Politechnika Koszalińska
Wydział Elektroniki i Informatyki

Dokumentacja do projektu z przedmiotu:

Zastosowanie Programowania Komponentowego

Semestr VI, Studia Dzienne
Kierunek: Informatyka
Rok akademicki 2025/2026

Temat projektu - Aplikacja „To do”

Projekt wykonał:

Adrian Bolek U-20025,
Michał Grabski U-20063,
data: 25.06.2025 r.

Spis treści

Wprowadzenie

Opis

Na potrzeby projektu zostały przygotowane dwie aplikacje. Obie służą do zarządzania listą zadań (tzw. „to-do list”). Umożliwiają dodawanie nowych zadań, przeglądanie istniejących, oznaczanie ich jako wykonane oraz wyświetlanie statystyk związanych z zadaniami.

Pierwsza aplikacja to wersja obiektowa, w której logika została zorganizowana w postaci klas C# (m.in. model danych TodoItem i kontekst TodoContext), a interfejs tworzą strony Razor (pliki .cshtml z kodem w C#).

Druga aplikacja to wersja komponentowa oparta na technologii Blazor – została rozbita na trzy niezależne komponenty (TaskDetailComponent, TaskListComponent, StatsComponent), które realizują poszczególne funkcjonalności aplikacji.

Cel

Celem projektu jest porównanie obu oraz zaimplementowanie tych samych funkcji przy użyciu różnych wzorców programistycznych.

Technologie

- Język C# (.NET 8.0)

- ASP.NET Core / Blazor Server

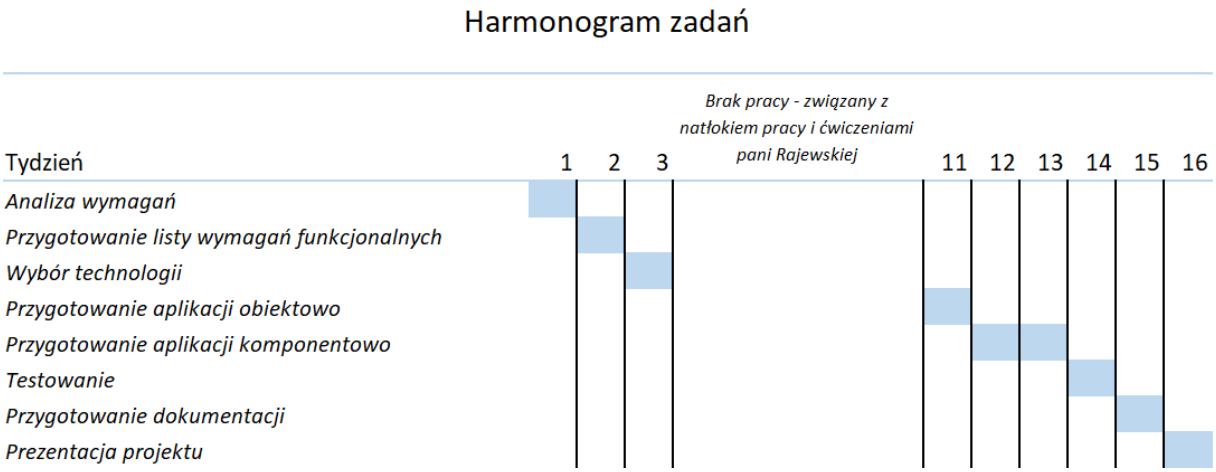
- Visual Studio 2022

- System.Text.Json – do serializacji danych zadań do formatu JSON (wykorzystywane w aplikacji komponentowej).

- SQLite - do serializacji danych zadań (wykorzystywane w aplikacji obiektowej).

- HTML/CSS – do tworzenia interfejsu użytkownika (pliki .cshtml i .css w katalogu wwwroot).

Diagram Gantta



Rys 1. Harmonogram zadań w postaci diagramu Gantta

Lista wymagań funkcjonalnych

- Dodawanie zadania: Użytkownik może utworzyć nowe zadanie, podając przynajmniej tytuł (opcjonalnie opis i datę).

