# Tao mikrousług

## Projektowanie i wdrażanie

Richard Rodger

MANNING

Helion

Tytuł oryginału: The Tao of Microservices

Tłumaczenie: Marcin Dzieszko

ISBN: 978-83-283-4807-3



Helion SA
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/taomik*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

# 2 Usługi

**Niniejszy rozdział opisuje zagadnienia:**

- udoskonalenie koncepcji mikrousług,
- przegląd głównych wariantów architektury mikrousług,
- porównanie monolitów z mikrousługami,
- użycie konkretnego przypadku do zbadania mikrousług,
- myślenie o mikrousługach jak o komponentach oprogramowania.

Aby być świadomym konsekwencji i kompromisów związanych z przejściem do nowej architektury, musisz zrozumieć, w jaki sposób różni się ona od poprzedniej architektury oraz jak nowe metody rozwiązują dawne problemy. Jakie są istotne różnice między architekturą monolityczną a architekturą opartą na mikrousługach? Jakie są nowe sposoby myślenia? I wreszcie jak mikrousługi rozwiązują problemy rozwoju oprogramowania korporacyjnego?

**Mikrousługa** to jednostka rozwoju oprogramowania. Architektura mikrousługi tworzy model mentalny, który umożliwia postrzeganie zagadnienia w uproszczony sposób. Książka ta przedstawia mikrousługi jako komponenty oprogramowania najbliższe ideałowi. Są to idealne artefakty do precyzyjnego zastosowania w produkcji. Łatwo oszacować ich ilość, aby zapewnić prawidłowe działanie. Podstawą mikrousług jest przekonanie, że powyższe aspekty ich architektury zapewniają szybki, praktyczny i skuteczny sposób tworzenia wartości biznesowej za pomocą oprogramowania. Przyjrzyjmy się szczegółom, aby zobaczyć, jak to działa w praktyce.

## 2.1. Definicje mikrousług

Termin *mikrousługa* jest z natury rozmyty. Jest to społeczny skutek rosnącej popularności tej architektury. Kiedy używamy wspomnianego terminu, powinniśmy sprecyzować, jak go rozumieć. W znacznej części artykułów na temat mikrousług wyrażany jest taki sam stosunek do rozwoju oprogramowania, ale używa się w nich różnych definicji tego kluczowego pojęcia. Akceptowanie wielu słabych definicji z kolei ogranicza nasze myślenie i stanowi łatwy cel krytyki. Przeanalizujmy kilka proponowanych definicji:

- *Mikrousługi są samodzielnymi komponentami oprogramowania, które mają nie więcej niż 100 linii kodu.* Ta definicja oddaje chęć tworzenia małych mikrousług utrzymywanych przez jednego programistę, a nie przez cały zespół. To odwołanie do idei, że ekstremalna prostota ma ekstremalne zalety: 100 linii kodu może być szybko i pewnie sprawdzone pod kątem błędów[1]. Niewielki fragment kodu można również w razie potrzeby łatwo usunąć i przepisać na nowo. Są to pożądane cechy w przypadku mikrousług, ale nie wyczerpuje to wszystkich zagadnień. Na przykład nie są podniesione kwestie dotyczące wdrażania i komunikacji między usługami. Podstawową słabością tej definicji jest również użycie arbitralnego ograniczenia liczby linii kodu, które traci sens, jeśli zmienimy język programowania. Ponieważ rozważamy inne definicje, zachowajmy pragnienie tworzenia kodu usługi wystarczająco niewielkiego, aby łatwo można go było zweryfikować i wyrzucić, jeśli zajdzie taka potrzeba.
- *Mikrousługi są wdrażanymi niezależnie procesami komunikującymi się asynchronicznie przy użyciu lekkich mechanizmów; skupiają się na konkretnych możliwościach biznesowych oraz działają w zautomatyzowanym środowisku, niezależnie od platformy i użytych języków.* Część definicji mikrosystemów ma charakter ogólny, zawierając wszystkie możliwe aspekty. Te definicje posiadają długą listę pożądanych atrybutów. Czy te atrybuty są uporządkowane według ważności? Czy ich lista jest wyczerpująca? Czy są dobrze zdefiniowane? Ogólne definicje dają poczucie, jakbyś był w odpowiedniej galaktyce, ale nie zapewniają wskazówki, jak dostać się do systemu mikrousługi. Zapraszają do niekończącej się debaty nad definicjami atrybutów. Czym, na przykład, jest naprawdę lekki mechanizm komunikacji?[2] To, co możemy wziąć z tych definicji,

[1] C. A. R. Hoare, twórca algorytmu *quicksort*, w swoim wykładzie z okazji otrzymania Nagrody Turinga (Turing Award Lecture) w 1980 r. wypowiedział słynne zdanie: „Istnieją dwa sposoby tworzenia projektu oprogramowania: jednym z nich jest uczynienie go tak prostym, że *oczywiście* nie ma w nim żadnych braków, a innym sposobem jest uczynienie go tak skomplikowanym, że nie ma w nim *oczywistych* braków".

[2] Niemożliwe jest wygranie wojny definicji. Gdy tylko przedstawisz ostateczny kontrprzykład, Twój przeciwnik zaprzeczy, że tenże jest w rzeczywistości przykładem dotyczącym omawianego tematu. Brytyjski filozof Antony Flew dostarcza kanonicznego przykładu tej taktyki, którą można sparafrazować w następujący sposób — Robert: „Wszyscy Szkoci noszą kilt!"; Hamish: „Mój wujek Duncan nosi spodnie"; Robert: „Tak, ale żaden *prawdziwy* Szkot tak nie robi".

to działający zestaw pomysłów, które można wykorzystać w praktyce, ale które same z siebie nie zapewniają wystarczającej przejrzystości.

- *Mikrousługi to miniserwery webowe oferujące niewielkie API, które akceptują i zwracają dokumenty typu JSON.* Jest to z pewnością powszechna implementacja. I to są rzeczywiście mikrousługi. Ale jak duże one są? I w jaki sposób ta definicja odnosi się do wszystkich innych problemów, takich jak niezależne wdrażanie? Ta definicja jest jednocześnie zbyt normatywna w niektórych kwestiach i niewystarczająco normatywna w innych. To definicja według pewnego archetypu. Niewielu nie zgodziłoby się z tym, że są to mikrousługi. A jednak wykluczona tu została większość ciekawych wzorów architektonicznych, w szczególności tych, które korzystają z asynchronicznych komunikatów. Ta definicja jest nie tylko słaba, ale również i niebezpieczna. Dowody empiryczne z wdrożonych systemów u klienta sugerują, że często taki model prowadzi do silnie powiązanych usług, które muszą być wdrożone razem[3]. Kluczowy wniosek z tej nieudanej definicji jest taki, że ograniczanie się do myślenia tylko w kategoriach API usług webowych uniemożliwia nam docenianie zasadniczych możliwości, które może przynieść szersza koncepcja. Definicja powinna pobudzać nasze myślenie, a nie je ograniczać.
- *Mikrousługa to niezależny komponent oprogramowania, na którego zbudowanie i wdrożenie potrzeba nie więcej niż jednej iteracji.* W tej definicji nacisk kładziony jest na ludzką stronę architektury. Wyrażenie *niezależny komponent oprogramowania* jest sugestywne i na tyle obszerne, że definicja ta stara się również zawierać strategie implementacji. Mikrousługi to komponenty oprogramowania wykorzystujące powszechne zrozumienie tego terminu[4]. Definicja ta wyraża pragnienie, aby mikrousługi rzeczywiście były „mikro" poprzez ograniczenie zasobów potrzebnych do ich napisania: jedna iteracja to wszystko, w co musisz zainwestować. Jest to ukłon w stronę stałego dostarczania — musisz być w stanie wdrożyć komponent w ciągu iteracji. Definicja ta jest ostrożna, unika wzmianek dotyczących procesów systemu operacyjnego, sieci, przetwarzania rozproszonego i protokołów komunikatów; żadne z nich nie jest esencjonalną właściwością[5].

Musimy zaakceptować fakt, że nie jesteśmy dziećmi w wieku szkolnym, ale profesjonalnymi programistami, i że żyjemy w skomplikowanym świecie dorosłych. Nie ma uporządkowanej definicji mikrousług, a każda wybrana przez nas definicja ogranicza

[3] W moim poprzednim wcieleniu konsultanta kierowałem swoimi biednymi zespołami tak, aby zbudowały wiele dużych systemów. Związaliśmy się najwspanialszymi węzłami gordyjskimi.

[4] **Komponenty oprogramowania** to niezależne, rozszerzalne, dobrze zdefiniowane bloki konstrukcyjne wielokrotnego użytku.

[5] Procesy Erlanga to z pewnością mikrousługi lub, co być może jest bardziej poprawne, nanousługi! Usilnie zalecam Ci przeczytanie doktoratu Joe'ego Armstronga, w którym opisano wszystkie szczegóły: *Making Reliable Distributed Systems in the Presence of Software Errors*, Royal Institute of Technology, 2003, *http://erlang.org/download/armstrong_thesis_2003.pdf*.

nasze myślenie. Zamiast więc szukać definicji, która zależy od parametrów liczbowych lub próbuje być wyczerpująca albo, na odwrót, zbyt wyspecjalizowana, powinniśmy dążyć do opracowania struktury koncepcyjnej, która jest *wytwórcza*. Pojęcia w ramach tej struktury pozwolą na dokładne zrozumienie nieodłącznych kompromisów związanych z architekturą mikrousługi. Następnie zastosujemy te pojęcia w danym kontekście, aby dostarczyć działające oprogramowanie[6].

## 2.2. Studium przypadku: wydanie cyfrowe gazety

Większość rozdziałów w tej książce wykorzystuje studium przypadku, aby poddać pod dyskusję praktyczne przykłady opisywanych pojęć. Poddane analizie będą systemy oprogramowania, które muszą dostarczyć szereg funkcjonalności; w każdym rozdziale zastanowimy się, jak architektura mikrousług może je implementować. Dla każdego systemu skoncentrujemy się na podzbiorze tych funkcjonalności, które dotyczą tematu omawianego w danym rozdziale. Rozdział 9. to pełne studium przypadku, w tym kod, który stanowi praktyczny przykład systemu mikrousług z wykorzystaniem technik architektonicznych opracowanych w tej książce.

Nasze studium w tym rozdziale dotyczy cyfrowej edycji gazety. Opiszmy ten system dokładnie, zaczynając od celów biznesowych. Generują one wymagania, które będziemy musieli określić nieformalnie. W tym rozdziale przyjrzymy się częściowym implementacjom tych nieformalnych wymagań przy użyciu mikrousług.

### 2.2.1. Cele biznesowe

Gazeta oferuje zarówno darmowe, jak i płatne treści. Aby wyświetlić płatne treści, użytkownicy muszą dokonać płatnej subskrypcji. Przychody gazety są generowane przez subskrypcje i reklamy. Reklama może być kierowana do konkretnego użytkownika lub skupiać się na odpowiedniej treści, aby być bardziej skuteczna. Aby zwiększyć przychody z reklam, czas korzystania z witryny przez użytkowników powinien być zmaksymalizowany.

Osoby pracujące nad gazetą, korzystające z witryny, powinny mieć możliwość publikowania artykułów w sposób ciągły za pomocą systemu zarządzania treścią. Powinny móc również przeglądać statystyki dotyczące treści, które opublikowały, aby uzyskać informacje zwrotne na temat ich skuteczności.

Aby zmaksymalizować dostęp do gazety, jest ona dostarczana za pośrednictwem strony internetowej i aplikacji mobilnych. Powinna być również dostarczona wyszukiwarka do przeszukiwania treści artykułów darmowych i płatnych, zoptymalizowana pod kątem uzyskania jak największej trafności.

### 2.2.2. Wymagania nieformalne

Na podstawie celów biznesowych możesz przedstawić listę nieformalnych wymagań. Wymagania będą wpływać na Twoje decyzje wdrożeniowe:

[6] Mikrousługi są tematem wartym całej książki — wybiegają daleko poza ramy szablonowej definicji. Zaraz, zaraz…

- Treść składa się z artykułów, z których każdy ma osobną stronę.
- Istnieją również specjalne strony z listami artykułów, takie jak strona tytułowa, a także strony pod kątem zainteresowań.
- Witryna internetowa oraz aplikacja we wszystkich wersjach powinny korzystać z opartego na REST wspólnego API, obsługiwanego przez część serwerową systemu.
- System powinien dostarczać statyczne wersje treści podstawowych, aby wyszukiwarki internetowe mogły je zindeksować, ale może również dostarczać dynamicznie tworzoną zawartość dodatkową.
- System musi implementować koncepcję użytkowników, zarówno czytelników, jak i autorów, z odpowiednimi uprawnieniami dla różnych poziomów dostępu.
- Treści na stronach należy kierować do bieżącego użytkownika, dopasowując zawartość do jego profilu za pomocą reguł biznesowych lub algorytmów optymalizacyjnych.
- Witryna jest ciągle rozwijana, podlega ostrej konkurencji z innymi internetowymi gazetami, więc nowe funkcjonalności muszą być dodawane szybko. Zmiany te obejmują specjalne krótkoterminowe miniaplikacje, takie jak przekazujące interaktywne treści podczas wyborów.

