

Zbigniew **FRYŻLEWICZ** / Dariusz **PARZYGnat** / Łukasz **PRZERADA**

SERVERLESS

na platformie *Azure*



Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Barbara Lepionka

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/servaz>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/servaz.zip>

ISBN: 978-83-283-5068-7

Copyright © Helion 2019

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	7
Rozdział 1. Serverless	11
1.1. „Bezserwerowy”	11
1.2. Od monolitu do funkcji	12
1.3. Od architektury klient-serwer do sterowanej zdarzeniami	14
1.4. FaaS — świat bez wad?	16
1.5. Historia FaaS	17
1.6. FaaS od różnych dostawców	18
1.7. Podsumowanie	18
Rozdział 2. Azure Functions — zacznijmy!	19
2.1. Pierwsza funkcja Hello Azure!	20
2.2. JavaScript i funkcje	26
2.3. C# script i funkcje	32
2.4. Wyzwalacze, wiązania i gotowe szablony	38
2.5. Podsumowanie	45
Rozdział 3. Azure CLI, Azure Function Tools for VS	47
3.1. Azure CLI	48
3.1.1. Logowanie do platformy Azure	49
3.1.2. Utworzenie grupy zasobów	49
3.1.3. Utworzenie konta w usłudze magazynu (Azure Storage)	50
3.1.4. Utworzenie aplikacji funkcji (Function App)	51
3.2. Azure Function Tools dla Visual Studio	53
3.2.1. Rozpoczęcie pracy z Visual Studio	53
3.2.2. Dodanie funkcji do aplikacji funkcji	56
3.2.3. Uruchomienie i debugowanie funkcji	61
3.2.4. Wdrożenie na platformie Azure	65
3.3. Podsumowanie	69

Rozdział 4. Durable Functions	71
4.1. Kluczowe pojęcia	72
4.1.1. Funkcja aktywności, funkcja orkiestracji	72
4.1.2. DurableOrchestrationContext	73
4.1.3. Nowe typy wiązań	74
4.1.4. Uruchomienie funkcji	75
4.2. Durable Functions pod maską	77
4.2.1. Task Hub	77
4.2.2. Skalowanie	78
4.3. Event sourcing	78
4.4. Ograniczenia związane z Durable Functions	82
4.5. Podsumowanie	83
 Rozdział 5. FaceAggregator — wizja i architektura	85
5.1. Założenia projektowe	85
5.2. Usługi i komponenty wsparcia	86
5.2.1. Cognitive Services	86
5.2.2. Azure Storage — Blob	86
5.2.3. Azure Storage — Queue	87
5.2.4. Cosmos DB	88
5.2.5. Twilio SMS	88
5.2.6. Logic Apps	88
5.2.7. Application Insights	89
5.2.8. Azure DevOps	89
5.2.9. Azure Functions Proxy	89
5.2.10. Google Sign-In	89
5.3. Architektura aplikacji	90
5.4. Projekt interfejsu użytkownika	93
5.5. Podsumowanie	93
 Rozdział 6. Zarządzanie zdjęciami	95
6.1. Frontend aplikacji FaceAggregator	96
6.2. Konfiguracja narzędzi i uruchomienie aplikacji	98
6.3. Funkcja pobierająca zasoby użytkownika	99
6.4. Funkcja dodająca zdjęcia do kontenera	106
6.5. Funkcja tworząca miniaturki zdjęć	110
6.6. Funkcje zwracające zdjęcia	112
6.7. Funkcja usuwająca i funkcja zmieniająca nazwę zdjęcia	115
6.8. Podsumowanie	119

Rozdział 7. CI/CD z Azure DevOps w tle	121
7.1. Przygotowanie	122
7.2. Continuous deployment	122
7.3. Azure DevOps	125
7.4. Udostępnienie frontendu za pomocą Azure Function Proxy	130
7.5. Podsumowanie	134
 Rozdział 8. Integracja aplikacji z Face API	 135
8.1. Zakładka Face Recognition	136
8.2. Funkcja pobierająca drzewo katalogów użytkownika	138
8.3. Funkcja przyjmująca zlecenia użytkownika	139
8.4. Utworzenie usługi Face API	141
8.5. Sprawdzanie zgodności zdjęć z wymaganiami	145
8.6. Funkcja trenująca model	154
8.7. Funkcja przetwarzająca wybrane zdjęcia	159
8.8. Wdrożenie projektu	166
8.9. Podsumowanie	169
 Rozdział 9. Autoryzacja i uwierzytelnianie	 171
9.1. Skonfigurowanie dostawcy tożsamości	171
9.2. Konfiguracja uwierzytelniania w Azure	175
9.3. Pobieranie tożsamości w kodzie	176
9.4. Atrybut filtrujący	178
9.5. Podsumowanie	180
 Rozdział 10. Łączenie z zewnętrznymi serwisami	 181
10.1. Utworzenie aplikacji Logic App	181
10.2. Pierwszy wyzwalacz i akcja	184
10.3. Wyrażenia warunkowe i pętla	187
10.4. Testowanie przepływu i widok tekstowy	191
10.5. Podsumowanie	194
 Rozdział 11. Testowanie	 195
11.1. Testowanie logiki biznesowej	195
11.2. Testowanie metod funkcji	202
11.3. Wdrożenie testów do chmury	211
11.4. Podsumowanie	213

Rozdział 12. Monitoring	215
12.1. Application Insights	215
12.2. Application Insights Analytics	222
12.3. Podsumowanie	226
Podsumowanie	227
Bibliografia	229
Książki	229
Źródła internetowe	229
Załącznik A. Portal Azure	231
A.1. Elementy pulpitu nawigacyjnego	231
A.2. Wyszukiwanie zasobów i usług	233
A.3. Grupy zasobów	234
A.4. Konto usługi Storage	236
A.4.1. Punkty końcowe konta usługi Storage	237
A.4.2. Tworzenie i usuwanie konta magazynu	238
A.5. Podsumowanie	240
Załącznik B. Azure Cosmos DB	241
B.1. Utworzenie konta Cosmos DB	241
B.2. Skonfigurowanie bazy danych i kolekcji	243
B.3. Podsumowanie	244
Załącznik C. Słownik terminów	245
Załącznik D. Narzędzia i systemy użyte w przykładach	251
Skorowidz	253

Wstęp

To jest książka dla programistów, którzy chcą poznać nowy, ciekawy świat architektury i przetwarzania serverless. Świat, który uwalnia od zarządzania serwerami, infrastrukturą i systemami operacyjnymi, bo wszystko zapewnia dostawca platformy chmurowej. Od teraz cały swój czas i wysiłek będziecie mogli skierować na projektowanie i budowanie innowacyjnych aplikacji i szybsze dostarczanie ich na rynek, a tym samym uzyskanie przewagi konkurencyjnej.