- Wyświetlanie listy zadań: Aplikacja pokazuje wszystkie zapisane zadania, zarówno wykonane, jak i niewykonane.

- Oznaczanie wykonania zadania: Istnieje możliwość oznaczenia zadania jako zakończone (co zmienia jego status).

- Usuwanie zadania: Użytkownik może usunąć zadanie z listy.

- Wyświetlanie statystyk: Aplikacja prezentuje statystyki dotyczące zadań, np. całkowitą liczbę zadań, liczbę ukończonych i nieukończonych, ewentualnie dodatkowe zestawienia (zgodnie z konfiguracją statystyk w pliku JSON).

- Trwałe przechowywanie danych: Dodane/zmodyfikowane zadania są zapisywane w trwałym magazynie (np. w pliku JSON lub w bazie danych) i odczytywane przy uruchomieniu aplikacji.

Aplikacja obiektowa

Rys 3. UI zakładki Statystyki aplikacji 1 stworzonej obiektowo

Aplikacja TodoAppv2 to klasyczna aplikacja ASP.NET Core z Razor Pages i bazą danych. Główne elementy to:

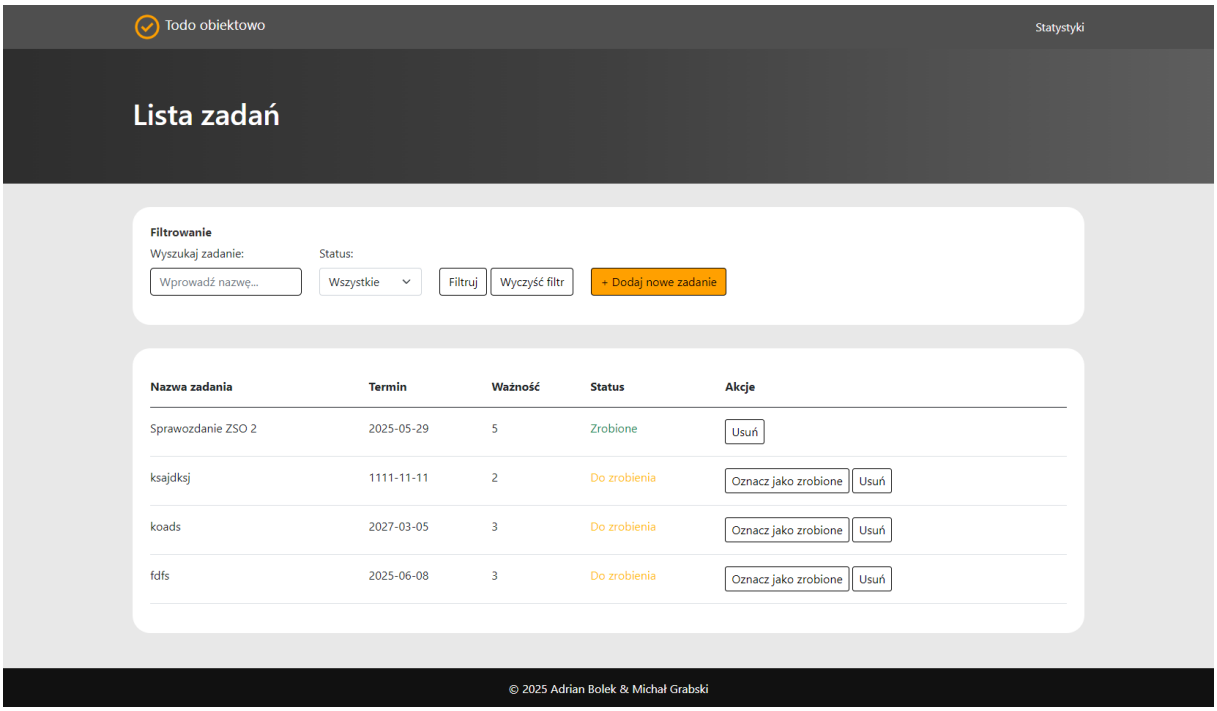
- Model danych: Klasa TodoItem reprezentuje zadanie (Id, Tytuł, Opis, DataDodania, IsCompleted), a TodoContext to kontekst bazy danych oparty na Entity Framework Core.