### 2.2.3. Podział funkcjonalny

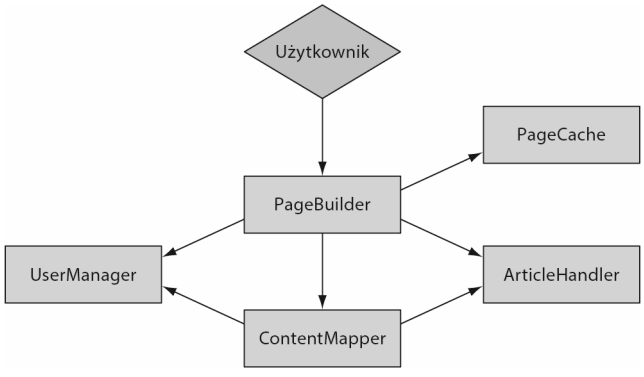
Z czysto funkcjonalnej perspektywy i bez odniesienia do żadnej architektury można stwierdzić, że wspomniane wymagania już teraz pozwalają zastanowić się nad tym, jak wdrożyć system gazety. Oto kilka rzeczy, które ten system powinien wykonywać:

- obsługa zawartości artykułów oraz operacje odczytu, zapisu i zapytań;
- tworzenie zawartości strony i dostarczanie pamięci podręcznej używanej do skalowania;
- obsługa kont użytkowników: logowanie, wylogowanie, profile itd.;
- dostarczanie ukierunkowanych treści i mapowanie tożsamości użytkowników na odpowiednie artykuły.

Te funkcjonalności sugerują niektóre komponenty oprogramowania, które powinieneś zbudować. Przez chwilę udajmy, że są one reprezentowane przez klasy zorientowane obiektowo:

- `ArticleHandler` — zapewnia operacje na zawartości artykułu;
- `PageBuilder` — generuje strony;
- `PageCache` — zarządza pamięcią podręczną strony;
- `UserManager` — zarządza użytkownikami;
- `ContentMapper` — decyduje o tym, jakie treści pokazać użytkownikowi.

Możesz nawet narysować możliwe zależności między tymi komponentami, tak jak pokazano na rysunku 2.1.




**Rysunek 2.1.** Możliwa architektura komponentów dla systemu gazety internetowej

Czy to właściwe komponenty? Czy są to właściwe zależności? Jeszcze za wcześnie, by to stwierdzić. Czy to są mikrousługi? Być może. Architektura mikrousług musi zapewniać proces analityczny decydujący o tym, które mikrousługi zbudować. Jakoś musisz przejść od nieformalnych wymagań do konkretnego zestawu usług w produkcji. Aby rozpocząć ten proces, przyjrzyjmy się bliżej właściwościom architektur mikrousług i sposobom ich konstruowania.

## 2.3. Architektury mikrousług

Jeśli przyjmiemy, że mikrousługi powinny porozumiewać się ze sobą za pomocą komunikatów, i chcemy, aby były niezależne, oznacza to, że muszą mieć dobrze zdefiniowany interfejs komunikacyjny. Odrębne komunikaty są najbardziej naturalnym mechanizmem do definiowania tego interfejsu[7].

Zrozumienie, że komunikacja między usługami może być określona w kategoriach komunikatów, pozwala jeszcze lepiej uświadomić sobie dynamiczną naturę mikrousług. Na jednym poziomie abstrakcji musisz zrozumieć, które usługi komunikują się ze sobą. W praktyce wiedza ta jest mniej użyteczna, niż mogłoby się wydawać. Wraz ze wzrostem liczby usług rośnie również liczba połączeń między nimi i pełna sieć interakcji staje się trudna do wizualizacji. Jednym ze sposobów na złagodzenie tej złożoności jest przyjęcie podejścia opisania systemu od strony poszczególnych komunikatów. Zauważmy, że usługi i komunikaty to dwa aspekty tej samej struktury. Często lepiej jest patrzeć na system mikrousług od strony komunikatów, które przechodzą przez ten system, a nie od strony usług, które przetwarzają odbierane komunikaty. Z tej perspektywy można analizować sposoby interakcji, znajdować typowe wzorce i tworzyć projekt architektury mikrousług.

[7] Nie wyklucza to innych mechanizmów komunikacyjnych, takich jak transmisja strumieniowa danych, ale są one zazwyczaj używane do specjalnych zastosowań lub jako warstwa transportowa dla osadzonych wiadomości.

### 2.3.1. Architektura miniserwerów webowych

W architekturze miniserwerów webowych mikrousługi to nic innego jak serwery webowe, które oferują niewielkie interfejsy oparte na REST. Komunikaty to żądania i odpowiedzi HTTP. Treść komunikatu to dokumenty JSON lub XML albo proste zapytania. Jest to architektura synchroniczna. Żądania HTTP wymagają odpowiedzi. Przyjmiemy to jako punkt wyjściowy, a następnie zastanowimy się, jak sprawić, by te miniserwery były bardziej podobne do komponentów oprogramowania.

Każda mikrousługa musi znać lokalizację innych usług, z którymi chce się komunikować. Jest to jednocześnie i ważna cecha, i słabość miniserwerów webowych. Kiedy jest to tylko kilka usług, możesz skonfigurować ręcznie połączenia między nimi, ale szybko staje się to niemożliwe do zarządzania, kiedy liczba usług rośnie. Standardowym rozwiązaniem jest mechanizm wykrywania usług.

Aby zapewnić wykrywanie wszystkich usług, musisz uruchomić odpowiednią usługę w swoim systemie, która przechowuje listę wszystkich mikrousług i ich lokalizacji w sieci. Każda mikrousługa musi przesłać zapytanie do usługi wykrywania, aby znaleźć inne usługi, z którymi chce się komunikować. Niestety to rozwiązanie ma dużo ukrytej złożoności. Po pierwsze, utrzymywanie usługi wykrywania, która przechowuje informacje zgodne z rzeczywistością, jest nietrywialne — napisanie dobrej implementacji wykrywania jest trudne[8]. Po drugie, mikrousługi muszą przechowywać wiedzę o innych usługach, którą uzyskają od usługi wykrywania, i muszą radzić sobie z problemem sprawdzania poprawności tej wiedzy. Po trzecie, przechowywanie tych informacji prowadzi do ścisłej zależności między usługami. Dlaczego? Rozważ to, że wewnątrz kodu monolitycznego potrzebne jest odwołanie do obiektu w celu wywołania jego metody. Teraz robisz to samo, tylko że poprzez sieć — potrzebujesz lokalizacji sieciowej i punktu końcowego adresu URL. Jeśli używasz wykrywania usług, wprowadzasz potrzebę dostarczenia dodatkowego kodu i modułów do swoich usług, by współpracowały z mechanizmem wykrywania.

W najprostszej konfiguracji ta architektura jest typu punkt-punkt. Mikrousługi komunikują się ze sobą bezpośrednio. Możesz rozszerzyć tę architekturę dzięki bardziej elastycznym wzorcom komunikatów poprzez używanie inteligentnego równoważenia obciążenia. Aby skalować daną mikrousługę, umieść moduł równoważenia obciążenia protokołu HTTP[9] tak, aby obsługiwał ruch dla zbioru instancji mikrousług. Będziesz potrzebował tego dla każdej mikrousługi, którą chcesz skalować. Zwiększy to złożoność Twojego wdrożenia, ponieważ musisz również zarządzać konfiguracjami modułu równoważenia obciążenia, tak samo jak swoimi mikrousługami.

---

[8] Dostępne są względnie wydajne implementacje wykrywania usług: ZooKeeper (*https://zookeeper.apache.org/*), Consul (*https://consul.io/*), etcd (*https://github.com/coreos/etcd*) i inne. Żadna z nich nie jest w pełni zgodna z wymaganiami w kwestii odporności na błędy i sprawdzania spójności danych, mimo że wszystkie są przygotowane do pracy w środowiskach produkcyjnych. Sprawdź artykuły z serii „Jepsen" autorstwa Kyle'a Kingsbury'ego na *https://aphyr.com/tags/jepsen*, gdzie znajduje się szczegółowa analiza.

[9] Odpowiednie komponenty równoważenia obciążenia to NGINX (*http://nginx.org/*), HAProxy (*http://www.haproxy.org/*) i Eureka (*https://github.com/Netflix/eureka*).

Jeśli sprawisz, że Twój moduł równoważenia obciążenia będzie inteligentny, będziesz mógł zacząć czerpać większe korzyści z mikrousług. Nic na to nie wskazuje, że wszystkie mikrousługi zarządzane przez dany moduł muszą mieć tę samą wersję tej samej mikrousługi. Możesz częściowo wdrożyć i przetestować nowe wersje mikrousługi w środowisku produkcyjnym, dodając je do danej grupy równoważenia obciążenia. Jest to prosty sposób na uruchamianie wielu wersji tej samej mikrousługi w tym samym czasie.

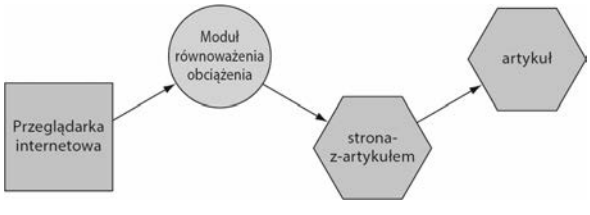
Możesz umieścić różne mikrousługi za tym samym modułem równoważenia obciążenia i użyć tego komponentu, aby dopasować do wzorca właściwości przychodzących komunikatów i następnie przypisać je do odpowiedniego typu mikrousługi[10]. Rozważ moc, jaką Ci to daje — możesz zwiększyć funkcjonalność systemu poprzez dodanie nowej mikrousługi i zaktualizowanie zasad modułu równoważenia obciążenia. Nie trzeba zmieniać, aktualizować, ponownie wdrażać ani w żaden inny sposób dotykać innych uruchomionych usług. Zdolność do wykonywania tego rodzaju małych, niewiele znaczących i niskiego ryzyka zmian w systemie produkcyjnym w dużym stopniu stanowi o atrakcyjności architektury mikrousługi. To sprawia, że ciągłe dostarczanie kodu do systemu produkcyjnego jest dużo bardziej możliwe.

**Równoważenia obciążenia po stronie klienta**

Moduł równoważenia obciążenia nie musi być oddzielnym procesem mającym dostęp do nasłuchujących mikrousług. Możesz użyć biblioteki po stronie klienta, osadzonej w mikrousłudze klienta, aby przeprowadzić inteligentne równoważenie obciążenia. Zaletą tego rozwiązania jest to, że nie musisz martwić się wdrażaniem i konfigurowaniem wielu systemów równoważenia obciążenia w sieci. Taki system może korzystać z mechanizmu wykrywania usług, aby określić, gdzie wysłać komunikaty.

## 2.4. Diagramy mikrousług

Narysujmy niektóre z tych konfiguracji, aby ułatwić ich wizualizację. Tradycyjne diagramy sieciowe są mniej przydatne dla mikrousług, ponieważ jest dla nich znacznie więcej komponentów i zawsze jesteśmy bardziej zainteresowani przepływem komunikatów niż ich zwykłym istnieniem. Rysunek 2.2 pokazuje prosty system typu punkt-punkt: część strony gazety. Później zbudujemy pełną strukturę, ale skupmy się najpierw na interakcjach między mikrousługami budującymi stronę artykułu.




**Rysunek 2.2.** Budowanie strony artykułu

[10] Jednym ze sposobów, aby to zrealizować, jest użycie modułów rozszerzeń dla serwerów takich jak NGINX. Możesz również rozwinąć własne rozwiązanie, używając platformy np. Node.js (*https://nodejs.org/*).

Usługa *artykuł* przechowuje dane artykułu. Usługa *strona-z-artykułem* tworzy kod HTML dla danego artykułu. Każdy artykuł ma własny unikatowy adres URL strony. Inteligentny system równoważenia obciążenia tworzy trasę dla żądań adresu URL artykułu pochodzących od klientów przeglądarek internetowych do usługi *strona-z-artykułem*.

Przyjmijmy za pewnik, że są to usługi do zbudowania. Widać, że różnią się od bardziej tradycyjnych elementów zorientowanych obiektowo, sugerowanych wcześniej (`PageBuilder`, `ArticleHandler`). W odpowiednim czasie zaczniesz tworzyć te usługi na podstawie komunikatów, które definiują system. Teraz zobaczmy, jak konwencje widoczne na diagramie mogą pomóc w zademonstrowaniu projektu systemu.

Na rysunku 2.2 linie ciągłe reprezentują komunikaty synchroniczne. To oznacza, że usługa klienta oczekuje natychmiastowej odpowiedzi ze strony nasłuchującej usługi, czyli że nie może kontynuować swojej pracy bez tej odpowiedzi. Strzałki są skierowane od usługi klienta w stronę usługi nasłuchującego. Strzałki są ciągłe, co oznacza, że usługa nasłuchująca *konsumuje* komunikat, czyli że nikt inny go nie widzi.

Mikrousługi są reprezentowane przez sześciokąty. Elementy zewnętrzne dla systemu (np. przeglądarka internetowa) są reprezentowane jako prostokąty, a elementy wewnętrzne w systemie (np. moduł równoważenia obciążenia) — jako koła. W przypadku mikrousług sześciokąt nie oznacza jednej mikrousługi, ale jedną lub więcej działających instancji mikrousługi tego samego rodzaju. Trzeba o tym pamiętać. W systemie produkcyjnym prawie nigdy nie uruchamiasz tylko jednej instancji danej mikrousługi.

## 2.5. Drzewo zależności mikrousług

Mikrousługi z założenia są zależne od siebie, ponieważ każda wykonuje tylko niewielką część pracy dla dowolnego żądania HTTP lub dla innego zadania w systemie. W synchronicznej architekturze typu punkt-punkt, którą moglibyśmy nazwać **mikrousługami podstawowymi**, drzewo zależności może być trudne do zarządzania w miarę wzrostu.

