**Pubilsher-Subscriber, CQRS**

**Singleton to wzorzec projektowy, który zapewnia, że dana klasa ma tylko jedną instancję w całej aplikacji i umożliwia globalny dostęp do tej instancji.**

**Analiza Implementacji Wzorca Singleton**

**Ten kod prezentuje dobrze zaimplementowany wzorzec projektowy Singleton z wykorzystaniem klasy ConfigurationManager.**

**Ogólna Struktura**

**Kod składa się z dwóch głównych części:**

1. **Klasa**[**Program**](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**z metodą [Main](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "), która demonstruje wzorzec Singleton**
2. **Klasa ConfigurationManager, która implementuje wzorzec Singleton**

**Klasa Program**

**Metoda [Main](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") demonstruje kluczowe cechy wzorca Singleton:**

1. **Dostęp do Instancji: Tworzy dwie referencje do singletona**

**ConfigurationManager**

**var config1 = ConfigurationManager.Instance;**

**var config2 = ConfigurationManager.Instance;**

1. **Weryfikacja Instancji: Potwierdza, że obie referencje wskazują na ten sam obiekt**

**Console.WriteLine($"Are both references the same instance? {ReferenceEquals(config1, config2)}");**

1. **Współdzielony Stan: Pokazuje, że aktualizacje przez jedną referencję są widoczne przez drugą**

**config1.UpdateSetting("ApiUrl", "https://api.newdomain.com/v2");**

**Console.WriteLine($"Updated API URL (from config2): {config2.GetSetting("ApiUrl")}");**

1. **Test Bezpieczeństwa Wątków: Wykorzystuje zadania równoległe do weryfikacji bezpiecznej inicjalizacji wątków**

**Parallel.For(0, 5, i => {**

**Console.WriteLine($"Thread {i}: Using instance {ConfigurationManager.Instance.GetHashCode()}");**

**});**

**Klasa ConfigurationManager (Implementacja Singleton)**

**Ta klasa demonstruje najlepsze praktyki wzorca Singleton:**

1. **Klasa Sealed: Słowo kluczowe sealed zapobiega dziedziczeniu, które mogłoby złamać wzorzec**
2. **Bezpieczna Leniwa Inicjalizacja: Użycie Lazy<T> zapewnia, że dokładnie jedna instancja jest tworzona nawet w scenariuszach wielowątkowych**

**private static readonly Lazy<ConfigurationManager> \_lazyInstance =**

**new Lazy<ConfigurationManager>(() => new ConfigurationManager());**

1. **Prywatny Konstruktor: Zapobiega tworzeniu nowych instancji przez kod zewnętrzny**

**private ConfigurationManager() { ... }**

1. **Publiczny Statyczny Punkt Dostępu: Zapewnia kontrolowany dostęp do pojedynczej instancji**

**public static ConfigurationManager Instance => \_lazyInstance.Value;**

1. **Praktyczna Implementacja: Klasa zarządza ustawieniami konfiguracji za pomocą:**
   * **Słownika przechowującego ustawienia**
   * **Metod do pobierania stringów połączenia i ustawień**
   * **Metod do aktualizacji ustawień**

**Kluczowe Cechy Implementacji**

1. **Bezpieczeństwo Wątków: Klasa Lazy<T> obsługuje całą synchronizację wątków**
2. **Lazy loading: Instancja jest tworzona tylko przy pierwszym dostępie**
3. **Czysta Enkapsulacja: Prywatny konstruktor wymusza wzorzec Singleton**

**Bridge** to strukturalny wzorzec projektowy, który **oddziela abstrakcję od implementacji**, tak aby mogły być rozwijane niezależnie.  
Chodzi o rozdzielenie hierarchii klas na dwie niezależne części:

* **Abstrakcję** (interfejs / logikę ogólną)
* **Implementację** (szczegóły wykonania)

Łączy się je **za pomocą referencji**.