- Strona główna (Index): Zawiera formularz dodawania i listę zadań. W kodzie-behind (Index.cshtml.cs) dane są pobierane z bazy (GET) i zapisywane (POST).

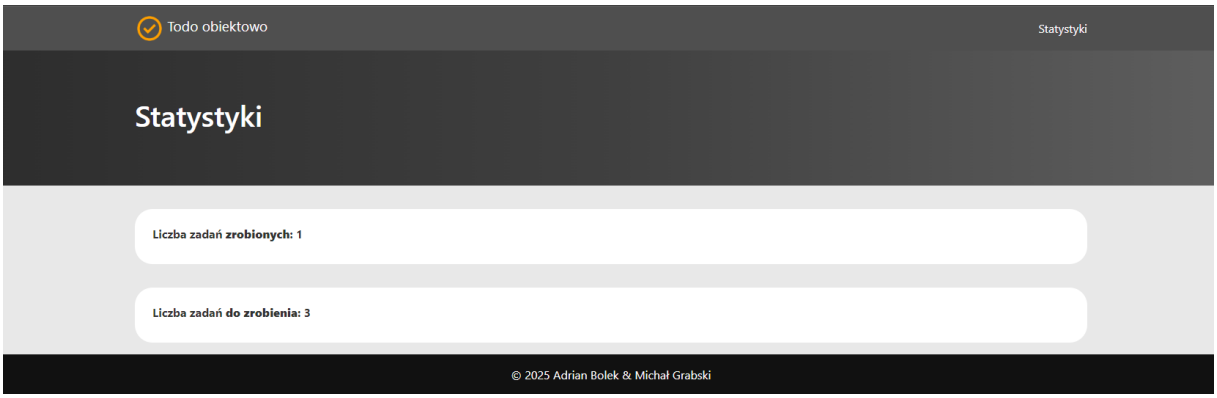
- Zarządzanie zadaniami: Zadania można oznaczać jako ukończone lub usuwać. Operacje te są obsługiwane w TodoContext i aktualizują bazę danych.

- Statystyki (Stats): Oddzielna strona pokazuje liczbę zadań, wykonanych i niewykonanych, na podstawie danych z bazy.

- Trwałość danych: Wszystkie dane są przechowywane w bazie danych, bez użycia JSON.



Rys 2. UI strony głównej aplikacji 1 stworzonej obiektowo



Rys 3. UI zakładki statystyki aplikacji 1 stworzonej obiektowo

Aplikacja Komponentowa

W aplikacji Komponentowo funkcjonalność została podzielona na trzy niezależne komponenty Blazor (znajdują się w katalogu Components): TaskDetailComponent, TaskListComponent i StatsComponent.

Komponent 1 - TaskDetailComponent

Opis: Ten komponent zawiera formularz dodawania nowego zadania. Użytkownik wprowadza dane zadania (np. tytuł, opis, datę) do pól formularza i zatwierdza.

Zmienne prywatne

```
public class TaskDetailComponent
{
    private TaskListComponent _taskList;
    private string _draftFilePath = "";
    private bool _hasDraft = false;
    private string _lastError = string.Empty;
    private DateTime _lastEdited = DateTime.MinValue;
```

-_taskList - lista dostępnych rzeczy ToDo

-_draftFilePath - ścieżka do wersji roboczej zadania.

-_hasDraft - czy istnieje wersja robocza.

-_lastError - ostatni napotkany błąd.

-_lastEdited - data ostatniej edycji.