**Pułapka rozproszonego monolitu**

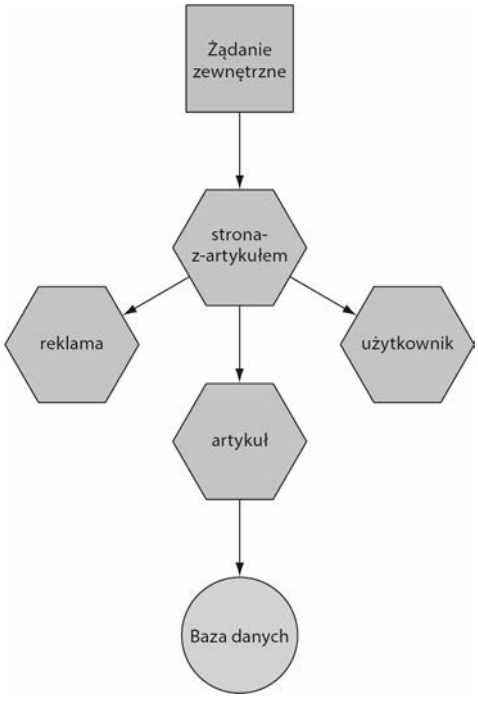
Rozproszony monolit jest nieprzyjemną pułapką czekającą na początkujących twórców mikrousług, którzy naiwnie wykorzystują tradycyjne wzorce obiektowe w kontekście mikrousług. W głównym nurcie języków zorientowanych obiektowo należy podać sygnatury dokładnej metody i typu obiektu. Jeśli typy nie będą zgodne, pojawi się błąd kompilacji. (Czy to stanowi prawdziwą korzyść, jeśli chodzi o tworzenie oprogramowania, czy też nie, to już temat dyskusji na inną okoliczność).

W architekturze mikrousług niedopasowanie typu nie jest błędem kompilacji, ale błędem czasu wykonywania, który może położyć Twój system. Używanie typów ścisłych oznacza, że budujesz rozproszony monolit, w którym wywołania metod są uruchamiane w sieci.

O wiele łatwiej jest zbudować tradycyjny monolit! Aby zyskać korzyści z architektury mikrousług, musisz zostawić niektóre z najlepszych praktyk pochodzących ze świata monolitycznego.

Głównym zagrożeniem są zwłaszcza *zależności usług*, w których jedna mikrousługa lub ich większa liczba staje się współzależna, zatem nowe wersje muszą być wdrożone w tym samym czasie. To szczególnie łatwo może nastąpić, jeśli używasz bibliotek serializacji obiektów, które wymagają pełnej zgodności wszystkich właściwości znajdujących się w komunikatach JSON lub XML. Dodaj pole do encji w jednej mikrousłudze, a będziesz musiał je dodać do wszystkich mikrousług, które używają tej jednostki. I zanim się obejrzysz, skończysz z rozproszonym monolitem — najgorszym z obu światów.

Wróćmy do studium przypadku. Wyświetlanie artykułu w gazecie wymaga wykonania większej liczby czynności niż pobieranie danych artykułu i jego formatowanie. Niektóre z tych działań pokazano na rysunku 2.3. Prawdopodobnie masz do czynienia z aktywnie zalogowanym użytkownikiem; musisz wyświetlić jego status w polu u góry strony, gdzie użytkownik może się wylogować lub wybrać zarządzanie swoim kontem. To sugeruje wybranie mikrousługi z odpowiedzialnością za użytkowników. Będziesz miał do czynienia z usługą reklamową, ponieważ to część biznesowego modelu dla gazety.

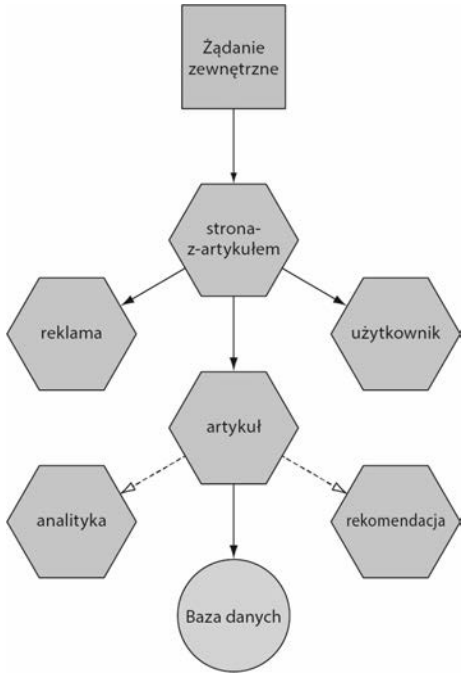



**Rysunek 2.3.** Budowanie strony artykułu z całkowitą zawartością

Usługa *strona-z-artykułem* pobiera treść z usług: *reklama*, *użytkownik* i *artykuł*. Nie ma sensu przesyłać żądań po kolei, czekając na pomyślną odpowiedź z poprzedniego, zanim zostanie wysłane następne. Zamiast tego musisz wysłać wszystkie żądania w tym samym czasie i odebrać odpowiedzi, gdy tylko przyjdą. Zaimplementowanie tego nie jest wielką filozofią, ale na pewno trudniej jest wtedy utrzymać porządek w kodzie. Musisz rozwinąć pewne abstrakcje związane z wysyłaniem i odbieraniem komunikatów, aby można było ujednolicić warstwę komunikacji między usługami.

Na rysunku 2.3 możesz zobaczyć, że baza danych jest eksponowana przez usługę *artykuł*. Nigdy nie pokazuj swoich decyzji implementacyjnych innym usługom! To w zasadzie złota reguła. Jedną z największych korzyści, które powinieneś uzyskać w zamian za dodatkową złożoność zarządzania mikrousługami, jest to, że w systemie produkcyjnym będziesz miał możliwość zmiany niemal wszystkiego niezależnie od innych elementów. Powinieneś być w stanie zmienić bazę danych prawie bez konieczności ponownego uruchamiania usługi *strona-z-artykułem*.

Dzięki usłudze *strona-z-artykułem* możesz policzyć, ile razy artykuł był czytany, ale obarczenie wspominanej usługi tą funkcją nie jest dobrym pomysłem. Mogą być jeszcze inne rzeczy wykonywane przez różne usługi, które to rzeczy chciałbyś zrobić, gdy artykuł jest czytany (np. optymalizacja silnika rekomendacji). Jednym ze sposobów odłączenia tych funkcjonalności od usługi *strona-z-artykułem* jest użycie *asynchronicznego* komunikatu, oznaczonego kropkowaną linią na rysunku 2.4. Usługa *strona-z-artykułem* wysyła komunikat, który zawiera informację o zdarzeniu, że artykuł został przeczytany, ale usługi tej „nie obchodzi", ile osób otrzymało tę odpowiedź lub ile osób może jej potrzebować.




**Rysunek 2.4.** Informowanie innych usług o tym, że artykuł został przeczytany

W tym przypadku usługi *analityka* i *rekomendacja* nie *konsumują* odbieranych komunikatów. Te komunikaty są zamiast tego *monitorowane*, jak na to wskazują otwarte groty strzałek. Aby to osiągnąć, możesz użyć kolejki przechowującej zduplikowane komunikaty[11]. Istotne jest myślenie na właściwym poziomie architektonicznym. Liczy

[11] Jednym z wielu sposobów osiągnięcia tego jest użycie funkcjonalności publikowania/subskrybowania brokera Redis: *https://redis.io/commands/pubsub*.

się przede wszystkim nie sposób implementacji takiej interakcji z komunikatami, ale fakt, że są to komunikaty asynchroniczne i monitorowane.

**„Nie powtarzaj się" nie jest złotą regułą**

Mikrousługi pozwalają bezpiecznie naruszać zasadę „nie powtarzaj się" (ang. *DRY — don't repeat yourself*). W tradycyjnym projektowaniu oprogramowania zaleca się generalizację powtarzających się bloków kodów, aby nie skończyć na utrzymywaniu wielu kopii nieznacznie różniącego się kodu. Projektowanie mikrousług polega na czymś dokładnie odwrotnym: każda mikrousługa może iść własną drogą. Antywzorcem jest tutaj szukanie wspólnej logiki biznesowej (infrastruktura jest, jak zawsze, wyjątkiem) i próba napisania jak najbardziej ogólnych modułów do wykorzystania w wielu mikrousługach. Dlaczego? Ponieważ ogólny kod jest złożony, musi zajmować się przypadkami wyjątkowymi i jest główną przyczyną narastającego długu technicznego.

Ogólne reguły biznesowe oraz modele domen stają się z czasem skomplikowane, ponieważ ogólny przypadek nie jest wystarczający, aby poradzić sobie ze złożonością realnego świata. Lepiej zachować wszystko oddzielnie, na prostszych zasadach specyficznych dla danego przypadku i na małych modelach odpowiadających poszczególnym mikrousługom. Dzięki temu Twoje mikrousługi będą niezależne i pozwolą programistom pracować równolegle na prostszych fragmentach kodu.

Wraz z rozwojem systemu rośnie również drzewo zależności między usługami zarówno wszerz, jak i wzdłuż. Na szczęście doświadczenia w tej dziedzinie sugerują[12], że wszerz rośnie szybciej niż wzdłuż. Gdy drzewo w końcu się rozrasta, zaczynają się pojawiać problemy związane z opóźnieniem przetwarzania komunikatów. Oto sedno problemu: czas reakcji w sieci ulega wypaczeniu, kiedy większość odpowiedzi wraca szybko, ale niektóre wymagają dużo więcej czasu niż przeciętnie. Dlatego aby wyznaczać cele wydajnościowe, używamy percentyli[13], ponieważ średnia nie ma charakteru informacyjnego. Gdy wiele elementów komunikuje się w krótkim okresie, czasy reakcji najgorszych przypadków rosną znacznie szybciej niż przeciętnych i niska wydajność w niewielkiej liczbie przypadków staje się w efekcie przestojem i następuje przekroczenie limitów czasu.

Jak sobie radzisz z tym problemem? Jednym ze sposobów jest scalanie usług[14], aby zmniejszyć zapotrzebowanie ruchu sieciowego. To jest poprawna optymalizacja wydajności, szczególnie w dojrzałych systemach. Znacznie mniej boli upewnianie się, że Twój kod infrastruktury jest w dobrej kondycji i że prace związane z tworzeniem sieci i usługą wyszukiwania są wyabstrahowane względem głównej logiki biznesowej mikrousług.

---

[12] Cenną inwestycją w zrozumienie architektury mikrousługi jest obejrzenie konferencji wideo, których wiele jest dostępnych w internecie. Omówiono w nich takie kwestie z praktycznego punktu widzenia w działających systemach produkcyjnych.

[13] *Percentyl* pokazuje, jaki procent odpowiedzi pojawił się w danym czasie. Na przykład percentyl *90% — 500 ms czasu reakcji* oznacza, że 90% odpowiedzi trwało krócej niż 500 ms.

[14] Scalanie usług to w 100% akceptowalna optymalizacja wydajności i tylko wydajności. Niemniej tracisz wtedy wiele korzyści z architektury mikrousługi. Wytyczna, by tworzenie mikrousługi zajmowało co najwyżej iterację, to także tylko wytyczna. Ale otrzymujesz wynagrodzenie za wydanie swojego zawodowego osądu w tych sprawach.

### 2.5.1. Architektura komunikatów asynchronicznych

Jako kompletną alternatywę względem podejścia typu punkt-punkt może warto przetransportować wszystkie Twoje komunikaty za pośrednictwem kolejki komunikatów? W tej architekturze masz jedną lub więcej kolejek, które obsługują wszystkie komunikaty. Usługi klienckie publikują komunikaty w kolejce, a usługi odsłuchowe je odczytują.

Korzystanie z kolejki komunikatów zapewnia większą elastyczność za cenę zwiększonej złożoności systemu. Kolejka taka jest kolejnym punktem, gdzie mogą wystąpić awarie, i wymaga takiej samej dbałości i uwagi jak baza danych w systemie produkcyjnym. Ponadto, w ramach skalowania systemu, kolejki komunikatów muszą być dystrybuowane podobnie jak bazy danych.

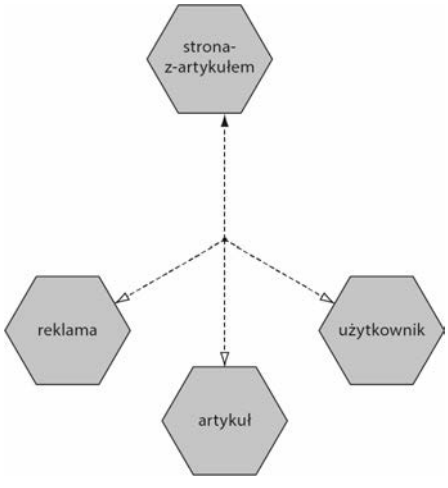
Musisz zdecydować, jak przekierowywać komunikaty. Z kolejką Twoje usługi przynajmniej nie muszą znać lokalizacji w sieci innych usług. Nadal jednak muszą wiedzieć, jak znaleźć kolejkę. Musisz używać tzw. tematów komunikatów, aby odpowiednio przekierowywać przychodzące komunikaty, i Twoje usługi także muszą o nich wiedzieć. Zrozum lepiej to podejście, przyglądając się pewnej strategii: **Rozpraszanie-Zbieranie**.