Warto poznać podstawowe zalety architektury i przetwarzania serverless. Po pierwsze zredukowane są obowiązki związane z zarządzaniem i bieżącym administrowaniem serwerami i konieczną infrastrukturą. Deweloperzy dostają do dyspozycji w pełni zarządzalne usługi i za pomocą stosunkowo niewielkiej ilości kodu mogą rozwiązywać problemy biznesowe oraz budować i wdrażać aplikacje. Po drugie aplikacja zbudowana w architekturze serverless automatycznie się skaluje, niezależnie jakie byłoby obciążenie. Co więcej, reakcja na wahania obciążenia jest bardzo szybka. W ciągu pojedynczych sekund może powstawać wiele tysięcy instancji funkcji — w reakcji na zdarzenia i wyzwalacze — i to bez zmian w konfiguracji aplikacji. I wreszcie — w architekturze serverless inaczej jest naliczany rachunek za użytkowanie aplikacji. Przetwarzanie serverless jest sterowane zdarzeniami, co oznacza, że zasoby są alokowane w reakcji na pojawiające się zdarzenia. Użytkownik jest obciążany tylko za faktycznie zużyty czas — liczony w milisekundach — i zasoby potrzebne do wykonania kodu aplikacji.

Microsoft Azure to jedna z kilku platform chmurowych o zasięgu globalnym, która w swojej ofercie zapewnia usługi oparte na architekturze serverless. Klasyczne modele dostarczania usług chmurowych to IaaS, PaaS i SaaS [URL: NIST], [Fry 15]. Kilka lat temu ten klasyczny model został uzupełniony i pojawiły się w nim usługi klasyfikowane jako FaaS (ang. *Function as a Service*). FaaS to nic innego jak dostarczanie usług na bazie architektury serverless, ale z inną implementacją wewnętrzną w stosunku do modelu PaaS. W modelu PaaS deweloper również nie musi bezpośrednio zarządzać zasobami. Jednak usługi, z których korzysta, mają inną architekturę implementacyjną, co rzutuje na ich skalowalność. W większości systemów PaaS cały czas jest uruchomiony i działa jeden proces serwera. Nawet gdy jest włączone autoskalowanie, to nowe procesy — o relatywnie długim czasie kreacji, zamykania — są dodawane/usuwane na tej samej maszynie. To oznacza, że skalowalność jest bardziej

„widoczna” dla dewelopera. W systemach FaaS oczekuje się, że każda z funkcji startuje w ciągu pojedynczych milisekund, tak aby obsłużyć indywidualne żądanie. W przeciwieństwie do tego w systemach PaaS kod aplikacji jest uruchomiony na pojedynczym, długo działającym wątku i obsługuje wiele żądań. Ta różnica jest widoczna przede wszystkim w kosztach. W usługach FaaS są one naliczane w relacji do czasu wykonania funkcji, w usługach PaaS są naliczane w relacji do czasu działania wątku, na którym wykonuje się aplikacja serwerowa.

Platforma Microsoft Azure dostarcza obecnie ponad 100 różnych zasobów i usług, w tym 3 oparte na architekturze serverless. Są to: *Azure Functions*, *Azure Logic Apps* i *Azure Event Grid*. Jeśli spojrzymy na te trzy usługi przez pryzmat procesów biznesowych i przepływów, to *Azure Functions* jest odpowiednia do zbudowania pojedynczego kroku w przepływie, a *Logic Apps* dostarcza mechanizmy do sterowania całym przepływem i integracją z serwisami zewnętrznymi, często typu SaaS. *Azure Event Grid* ułatwia natomiast tworzenie aplikacji opartych na zdarzeniach, poprzez dostarczenie jednorodnego mechanizmu subskrypcji, powiadamiania i obsługi zdarzeń. *Azure Event Grid* jest dodatkowo klasyfikowana jako usługa reaktywna.

Książka składa się z 12 rozdziałów, 4 załączników i jest przeznaczona dla programistów z dobrą znajomością języka C# i podstawową znajomością platformy Microsoft Azure.

W rozdziale 1., zatytułowanym „Serverless”, opisujemy genezę architektury serverless, jej zalety i związane z nią zagrożenia.

Rozdział 2., zatytułowany „Azure Functions — zacznijmy!”, zawiera podstawowe informacje o *Azure Functions*, zilustrowane przykładami funkcji w językach JavaScript i C# script. Przykłady, wszystkie o charakterze demonstracyjnym, zostały zbudowane i uruchomione w trybie online z wykorzystaniem portalu Azure.

W rozdziale 3., zatytułowanym „Azure CLI, Azure Function Tools for VS”, opisaliśmy środowiska, które służą do wytwarzania aplikacji w oparciu o *Azure Functions*, całkowicie lub częściowo w trybie offline. Analizowane przykłady obejmowały zarówno funkcje umieszczone w serwisie GitHub, jak i funkcję w języku C#, zbudowaną w środowisku VS 2017, uruchomioną w lokalnym emulatorze i po przetestowaniu wdrożoną na platformie Azure.

W rozdziale 4., zatytułowanym „Durable Functions”, opisaliśmy *Durable Task Framework* i jego zastosowanie do realizacji przepływów w aplikacji zbudowanej z *Azure Functions*. Framework i zbudowane na podstawie jego szablonów funkcje wykorzystamy praktycznie w rozdziale 8.