Konstruktor

```
public TaskDetailComponent()
{
    _taskList = new TaskListComponent();
    _draftFilePath = "draft_task.json";
    _hasDraft = false;
    _lastError = string.Empty;
    _lastEdited = DateTime.MinValue;
}
```

Settery i gettery

```
public TaskListComponent GetTaskList() => _taskList;

public void SetTaskList(TaskListComponent taskList) => _taskList = taskList;

public string GetDraftFilePath() => _draftFilePath;

public void SetDraftFilePath(string path) => _draftFilePath = path;

public bool GetHasDraft() => _hasDraft;

public void SetHasDraft(bool hasDraft) => _hasDraft = hasDraft;
Odwołania: 0
public string GetLastError() => _lastError;

public void SetLastError(string error) => _lastError = error;

public DateTime GetLastEdited() => _lastEdited;

public void SetLastEdited(DateTime dateTime) => _lastEdited = dateTime;
```

Customizacja: Użytkownik może przypisać kategorię do zadania, wybierając ją z rozwijanej listy. Dostępne są szablony nazw (np. „Zrobić sprawozdanie do...”, „Kupić...”), które po kliknięciu automatycznie wstawiają przykładowy tekst do pola „Tytuł”.

Serializacja: Komponent zapisuje tworzony obiekt task do lokalnego pliku JSON już w trakcie edycji formularza – np. po zmianie wartości w polu wejściowym. Dzięki temu po ponownym uruchomieniu aplikacji formularz zostaje automatycznie uzupełniony poprzednimi danymi użytkownika (tzw. „szkic zadania”).Po zatwierdzeniu formularza komponent odczytuje aktualną listę zadań z tasks.json, dodaje nowe zadanie, a następnie ponownie zapisuje pełną listę.

Komponent 2 - TaskListComponent

Opis: Ten komponent wyświetla listę zadań pobranych z magazynu. Pozwala także na interakcję – oznaczanie zadania jako ukończone lub usuwanie zadania.

Zmienne prywatne

```
Odwołania: 14
public class TaskListComponent
{
    private string filePath = "data/tasks.json";
    private List<TaskItem> tasks = new();
    private int nextId = 1;
    private bool _isDirty = false;
    private string _listName = "Default List";
```

-filePath - ścieżka do pliku z zadaniami.

-tasks - lista dostępnych rzeczy ToDo.

-nextId - następny identyfikator zadania.

-_isDirty - czy dane zostały zmodyfikowane i nie zapisane.

-_listName - nazwa listy zadań.

Konstruktor

```
Odwołania: 2
public TaskListComponent()
{
    _isDirty = false;
    _listName = "Default List";
    LoadFromFile();
}
```

Settery i gettery

```
Odwołania: 0
public string GetFilePath() => filePath;
Odwołania: 0
public void SetFilePath(string path) => filePath = path;
Odwołania: 0
public List<TaskItem> GetTasks() => tasks;
Odwołania: 0
public void SetTasks(List<TaskItem> taskList) => tasks = taskList;
Odwołania: 0
public int GetNextId() => nextId;
Odwołania: 0
public void SetNextId(int id) => nextId = id;
Odwołania: 0
public bool GetIsDirty() => _isDirty;
Odwołania: 0
public void SetIsDirty(bool isDirty) => _isDirty = isDirty;
Odwołania: 0
public string GetListName() => _listName;
Odwołania: 0
public void SetListName(string name) => _listName = name;
```

Customizacja: Użytkownik może:

- Oznaczyć zadanie jako wykonane (np. przez checkbox lub przycisk).

- Usunąć zadanie z listy.

Serializacja: Gdy komponent się uruchamia, odczytuje zadania z pliku tasks.json i wyświetla je użytkownikowi. Gdy użytkownik coś zmienia (np. oznaczy zadanie jako ukończone lub je usunie), komponent:

- aktualizuje dane w liście tasks,

- zapisuje całą listę z powrotem do pliku JSON - dzięki temu zmiany są zapamiętane przy kolejnym uruchomieniu aplikacji.

Komponent 3 - StatsComponent

Opis: Komponent służy do prezentacji statystyk zadań. Pokazuje: liczbę wszystkich zadań,liczbę ukończonych, procentowy postęp wykonania.

