WPF Deep Dive

Windows Presentation Foundation – w skrócie WPF – jest jednym z dwóch .NETowych frameworków ułatwiających budowanie interfejsu użytkownika aplikacji desktopowych – swój monopol WPF (w wersji klasycznej i UWP) dzieli tylko z Windows Forms, będącymi w zasadzie opakowaniem niskopoziomowych mechanizmów WinAPI. W 2014 roku na tych łamach pojawił się cykl artykułów wprowadzających w podstawy WPF, który pomagał w miarę łagodnie wdrożyć się w to środowisko. Przyszedł czas na to, by sięgnąć głębiej i bardziej wnikliwie przyjrzeć się mechanizmom, które rządzą WPFem.

WYMAGANIA

Artykuły z serii WPF Deep Dive pisane są w założeniu dla tych osób, które wiedzą już trochę o WPF i miały szansę napisać w tym frameworku jedną czy dwie aplikacje, ale chciałyby dowiedzieć się o nim czegoś więcej. Nie znajdziemy tu więc na pewno szczegółowej listy różnic pomiędzy WPF i Windows Forms, porad, kiedy stosować pierwszy z frameworków, a kiedy drugi, nie będę też pisał o architekturze MVVM ani o podstawach XAMLa – czytelnik z tymi tematami powinien już być zaznajomiony. W zamian postaram się opowiedzieć bardziej szczegółowo o różnych aspektach WPF, zahaczając od czasu do czasu o jego wewnętrzną implementację (która w dużej części od 4 grudnia 2018 dzięki uprzejmości Microsoftu jest dostępna na zasadach open source). Dowiemy się trochę o tym, jak WPF działa „pod maską", i spróbujemy zobaczyć, jak mocno możemy nagiąć zasady tego i tak bardzo elastycznego frameworka.

Ponieważ nie jest to z zamierzenia kurs, poruszane zagadnienia nie będą ułożone w jakimś ściśle określonym porządku – postaram się za każdym razem opowiedzieć o innych aspektach WPF, którym z jakiegoś powodu warto poświęcić trochę więcej czasu. I dlatego też, bez zbędnego przedłużania, dzisiaj zaczniemy od następującego tematu:

WIELOWĄTKOWOŚĆ W WPF

Za każdym razem, gdy zabierzecie się za projektowanie własnego frameworka oferującego budowanie interfejsu użytkownika, staniecie w pewnym momencie przed pytaniem: a co z wielowątkowością? I będziecie musieli podjąć jedną z dwóch decyzji: pozwolić albo zabronić. Tylko uwaga: nie chodzi tu bynajmniej o to, czy użytkownik waszego frameworka będzie mógł korzystać z wielowątkowości w ogóle. Kluczowe jest raczej to, czy pozwolicie mu na wielowątkowy dostęp do waszych kontrolek.

Z czego wynika problem? Odpowiedź jest dosyć prosta: wyścig. Wyobraźmy sobie, że po wywołaniu jakiejś metody wasza kontrolka wykonuje operacje A, B i C. Powiedzmy, że jeden wątek wywołał tę metodę i zdążyła się już zakończyć operacja A. W międzyczasie jednak drugi wątek również wywołał tę samą metodę, ale z innymi parametrami. W pierwszym wątku wykonywana więc będzie operacja B, zaś w drugim A – która może zmienić wewnętrzny stan kontrolki według nowych parametrów. Operacje B i C z pierwszego wątku nie będą na to przygotowane i w efekcie mamy całą paletę możliwych rezultatów: od prawidłowego działania (bo przypadkowo może zdarzyć się, że nie naruszymy wewnętrznego stanu kontrolki), przez działanie nieprawidłowe, ale niepowodujące błędu (np. rozjechane wartości pól klasy reprezentującej kontrolkę – chyba najgorsza możliwa opcja), aż po rzucony wyjątek (pomimo dwóch prawidłowych wywołań).

Istnieją dwa rozwiązania tego problemu.

Bez obsługi wielowątkowości

Prostsze z nich polega na wprowadzeniu kontraktu: przyjmujemy, że do kontrolek dostęp może mieć tylko jeden wątek – zwany zwykle wątkiem UI (ang. *User Interface* – interfejs użytkownika). Przeważnie jest to również ten sam wątek, w którym kontrolki zostały powołane do życia.

Pojawiają się jednak dwa nowe problemy. Co stanie się, jeżeli ktoś mimo wszystko spróbuje dostać się do kontrolek spoza wątku UI, łamiąc wspomniany kontrakt? I z drugiej strony, jak umożliwić wykonanie takiej operacji w bezpieczny sposób?

Pierwszy z problemów różne istniejące frameworki rozwiązują w różny sposób. VCL, z którego korzystamy w Delphi, po prostu przyjmuje twarde założenie, że programista będzie pracował z kontrolkami tylko z głównego wątku. Próba dostępu do nich z poziomu innego wątku powiedzie się, ale skończy się na jeden z opisanych wcześniej sposobów (tzw. niezdefiniowane zachowanie). I, wbrew pozorom, takie podejście jest całkiem sensowne – w końcu po to właśnie są kontrakty, czyli wymagania, które stawiane są użytkownikom różnych bibliotek.

WPF jest w tej kwestii znacznie bardziej restrykcyjny i aktywnie weryfikuje, czy kod pracujący z kontrolką wykonuje się na wątku UI czy też nie – i w drugim przypadku natychmiast rzucany jest wyjątek. Dla programisty jest to rozwiązanie znacznie wygodniejsze, bo jeżeli w ferworze walki z kodem aplikacji pomyli się gdzieś i złamie wspomnianą zasadę (o co, wbrew pozorom, wcale nie jest tak trudno), to dowie się o tym stosunkowo szybko i bezboleśnie (uwierzcie, nie chcecie szukać przyczyny niezdefiniowanego zachowania, to może trwać tygodniami – z własnego doświadczenia).

Co zaś z drugą kwestią – bezpiecznego dostępu do kontrolek z poziomu innych wątków? Otóż ten dylemat rozwiązywany jest najczęściej przy użyciu kolejki komunikatów. Jeżeli inny wątek chce zmienić coś w kontrolce, taka operacja musi zostać przeniesiona do głównego wątku. W takim przypadku do wspomnianej kolejki jest