Rozdział 5., zatytułowany „FaceAggregator — wizja i architektura”, to opis założeń projektowych, składniki architektury oraz makieta interfejsu aplikacji, której implementację i wdrożenie do chmury opisujemy w rozdziałach 6. – 10.

Rozdział 6., zatytułowany „Zarządzanie zdjęciami”, zawiera opis wszystkich funkcji odpowiedzialnych za zarządzanie zdjęciami użytkowników. Ich działanie zostało też częściowo przetestowane w środowisku lokalnym, zarówno z użyciem przygotowanego frontendu (aplikacja Angular do pobrania z GitHub), jak i aplikacji Postman. Całość, zbudowana jako rozwiązanie VS 2017 z projektami FaceAggregatorFrontend i ServerlessImageManagement, tworzy pierwszą, szkieletową wersję aplikacji FaceAggregator.

W rozdziale 7., zatytułowanym „CI/CD z Azure DevOps w tle”, opisujemy wdrożenie pierwszej wersji aplikacji FaceAggregator na platformę Azure. Proces jest zautomatyzowany dzięki wykorzystaniu usługi Azure DevOps. W tym rozdziale opisujemy również, jak udostępnić frontend aplikacji, korzystając z funkcjonalności Proxy. Efektem działań w tym rozdziale jest druga wersja aplikacji.

W rozdziale 8., zatytułowanym „Integracja aplikacji z Face API”, opisujemy budowę trzeciej wersji aplikacji FaceAggregator. Aplikację zbudowaną w rozdziale 6. uzupełniamy o funkcje związane z rozpoznawaniem osób na zdjęciach. Uzupełnienie wymagało napisania kilku nowych funkcji i, co najważniejsze, integracji z usługą Face API, dostępną w ramach *Cognitive Services* platformy Azure. Funkcje i ich orkiestrację zbudowaliśmy, wykorzystując wspomniany w rozdziale 4. *Durable Task Framework*.

W rozdziale 9., zatytułowanym „Autoryzacja i uwierzytelnianie”, konfigurujemy dostawcę tożsamości i integrujemy go z budowaną aplikacją. Teraz każdy z użytkowników aplikacji FaceAggregator ma własne konto i chroniony dostęp do swoich zasobów. Na zakończenie tego rozdziału mamy wdrożoną do chmury czwartą wersję aplikacji FaceAggregator.

W rozdziale 10., zatytułowanym „Łączenie z zewnętrznymi serwisami”, pokazujemy, jak wykorzystać usługę *Azure Logic Apps* do zbudowania przepływu i integracji z serwisem SMS *Twilio*. Dzięki temu uzyskujemy możliwość wysłania do użytkownika aplikacji FaceAggregator powiadomień o stanie realizacji jego zlecenia.

W rozdziale 11., zatytułowanym „Testowanie”, omawiamy zasady testowania w usługach *Azure Functions*. Pokazujemy również, jak można testować bardziej skomplikowaną logikę biznesową i jak „wynieść” ją poza funkcje. W zakończeniu rozdziału omawiamy aktualizację procesu CI/CD i dodanie nowego kroku związanego z testowaniem. Na zakończenie tego rozdziału mamy wdrożoną do chmury piątą i ostatnią wersję aplikacji FaceAggregator.

W rozdziale 12., zatytułowanym „Monitoring”, omawiamy usługi *Application Insights* i *Application Insights Analytics* i ich zastosowanie do monitorowania bieżącego stanu aplikacji FaceAggregator i historii jej działania.

Wymienione rozdziały są poprzedzone wstępem, a zakończone podsumowaniem i wykazem bibliografii oraz referencji do użytych w przykładach narzędzi i podsystemów. Uzupełnieniem treści rozdziałów są cztery załączniki o nazwach, które jednoznacznie określają ich zawartość.

Autorzy i Wydawnictwo Helion serdecznie zapraszają do lektury, a pod adresem *serverless2018@gmail.com* czekają na wszelkie uwagi, komentarze lub pytania związane z zawartością książki.

Rozdział 1.

Serverless

Zanim przejdziemy do tworzenia własnych rozwiązań, warto zadać sobie kilka pytań. W tym rozdziale postaramy się przyjrzeć hasłu *serverless*. Omówimy, jak wyglądała ewolucja architektury, aż do miejsca, w którym się obecnie znajdujemy. Przedstawimy również głównych dostawców usług FaaS oraz wskażemy różnice i podobieństwa, jakie pomiędzy nimi występują. Przeprowadzimy także analizę usług „bezserwerowych” pod kątem biznesowym — jaką przewagę mogą zapewnić deweloperom, a w efekcie także ich klientom.

1.1. „Bezserwerowy”

Niezwykłe dynamiczny rozwój i miliony inwestowane przez wielkie korporacje powodują, że w otaczającej nas rzeczywistości co rusz pojawiają się nowe idee związane z chmurą obliczeniową. Niewątpliwie jedną z nich jest *serverless*, która rozpala wyobraźnię deweloperów. Co kryje się za tym pojęciem? W artykule [URL: Robin] zaprezentowano dwie interpretacje:

1. *Serverless* określa aplikację, która w dużym stopniu lub całkowicie bazuje na innych serwisach i to ich używa do zarządzania oraz sterowania logiką i stanem. To zwykle są aplikacje typu *single page*, które reprezentują idee grubego klienta, a które do operacji stricte backendowych wykorzystują gotowe usługi, np. *Auth0* do uwierzytelniania. Klocki, które wykorzystuje się do budowania tego typu aplikacji, są bezobsługowe i za zarządzanie nimi i aktualizacje odpowiada ich dostawca.
2. *Serverless* może oznaczać również aplikację, która posiada logikę po stronie serwera, ale inaczej niż w tradycyjnym podejściu jest ona wykonywana w bezstanowych kontenerach, które często są sterowane przez zdarzenia i są realizowane w pełni przez dostawcę tego typu usługi. W tym kontekście nawet lepiej używać pojęcia FaaS (ang. *Function as a Service*).