Większość rodzajów treści jest przydatna nawet wtedy, gdy nie są kompletne lub całkowicie poprawne. W przykładzie z internetową gazetą pokazywanie nieaktualnej wersji artykułu pobranej z pamięci podręcznej jest zdecydowanie lepsze z perspektywy biznesowej niż wyświetlanie błędu strony, jeśli usługa *artykuł* akurat działa niepoprawnie. Liderzy większości organizacji wolą utrzymywać swój biznes otwarty, nawet jeśli nie mogą zaoferować pełnej usługi[15]. To jest po prostu sprawa zdroworozsądkowego biznesu. Firmy chcą być dostępne dla swoich klientów, nawet jeśli ich produkty nie są spójne. Co więcej, klienci również mają takie preferencje — kanapka z szynką i serem z serwisu hotelowego, kiedy przyleciałeś o 2 nad ranem, jest lepsza niż brak jedzenia!

Rozważmy asynchroniczne podejście do budowania strony artykułu. Taka strona składa się z wielu elementów: statusu użytkownika, reklam, tekstu i metadanych artykułu, miniprofilu autora, linków z powiązanymi treściami itd. Strona jest nadal przydatna, nawet jeśli większość tych elementów się na niej nie pojawia. Wskazuje to na *rozpraszanie* komunikatów w mikrousługach odpowiedzialnych za generowanie zawartości, a następnie asynchroniczne *zbieranie* odpowiedzi przed upływem wyznaczonego czasu. Każda usługa dostaje, powiedzmy, 200 ms na odpowiedź. Jeśli nie wyrobi się w tym czasie, jej element nie jest wyświetlany, ale przynajmniej użytkownik coś otrzymuje. Ta technika ma także tę zaletę, że użytkownik ma odczucie, że Twoja strona działa znacznie szybciej, ponieważ dostarczanie stron nie jest spowalniane przez wolne usługi.

[15] Czy banki odmawiają przetwarzania płatności, gdy nie mogą dokonywać transakcji typu ACID? Czy kiedykolwiek przekroczyłeś swój dopuszczalny debet? Banki rozwiązują ten problem poprzez reguły biznesowe (opłaty karne), a nie reguły informatyczne, które mogłyby zaszkodzić ich działalności.

Na rysunku 2.5 usługa *strona-z-artykułem* wysyła komunikat asynchroniczny. Usługi *artykuł*, *reklama* i *użytkownik* monitorują, ale nie konsumują tego komunikatu. Robią swoje i generują odpowiedzi. Ich odpowiedzi są również asynchroniczne. Usługa *strona-z-artykułem* konsumuje te odpowiedzi, co jest wskazane przez wypełnioną strzałkę. Strzałka ta jest przesunięta względem środka sześciokąta reprezentującego usługę *strona-z-artykułem*, aby wskazać, że usługa ta jest źródłem tego przepływu komunikatu. Ten wzór jest powszechny; schemat „streszcza" rozpraszanie i zbieranie jedną przerywaną linią. Pamiętaj, że te elementy nie są indywidualnymi przypadkami usług, ale raczej reprezentują wiele instancji[16].




**Rysunek 2.5.** Wzorzec Rozpraszanie-Zbieranie

Kolejka komunikatów powoduje, że łatwiej jest zaimplementować strategię Rozpraszanie-Zbieranie i w ogóle znacznie bardziej pasuje ona do asynchronicznych wzorców. W praktyce tworzysz temat *żądania* dla usługi *strona-z-artykułem*, gdzie wstawiane są *komunikaty-żądania*, a także temat *odpowiedzi* dla usług dostarczających treść, by mogły tam wstawiać *komunikaty-odpowiedzi*. Musisz również zidentyfikować i odpowiednio oznaczyć komunikaty, aby mogły zostać zignorowane przez usługi, które nie są nimi zainteresowane. Ale ostrożnie, jest wiele różnorodnych trybów awarii kolejek komunikatów. W rozdziale 3. bardziej szczegółowo zbadamy wzorce komunikatów i ich tryby awarii.

Czym kierują się ludzie przy wyborze w systemie produkcyjnym między strategią synchroniczną a asynchroniczną? Prawie wszystkie systemy produkcyjne są hybrydami. Asynchroniczne kolejki komunikatów mają większą tolerancję na błędy i umożliwiają łatwiejszą dystrybucję zadań. Dodawanie nowych usług jest łatwe i nie musisz się zbytnio martwić mechanizmem wykrywania usług. Z drugiej strony synchroniczna komunikacja typu punkt-punkt to absolutna konieczność, kiedy zależy Ci na niewielkim opóźnieniu. Również w początkowym stadium projektu znacznie szybciej można rozpocząć, używając komunikacji punkt-punkt.

[16] Netflix, główny zwolennik mikrousług, zwykle jest wdrażany w jednostkach grupy Amazon Web Services Auto Scaling. Nie ma tu mowy o myśleniu w kategoriach pojedynczych maszyn lub kontenerów.

## 2.6. Projekty monolityczne a projekty mikrousługowe

Monolityczna architektura oprogramowania powoduje negatywne konsekwencje, które w opinii wielu są podstawowymi wyzwaniami dotyczącymi całego rozwoju oprogramowania. Po bliższym, krytycznym zbadaniu to założenie można postawić na głowie. Wiele wyzwań i wiele rzekomych rozwiązań tych wyzwań wynika bezpośrednio z inżynierskich wpływów architektury monolitycznej i staje się dyskusyjnych przy innym inżynierskim podejściu.

Istnieją trzy konsekwencje monolitów. Pierwsza jest taka, że wszyscy członkowie zespołu programistów muszą starannie koordynować swoje działania w taki sposób, aby nawzajem się nie blokować. Jest jeden kod bazowy, więc jeśli pojedynczy programista zepsuje kompilację, to wszyscy deweloperzy zostaną zablokowani. Kod naturalnie zmierza w kierunku głębokiej, wielowymiarowej zależności. Jest to konsekwencja postrzegania najlepszych praktyk. Refaktoryzacja wspólnego kodu do bibliotek współdzielonych tworzy głębokie drzewa zależności. Wymagana liczba poprawek, kiedy zmieniana jest struktura kodu, jest wykładniczo proporcjonalna do głębokości tej struktury w drzewie zależności. Zespoły często podejmują racjonalny wybór, by zapakować złożoność w coraz większą liczbę warstw, próbując ją ukryć i opanować. Monolity znacznie zwiększają koszty pracy równoległej i tym samym spowalniają rozwój oprogramowania.

Drugą konsekwencją monolitów jest szybkie uzyskanie długu technicznego. Nie ma naturalnej siły ograniczającej. Dobrze skonstruowany, prawidłowo rozłączny, czysty, zorientowany obiektowo początkowy projekt jest zbyt słaby, aby oprzeć się natychmiastowym potrzebom całego zespołu pracującego pod presją czasu, który to zespół chce jak najszybciej dostarczyć bieżące funkcjonalności. Jest zbyt wiele sposobów, by jeden fragment kodu mógł wpłynąć na inny fragment — zbyt wiele sposobów na tworzenie zależności.

Pierwszym przykładem jest korozja struktury danych. Biorąc pod uwagę wstępną definicję wymagań, starsi programiści i architekci projektują odpowiednie struktury danych, by opisać istotę problemu. Są to wspólne struktury danych i muszą być dostosowane do wszystkich znanych wymagań oraz przewidywać przyszłe wymagania, więc mają tendencję do większej złożoności. Głębokie zagnieżdżenie odniesień między strukturami jest znakiem ostrzegawczym przed próbą bycia ponadczasowym. Niestety świat często potrafi przechytrzyć naszą skromną inteligencję, a struktury danych nie mogą sprostać rzeczywistym potrzebom biznesowym w miarę ich pojawiania się. Zespół jest zmuszony do wprowadzania prowizorki, niejawnych konwencji i tworzenia rozszerzeń ad hoc[17].

Później nowi członkowie zespołu oraz młodsi programiści, nie rozumiejąc sił rządzących strukturami danych, mogą wprowadzać subtelne i podstępne błędy, które powodują, że cały zespół musi poświęcić nieproporcjonalnie dużo czasu na poprawki

[17] Deklaratywne struktury są zwykle najlepszą opcją do odzwierciedlania świata, ponieważ można nimi manipulować w powtarzalny, spójny, umyślnie ograniczony sposób. Jeśli wprowadzisz sposoby osadzania kodu wykonywalnego do obsługi specjalnych przypadków takich jak „darmowa karta wyjścia z więzienia", sprawy mogą się szybko skomplikować i skończysz w więzieniu dłużników technicznych.

i rozwiązania poprawiające wydajność[18]. Ostatecznie zespół musi podjąć szeroko zakrojone działania związane z refaktoryzacją, aby odzyskać pewną miarę prędkości programowania. Globalnie udostępnione struktury danych i modele są tak samo złe jak zmienne globalne i nie mają naturalnej obrony przed technicznym długiem.

Trzecią konsekwencją monolitów jest to, że istnieją tylko wdrożenia typu „wszystko albo nic". Masz starą wersję monolitu uruchomionego w produkcji oraz nową wersję gotową do wdrożenia. Jeżeli chcesz zaktualizować system produkcyjny bez wpływu na działalność biznesową, czeka Cię bardzo stresujący weekend.

Być może jesteś bardziej wyrafinowany i stosujesz wdrożenia metodą *blue-green*, aby zmniejszyć ryzyko[19]. Nadal musisz zużywać energię na budowanie infrastruktury *blue-green* i wciąż niewiele Ci to pomoże, jeśli musisz przeprowadzić migracje schematu bazy danych, ponieważ niełatwo jest przywrócić poprzednią wersję bazy.

Podstawowym problemem jest to, że każda zmiana w systemie produkcyjnym wymaga przeniesienia całego kodu bazowego. Istnieje duże ryzyko niepowodzenia wdrożenia na wszystkich poziomach systemu. Brak wystarczającej ilości testów jednostkowych, akceptacyjnych, integracyjnych, instrukcji testowania oraz możliwości testowania wersji próbnych w systemie produkcyjnym pozwala się zorientować, jak duże jest prawdopodobieństwo niepowodzenia wdrożenia. Przyczyny awarii są często bezpośrednimi konsekwencjami warunków produkcyjnych, których nie da się zasymulować. Wystarczy, że dane produkcyjne (do których możesz nawet nie mieć dostępu z powodu zasad zachowania poufności informacji) posiadają tylko jeden nieprzewidziany aspekt, a już będzie to powodować krytyczne awarie. Trudno jest zweryfikować wydajność przy użyciu danych testowych — dane produkcyjne mogą być nawet kilka rządów wielkości większe. Użytkownicy mogą zachowywać się w nieprzewidywalny sposób, szczególnie podczas korzystania z nowych funkcjonalności. Może to doprowadzić do awarii systemu, ponieważ zespół nie był w stanie sobie wyobrazić takich przypadków użycia. Ryzyko związane z wdrożeniami w systemach monolitycznych powoduje powolne, rzadkie publikowanie nowych funkcjonalności, co powstrzymuje przed szybkim wprowadzaniem zmian.

Wyzwania inżynieryjne, bo takimi właśnie są, nie mogą zostać rozwiązane przez dowolne podejście do zarządzania projektami. Dotyczą innego zakresu problemów. A jednak niemal powszechnie i wyłącznie firmy próbują je rozwiązać metodami rozwoju oprogramowania i technikami zarządzania projektem. Zamiast cofnąć się o krok i szukać prawdziwego powodu, dla którego projekty są realizowane z opóźnieniem i prze-

[18] Pouczający przykład od byłego klienta: klient dodał kolumnę w bazie danych dla treści XML, aby móc przechowywać niewielkie ilości nieustrukturyzowanych danych. Schemat dla tego XML-a zawierał szereg elementów, które można było powtórzyć do przechowywania list. Te listy były nieograniczone. Źródłem problemu było to, że niewielka liczba użytkowników generowała bardzo długie listy, co prowadziło do przechowywania dokumentów XML o ogromnej zawartości. To z kolei wywoływało dziwne i niesamowite błędy w procesie odśmiecania pamięci, których przez długi czas nie można było powiązać z główną przyczyną.

[19] Strategia *blue-green* oznacza, że przez cały czas są produkowane dwie wersje systemu: niebieska i zielona. Tylko jedna z nich jest aktualnie działającym systemem. Aby wdrożyć nową wersję, uaktualnij ją na partycji, która jest offline, a następnie uruchom ją, wyłączając aktualnie działającą.

kraczają budżet, osoby odpowiedzialne za rozwój oprogramowania raczej obwiniają się za złe wykonanie. A przecież żadne dobre wykonanie nie pozwoli Ci wybudować drapaczy chmur, jeśli założenia są bzdurne[20]. Rozwiązanie leży gdzieś indziej.

Architektura mikrousług jako podejście inżynierskie pozwala nam, twórcom oprogramowania, ponownie zapoznać się z naszymi najlepszymi praktykami i zapytać, czy naprawdę mogą one sprawić, by dostarczanie nowych funkcjonalności było szybsze i bardziej przewidywalne. Czy może są one jedynie kiepskim sposobem na złagodzenie fundamentalnych problemów rozwoju monolitycznego? Frederick P. Brooks w swojej wydanej w 1975 r. książce *Mityczny osobomiesiąc. Eseje o inżynierii oprogramowania* szczegółowo wyjaśnia wyzwania związane z rozwojem projektów monolitycznych[21]. Następnie sugeruje zestaw technik i praktyk nie po to, by rozwiązać problem, ale żeby go ograniczyć i złagodzić. Zasadnicze przesłanie jest takie: brak jakiegokolwiek panaceum — żadne techniki zarządzania projektami nie mogą zapobiec inżynierskim deficytom architektury monolitycznej.

### 2.6.1. Jak mikrousługi zmieniają zarządzanie projektem

Inżynierskie własności architektury mikrousług mają bezpośredni wpływ na wysiłek związany z takim zarządzaniem projektem, by zapewnić jego pomyślną realizację. Jest mniejsze zapotrzebowanie na szczegółowe zarządzanie zadaniami i na wiele bezużytecznych ceremonii o charakterze czysto metodycznym[22]. Zarządzanie projektami opartymi na mikrousługach jest dużo lżejsze. Przeanalizujmy teraz implikacje tego podejścia.