Zmienne prywatne

```
public class StatsComponent
{
    private TaskListComponent _taskList;
    private string _configFilePath = "";
    private int _taskLimit = 100;
    private string _componentName = "StatsComponent";
    private bool _autoSaveEnabled = true;
```

_taskList - lista dostępnych rzeczy ToDo

_configFilePath - ścieżka do pliku konfiguracyjnego.

_taskLimit - maksymalna liczba rzeczy ToDo.

_componentName - nazwa komponentu.

_autoSaveEnabled - czy automatyczne zapisywanie jest włączone.

Konstruktor

```
public StatsComponent()
{
    _taskList = new TaskListComponent();
    _configFilePath = "stats_config.json";
    _taskLimit = 100;
    _componentName = "StatsComponent";
    _autoSaveEnabled = true;
}
```

Settery i gettery

```
public TaskListComponent GetTaskList() => _taskList;
Odwołania: 0
public void SetTaskList(TaskListComponent taskList) => _taskList = taskList;
Odwołania: 0
public string GetConfigFilePath() => _configFilePath;
Odwołania: 0
public void SetConfigFilePath(string path) => _configFilePath = path;
Odwołania: 0
public int GetTaskLimit() => _taskLimit;
Odwołania: 0
public void SetTaskLimit(int limit) => _taskLimit = limit;
Odwołania: 0
public string GetComponentName() => _componentName;
Odwołania: 0
public void SetComponentName(string name) => _componentName = name;
Odwołania: 0
public bool GetAutoSaveEnabled() => _autoSaveEnabled;
Odwołania: 0
public void SetAutoSaveEnabled(bool enabled) => _autoSaveEnabled = enabled;
```

```
public int Priority
{
    get => _priority;
    set
    {
        if (value < 1) _priority = 1;
        else if (value > 5) _priority = 5;
        else _priority = value;
    }
}
```

```
public string Title
{
    get => _title;
    set
    {
        while (true)
        {
            if (!IsValidTitle(value))
            {
                throw new ArgumentException("Tytuł nie może zawierać znaków specjalnych oraz emotek.");
            }
            else
            {
                _title = value;
                break;
            }
        }
    }
}
```

Customizacja: Użytkownik może zmienić kolor tła tej sekcji za pomocą pola typu color input. Wybrany kolor można zapisać, a po ponownym otwarciu aplikacji będzie on przywrócony.

Serializacja: Komponent odczytuje zadania z pliku tasks.json – tak samo jak inne komponenty – i zlicza je, aby obliczyć statystyki (ilość wykonanych, % postępu). Dodatkowo zapisuje wybrany kolor tła do pliku konfiguracyjnego (np. stats_config.json), by móc go przywrócić przy następnym uruchomieniu aplikacji.

Spełnienie założeń

Cechy programowania komponentowego	Realizacja w aplikacji
Podział na niezależne komponenty	3 wyraźnie oddzielone komponenty: TaskDetailComponent, TaskListComponent, StatsComponent
Każdy komponent ma własny stan i logikę	Każdy działa samodzielnie
Możliwość ponownego użycia komponentów	Komponenty można osadzić w różnych miejscach projektu przez prosty tag, np. <TaskListComponent />
Encapsulacja (enkapsulacja)	Każdy komponent ukrywa swoje dane i funkcje – użytkownik widzi tylko jego działanie
Brak silnego powiązania z resztą systemu (luźne sprzężenie)	Komponenty nie zależą bezpośrednio od siebie – każdy można rozwijać i testować niezależnie
Personalizacja i konfiguracja komponentów	Zastosowanie customizacji
Serializacja i zarządzanie lokalnym stanem	Komponenty samodzielnie odczytują/zapisują dane do JSON (tasks.json, stats_config.json)

Testowanie

TaskDetailComponent

Nazwa testu	Opis	Dane wejściowe	Oczekiwany wynik	Wynik
Dodanie poprawnego zadania	Użytkownik wpisuje wszystkie dane i klika „Dodaj”	Tytuł: 'Zrobić projekt', Priorytet: 1, Kategoria: 'Studia', Termin: 24.06.2025	Zadanie zapisane do pliku tasks.json i widoczne na liście	Zaliczony