Upraszczając, *serverless* oznacza nowe podejście do tworzenia oprogramowania, w którym wykorzystuje się gotowe usługi, np. usługę uwierzytelniania, przechowywania danych itp. Zaletą tych elementów jest bezobsługowość, tzn. wszystkimi aktualizacjami i zarządzaniem sprzętem, na którym udostępniane są te usługi, zajmuje się dostawca chmury. Nie wszystkie

klocki spełniają jednak wymagania biznesu, a integracja pomiędzy nimi może działać inaczej, niż to zakładaliśmy. Wtedy z pomocą przychodzą usługi realizujące pojęcie *Function as a Service* (FaaS). Są to platformy odpowiedzialne za wykonanie wprowadzonego kodu, które również cechują się bezobsługowością. Jeżeli gotowe elementy nie są w pełni satysfakcjonujące albo brakuje takich spełniających oczekiwania biznesu, to w świecie serverless definiujemy je w formie kilku funkcji. Jednak nie przejmujemy się ich wykonywaniem i skalowalnością, tylko wdrażamy je bezpośrednio do platform FaaS. Tutaj łatwo znaleźć historyczną analogię, kiedy mało kto posiadał komputer i programy zapisane na kartach perforowanych wykonywano na komputerach uczelnianych. Różnica polega na tym, że teraz usługodawca zapewnia, że wprowadzony kod zostanie wykonany oczekiwaną ilość razy w określonym czasie. Ten sposób gwarantuje niemal natychmiastową skalowalność i obciąża rachunek usługobiorcy tylko za czas i zasoby, które są faktycznie wykorzystywane w trakcie wykonywania wprowadzonego kodu. *Function as a Service* to pojęcie często utożsamiane z pojęciem serverless, ale nie tożsame. Można je traktować jako usługę wchodzącą w skład architektury serverless.

1.2. Od monolitu do funkcji

Z architekturą monolityczną — rysunek 1.1 — miał do czynienia chyba każdy z deweloperów. Cechą charakterystyczną takiej architektury jest zamknięcie wszystkich domen biznesowych w jednym niepodzielnym systemie. Dzięki takiemu podejściu szybciej można pokazać klientowi jakąś wartość biznesową. Z drugiej strony ten typ architektury prowadzi do zwiększenia złożoności i skomplikowania systemu. Wiele komponentów posiada zależności i ciężko wydzielić ich logikę biznesową. Przez takie podejście dużym wyzwaniem staje się skalowanie. Systemy oparte na architekturze monolitycznej nie są dostosowane w szczególności do skalowania wszcz (tworzenia kilku instancji aplikacji).



RYСУNEK 1.1. Przykład architektury monolitycznej — zamknięcie wszystkich domen sklepu internetowego w jednym systemie

Po prezentacji Dockera w 2013 roku nastała era mikroserwisów. W architekturze tej chodzi o podział systemu w oparciu o jego domeny. Czyli dla przykładu, projektując internetowy sklep — rysunek 1.2 — możemy wydzielić takie fragmenty, jak np. katalog produktów, obsługa zamówienia, potwierdzanie tożsamości, i każda z tych części może stanowić oddzielny mikroserwis z własnym modelem danych. Oczywiście przy tego typu architekturze zawsze pojawia się pytanie, jak duże powinny być nasze mikroserwisy. Wskazówek lub odpowiedzi w tej kluczowej kwestii najlepiej szukać w literaturze technicznej, u ekspertów lub ucząc się metodą prób i błędów. Wielką zaletą mikroserwisów jest ich niezależny charakter. Każdy z zaplanowanych mikroserwisów może być przygotowywany przez oddzielny zespół oraz w innej technologii. Ponadto takie podejście jest łatwe w skalowaniu w porównaniu z architekturą monolityczną. Na pierwszy rzut oka ta architektura rozwiązuje wszystkie istniejące problemy w podejściu monolitycznym. Jednak nic nie jest bez wad. W mikroserwisach o wiele więcej czasu potrzeba na integrację poszczególnych systemów oraz wdrożenie całej aplikacji. Ponadto podczas tworzenia nowego projektu czasochłonne jest utworzenie fundamentów, tj. skonfigurowanie komunikacji pomiędzy serwisami, przygotowanie warstw persystencji itp. W porównaniu z architekturą monolityczną pierwsze funkcjonalności są dostarczane później. Z tego względu podejście to może nie być odpowiednie dla aplikacji z prostą logiką biznesową, przeznaczonej dla małego grona odbiorców.



RYСУNEK 1.2. Przykład architektury mikroserwisów — każda domena sklepu internetowego wydzielona jako osobny mikroserwis

Podział na funkcje jest kolejnym krokiem w ewolucji architektury. Projekt rozbity jest na jeszcze mniejsze, niezależnie uruchamiane części, często nazywane nanoserwisami. Wracając do analogii sklepu internetowego — rysunek 1.3 — jedna funkcja może reprezentować funkcjonalność zapisu zamówienia, a inna może być odpowiedzialna za pobranie o nim danych. Tak małe kawałki kodu o wiele łatwiej jest skalować. Przy użyciu platform FaaS nowe



RYСУNEK 1.3. Przykład architektury opartej na funkcjach — każda funkcjonalność sklepu internetowego wydzielona jako osobna funkcja

instancje funkcji pojawiają się w zależności od ilości zapytań. Jeżeli chodzi o minusy tego rozwiązania, to — podobnie jak w przypadku mikroservisów — najwięcej uwagi należy zwrócić na integrację oraz wdrożenie aplikacji opartej na funkcjach.

1.3. Od architektury klient-serwer do sterowanej zdarzeniami

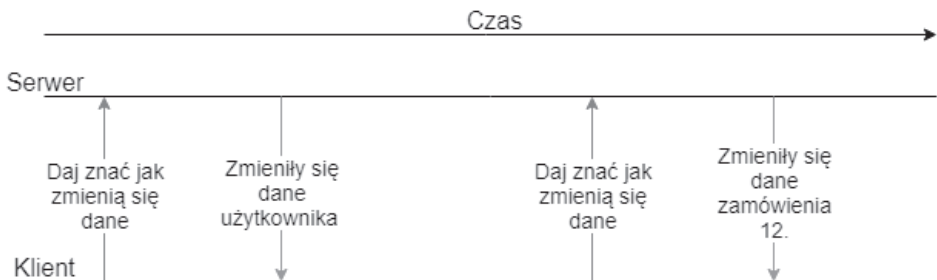
Świat serverless wprowadza zmiany nie tylko do architektury aplikacji, ale też do sposobu komunikacji. Zmiana polega na przejściu z komunikacji opartej na odpytywaniu (ang. *pooling*) do bazującej na zdarzeniach. Spróbujmy przedstawić tę transformację na podstawie komunikacji klient-serwer.