### 2.6.2. Jednolitość ułatwia estymację

Mikrousługi są małe i dobrą praktyką jest ograniczanie ich co najwyżej do jednej iteracji pracy jednego dewelopera. Szacowanie z punktu widzenia mikrousług jest więc znacznie łatwiejsze niż ogólne szacowanie nakładu pracy na tworzenie oprogramowania, ponieważ musisz wtedy podzielić pracę na kawałki wielkości jednej iteracji. To ważna obserwacja. Tradycyjne systemy monolityczne składają się z heterogenicznych komponentów o różnych rozmiarach i różnej złożoności. Dokładne oszacowanie jest niezwykle trudne, ponieważ każdy element jest szczególnym przypadkiem i ma wieloaspektowy zestaw interakcji z innymi komponentami poprzez wywołania metod,

[20] Od czasów neolitu używano konstrukcji wikliniarsko-glinianych — jest to doskonała technika budowlana, dzięki czemu można zbudować do trzech lub czterech małych pięter. Ale to nie pomoże Ci zbudować Empire State Building.

[21] Brooks był kierownikiem projektu systemu mainframe IBM System/360 i jako pierwszy zapisał obserwację, że dodawanie kolejnych programistów do projektu, który już jest opóźniony, sprawia, że staje się on jeszcze bardziej opóźniony.

[22] Aby oszczędzić sobie zażenowania, żadna metodyka nie zostanie tutaj nazwana. Ale wiesz, kim jesteś. Jeśli istniałoby takie podejście do zarządzania projektem do rozwoju oprogramowania, które pozwalałoby na konsekwentne dostarczanie nowych funkcji w wielu różnych rodzajach zespołów, to już kierowalibyśmy się ku rozwiązaniu. Jednak w tej kwestii nie widzimy oznak znaczących postępów.

współdzielone obiekty, struktury danych oraz schematy bazy danych. Rezultatem jest projektowa lista zadań, która dostosowuje się do wymagań architektury systemu[23].

W przypadku mikrousług limit złożoności jednej iteracji wymusza jednorodność komponentów, co zwiększa dokładność szacowania. W praktyce jeszcze większa dokładność może zostać osiągnięta przez klasyfikowanie mikrousług na, powiedzmy, trzy poziomy złożoności oraz przez klasyfikowanie programistów na trzy poziomy doświadczenia i dopasowanie mikrousług do programistów. Na przykład mikrousługa poziomu 1. może zostać ukończona przez programistę poziomu 1. w jednej iteracji, podczas gdy mikrousługa na poziomie 3. wymaga programisty poziomu 3., by ukończyć ją w takim samym czasie. Takie podejście daje znacznie większą dokładność niż ogólne szacowanie przy użyciu *punktów historyjek użytkownika*, które nie uwzględniają różnic w możliwościach programistów. Projekt oparty na mikrousługach może być dokładnie zaplanowany za pomocą sensownego, znaczącego mapowania mikrousług na iteracje. Omówimy ten pomysł bardziej szczegółowo w części II tej książki.

**Dlaczego ocena oprogramowania jest trudna?**

Dlaczego tak trudno jest oszacować złożoność komponentów większego systemu? Ścisła zależność, która niezmiennie występuje w architekturach monolitycznych, oznacza, że programowanie w późniejszych etapach projektu jest wykładniczo wolniejsze, ponieważ wstępne szacunki komponentów na późnym etapie są nadmiernie optymistyczne. Ta wykładnicza powolność wynika z faktu matematycznego (znanego też jako prawo Metcalfa), że liczba możliwych połączeń między węzłami w sieci wzrasta proporcjonalnie do kwadratu liczby tych węzłów.

I jest jeszcze jeden czynnik: ludzka psychika cierpi na wiele uprzedzeń poznawczych, np. nie jesteśmy dobrzy w pracy z prawdopodobieństwami. Wiele z tych uprzedzeń może nie pozwalać na dokładne oszacowanie projektu. Oto przykład: *zakotwiczenie* jest tendencją polegającą na skupianiu się na pierwszej wartości, którą usłyszysz. Ale złożoność komponentów oprogramowania i tym samym czas zakończenia pracy nad nimi podlega tzw. „prawu mocy": większość zajmuje krótki czas, ale na część potrzeba dużo więcej czasu[24]. I zakotwiczenie na wstępnych szacunkach wpływa negatywnie na kolejne, gdy złożoność systemu rośnie.

Największe i najtrudniejsze komponenty są niedoszacowane, ponieważ większość pracy szacunkowej dotyczy małych komponentów. Stary dowcip o tym, że ostatnie 10% harmonogramu zajmuje 90% czasu, ma w sobie sporo prawdy.

### 2.6.3. Jednorazowy kod tworzy bardziej przyjazne zespoły

Kod mikrousługi jest jednorazowy. Dosłownie można go wyrzucić. Dowolna mikrousługa jest warta jednej iteracji pracy jednego dewelopera. Gdyby mikrousługa była źle napisana, nie spełniałaby wymagań w wybranym języku lub stałaby się niepotrzebna ze względu na zmienione wymagania, można by ją wycofać bez głębszego za-

[23] Trochę ironicznie — oszacowanie Fibonacciego (gdzie tzw. *punkty historyjek użytkownika* projektu *agile* muszą być liczbami Fibonacciego: 1, 2, 3, 5, 8, 13, …) jest wystarczającym dowodem na to, że osiągnięto lokalne maksimum w dokładności szacowania systemów monolitycznych.

[24] „Prawa mocy" opisują wiele zjawisk, w których małe przyczyny mogą wywoływać ponadprzeciętne skutki, np. czas trwania trzęsienia ziemi, wynagrodzenia kadry kierowniczej i częstotliwość występowania liter w tekście.

stanowienia. To miałoby zdrowy wpływ na dynamikę zespołu: nikt nie byłby emocjonalnie związany ze swoim kodem ani nie robiłby się z tego powodu zaborczy.

Przypuśćmy, że Alicja myśli, iż mikrousługa A, napisana przez Roberta kilka iteracji wcześniej w języku Java, będzie dwa razy wydajniejsza w C++. Powinna się podjąć wykonania tego zadania! To, tak czy inaczej, tylko inwestycja w dodatkową iterację. Jeśli próba zakończy się porażką, zespół nie zostanie postawiony w gorszej sytuacji, ponieważ wciąż ma kod Java Roberta.

Świadomość, że każda mikrousługa może żyć lub musi umrzeć w zależności od tego, czy jest przydatna, jest naturalną metodą ograniczającą złożoność. Złożoność czyni Cię słabym. Lepiej napisać nową mikrousługę specjalnego przypadku, niż rozszerzać istniejącą. Jeśli zarezerwujesz, powiedzmy, 20% iteracji na przepisanie i obsługę nieprzewidzianych, specjalnych przypadków, możesz mieć większą pewność, że jest to tylko możliwość, a nie ogólna polityka negocjacyjna[25].

### 2.6.4. Homogeniczne komponenty pozwalają na heterogeniczną konfigurację

Możesz grupować mikrousługi w różne klasy z różnymi ograniczeniami biznesowymi. Niektóre z nich mają krytyczne znaczenie oraz wysokie obciążenie — np. mikrousługa w kasie witryny e-commerce. Inne to podstawowe funkcjonalności, ale nie tak krytyczne. Jeszcze inne miło byłoby mieć, ale ich awaria nie jest tak istotna. Warto zadać sobie pytania: Czy jest konieczne, by wszystkie usługi różnego rodzaju miały ten sam poziom jakości? Czy wszystkie potrzebują takiego samego poziomu pokrycia testami jednostkowymi? Czy jest sens wkładać taki sam wysiłek w kontrolę jakości po równo dla wszystkich mikrousług? Nie. Nie ma uzasadnionego przypadku biznesowego dla takiego poglądu.

Powinieneś wydatkować swój wysiłek tam, gdzie to się liczy. Powinieneś mieć różne poziomy pokrycia testami jednostkowymi dla różnych klas mikrousług. Podobnie jak są różne wymagania dotyczące wydajności, bezpieczeństwa danych, bezpieczeństwa i skalowania. Kod bazowy systemu monolitycznego musi spełniać najwyższe standardy we wszystkich dziedzinach; to podstawa inżynierska. Mikrousługi pozwalają lepiej skoncentrować się na szczegółach, jeśli chodzi o ograniczanie zasobów programistów, gdy ma to znaczenie[26].

Mikrousługi sprawiają, że udane awarie są naprawdę udane. Typowy system oprogramowania musi często przechodzić testy akceptacyjne dla użytkownika. W praktyce oznacza to, że lista z testami nie zostanie zaakceptowana, dopóki wszystkie testy na wszystkie funkcjonalności nie zostaną pomyślnie wykonane. Odsuń się na chwilę od

[25] Szacowanie projektu oprogramowania często przekształca się w grę polityczną. Twórcy oprogramowania dają optymistyczne szacunki, by zdobyć dodatkowe punkty. Interesariusze biznesowi, już doświadczeni przez nieudane projekty, zdecydowanie wymagają harmonogramu na wszystkie funkcjonalności. Ostateczny harmonogram jest określany w wyniku ciężkiego targowania się, a nie poprzez inżynierię. Obie strony mają swoje uzasadnione potrzeby, ale kończy się sytuacją, w której wszyscy tracą, ponieważ przekazywanie tych potrzeb nie jest politycznie bezpieczne.

[26] Sposób kwantyfikacji tych szczegółowych pomiarów omówiono w rozdziale 6.

tego i zadaj sobie pytanie, czy jest to dobry sposób na zapewnienie zgodności dostarczonego oprogramowania z pierwotnie postawionymi celami biznesowymi. Jak ktokolwiek może być pewny, że dana funkcjonalność dostarcza rzeczywistą wartość, dopóki nie zostanie to potwierdzone w systemie produkcyjnym? Być może pewne funkcje nigdy nie będą używane lub okażą się zbyt skomplikowane. Być może brakuje Ci krytycznej funkcji, o której nikt wcześniej nie pomyślał. Testy akceptacyjne użytkownika traktują wszystkie funkcjonalności jako mające taką samą wartość. W praktyce okazuje się, że zespół dostarcza w większości kompletny system z najczęściej losowym podzbiorem pierwotnie pożądanych funkcji. Po wielu narzekaniach zostaje to zaakceptowane, ponieważ firma potrzebuje systemu, aby przejść do produkcji.

Podejście oparte na mikrousługach nie zmienia rzeczywistości, w której to zasoby programistów są ograniczone, i ostatecznie może nie być wystarczająco dużo czasu, aby wszystko zbudować. Podejście to jest takie jak przy przeszukiwaniu wszerz. Większość projektów przyjmuje natomiast podejście jak przy przeszukiwaniu w głąb: historyjki użytkownika są przydzielone do iteracji, a zespół „wypala" wymagania na wykresie spalania sprintu. Na końcu według oryginalnego harmonogramu masz, powiedzmy, 80% ukończonych funkcji i 20% nietkniętych. W podejściu „wszerz" dostarczasz niekompletne wersje wszystkich funkcji. Pod koniec projektu masz 100% funkcji, które są w większości kompletne, ale znacząca liczba przypadków szczególnych nie została zakończona. Które z tych podejść jest lepsze do zaprezentowania w żyjącym systemie produkcyjnym? Dzięki podejściu „wszerz" zajmujesz się wszystkimi przypadkami biznesowymi na pewnym poziomie. Nie marnujesz sił, by w pełni zrealizować funkcje, które okazują się nie mieć żadnej wartości. I dajesz biznesowi okazję, by jeszcze podczas trwania projektu przekierować wysiłek bez rezygnacji z całych funkcjonalności — to znacznie ułatwi Ci dyskusję z interesariuszami. Mikrousługi umożliwiają również alokację zasobów zespołu deweloperskiego po skończeniu projektu w sposób bardziej wydajny i przyjazny.

### *2.6.5. Istnieją różne rodzaje kodów*

Mikrousługi umożliwiają oddzielenie kodu logiki biznesowej od kodu infrastruktury. Kod logiki biznesowej wynika bezpośrednio z wymagań biznesowych. Jest to określone przez optymistyczne przypuszczenia interesariuszy, biorących pod uwagę niepełną i nieadekwatną informację biznesową. W naturalny sposób podlega ona szybkim zmianom, ma w sobie ukrytą złożoność i może stać się nieaktualna. Wtłoczenie logiki biznesowej do jednostek mikrousług jest praktycznym, inżynierskim podejściem do zarządzania szybkimi zmianami.

W systemie jest jeszcze inny rodzaj kodu: kod infrastruktury. To właśnie tutaj jest miejsce na integrację, algorytmy, manipulowanie strukturami danych, analizę składniową oraz kod narzędziowy. Ten kod jest mniej zależny od kaprysów świata biznesu. Często jest to stosunkowo kompletna specyfikacja techniczna, interfejs API lub są to specyficzne wymagania. Kod ten można bezpiecznie przechowywać oddzielnie od kodu logiki biznesowej, więc nie spowalnia to kodu biznesowego ani negatywne, incydentalne przypadki logiki biznesowej nie mają na niego wpływu.

Problem z większością architektur monolitycznych polega na tym, że te dwa typy kodu — logiki biznesowej i infrastruktury — w końcu zostają zmieszane ze sobą, co może mieć negatywny wpływ na wydajność zespołu i poziom zadłużenia technicznego. Logika biznesowa należy do mikrousług, infrastruktura należy do bibliotek oprogramowania. Umiejętność prawidłowego rozdzielania wysiłku programistycznego zgodnie z tym podziałem sprawia, że dalsze szacowanie jest bardziej dokładne i zwiększa przewidywalność harmonogramu projektu.

