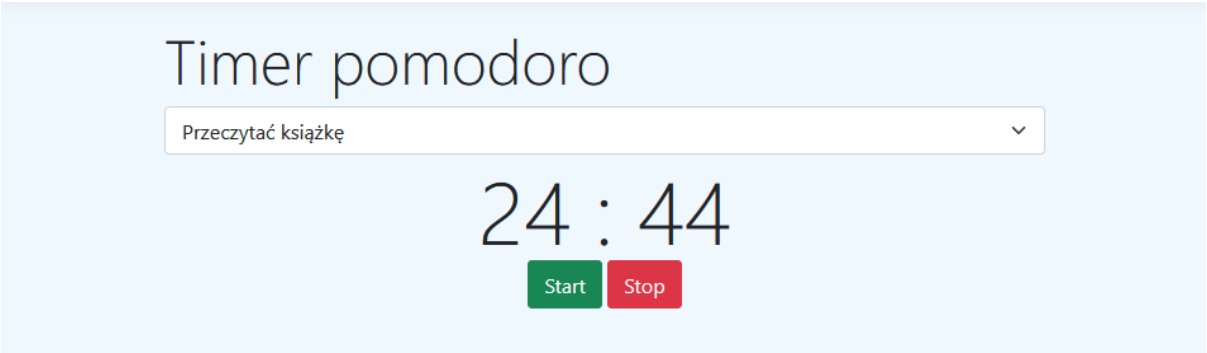
Timer pomodoro - widok Index kontrolera TimerController:



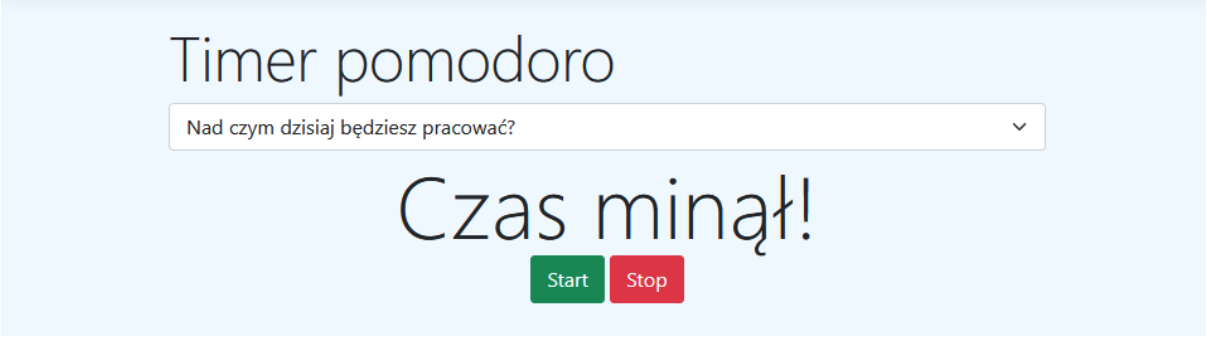
Rozwijana lista zadań:



Timer w trakcie działania:



Timer po zakończeniu odliczania:



Funkcja StartTimer w widoku Index kontrolera TimerController:

```
function StartTimer() {
    if (interval == null) {
        var end = new Date();
        end.setMinutes(end.getMinutes()+2);
        interval = setInterval(function () {
            var now = new Date();
            var distance = end - now;
            if (distance < 0) {
                clearInterval(interval);
                document.getElementById('timecountdown').innerHTML = 'Czas minął!';
                return;
            }
            var _minutes = Math.floor(distance / _minute);
            var _seconds = Math.floor((distance % _minute) / _second);
            document.getElementById('timecountdown').innerHTML = _minutes;
            document.getElementById('timecountdown').innerHTML += ' : ' + _seconds;
        }, 1000);
    }
}
```

Funkcja StopTimer w widoku Index kontrolera TimerController:

```
function StopTimer() {
    clearInterval(interval);
    interval = null;
    var _minutes = 25;
    var _seconds = 0;
    document.getElementById('timecountdown').innerHTML = _minutes;
    document.getElementById('timecountdown').innerHTML += ' : ' + _seconds;
}
```