Na początku wykorzystywano prostą zasadę: zapytanie-odpowiedź. Za każdym razem, gdy klient chciał dowiedzieć się np. o zmianach, to wysłał zapytanie do serwera i od razu otrzymywał odpowiedź. Komunikat od serwera przychodził zawsze, niezależnie, czy dane zostały zmienione, czy nie — rysunek 1.4. Minusem takiego rozwiązania jest przypadek, gdy zmiana pojawia się tuż po wysłaniu komunikatu z serwera i klient dowiaduje się o niej w momencie, gdy znowu wyśle zapytanie.

Kolejnym etapem w rozwoju komunikacji klient-serwer jest wykorzystanie poolingu do subskrypcji zapytań. Sposób polega na acyklicznym wysyłaniu do serwera monitów z prośbą o poinformowanie, gdy zmienią się wnioskowane dane. Gdy do takiej zmiany dojdzie, to klient zostanie o niej poinformowany — rysunek 1.5. W tym schemacie komunikacji — po każdej odpowiedzi — klient musi subskrybować się ponownie, jeżeli nadal interesują go zmiany dotyczące wskazanego typu danych.

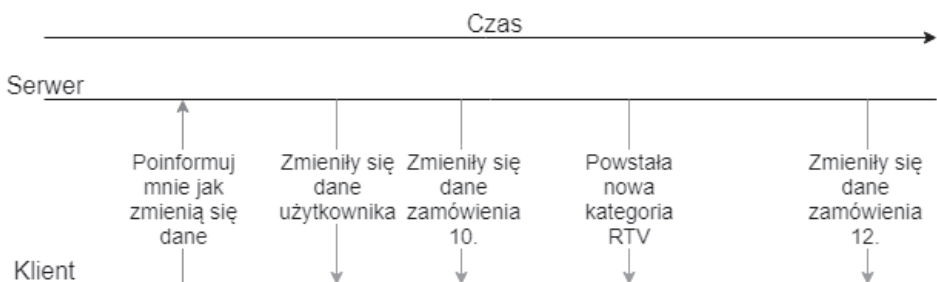


RYСУNEK 1.4. Przykład architektury bazującej na zasadzie zapytanie-odpowiedź



RYСУNEK 1.5. Przykład architektury bazującej na wykorzystaniu poolingu do subskrypcji zapytań

Podójście, które zasadniczo zmienia komunikację i jest wykorzystywane w świecie serverless, opiera się na zdarzeniach. W odniesieniu do architektury klient-serwer polega na zwróceniu się klienta do serwera, żeby informował go np. o wszystkich zmianach danych — rysunek 1.6. Gdy dane zmieniają się na serwerze, to informacja o tym bezzwłocznie zostanie przesłana do klienta. Zalety takiego rozwiązania są oczywiste. Klient jednokrotnie zgłasza się do serwera i komunikaty przesyłane są tuż po wystąpieniu zdarzenia.



RYСУNEK 1.6. Przykład architektury bazującej na zdarzeniach

1.4. FaaS — świat bez wad?

Wykorzystanie FaaS wydaje się być ciekawym rozwiązaniem. Spróbujmy przeanalizować zasadnicze cechy tego podejścia.

Jak wspomniano, zastosowanie funkcji pozwala wyeliminować koszty operacyjne. Redukcję tę można rozpatrywać w dwóch aspektach — kosztów infrastruktury i personalnych. Kolejnym aspektem będzie optymalizacja kosztów skalowania. Wszystkie rachunki będą dotyczyły tylko i wyłącznie mocy obliczeniowej, która była potrzebna do obsłużenia wygenerowanego ruchu. W kwestii skalowalności dostaje się również inne korzyści. Nie trzeba się już dłużej zastanawiać nad ilością ruchu, jaki wygeneruje zbudowana aplikacja, ponieważ mamy pewność, że każde zapytanie zostanie obsłużone (oczywiście jeżeli nie istnieje inne wąskie gardło). Nie mniej ważną kwestią jest szybkość testowania nowych rozwiązań, a także wprowadzenie nowych funkcjonalności na rynek. Wyobraźmy sobie, że chcemy przetestować możliwość informowania użytkowników o pojawiających się błędach przez SMS-y. W świecie serverless może się to sprowadzić do dodania jednej funkcji przesyłającej zebrane dane do dostawcy usługi wysyłania wiadomości. Dodatkowym atutem wybrania tego typu podejścia jest status organizacji przyjaźniejszej środowisku naturalnemu. W jaki sposób? Wybierając FaaS, korzystamy z zalet dzielonej infrastruktury — gdy nasza aplikacja nie potrzebuje mocy procesorów, to korzysta z nich ktoś inny. Dzięki temu wzrasta średnie wykorzystanie mocy, ale spada ilość potrzebnych na świecie serwerów, których produkcja i funkcjonowanie (zużycie energii, emisja ciepła) nie są przyjazne środowisku naturalnemu.

Ciekawym aspektem jest kwestia przygotowania i złożoności wdrożenia. Z jednej strony wielkim plusem korzystania z platform FaaS jest brak potrzeby przygotowywania skryptów startujących, czy też specjalnych instalatorów. Wystarczy, że nasz kod zostanie wgrany na platformę FaaS, i już możemy korzystać z zaimplementowanej funkcjonalności. Z drugiej jednak strony pojawia się kwestia wersjonowania poszczególnych funkcji oraz wdrażania poprawek do grupy funkcji. Duża granulacja rozwiązania i dość ubogi, jak na chwilę obecną, zestaw narzędzi ułatwiających pracę z tego typu rozwiązaniami mogą powodować, że decyzje o korzystaniu z FaaS mogą być wstrzymywane.