## 2.7. Jednostka oprogramowania

Powyższa dyskusja pokazuje, że mikrousługi są niezwykle użyteczne jako jednostki strukturalne programowania. Czy można je uznać za fundamentalne jednostki, tak jak obiekty, funkcje lub procesy? Tak, ponieważ dają nam potężny model myślenia koncepcyjnego o projekcie systemu.

Istotą problemu, który próbujemy rozwiązać, są różne rodzaje skalowania wielowymiarowego: skalowanie systemów oprogramowania w produkcji, skalowanie złożoności oprogramowania, które tworzy te systemy, i skalowanie zespołów programistów, którzy je budują. Potęga koncepcji mikrousług wynika z tego, że oferuje ona zunifikowane rozwiązanie dla wielu różnych problemów związanych ze skalowaniem.

Problemy ze skalowaniem są trudne, ponieważ mają charakter wykładniczy. Nie ma trzyipółmetrowych ludzi, jako że podwojenie wysokości oznacza ośmiokrotne zwiększenie objętości ciała, a materiały, z jakich zbudowane są nasze ciała i ich architektura, nie mogą poradzić sobie ze zwiększoną wagą[27]. Problemy ze skalowaniem mają tę charakterystykę. Zwiększanie liniowo jednego parametru wejściowego powoduje nieproporcjonalnie przyspieszoną zmianę w innych aspektach systemu.

Jeśli podwoisz rozmiar zespołu oprogramowania, nie podwoisz prędkości wyjściowej. W zespołach większych niż kilka osób dodawanie kolejnych osób może nawet spowolnić postępy[28]. Podwój złożoność systemu oprogramowania, a nie podwoisz liczby błędów; zwiększysz ją o kwadrat rozmiaru kodu bazowego. Podwój liczbę klientów do obsłużenia i — zanim się obejrzysz — będziesz musiał zarządzać systemem rozproszonym.

Skalowaniem można zająć się w dwóch zasadniczych wymiarach[29]: pionowym i poziomym. **Skalowanie pionowe** oznacza tworzenie tego, co jest większe, silniejsze lub szybsze. To działa do momentu, aż fizyczne, matematyczne lub funkcjonalne aspekty systemu dosięgną swoich ograniczeń strukturalnych. Tak więc nie możesz wciąż kupować coraz mocniejszych maszyn. Skalowanie pionowe ma tendencję do wykładniczego spadku skuteczności i wykładniczego wzrostu kosztów, co daje w praktyce duże ograniczenia. Mimo to nie bój się skalować w pionie, kiedy możesz sobie na to pozwolić — sprzęt jest znacznie tańszy niż deweloperzy.

---

[27] Musisz również podwoić szerokość i głębokość, aby zachować proporcje; stąd $2^3$.

[28] Amazon stworzył naukową zasadę dotyczącą wielkości zespołu programistycznego: musi być możliwe nakarmienie całego zespołu nie więcej niż dwiema pizzami.

[29] Możesz dodawać wymiary i uzyskiwać sześciany albo hipersześciany skali. Dzięki temu oczywiście udoskonalisz swoją analizę, ale dwa wymiary będą w sam raz do podejmowania decyzji.

**Skalowanie poziome** wymyka się twardym ograniczeniom. Zamiast zwiększać moc każdego elementu, po prostu dodawaj kolejne. Nie ma żadnych podstawowych ograniczeń tak długo, jak długo Twój system jest przeznaczony do skalowalności liniowej. Większość systemów nie jest, ponieważ ma nieodłączne ograniczenia komunikacyjne, które wymagają zbyt wielu elementów do komunikowania się ze zbyt wieloma innymi elementami.

Systemy biologiczne obejmujące miliardy pojedynczych komórek pokonały ograniczenia skalowania poziomego poprzez komunikację tak bardzo lokalną, jak to tylko możliwe. Komórki komunikują się ze swoimi bliskimi sąsiadami i robią to asynchronicznie za pomocą dopasowywania wzorców o nieukierunkowanych sygnałach hormonalnych. Powinniśmy wyciągnąć lekcję z tej architektury!

Skalowanie o dużej pojemności powstaje, gdy system składa się z dużej liczby niezależnych, homogenicznych jednostek. Brzmi znajomo? Podstawowe cechy mikrousług nadają się do skutecznego skalowania — nie tylko pod względem obciążenia, ale także pod względem złożoności.

## 2.8. Wymagania dotyczące komunikatów do usług

Wróćmy na ziemię. Jak zastosować te pomysły w praktyce? Weźmy pod uwagę system gazety internetowej i przeprowadźmy dalszą analizę. Musisz wiedzieć, jakie usługi zbudować — jak się tam dostać?

Próba odgadnięcia odpowiednich usług nie jest szczególnie skuteczna, aczkolwiek Twoja intuicja dotycząca tego, co czyni usługę dobrą, będzie się poprawiać z czasem. Bardziej przydatne jest rozpoczęcie od strony komunikatów. W szczególności podziel każde wymaganie na zestaw komunikatów, które opisują działania spełniające dane wymaganie. Następnie przyporządkuj komunikaty do usług, dbając o utrzymanie usług niewielkich rozmiarów. Bardziej złożone usługi mogą obsłużyć więcej komunikatów, ale należy je przypisać bardziej doświadczonym członkom zespołu. Nie jest konieczna pełna implementacja wszystkich komunikatów natychmiast, powinieneś dążyć raczej do osiągnięcia podejścia „wszerz" niż podejścia „w głąb" i dostarczyć co najmniej podstawową implementację w przeciągu pierwszych kilku iteracji.

Zróbmy tak dla strony z gazetami. Tabela 2.1 zawiera listę wymagań z odpowiadającymi im komunikatami. To jest pierwsze podejście i możesz zmienić ten zestaw w trakcie trwania projektu. Różni się to od tradycyjnego podejścia, gdzie można by pomyśleć o tym, jakie podmioty tworzą system. Zamiast tego myśl w kategoriach działań — odpowiedz na pytanie „Co się stanie?". Zauważysz, że ta analiza udoskonala wcześniejsze eksperymenty z usługą *artykuł* poprzez eksplorację możliwych wariantów interakcji między usługami. Jest to celowe, abyś mógł dostrzec elastyczność, którą zapewnia to podejście. Zmodyfikujesz architekturę ponownie, zanim skończysz.

**Tabela 2.1. Mapowanie wymagań na komunikaty**

| Wymaganie | Komunikaty |
|---|---|
| Strony artykułu | *buduj-artykuł*, *pobierz-artykuł*, *artykuł-wyświetlany* |
| Strony list artykułów | *buduj-listę-artykułów*, *podaj-listę-artykułów* |
| API REST | *pobierz-artykuł*, *dodaj-artykuł*, *usuń-artykuł*, *podaj-listę-artykułów* |
| Zawartość statyczna i dynamiczna | *potrzebujesz-danych-artykułu*, *podaj-dane-artykułu* |
| Zarządzanie użytkownikiem | *loguj*, *wyloguj*, *rejestruj*, *pobierz-profil* |
| Kierowanie zawartości do odbiorcy | *potrzebujesz-danych-użytkownika*, *podaj-dane-użytkownika* |
| Specjalne miniaplikacje | Zależne od API |

Niektóre aktywności będą współdzielić komunikaty. Tego można było się spodziewać. W dużych systemach stworzyłbyś przestrzeń nazw dla komunikatów, ale dla naszych celów nie jest to konieczne. Powinieneś również jasno określić cel swoich komunikatów, opisując czynności, które mają reprezentować:

- *buduj-artykuł* — konstruuje stronę HTML artykułu;
- *pobierz-artykuł* — pobiera dane artykułu;
- *artykuł-wyświetlany* — zapowiada oglądanie artykułu;
- *buduj-listę-artykułów* — konstruuje stronę z listą artykułów;
- *podaj-listę-artykułów* — pyta magazyn artykułów;
- *dodaj-artykuł* — dodaje artykuł do magazynu artykułów;
- *usuń-artykuł* — usuwa artykuł z magazynu artykułów;
- *potrzebujesz-danych-artykułu* — wyraża potrzebę pobrania zawartości strony z artykułem;
- *podaj-dane-artykułu* — pobiera pewne elementy zawartości strony z artykułem;
- *loguj* — loguje użytkownika;
- *wyloguj* — wylogowuje użytkownika;
- *rejestruj* — rejestruje nowego użytkownika;
- *pobierz-profil* — pobiera profil użytkownika;
- *potrzebujesz-danych-użytkownika* — wyraża potrzebę pobrania treści pod kątem osoby odwiedzającej witrynę;
- *podaj-dane-użytkownika* — zbiera niektóre treści ukierunkowane na użytkownika.

Komunikaty te można następnie zorganizować w usługi. Dla każdej usługi musisz zdefiniować komunikaty przychodzące i wychodzące; patrz tabele 2.2 – 2.9. Musisz także zdecydować, czy komunikat ma być synchroniczny, czy asynchroniczny (asynchroniczny oznaczony jest przez „(A)" w tabelach). Komunikaty synchroniczne oczekują natychmiastowej odpowiedzi — załóżmy, że są one domyślne. Musisz również zdecydować, czy komunikat ma być konsumowany, czy po prostu monitorowany przez usługę. Konsumowane komunikaty nie mogą być już odbierane przez inne usługi — załóżmy, że są domyślne.

**Tabela 2.2. Usługa strona-z-artykułem**

| | |
|---|---|
| Wejściowe | buduj-artykuł, buduj-listę-artykułów, podaj-dane-artykułu (A), podaj-dane-użytkownika (A) |
| Wyjściowe | *pobierz-artykuł*, *potrzebujesz-danych-artykułu (A)*, *potrzebujesz-danych-użytkownika (A)* |
| Notka | Nie przyjmujemy żadnych założeń co do sposobu konstruowania stron HTML. Być może świadczące usługi wysłały HTML, a może tylko metadane. |

**Tabela 2.3. Usługa strona-z-listą-artykułów**

| | |
|---|---|
| Wejściowe | buduj-listę-artykułów, podaj-dane-użytkownika (A) |
| Wyjściowe | *podaj-listę-artykułów*, *potrzebujesz-danych-użytkownika (A)* |

**Tabela 2.4. Usługa artykuł**

| | |
|---|---|
| Wejściowe | pobierz-artykuł, dodaj-artykuł, usuń-artykuł, podaj-listę-artykułów |
| Wyjściowe | *dodaj-element-do-cache (A)*, *pobierz-element-z-cache* |
| Notka | Ta usługa komunikuje się z usługą cache, by zachowywać artykuły. |

**Tabela 2.5. Usługa cache**

| | |
|---|---|
| Wejściowe | pobierz-element-z-cache, dodaj-element-do-cache(A) |
| Wyjściowe | Brak. |
| Notka | Komunikaty cache nie pochodzą z listy wymagań. Zamiast tego korzystamy z naszego doświadczenia architektów oprogramowania — potrzebujemy pamięci cache w systemie, aby zapewnić odpowiednią wydajność. Takie komunikaty powstają naturalnie w wyniku analizy systemu. |

**Tabela 2.6. Usługa gateway-api**

| | |
|---|---|
| Wejściowe | Brak. |
| Wyjściowe | *buduj-artykuł*, *buduj-listę-artykułów*, *pobierz-artykuł*, *podaj-listę-artykułów*, *dodaj-artykuł*, *usuń-artykuł*, *loguj*, *wyloguj*, *rejestruj*, *pobierz-profil* |
| Notka | Komunikaty przychodzące do tej usługi to tradycyjne wywołania HTTP typu REST, a nie komunikaty mikrousług. Ta usługa konwertuje je na wewnętrzne komunikaty mikrousług. |

**Tabela 2.7. Usługa użytkownik**

| | |
|---|---|
| Wejściowe | loguj, wyloguj, rejestruj, pobierz-profil |
| Wyjściowe | *podaj-dane-artykułu* |

**Tabela 2.8. Usługa reklama**

| | |
|---|---|
| Wejściowe | potrzebujesz-danych-artykułu |
| Wyjściowe | *podaj-dane-artykułu* |