TaskListComponent

Nazwa testu	Opis	Dane wejściowe	Oczekiwany wynik	Wynik

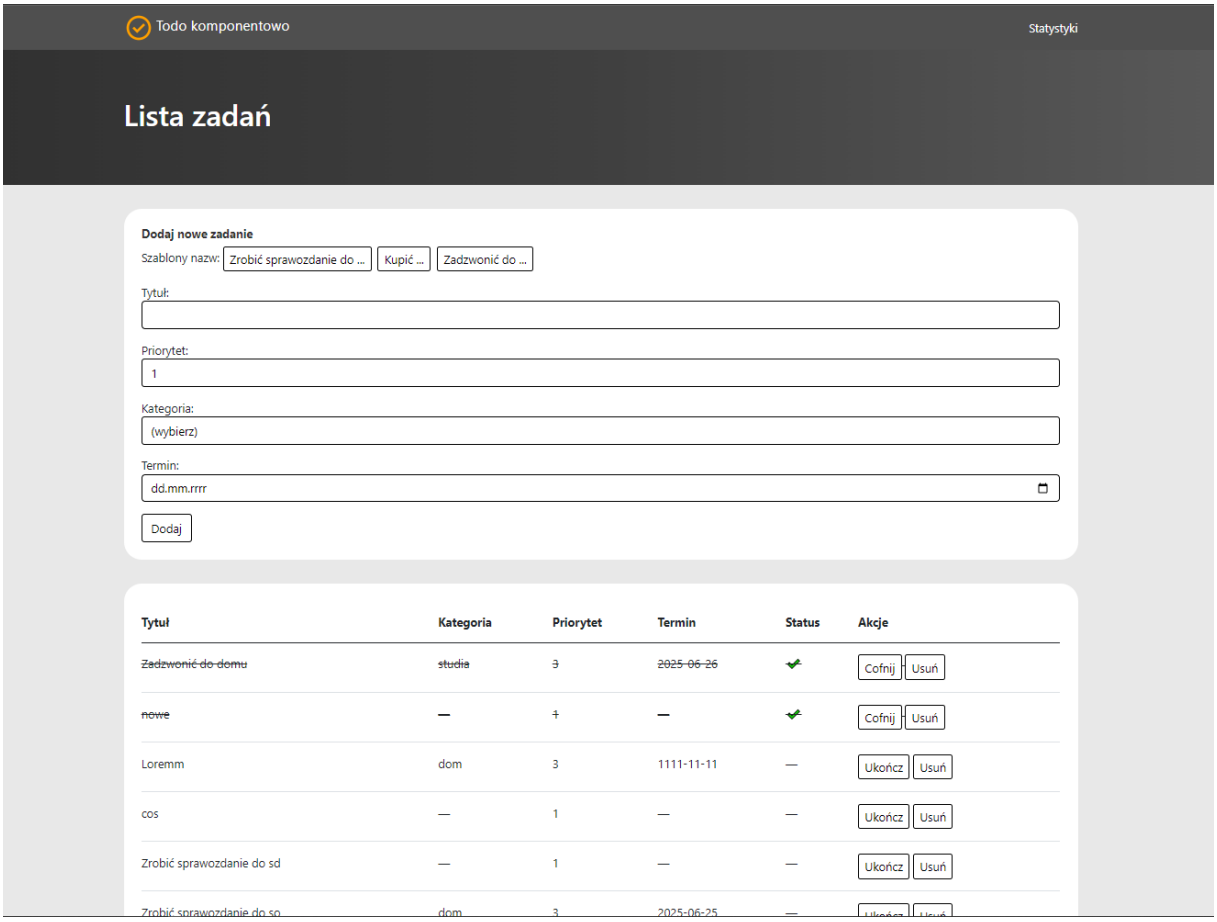
Oznaczenie zadania jako ukończone	Użytkownik klika przycisk „ukończ” przy jednym z zadań	Zadanie: 'Kupić mleko', IsCompleted: false	Zadanie zmienia status na ukończone, zapis do tasks.json, zmiana w widoku	Zaliczony