Czy to wszystkie wady, jakie ma FaaS? Największym problemem jest testowanie. W nowo tworzonych funkcjach bardzo często wykorzystujemy inne usługi serverless z portfolio danego dostawcy chmury, np. usługę przechowywania danych. W przypadku testów integracyjnych zapłacimy nie tylko za samo wywołanie w obrębie platformy FaaS, ale także za usługi, z których korzystamy. Z drugiej strony tylko w środowisku serverless można w całości odwzorować to, co się dzieje w środowisku produkcyjnym. Tylko małe firmy mogłyby przynajmniej w teorii pozwolić sobie na posiadanie tych samych zasobów obliczeniowych zarówno w środowisku produkcyjnym, jak i testowym. Patrząc na testowanie z innej perspektywy, możemy stwierdzić, że zmierza to w dobrym kierunku, ponieważ duża część dostawców FaaS dostarcza środowiska uruchomieniowe, które możemy wykorzystać w lokalnym środowisku. Kolejnym problemem może być uzależnienie się od dostawcy,

np. możemy wybrać język, który jest wspierany tylko u jednego dostawcy, i w przypadku podniesienia cen będziemy zmuszeni się przystosować. Wydaje się, że przejście pomiędzy dostawcami chmury będzie o tyle trudne, że najczęściej korzysta się ze wszystkich usług serverless, jakie dany dostawca chmury zapewnia, aby do minimum ograniczyć koszty operacyjne. Na koniec zdecyduje na pewno rachunek zysków i strat, czy lepiej poświęcić trochę czasu i dodać kolejne poziomy izolacji, które przynajmniej w teorii pozwolą nam migrować do innej chmury, czy wdrożyć nasz produkt szybciej na rynek i uzyskać przewagę nad konkurencją. Ostatnią, ale nie mniej ważną kwestią jest *multi-tenancy problem*. Jest on powszechnie znany w świecie serverless. Chodzi o przypadek, w którym kilka instancji oprogramowania dedykowanych dla różnych klientów jest wykonywanych w tej samej fizycznej infrastrukturze w tym samym czasie. Dostawcy chmury robią wszystko, co w ich mocy, żebyśmy czuli, że jesteśmy jedynymi klientami korzystającymi z ich usług. W takiej konfiguracji możemy napotkać problemy związane z bezpieczeństwem (jeden klient może uzyskać nieuprawniony dostęp do danych innego klienta), stabilnością (błąd w jednej instancji oprogramowania może wpłynąć na działanie innych instancji) i wydajnością (oprogramowanie wymagające dużej ilości czasu procesora może spowalniać program innego klienta). Niekorzystny wpływ innego klienta chmury na nasz biznes może zachodzić również z innego punktu widzenia. Wyobraźmy sobie, że korzystamy z jakiegoś limitowanego API, tzn. wykonujemy do niego kilka zapytań dziennie. Wszystko działa dobrze, dopóki ktoś nie będzie robił dokładnie tej samej operacji i jego wywołania nie będą wykonywane z tej samej puli adresów. Wtedy jednak może się okazać, że pośrednio będziemy wpływać na siebie, ponieważ wskazana usługa będzie szybciej blokowana. Wydaje się, że ogólnie multi-tenancy problem jest coraz rzadziej spotykany i może stać się marginalnym wraz ze wzrostem dojrzałości chmur.

1.5. Historia FaaS

Historia FaaS rozpoczęła się od platformy Zimki typu *pay as you go*. Platforma ta ujrzała światło dzienne w 2006 roku i była napisana w JavaScriptcie. Pomimo gwałtownego wzrostu zainteresowania i zysków Zimki zostało zamknięte w grudniu 2007 roku, a za główny powód podano to, że ten obszar nie jest kluczowy dla posiadającej platformę firmy.

W 2014 roku Amazon Web Services zaprezentował Lambdę. Była to pierwsza platforma FaaS wdrożona przez dostawcę usług z chmury publicznej. AWS Lambda na początku wspierała tylko Node.js. W momencie, kiedy AWS zaprezentował tę usługę, inni dostawcy chmury publicznej wąpili w powodzenie tego typu serwisu.

W 2016 roku pomysł wdrożony przez AWS został wreszcie doceniony przez innych dostawców. W lutym swoje rozwiązania zaprezentowały Google i IBM. Firma z siedzibą w Mountain View pokazała światu Google Cloud Functions, które w pierwszej fazie wspierało tylko funkcje napisane w Node.js. IBM natomiast przedstawił platformę OpenWhisk, w której korzystać można było z Node.js oraz technologii Swift.

Miesiąc później, w marcu 2016 swoje rozwiązanie zademonstrował ostatni z wielkich dostawców chmury publicznej — Microsoft. *Azure Functions* na początku wspierała przede wszystkim skryptowy język C# oraz platformę uruchomieniową Node.js.

1.6. FaaS od różnych dostawców

Platformy FaaS są jednymi z najszybciej rozwijających się w środowisku chmurowym. Każdemu z dostawców zależy na uzyskaniu przewagi i przekonaniu do siebie największego grona odbiorców. Stąd coraz więcej języków jest wspieranych przez różnych dostawców, poszerza się liczba wyzwalaczy inicjalizujących funkcje oraz liczba wyjść z funkcji, tj. miejsc, do których przekazywany jest wynik wykonania funkcji. Przykładem, jak bardzo zależy dostawcom na trafieniu do jak najszerzego grona odbiorców, jest Microsoft, który wprowadził wsparcie dla Javy w *Azure Functions*.

Oczywiście istnieją różnice pomiędzy implementacją i cechami FaaS u różnych dostawców, np. jedne funkcje mogą być testowane lokalnie, a inne nie. Szybkość zmian jest jednak tak duża, że coś, co wczoraj było niedostępne, jutro może być już wspierane. Z tego względu, gdy już będziecie zaczynać swój projekt oparty na FaaS, warto przejrzeć aktualną ofertę dostawców i ich plany na najbliższą przyszłość.