**Tabela 2.9. Usługa ukierunkowana-zawartość**

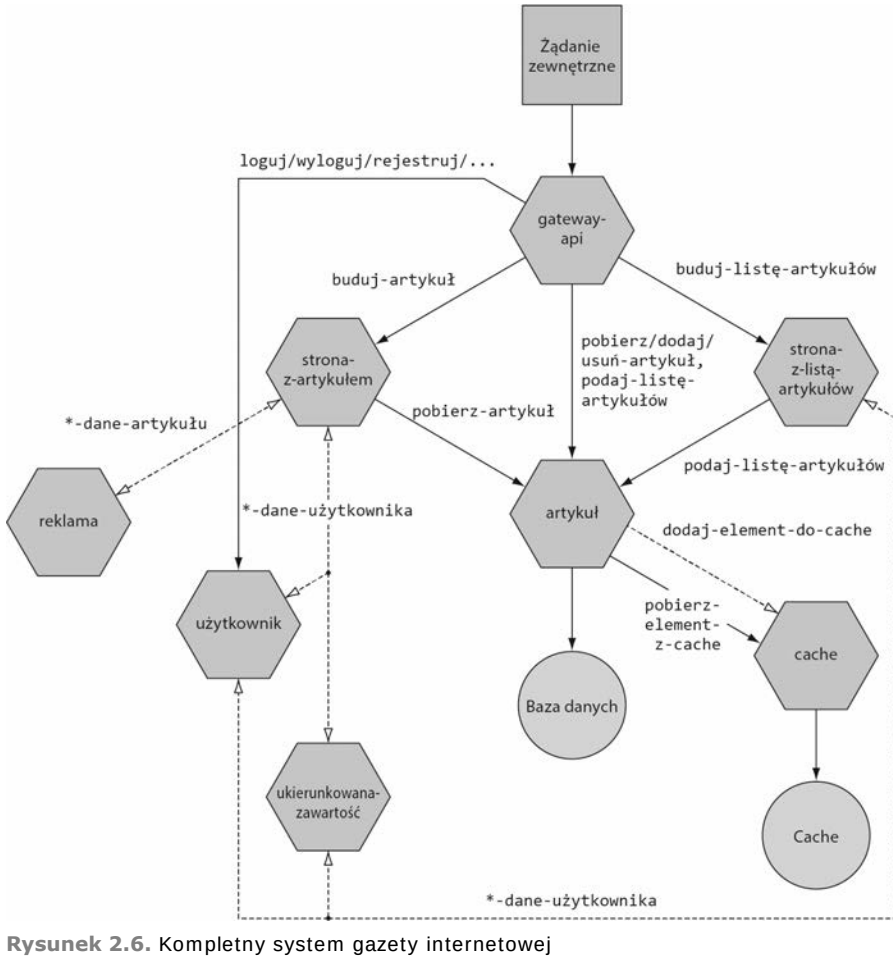
| | |
|---|---|
| Wejściowe | potrzebujesz-danych-użytkownika |
| Wyjściowe | *podaj-dane-użytkownika* |
| Notka | Na razie jest to prosta, początkowa implementacja, która udostępnia wywołanie „Zarejestruj teraz!” dla nieznanych użytkowników i pustą zawartość dla znanych, zalogowanych już użytkowników. Planowane jest rozszerzanie tej funkcjonalności poprzez dodawanie większej liczby usług. |

Lista usług ze wstępnej analizy może być oszacowana pod względem złożoności i dostosowana tak, aby początkowa wersja każdej usługi mogła zostać zbudowana w ramach jednej iteracji. Niektóre funkcjonalności usług są dodawane przyrostowo, więc późniejsze wersje również wymagają jednej iteracji. Uważaj, aby nie robić tego zbyt często, ponieważ takie usługi mogą się rozwinąć w kierunku złożoności i stać się niezbędne, a nie jednorazowe. Kiedy to możliwe, lepiej jest dodawać funkcjonalność, dodając usługę.

Listy wymagań, komunikatów i usług są jednym ze sposobów patrzenia na system. Spójrzmy teraz na architekturę systemu gazety internetowej za pomocą diagramu mikrousług.

## 2.9. Diagramy architektury mikrousług

Wcześniej w tym rozdziale zaprezentowaliśmy mniejsze części systemu. Teraz stwórzmy pełny schemat architektury systemu (patrz rysunek 2.6).




**Rysunek 2.6.** Kompletny system gazety internetowej

**UWAGA** W większości diagramów sieciowych połączenia między elementami przedstawiane są jako linie proste, często bez kierunku. Linie te wskazują ruch w sieci, ale niewiele więcej. Zakłada się, że elementy sieciowe są pojedynczymi instancjami. W systemie mikrousługowym lepiej przyjąć domyślnie jedną lub więcej instancji, ponieważ tak jest powszechnie przyjęte. Używam tej konwencji w diagramach w całej książce, co daje natychmiastowy wgląd w każde studium przypadku.

Kompletny system gazety zawiera i udoskonala podsystem artykułów, który widziałeś wcześniej. Synchroniczne i asynchroniczne przepływy komunikatów mogą być wyraźnie widoczne i odwzorowane z powrotem na komunikaty i specyfikację usług. Używaj tego diagramu jako przykładu odniesienia dla konwencji wizualnych występujących w tej książce.

Na tym diagramie systemu gazety stosuję następujące konwencje dla grupy elementów sieci:

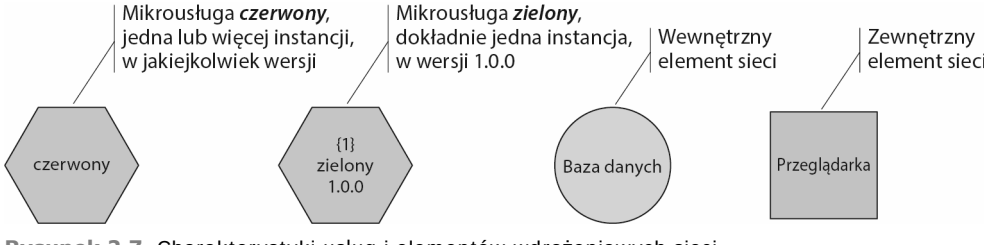
- Sześciokąty reprezentują mikrousługi.
- Koła reprezentują systemy wewnętrzne.
- Prostokąty reprezentują systemy zewnętrzne.

Systemy wewnętrzne to bazy danych, silniki buforujące, usługi katalogowe itd. Stanowią one oddzielną część infrastruktury względem mikrousług. System wewnętrzny również może składać się z własnych mikrousług, a kształt koła można wykorzystać do przedstawiania całych podsystemów złożonych właśnie z mikrousług.

Zakłada się, że cała komunikacja opiera się na komunikatach. Dotyczy to również elementów, które nie są mikrousługami, dzięki czemu można użyć tej samej konwencji do zaznaczania komunikacji z tymi elementami. Przypadki specjalne, takie jak przepływ danych strumieniowych, muszą być opatrzone dodatkowymi przypisami.

Takie diagramy mogą zawierać dalsze informacje, jak pokazano na rysunku 2.7:

- *ciągła linia graniczna* — jedna lub więcej instancji i wersji danego elementu;
- *przerywana linia graniczna* — rodzina powiązanych elementów;
- *nazwa* — wymagane; identyfikuje element lub rodzinę;
- *liczność* — opcjonalne; liczba istniejących instancji; ponad nazwą;
- *znacznik wersji* — opcjonalne; numer wersji tych instancji; poniżej nazwy.




**Rysunek 2.7.** Charakterystyki usług i elementów wdrożeniowych sieci

Ciągła linia graniczna wskazuje liczność jednego lub większej liczby elementów, co jest wartością domyślną. **Liczność** oznacza liczbę działających instancji[30]. Pełna lista wartości przyjmowanych jako liczność jest następująca:

- `?` — zero lub jedno wystąpienie;
- `*` — zero lub więcej instancji;
- `+` — jedno lub więcej wystąpień;
- `{n}` — dokładnie *n* wystąpień;
- `{n:m}` — liczba wystąpień między *n* i *m*;
- `{n:}` — co najmniej *n* wystąpień;
- `{:m}` — nie więcej niż *m* wystąpień.

Liczności numeryczne muszą zawsze znajdować się w nawiasach klamrowych, aby nie sugerować, że są numerami wersji.

Przerywana linia graniczna oznacza, że element składa się z grupy powiązanych usług. W tym przypadku liczność odnosi się do każdego członka rodziny i ta precyzyjna rozdzielczość pozwala na to, byś sam mógł rozdzielić poszczególnych członków.

Znacznik wersji pojawia się pod nazwą i jest opcjonalny. Jest zgodny ze standardem semantycznym[31] z wyjątkiem tego, że możesz pominąć wszystkie wewnętrzne numery, i wtedy zakłada się, że wynoszą one 0. Możesz nawet pominąć wszystkie numery i użyć tylko znacznika sufiksu. Użyj numeru wersji, gdy ważne jest, aby móc komunikować się z różnymi wersjami tej samej usługi, które jednocześnie występują w sieci.

### 2.9.1. Diagramy przepływów komunikatów

Kluczowe znaczenie ma zrozumienie przepływu komunikatów w systemie. W szczególności wszystkie komunikaty mają usługę klienta, który tworzy i wysyła komunikaty, oraz usługę, która nasłuchuje i odbiera. Wszystkie linie reprezentujące komunikaty łączące elementy muszą mieć kierunek wyznaczony przez strzałkę po stronie usługi odbierającej. Możesz przekazać tę informację za pomocą następujących konwencji:

- *ciągła linia* — komunikat synchroniczny, który oczekuje odpowiedzi;
- *przerywana linia* — komunikat asynchroniczny, który nie oczekuje odpowiedzi;
- *zamknięta strzałka* — komunikat jest konsumowany przez odbiorcę;
- *otwarta strzałka* — komunikat jest monitorowany przez odbiorcę.

Ponieważ linie komunikatów mogą być ciągłe lub przerywane, a groty strzałek mogą być zamknięte lub otwarte, istnieją cztery możliwości (zostaną one omówione bardziej szczegółowo w następnym rozdziale):

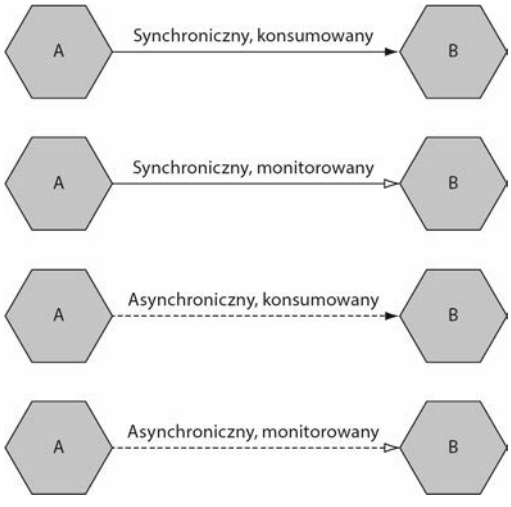
---

[30] Używam liczności do ujednoznacznienia strategii wdrażania omówionych w rozdziale 5.

[31] Identyfikatory wersji są zgodne ze wzorcem `MAJOR.MINOR.PATCH`. Możesz pominąć numery `MINOR` i `PATCH`. Zobacz: „Wersjonowanie semantyczne 2.0.0" — *http://semver.org/*.

- *„ciągła linia"-„zamknięta strzałka" — aktor synchroniczny* — tylko jedna instancja konsumuje komunikat i odpowiada na niego;
- *„ciągła linia"-„otwarta strzałka" — subskrybent synchroniczny* — wiele instancji monitoruje komunikat, a nadawca akceptuje pierwszą odpowiedź;
- *„przerywana linia"-„zamknięta strzałka" — aktor asynchroniczny* — tylko jeden odbiorca konsumuje komunikat;
- *„przerywana linia"-„otwarta strzałka" — subskrybent asynchroniczny* — wszyscy odbiorcy monitorują komunikat.

Rysunek 2.8 pokazuje, jak odzwierciedlić cztery interakcje.




**Rysunek 2.8.** Interakcja z komunikatami

Linie komunikatów mogą być dwukierunkowe, aby zmniejszyć bałagan na diagramie. Aby wskazać usługę-nadawcę, przesuń grot strzałki tak, żeby nie dotykała linii granicznej tego elementu.

Komunikaty mogą być przeznaczone dla wielu odbiorców, a ten sam komunikat może być odzwierciedlony poprzez oddzielne strzałki pochodzące od tej samej usługi. By uzyskać większą czytelność, można również podzielić strzałkę na wiele podstrzałek. Punkt podziału jest oznaczony symbolem kropki.

W komunikacji synchronicznej, kiedy masz wielu odbiorców, każdy komunikat jest dostarczany tylko do jednego odbiorcy zgodnie z jakimś algorytmem, który może być wskazany przez adnotację. Domyślnie używa się algorytmu karuzelowego. W komunikacji asynchronicznej komunikat jest dostarczany do wszystkich odbiorców. W obu przypadkach to, czy komunikat jest konsumowany, czy monitorowany, jest wskazane przez grot strzałki.

Linie komunikatów mogą być opatrzone przypisami z pełną nazwą komunikatu (będziesz jeszcze tego używał) lub skróconą nazwą (jak użyto w tym studium). Linie komunikatów można również opatrzyć poprzedzającymi numerami sekwencji. Mają one następujący format: `x.i.j.k...`, gdzie *x* jest literą, natomiast *i*, *j*, *k* są dodatnimi liczbami całkowitymi. Oddzielne sekwencje są oznaczone literą i nie ma domniemania

kolejności czasowej między nimi. Dodatnie liczby całkowite wskazują natomiast na czasową kolejność komunikatów w danej sekwencji. Tylko pierwsza liczba jest wymagana, a kropki separatora wskazują kolejność podsekwencji.

Każdą część diagramu można opisać za pomocą objaśnień w celu ujednoznacznienia interakcji między mikrousługami. Aby uniknąć pomyłki z prostokątami elementów zewnętrznych, objaśnienia składają się z linii łączącej opisywany obiekt z tekstem objaśniającym, sąsiadującym z pojedynczą poziomą lub pionową linią graniczną obiektu.

Diagramy mikrousług nie mają być częścią formalnej specyfikacji — są przeznaczone do komunikacji w zespole. Dlatego dopuszczalne jest pomijanie elementów ze względu na zwięzłość, nawet jeśli powoduje to niejednoznaczności. Diagramy zwłaszcza nie są przeznaczone do pokazywania mechanizmu transportu wybranego dla komunikatów, ponieważ zakłada się niezależność diagramów od warstwy transportowej. Jeśli chcesz wskazać określone mechanizmy transportu, użyj adnotacji.