**Analiza Implementacji Wzorca Bridge (Most)**

Ten kod prezentuje doskonałą implementację wzorca projektowego Bridge (Most) przy użyciu systemu powiadomień. Pozwól, że wyjaśnię całą implementację:

**Ogólna Struktura**

Kod składa się z czterech głównych komponentów:

1. Interfejs implementatora ([INotificationPlatform](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")) i jego konkretne implementacje
2. Abstrakcyjna klasa abstrakcji ([Message](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html))
3. Konkretne klasy rozszerzonej abstrakcji (różne typy wiadomości)
4. Klasa [Program](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) demonstrująca działanie wzorca

**Komponenty Wzorca Bridge (Most)**

**1. Interfejs Implementatora (Most)**

public interface INotificationPlatform

{

    void SendNotification(string recipient, string subject, string body);

}

Ten interfejs definiuje funkcjonalność, którą muszą zapewnić wszystkie platformy powiadomień. Działa jako "most" między abstrakcją (typami wiadomości) a implementacją (platformami dostarczania).

**2. Konkretne Implementacje**

Kod zawiera trzy konkretne implementacje:

* **EmailPlatform**: Symuluje wysyłanie powiadomień przez e-mail
* **SmsPlatform**: Symuluje wysyłanie powiadomień przez SMS
* **PushNotificationPlatform**: Symuluje wysyłanie powiadomień push

Każda implementacja obsługuje szczegóły dostarczania wiadomości na swój sposób, przestrzegając jednocześnie wspólnego interfejsu.

**3. Abstrakcja**

public abstract class Message

{

    protected INotificationPlatform \_platform;

    protected string \_recipient = "user@example.com"; // Domyślny odbiorca dla demonstracji

    public Message(INotificationPlatform platform)

    {

        \_platform = platform;

    }

    public void SetPlatform(INotificationPlatform platform)

    {

        \_platform = platform;

    }

    public abstract void Send(string content);

}

Abstrakcyjna klasa [Message](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html):

* Zawiera referencję do implementatora ([INotificationPlatform](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "))
* Umożliwia zmianę implementacji w czasie wykonywania poprzez [SetPlatform](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")
* Definiuje abstrakcyjną metodę [Send](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "), którą muszą implementować podklasy

**4. Rozszerzone Abstrakcje**

Kod zawiera cztery wyspecjalizowane typy wiadomości:

* **UrgentMessage**: Do pilnych powiadomień z symbolami ostrzegawczymi
* **StandardMessage**: Do zwykłych powiadomień informacyjnych
* **MarketingMessage**: Do treści promocyjnych ze specjalnym formatowaniem
* **ReminderMessage**: Do przypomnień z symbolami zegara

Każda rozszerzona abstrakcja implementuje metodę [Send](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") inaczej, formatując wiadomość zgodnie z jej konkretnym przypadkiem użycia.

**Klasa Program - Demonstracja**

Metoda [Main](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") pokazuje kluczowe zalety wzorca Bridge:

1. **Tworzenie Różnych Kombinacji**:

// Różne platformy

INotificationPlatform emailPlatform = new EmailPlatform();

INotificationPlatform smsPlatform = new SmsPlatform();

INotificationPlatform pushPlatform = new PushNotificationPlatform();

// Różne typy wiadomości korzystające z różnych platform

Message urgentMessage = new UrgentMessage(emailPlatform);

Message standardMessage = new StandardMessage(smsPlatform);

Message marketingMessage = new MarketingMessage(pushPlatform);

1. **Używanie Abstrakcji**:

urgentMessage.Send("Server outage detected! Please investigate immediately.");

standardMessage.Send("Your order #12345 has been processed.");

marketingMessage.Send("Check out our new summer sale with discounts up to 50%!");

1. **Dynamiczna Zmiana Platformy**:

urgentMessage.SetPlatform(smsPlatform);

urgentMessage.Send("Urgent: Database connection failure!");

1. **Dodawanie Nowych Typów Wiadomości Bez Zmiany Platform**:

Message reminderMessage = new ReminderMessage(pushPlatform);

reminderMessage.Send("Don't forget your appointment tomorrow at 2 PM.");

**Kluczowe Zalety Tej Implementacji**

1. **Oddzielenie**: Typy wiadomości i platformy powiadomień są całkowicie oddzielnymi hierarchiami
2. **Niezależna Ewolucja**: Możesz dodawać nowe typy wiadomości bez zmiany platform i odwrotnie
3. **Elastyczność W Czasie Wykonywania**: Implementacja może być zmieniana w czasie wykonywania
4. **Proste Rozszerzenie**: Obie hierarchie można rozszerzać niezależnie:
   * Nowe typy wiadomości: Utwórz nową podklasę [Message](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)
   * Nowe platformy: Utwórz nową implementację [INotificationPlatform](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ")
5. **Pojedyncza Odpowiedzialność**: Każda klasa koncentruje się na jednym aspekcie (albo formatowaniu wiadomości, albo jej dostarczaniu)

Ta implementacja doskonale demonstruje wzorzec Bridge poprzez oddzielenie tego, jakie wiadomości są wysyłane (abstrakcja) od tego, jak są wysyłane (implementacja), pozwalając obu aspektom zmieniać się niezależnie.

**Mediator** to wzorzec projektowy typu behawioralnego, który **centralizuje komunikację między obiektami** w systemie, eliminując bezpośrednie powiązania między nimi.  
Obiekty nie komunikują się ze sobą bezpośrednio, tylko za pośrednictwem wspólnego **mediatora**.

**Analiza Implementacji Wzorca Mediatora**

Ten kod prezentuje doskonałą implementację wzorca projektowego Mediator przy użyciu systemu współpracy/czatu zespołowego. Pozwól, że wyjaśnię całą implementację:

**Ogólna Struktura**

Kod składa się z czterech głównych komponentów:

1. Interfejs mediatora i jego konkretna implementacja ([IChatMediator](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") i [TeamChatMediator](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "))
2. Abstrakcyjna klasa kolegi ([TeamMember](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "))
3. Konkretne klasy kolegów ([Developer](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), [Tester](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) i [ProductManager](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "))
4. Klasa [Program](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), która demonstruje działanie wzorca

**Komponenty Wzorca Mediator**

**1. Interfejs Mediatora**

public interface IChatMediator

{

    void SendMessage(string message, TeamMember sender);

    void SendDirectMessage(string message, TeamMember sender, string receiverName);

    void SendMessageToRole(string message, TeamMember sender, TeamRole role);

    void RegisterColleague(TeamMember member);

    void NotifyAll(string message, TeamMember sender);

}

Ten interfejs definiuje kontrakt dla komunikacji między kolegami. Obsługuje różne typy wiadomości: ogólne wiadomości, wiadomości bezpośrednie, wiadomości oparte na rolach i powiadomienia.

**2. Konkretny Mediator**

public class TeamChatMediator : IChatMediator

{

    private readonly List<TeamMember> \_teamMembers = new List<TeamMember>();

    // Implementacja wszystkich metod komunikacyjnych...

}

[TeamChatMediator](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) utrzymuje listę członków zespołu i obsługuje całą logistykę komunikacji:

* Rejestrację członków zespołu
* Kierowanie wiadomości między członkami zespołu
* Obsługę różnych typów wiadomości (ogólne, bezpośrednie, specyficzne dla roli)
* Rozsyłanie powiadomień

**3. Abstrakcyjny Kolega**

public abstract class TeamMember

{

    protected readonly IChatMediator \_mediator;

    public string Name { get; }

    public TeamRole Role { get; }

    // Konstruktor i metody komunikacyjne...