1.7. Podsumowanie

W rozdziale tym staraliśmy się wyjaśnić, czym naprawdę jest serverless oraz jakie zmiany w architekturze aplikacji i komunikacji nastąpiły na przełomie lat i ostatecznie stały się podwaliną świata serverless.

Poznaliśmy różnych dostawców FaaS, sposób realizacji przez nich tego typu usług oraz cechy, które charakteryzują każdą z nich. Może to mieć znaczenie przy wyborze odpowiedniego dostawcy przy realizacji własnego projektu opartego na architekturze serverless.

Skorowidz

A

- agent, 245
 - pool, 245
 - queue, 245
- akcja, 184
- Angular, 245
- aplikacja
 - FaceAggregator, 96, 195
 - Logic App, 181
- aplikacje
 - funkcji, 57
 - uruchomienie, 98
 - wdrożenie, 121, 133
- Application
 - Insights, 89, 215
 - Insights Analytics, 222
 - Program Interface, 245
- architektura
 - aplikacji, 90
 - klient-serwer, 14
 - mikroserwisów, 13
 - monolityczna, 12
 - oparta na funkcjach, 14
 - sterowana zdarzeniami, 14
- atrybut filtrujący, 178
- autoryzacja, 171
- Azure, 227
 - CLI, 47, 48, 245
 - Cloud Shell, 48, 245
 - Cosmos DB, 241
 - baza danych, 243
 - kolekcja, 243
 - tworzenie konta, 241
 - DevOps, 89, 121, 125, 245
 - Function Proxy, 89, 130, 246

- Function Tools dla Visual Studio, 47, 53
- Functions, 19, 246
- Functions and Web Jobs Tools, 246
- grupy zasobów, 234
- Marketplace, 246
- pulpit nawigacyjny, 231
- Storage, 50, 86, 236
- Storage Explorer, 64
- tworzenie konta magazynu, 238
- usuwanie konta magazynu, 238
- wyszukiwanie zasobów i usług, 233

B

- backend, 85, 246
- baza danych, 243
- bezstanowość, 246
- biblioteki C# script, 35
- Bindings, 246
- BitBucket, 246
- Blob, 86, 246
- Blog Storage, 246
- Branch, 246

C

- C# script, 32, 247
- chmura
 - Azure, 65
 - wdrożenie testów, 211
- CI/CD, 121, 125

- ciągłe
 - dostarczanie, 123
 - wdrażanie, 122, 123
- Cloud Explorer, 56
- Cognitive Services, 86
- Cold start, 247
- continuous
 - delivery, 123
 - deployment, 121–125
 - integration, 121, 125
- Cosmos DB, 88

D

- debugowanie funkcji, 61
- definiowanie
 - funkcji, 24
 - kontenera, 21
- dodawanie
 - funkcji, 56
 - zdjęć, 106
- dostawca tożsamości, 171
- dostęp do kontenera, 131
- Dropbox, 247
- drzewo katalogów
 - użytkownika, 138
- Durable Functions, 71, 82
- działanie Durable Function, 79–82

E

- ekran
 - Application Insights Analytics, 224
 - edycji zapytania, 225

ekran
 Keys, 144
 Metrics Explorer, 220
 Overview, 144, 219
elementy pulpitu
 nawigacyjnego, 231
Event sourcing, 78

F

FaaS, Function as a Service,
 16, 247
Face API, 135, 141
FaceAggregator, 85
filtrowanie
 nieuwierzytelnionych
 zapytań, 178
formularz, 21, 182, 216
framework, 247
frontend, 85, 96, 130, 247
Function
 App, 247
 as a Service, 247
funkcja
 aktywności, 72
 dodająca zdjęcia do
 kontenera, 106
 Hello Azure!, 20
 Orchestration_HttpStart,
 76
 pobierająca zasoby
 użytkownika, 99, 138
 przetwarzająca wybrane
 zdjęcia, 159
 przyjmująca zlecenia
 użytkownika, 139
 trenująca model, 154
 trwała, 92
 tworząca miniaturki
 zdjęć, 110
 usuwająca zdjęcia, 115
 zmieniająca nazwę
 zdjęcia, 115
 zwracająca zdjęcia, 112
funkcje, 26, 32
 debugowanie, 61
 orkiestracji, 72

testowania, 62
uruchomienie, 61, 75
zatrzymanie
 wykonywania, 63
funkcjonalności Azure, 233

G

Google Sign-In, 89

I

instalowanie pakietu
 roboczego, 54
integracja aplikacji z Face
 API, 135
interfejs
 Azure Storage Explorer,
 64
użytkownika, 93

J

JavaScript, 26

K

klasa
 DurableOrchestrationConte
 xt, 73
klasy anonimowe, 36
kolekcja, 243
komponenty wsparcia, 86
komunikacja
 klient-serwer, 14
 oparta na odpytywaniu, 14
konfiguracja
 akcji, 186
 dostawcy tożsamości, 171,
 177
 Function App, 66
 funkcji, 40
 narzędzi, 98
 procesu wdrażania, 129
 uwierzytelniania, 175, 177
 wiązania Azure Queue
 Storage, 42

wysyłki e-maili, 190
wyzwalacza HTTP, 41
źródła wdrożenia, 124
konto
 Cosmos DB, 241
 usługi Storage, 236, 237
Kudu, 247
 Debug console, 30

L

liczby pseudolosowe, 83
Logic Apps, 88
logika biznesowa, 195
logowanie do platformy
 Azure, 49

Ł

łączenie z serwisami, 181

M

metoda, 247
 CallActivityAsync<T>, 73
 CallSubOrchestrator
 ↳ Async<T>, 73
 CreateCheckStatus
 ↳ Response, 74, 76
 CreateTimer<T>, 73
 GetInput<T>, 73
 GetStatusAsync, 74
 RaiseEventAsync, 74
 StartNewAsync, 74
 TerminateAsync, 74
 WaitForExternalEvent<T>,
 74
metody
 testowanie, 202
miniaturki zdjęć, 110
mockowanie, 248
monitoring, 215
moq, 248
multi-tenancy problem, 17