StatsComponent

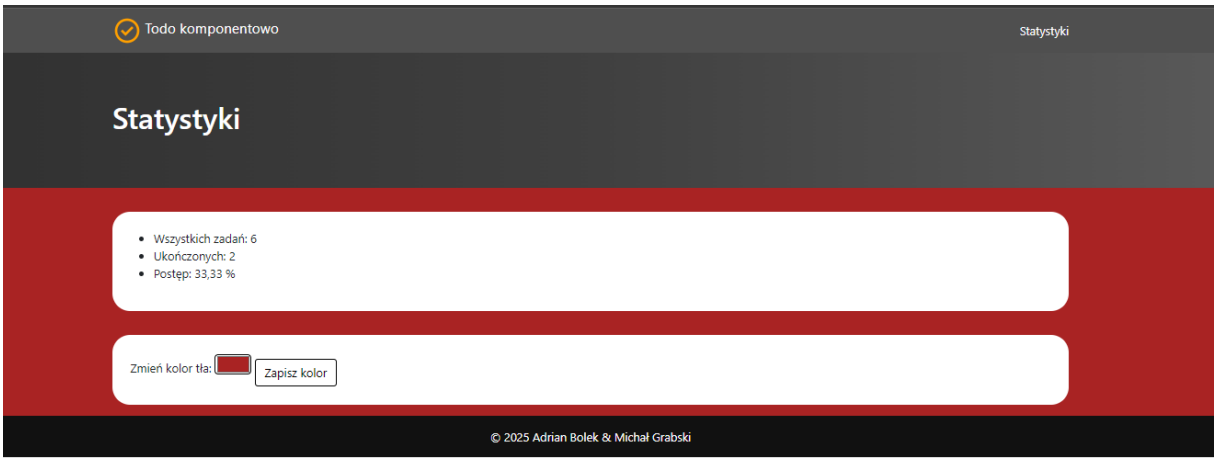
Nazwa testu	Opis	Dane wejściowe	Oczekiwany wynik	Wynik
Reakcja na zmianę ukończenia zadania	Użytkownik oznacza jedno z zadań jako ukończone	Przed: 2/6 ukończonych; Po: 3/6	Zaktualizowane statystyki: 3 ukończone, postęp: 50%	Zaliczony

Podsumowanie

Aplikacja komponentowa w pełni wykorzystuje podejście komponentowe. Każdy element funkcjonalny (dodawanie, lista, statystyki) został wydzielony jako niezależny komponent z własnym stanem i logiką. Komponenty można łatwo osadzać, testować i modyfikować osobno, co zwiększa przejrzystość i ponowną używalność kodu.



Rys 4. UI strony głównej aplikacji 2 stworzonej komponentowo



Rys 5. UI zakładki statystyki aplikacji 1 stworzonej komponentowo

Podsumowanie

Porównanie

Kryterium	Aplikacja obiektowa	Aplikacja komponentowa
Założenia	Programowanie w oparciu o klasy i obiekty, które reprezentują dane i logikę	Programowanie w oparciu o komponenty – niezależne, samodzielne moduły łączące logikę i UI
Jednostka strukturalna	Klasa (np. TodoItem, TodoContext, IndexModel)	Komponent Blazor (np. TaskDetailComponent, TaskListComponent)
Możliwość ponownego użycia	Ograniczona – klasy i logika są powiązane z konkretnymi stronami i kontekstem	Wysoka – komponent można osadzić wielokrotnie
Testowanie	Klasy testuje się oddzielnie (np. testy TodoContext), UI trudniejszy do izolowania	Komponenty są łatwiejsze do testowania jednostkowego i można je sprawdzać osobno, np. testy StatsComponent

Wnioski

Analiza obu aplikacji pokazuje, że ten sam zestaw funkcji można zrealizować różnymi podejściami. Obiektowa jest prosta, ale mniej elastyczna w modyfikacji UI. Komponentowo (Blazor) dzieli aplikację na mniejsze części, które można łatwo rozwijać i ponownie wykorzystywać. Komponenty mają własny stan i reagują dynamicznie, co poprawia wygodę użytkownika.