}

Klasa [TeamMember](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "):

* Przechowuje referencję do mediatora
* Udostępnia metody do wysyłania różnych typów wiadomości
* Definiuje wirtualne metody do odbierania wiadomości

**4. Konkretni Koledzy**

Zdefiniowano trzy wyspecjalizowane typy członków zespołu:

public class Developer : TeamMember

{

    public Developer(string name, IChatMediator mediator)

        : base(name, TeamRole.Developer, mediator) { }

}

public class Tester : TeamMember

{

    // Nadpisana obsługa wiadomości dotyczących błędów...

}

public class ProductManager : TeamMember

{

    // Nadpisana obsługa powiadomień...

}

Każdy typ kolegi:

* Posiada swoją własną rolę
* Może mieć specjalistyczne zachowanie podczas obsługi wiadomości (jak pokazano w klasach [Tester](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) i [ProductManager](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o "))

**Demonstracja w Klasie Program**

Metoda [Main](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") demonstruje kluczowe korzyści wzorca Mediator:

1. **Tworzenie Mediatora i Kolegów**:

TeamChatMediator chatRoom = new TeamChatMediator();

Developer alice = new Developer("Alice", chatRoom);

Developer bob = new Developer("Bob", chatRoom);

Tester charlie = new Tester("Charlie", chatRoom);

ProductManager dave = new ProductManager("Dave", chatRoom);

1. **Rejestracja w Mediatorze**:

chatRoom.RegisterColleague(alice);

chatRoom.RegisterColleague(bob);

chatRoom.RegisterColleague(charlie);

chatRoom.RegisterColleague(dave);

1. **Różne Typy Komunikacji**:
   * Ogólna komunikacja zespołowa
   * Bezpośrednie wiadomości między osobami
   * Wiadomości oparte na rolach (tylko dla określonych ról w zespole)
   * Powiadomienia dla całego zespołu

**Kluczowe Cechy Tej Implementacji**

1. **Pełne Rozdzielenie**: Członkowie zespołu nie wiedzą o sobie bezpośrednio; cała komunikacja odbywa się przez mediatora
2. **Wiele Wzorców Komunikacji**: Obsługuje różne typy wiadomości (ogólne, bezpośrednie, specyficzne dla roli, powiadomienia)
3. **Wyspecjalizowane Zachowanie**: Różne role zespołowe mogą obsługiwać wiadomości w różny sposób:
   * Testerzy automatycznie tworzą zgłoszenia dla wiadomości związanych z błędami
   * Kierownicy produktu aktualizują tablice statusu po otrzymaniu powiadomień
4. **Rozszerzalny Projekt**: Nowe typy członków zespołu mogą być dodawane bez zmiany istniejącego kodu
5. **Pojedyncza Odpowiedzialność**: Mediator obsługuje całą logikę routingu, podczas gdy członkowie zespołu skupiają się na swoich konkretnych zachowaniach

Ta implementacja doskonale demonstruje wzorzec Mediator poprzez centralizację złożonej logiki komunikacyjnej, jednocześnie zachowując rozdzielenie komponentów, co pozwala na łatwą rozszerzalność i konserwację

**Publisher-Subscriber** to wzorzec projektowy typu behawioralnego, w którym **nadawca (publisher)** wysyła komunikaty bez bezpośredniego wskazywania odbiorców.  
**Subskrybenci (subscribers)** rejestrują się do odbioru wybranych komunikatów i są automatycznie powiadamiani, gdy publisher publikuje zdarzenie.

 **Publisher** nie zna swoich subskrybentów — wysyła zdarzenie „w eter”.

 **Subscriber** decyduje, które zdarzenia go interesują i subskrybuje je.

 **Mechanizm rejestracji i powiadamiania** obsługuje pośrednik (Event Broker / Event Bus) lub sam publisher.

 Może działać synchronicznie lub asynchronicznie.

 Stosowany w systemach zdarzeniowych, komunikacji międzymodułowej, systemach rozproszonych.