N

narzędzia, 251
Newtonsoft.Json, 248
NuGet, 248

O

okno
 alertów, 221
 edycji elementu, 221
 Extract Interface, 201
 Monitor, 218
OneDrive, 248
OpenAPI, 248
operator warunkowy, 189

P

PaaS, Platform as a Service, 248
pakiet string, 31
pay-as-you-go, 248
pętla for each, 187
pętle nieskończone, 83
plan konsumpcyjny, 248
platforma Microsoft Azure, 227
pobieranie
 tożsamości w kodzie, 176
 zasobów, 99
połączenie ze źródłem kodu, 127
portal Azure, 231
Power BI, 248
Precompiled functions, 248
proces wdrażania, 129, 130
program Postman, 76
projekt interfejsu
 użytkownika, 93
przepływ, 191
przetwarzanie
 tekstu wiadomości, 188
 zdjęć, 159
pulpit nawigacyjny, 231

Q

Queue, 87

R

REST, 248
RESTful API, 248

S

serverless, 11, 248
serwisy zewnętrzne, 181
skalowanie, 78
sprawdzanie zgodności zdjęć, 145
Storage, 236, 237
storage account, 249
stos technologiczny, 249
strumień CI/CD, 121, 125
systemy, 251
szablon, 38
 Azure Functions, 24
 Durable Functions, 75
 HTTP trigger, 39, 57
 Manual trigger, 25
 procesu budowy projektu, 127

T

Task Hub, 77
testowanie, 195
 funkcji, 62, 207
 logiki biznesowej, 195
 metod funkcji, 202
 przepływu, 191
testy jednostkowe, 249
tożsamość, 176
trenowanie modelu, 154
trigger, 38, 184, 249
Twilio SMS, 88
tworzenie
 aplikacji funkcji, 51, 52
 aplikacji Logic App, 181

grupy zasobów, 49
konta Cosmos DB, 241
konta w usłudze
 magazynu, 50
organizacji, 125
planu hostingowego, 67
procesu budowy, 126
projektu, 126
usługi Face API, 141
typy wiązań, 74

U

udostępnienie frontendu, 130
uruchomienie
 aplikacji, 98
 funkcji, 61, 75
usługa, 86
 Application Insights Analytics, 223
 Azure Functions, 19
 Continuous Deployment, 121
 Face API, 141
 Logic App, 182
 Storage, 122, 236, 237
usługi Cognitive Services, 87
usuwanie
 konta magazynu, 238
 zdjęć, 115
uwierzelnianie w Azure, 171, 175

V

Visual Studio, 53, 56

W

wdrożenie, 65–68
 aplikacji, 121, 133, 166
 testów do chmury, 211
wiązania, 38
widok tekstowy, 191
wstrzykiwanie zależności, 209

wybór szablonu funkcji, 58
wykonywanie funkcji, 63
wykres kołowy, 226
wyrażenia warunkowe, 187
wyszukiwanie zasobów
i usług, 233
wyzwalacz, 38, 184, 249
wzorzec fan-out/fan-in, 151

X

xUnit, 249

Z

zakładka
 Face Recognition, 136
 Monitor, 218
zapytanie App Insights
 Analytics, 226
zarządzanie zdjęciami, 95
zasoby Azure, 234
zdjęcia, 95
 dodawanie do kontenera,
 106
 przetwarzanie, 159

 sprawdzanie zgodności,
 145
tworzenie miniaturek, 110
usuwanie, 115
zmienianie nazwy, 115
zwracanie, 112
zlecenia użytkownika, 139
zmienianie nazwy zdjęcia, 115
zwracanie zdjęć, 112

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

Odkryj platformę Microsoft Azure i możliwości architektury serverless!

- Poznaj usługi FaaS oferowane przez platformę Azure
- Twórz skalowalne aplikacje w architekturze serverless
- Naucz się praktycznie wykorzystywać potencjał chmury

Platformy chmurowe i oferowane przez nie usługi zdobyły serca całych rzesz programistów i inwestorów IT, którzy cenią sobie nie tylko ich wysoką skalowalność, niezawodność i bezpieczeństwo, lecz również stosunkowo niskie koszty, wygodę używania oraz możliwości szybkiego uruchamiania i udostępniania gotowych rozwiązań. Ostatnimi czasy szczególnie popularna jest architektura serverless, dzięki której tworzenie i wdrażanie wydajnych aplikacji sieciowych wymaga niewielkich nakładów pracy i jest możliwe w bardzo krótkim czasie.

Jeśli chcesz od praktycznej strony poznać proces budowania i wdrażania aplikacji wykorzystującej usługi FaaS oferowane przez chmurę firmy Microsoft, sięgnij po książkę *Serverless na platformie Azure*. Bez zbędnych wstępów otworzy przed Tobą świat nowoczesnej architektury, umożliwiającej szybkie i łatwe wprowadzanie na rynek wydajnych, skalowalnych i łatwych w utrzymaniu aplikacji biznesowych. Dzięki lekturze poznasz kroki niezbędne do opracowania rozwiązania opartego na Azure Functions, realizacji przepływów za pomocą Durable Task Framework i integracji aplikacji z zewnętrznymi serwisami przy użyciu Azure Logic Apps. Dowiesz się też, jak monitorować rozwiązanie za pomocą usługi Application Insights.

- Zalety i ograniczenia architektury i przetwarzania serverless
- Tworzenie aplikacji opartej na Azure Functions
- Realizacja przepływów za pomocą Durable Functions
- Integracja z zewnętrznymi serwisami przy użyciu Azure Logic Apps
- Wdrożenie aplikacji za pomocą narzędzi CI/CD w ramach usługi Azure DevOps
- Testowanie aplikacji opartej na Azure Functions
- Monitorowanie rozwiązania za pomocą Application Insights

Buduj, wdrażaj, oszczędzaj — pracuj wydajniej dzięki architekturze serverless!

	Sprawdź nasze szkolenia!  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI Sięgnij po więcej! ▶ 
 helion.pl  HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		ISBN 978-83-283-5068-7  9 788328 350687
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 49,00 zł