Materiały źródłowe

https://learn.microsoft.com/en-us/aspnet/core/blazor

https://learn.microsoft.com/en-us/aspnet/core/razor-pages

https://learn.microsoft.com/en-us/dotnet/api/system.text.json

Programowanie w ASP.NET Core - Esposito Dino

Karta oceny

	Karta oceny – ZPK [Podaj liczbę komponentów: ( )]	[0 v 1]
1	Czy utworzono grupę projektową?	1
2	Czy wybrano temat projektu ZPK?	1
3	Czy zamieściłeś w dokumentacji diagram Gantta przedstawiający zadania uczestników projektu w funkcji czasu?	1
4	Czy przedstawiono w dokumentacji opis projektu, cel projektu?	1
5	Czy przedstawiono w dokumentacji listę wymagań funkcjonalnych całego projektu?	1
6	Czy w dokumentacji na podstawie listy wymagań funkcjonalnych, wyodrębniono komponenty?	1
7	Czy wyodrębnione komponenty posiadają nazwę?	1
8	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada w dokumentacji krótki opis (zasadę) działania?	1
9	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada zmienne prywatne (minimum 5 zmiennych na każdy komponent)?	1
10	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada wartości „default” dla każdej zmiennej prywatnej?	1
11	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada konstruktor bezargumentowy?	1
12	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada zestaw funkcji typu setter/getter?	1
13	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada wartości do „customizacji”?	1
14	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada serializację?	1
15	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada wybrany przypadek testowy (funkcjonalny jednostkowy)?	1
16	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada wybrany przypadek testowy (funkcjonalny automatyczny) ?	0
17	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada aplikację testującą?	0
18	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada w dokumentacji przypadki wywołań?	0
19	Czy komponent x – gdzie (x=1,2,3,4,5...) posiada w dokumentacji zbiór cech (zestawienie), że jest komponentem, spełnia założenia programowania komponentowego?	1
20	Czy zbudowałeś aplikację zgodną z opisem funkcjonalnym, zrealizowaną komponentowo?	1
21	Czy zbudowałeś aplikację zgodną z opisem funkcjonalnym, zrealizowaną obiektowo (1 wersja)?	1
22	Czy porównałeś (zestawiłeś) w dokumentacji wybrane różnice w zakresie programowania komponentowego / obiektowego?	1
23	Czy utworzyłeś dokumentację użytkownika (uruchomieniową) dla developera Twojego komponetu_x, – gdzie (x=1,2,3,4,5...)	0
24	Czy utworzyłeś katalog repozytorium: ZPK_Lato2025_Nazwisko1_Nazwisko2_Nazwisko3..... - ilość nazwisk zgodna ze składem grupy?	1
26	Czy w repozytorium znajduje się dokumentacja (sprawozdanie) w postaci pliku otwartego oraz zamkniętego?	1
27	Czy w repozytorium znajduje się komponent x – gdzie (x=1,2,3,4,5...), wersja źródłowa, oraz katalog projektowy?	1
28	Czy w repozytorium znajduje się projekt aplikacji zbudowany z komponentów zgodny z opisem funkcjonalnym?	1
29	Czy w repozytorium znajdują się raporty z postępów prac?	1
30	Czy zamieściłeś w dokumentacji zestawienie użytych narzędzi programistycznych?	1
31	Czy zamieściłeś w dokumentacji odnośniki do materiałów źródłowych?	1
32	Czy zamieściłeś w dokumentacji wnioski?	1
	Suma uzyskanych punktów:	28 /32pkt

GitHub

Link do GitHuba z naszym projektem:

https://github.com/MeAsBob/ZPK