**CQRS (Command Query Responsibility Segregation)** to wzorzec architektoniczny, który **oddziela operacje odczytu danych (Query)** od operacji ich modyfikacji (Command).  
W odróżnieniu od klasycznego modelu, gdzie ten sam model danych służy zarówno do odczytu, jak i zapisu — CQRS dzieli te odpowiedzialności na osobne modele i interfejsy.

**Wyjaśnienie projektu Cloud\_patterns**

**Ogólny przegląd**

Projekt Cloud\_patterns to demonstracja dwóch ważnych wzorców projektowych w aplikacjach chmurowych: Publisher-Subscriber (Wydawca-Subskrybent) oraz CQRS (Command Query Responsibility Segregation). Aplikacja symuluje system handlu akcjami w czasie rzeczywistym, gdzie przetwarzane są dane giełdowe i dostarczane do wielu subskrybentów z minimalnym opóźnieniem.

**Architektura projektu**

Projekt jest podzielony na kilka kluczowych modułów:

**1. System zdarzeń (Events)**

Katalog [Events](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") zawiera podstawowe komponenty do obsługi zdarzeń:

* [IEvent](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - interfejs bazowy dla wszystkich zdarzeń
* [EventBase](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - abstrakcyjna implementacja IEvent
* [StockPriceUpdatedEvent](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - zdarzenie aktualizacji ceny akcji
* [StockVolumeUpdatedEvent](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - zdarzenie aktualizacji wolumenu akcji

**2. Wzorzec Publisher-Subscriber (PubSub)**

Katalog [PubSub](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) implementuje wzorzec Publisher-Subscriber:

* [EventBroker](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - broker zdarzeń symulujący usługi chmurowe jak Azure Event Hubs czy Service Bus

**3. Wzorzec CQRS**

CQRS dzieli model aplikacji na część do zapisu (Write) i odczytu (Read):

* **Write Model**: [StockWriteModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") zajmuje się aktualizacją danych
* **Read Model**: [StockReadModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") zajmuje się udostępnianiem danych do odczytu

**4. Źródło danych**

* [StockDataFeed](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - symuluje zewnętrzne źródło danych rynkowych

**5. Subskrybenci**

Katalog [Subscribers](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") zawiera różnych odbiorców danych:

* [SubscriberBase](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - bazowa klasa dla wszystkich subskrybentów
* [TradingDashboard](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - dashboard do monitorowania akcji
* [MobileApp](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - aplikacja mobilna
* [AnalyticsEngine](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) - silnik analityczny

**Przepływ danych**

1. [StockDataFeed](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) generuje losowe aktualizacje cen i wolumenów akcji
2. Aktualizacje trafiają do [StockWriteModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") (model zapisu)
3. [StockWriteModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) generuje odpowiednie zdarzenia
4. [EventBroker](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) dystrybuuje zdarzenia do zarejestrowanych subskrybentów
5. [StockReadModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) otrzymuje zdarzenia i aktualizuje swój stan
6. Klienci odpytują [StockReadModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) o aktualne dane

Wyjaśnienie kodu

1. **Inicjalizacja komponentów**:

var eventBroker = new EventBroker();

var stockWriteModel = new StockWriteModel(eventBroker);

var stockReadModel = new StockReadModel();

Tworzone są kluczowe komponenty systemu: broker zdarzeń, model zapisu i model odczytu.

1. **Subskrypcja zdarzeń**:

eventBroker.Subscribe<StockPriceUpdatedEvent>(stockReadModel.HandleStockPriceUpdate);

eventBroker.Subscribe<StockVolumeUpdatedEvent>(stockReadModel.HandleStockVolumeUpdate);

Model odczytu (read model) rejestruje się jako subskrybent zdarzeń związanych ze zmianami cen i wolumenów akcji.

1. **Tworzenie subskrybentów**:

var tradingDashboard = new TradingDashboard("Executive Dashboard", eventBroker);

var mobileApp = new MobileApp("Mobile Client", eventBroker);

var analyticsEngine = new AnalyticsEngine("Analytics Engine", eventBroker);

Tworzone są różne typy subskrybentów (dashboard, aplikacja mobilna, silnik analityczny), które reagują na zdarzenia.

1. **Uruchomienie symulacji**:

var cancellationTokenSource = new CancellationTokenSource();

var dataFeedTask = stockDataFeed.StartProcessing(cancellationTokenSource.Token);

var clientQueryTask = SimulateClientQueries(stockReadModel, cancellationTokenSource.Token);

Rozpoczynają się dwa równoległe zadania: symulacja źródła danych oraz symulacja klientów wykonujących zapytania.

1. **Oczekiwanie na zakończenie**:

Console.WriteLine("Press any key to stop the simulation...");

Console.ReadKey();

Aplikacja czeka na naciśnięcie dowolnego klawisza przez użytkownika.

1. **Zatrzymanie zadań i wyświetlenie statystyk**:

cancellationTokenSource.Cancel();

try {

    await Task.WhenAll(dataFeedTask, clientQueryTask);

} catch (OperationCanceledException) {

    Console.WriteLine("Simulation stopped.");

}

Console.WriteLine("\nStatistics:");

Console.WriteLine($"Total events published: {eventBroker.TotalEventsPublished}");

Po naciśnięciu klawisza, wszystkie zadania są anulowane i wyświetlane są statystyki.

**Metoda SimulateClientQueries**

static async Task SimulateClientQueries(StockReadModel readModel, CancellationToken cancellationToken)

{

    var stockSymbols = new[] { "MSFT", "AAPL", "AMZN", "GOOGL", "TSLA" };

    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)

    {

        var symbol = stockSymbols[random.Next(stockSymbols.Length)];

        var stock = readModel.GetStock(symbol);

        if (stock != null)

        {

            Console.WriteLine($"[QUERY] Client retrieved {symbol}: ${stock.CurrentPrice:F2} (Vol: {stock.Volume})");

        }

        await Task.Delay(500, cancellationToken);

    }

}

Ta metoda symuluje klientów wykonujących zapytania do modelu odczytu:

1. Definiuje tablicę symboli akcji i generator liczb losowych
2. W nieskończonej pętli (dopóki nie zostanie anulowana):
   * Losowo wybiera symbol akcji
   * Wykonuje zapytanie do modelu odczytu o dane tej akcji
   * Wyświetla pobrane informacje
   * Czeka 500 milisekund przed kolejnym zapytaniem

Ta metoda demonstruje, jak model odczytu (strona zapytań CQRS) służy do szybkiego pobierania danych bez wpływania na operacje zapisu.

**Korzyści implementacji**

1. **Skalowalność**: Separacja zapisu i odczytu (CQRS) pozwala na niezależne skalowanie tych operacji
2. **Wydajność**: Model odczytu jest zoptymalizowany pod kątem wydajnych zapytań
3. **Luźne powiązania**: Wydawcy nie wiedzą o subskrybentach, co ułatwia rozbudowę systemu
4. **Izolacja odpowiedzialności**: Każdy komponent ma jasno określoną rolę
5. **Odporność na awarie**: W rzeczywistym systemie chmurowym, każdy komponent mógłby działać niezależnie

**Praktyczne zastosowanie w rzeczywistej aplikacji chmurowej**

W prawdziwej implementacji chmurowej:

* [EventBroker](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) byłby zastąpiony przez Azure Event Hubs lub Service Bus
* [StockWriteModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) mógłby być implementowany jako Azure Functions
* [StockReadModel](vscode-file://vscode-app/c:/Users/michal.sermak/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) mógłby być przechowywany w Azure Cosmos DB
* Implementowano by odpowiednie mechanizmy obsługi błędów i odporności na awarie
* Dane byłyby serializowane i deserializowane podczas przesyłania

Ten projekt pokazuje, jak wzorce Publisher-Subscriber i CQRS mogą być wykorzystane do budowy wydajnych, skalowalnych systemów działających w czasie rzeczywistym w środowisku chmurowym.