## 2.10. Mikrousługi to komponenty oprogramowania

W tej książce twierdzę, że mikrousługi są doskonałymi, no, prawie doskonałymi komponentami oprogramowania. Warto przyjrzeć się temu twierdzeniu bardziej szczegółowo. Komponenty oprogramowania mają relatywnie dobrze określone cechy i istnieje ogólna zgoda w kwestii zrozumienia najważniejszych z nich. Zobaczmy, jak mikrousługi wypadają w porównaniu z tym rozumieniem.

### 2.10.1. Enkapsulacja

Komponenty oprogramowania są samodzielne. Zawierają zestaw semantycznie spójnych działań i danych. Świat zewnętrzny nie jest wtajemniczony w tę wewnętrzną reprezentację i nie może jej zanieczyszczać. Podobnie komponent nie ujawnia wewnętrznej implementacji. Celem tej cechy jest możliwość wymiany komponentów.

Mikrousługi zapewniają hermetyzację w bardzo silny sposób — znacznie silniejszy niż takie językowe konstrukcje, jak moduły czy klasy. Każda mikrousługa musi założyć fizyczne oddzielenie od innych mikrousług, może komunikować się z nimi tylko za pośrednictwem komunikatów i nie ma żadnej furtki do wewnętrznych elementów innych mikrousług. Stworzenie takiej furtki wymaga od programistów większego wysiłku, więc enkapsulacja jest dobrze zachowana przez cały okres istnienia systemu.

### 2.10.2. Wielokrotne użycie

Ponowne użycie jest Świętym Graalem rozwoju oprogramowania. Dobry komponent może być używany ponownie w wielu różnych systemach przez długi czas. W praktyce jest to trudne do osiągnięcia, ponieważ każdy system ma własne potrzeby. Komponent ewoluuje z czasem, by w końcu i tak posiadać różne wersje. Możliwość ponownego użycia oznacza również rozszerzalność: powinno być łatwe ponowne użycie komponentu w nowym kontekście bez konieczności ciągłego jego modyfikowania. Celem tej cechy jest to, aby komponenty były przydatne poza jednym, określonym projektem.

Mikrousługi są z natury wielokrotnego użytku, ponieważ są usługami sieciowymi, które mogą być wywołane przez każdego. Nie musisz się martwić o integrację kodu lub konsolidację bibliotek. Mikrousługi spełniają wymagania dotyczące wersjonowania i rozszerzalności nie z powodu zwiększania możliwości pojedynczej mikrousługi[32], ale ze względu na to, że pozwalają systemowi na dodawanie nowych mikrousług dla specjalnych przypadków, a następnie na wykorzystanie routingu komunikatów do uruchomienia nowej usługi. Omówimy to szczegółowo w rozdziale 3.

### 2.10.3. Dobrze zdefiniowane interfejsy

Interfejs oferowany przez komponent to pełna definicja jego umowy z otaczającym światem zewnętrznym. Ten interfejs powinien mieć wystarczającą, ale nie za dużą ilość szczegółów, aby umożliwić komponentowi zmianę jego implementacji lub zainstalowanie go w innym systemie. Celem tej cechy jest umożliwienie swobodnego wyboru komponentów.

Mikrousługi wykorzystują komunikaty i tylko komunikaty, aby komunikować się z otoczeniem. Te komunikaty mogą być jawnie wylistowane, a ich zawartość może być ograniczona w razie potrzeby[33]. Mikrousługi mają intencjonalnie dobrze zdefiniowane, ale niekoniecznie ścisłe interfejsy.

### 2.10.4. Kompozycyjność

Prawdziwa moc komponentów do przyspieszenia rozwoju oprogramowania nie pochodzi z wielokrotnego użytku, który jest jedynie akceleratorem liniowym, ale z łączenia komponentów, co pozwala robić o wiele więcej interesujących rzeczy, niż każdy komponent może zrobić osobno. Komponenty mogą być składane razem w bardziej sprawne komponenty, które z kolei mogą być składane w jeszcze większe systemy[34]. Cecha ta sprawia, że rozwój oprogramowania jest przewidywalny poprzez deklarację zachowania systemu, a nie poprzez konstruowanie go.

Mikrousługi łatwo ulegają kompozycji, ponieważ sieciowym przepływem komunikatów można manipulować zgodnie z potrzebami. Na przykład jedna mikrousługa może opakować inną poprzez przechwytywanie wszystkich komunikatów tej ostatniej, modyfikowanie ich w jakiś sposób i przekazywanie dalej do opakowanej usługi.

---

[32] Tradycyjnie systemy komponentów bazują na hookach API, aby rozszerzyć funkcjonalność. Jest to z natury nieopłacalne podejście, ponieważ nie jest jednolite — każdy komponent i hook API stanowią odosobniony przypadek.

[33] Oprzyj się pokusie używania schematów wiadomości i egzekwowania umów między usługami. Może się to wydawać dobrym pomysłem, dopóki nie zapędzisz się w kozi róg. Mikrousługi są odpowiednie w nieuporządkowanej logice biznesowej, w której rygorystyczne schematy codziennie umierają.

[34] Najbardziej udaną architekturą komponentów są UNIX-owe potoki. Ograniczając interfejs integracji do strumienia bajtów, poszczególne narzędzia wiersza poleceń można komponować w złożone potoki przetwarzania danych. Kompozycyjna moc tej architektury jest główną przyczyną sukcesu systemu operacyjnego.

### 2.10.5. Mikrousługi jako komponenty w praktyce

Przykładem użyteczności mikrousług jako komponentów jest interakcja z komunikatem *opakowanie-cache*. Ta interakcja szczególnie pokazuje kompozycję usług jako potężną technikę rozszerzania systemów na żywo. W tym przykładzie jednostkowa usługa, np. *artykuł* z systemu gazety, obsługuje komunikaty aktywności dla leżącej poniżej jednostki obsługującej dane artykułu. Większość z tych komunikatów jest związana z dostępem do danych. Istnieje jednak jedna słabość związana z tą techniką: usługa *artykuł* musi wiedzieć o usłudze *cache*! Czyli potrzebna jest dodatkowa logika. Usługa *artykuł* byłaby mniejszą i lepszą mikrousługą, gdyby nic nie wiedziała o *cache*. Zwracaj uwagę na tego typu zależności.

Stworzyłem tutaj jeden taki przykład w celach dekonstrukcji, ale łatwo jest skończyć z niepotrzebnymi zależnościami.

Alternatywną konfiguracją jest wprowadzenie usługi *cache-artykułu*, która przechwytuje wszystkie komunikaty przeznaczone dla usługi *artykuł* i przekazuje dalej większość z nich. Ale komunikaty *pobierz-artykuł*, *dodaj-artykuł* i *usuń-artykuł* powodują również wstrzykiwanie i usuwanie artykułów do i z pamięci cache. Z perspektywy reszty systemu komunikaty dotyczące artykułów obsługiwane są w ten sam sposób; nic się nie zmieniło. Ale dostajemy buforowanie artykułów! Skomponowaliśmy razem usługi *cache-artykułu* i *artykuł*.

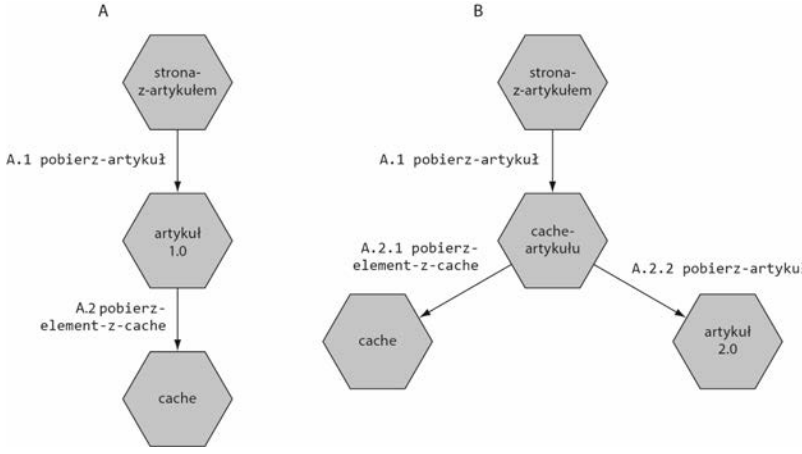
Aby to zadziałało w praktyce, musisz rozplanować interakcje z komunikatami. Zazwyczaj będziesz chciał wprowadzić ten typ zmiany w systemie uruchomionym w produkcji, bez przerwy w świadczeniu usług. Jednym ze sposobów na to jest użycie inteligentnego systemu równoważenia obciążenia do zarządzania usługą *artykuł*. Aby dodać *cache-artykułu*, zaktualizuj konfigurację modułu równoważenia obciążenia tak jak pokazano na rysunku 2.9. Będziesz potrzebował systemu równoważenia obciążenia, który poradzi sobie ze zmianą konfiguracji na żywo[35].

Innym sposobem jest użycie kolejki komunikatów[36]. Można wprowadzić *cache-artykułu* jako kolejnego subskrybenta, a następnie sprawić, żeby *artykuł* nie miał bezpośredniego kontaktu z kolejką komunikatów (zgodnie z krokami pokazanymi na rysunku 2.10). Wtedy usługi *artykuł* oraz *cache-artykułu* używałyby komunikacji typu punkt-punkt. Albo mógłbyś użyć oddzielnego tematu kolejki, gdybyś chciał uniknąć dodatkowego obciążenia związanego z wykrywaniem usług.

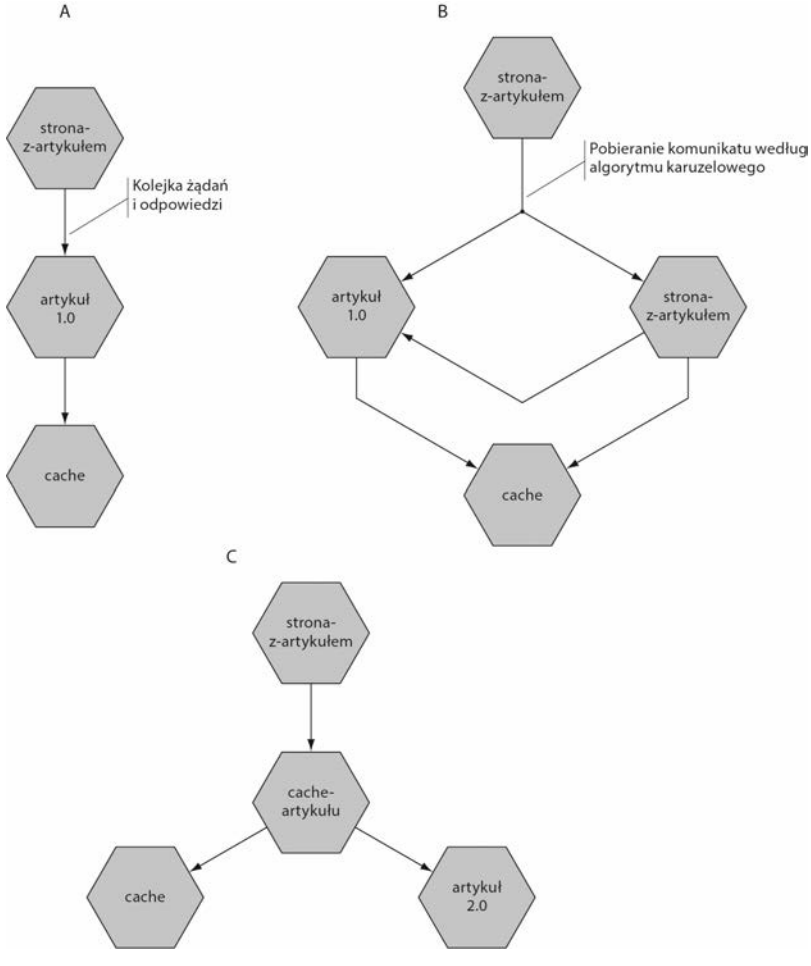
---

[35] Najlepszym wyborem są tu moduły równoważenia obciążenia specjalnie zaprojektowane dla mikrousług. Wypróbuj następujące: Eureka (*https://github.com/Netflix/eureka*), Synapse (*https://github.com/airbnb/synapse*) oraz Baker Street (*http://bakerstreet.io/*).

[36] Kolejki wiadomości są z założenia asynchroniczne, ale to nie oznacza, że przepływy wiadomości są z natury asynchroniczne. Wiadomości synchroniczne to te, które wymagają odpowiedzi, aby klient mógł kontynuować pracę. Mogą być dostarczane za pośrednictwem kolejki wiadomości przy użyciu tematów żądania i odpowiedzi dla każdego typu wiadomości lub przez osadzenie adresu sieciowego ścieżki zwrotnej jako metadanych w wiadomości żądania.




**Rysunek 2.9.** Rozszerzanie usługi artykuł bez modyfikacji




**Rysunek 2.10.** Wprowadzanie nowej funkcjonalności do działającego systemu

Bardzo ważną zasadą, na którą należy zwrócić uwagę, jest to, że aby poprawić i zmodyfikować funkcjonalność systemu w odniesieniu do buforowania artykułów, nie musisz rozszerzać istniejących usług[37]. Nie sprawiasz tym samym, że istniejące usługi staną się bardziej złożone. Główne działania polegają na dodawaniu do żyjącego systemu i usuwaniu z niego usług, po jednej naraz, bez przerywania jego działania.

Na każdym etapie możesz zweryfikować system, sprawdzając jego zachowanie i upewniając się, że nic nie jest zepsute. Potem na każdym kroku, jeśli coś zepsułeś, możesz łatwo przywrócić system do znanego, działającego stanu. Właśnie w ten sposób mikrousługi dokonują wdrożenia bez ryzyka.

## 2.11. Wewnętrzna struktura mikrousługi

Podstawowym celem mikrousługi jest wdrożenie logiki biznesowej. Powinieneś być w stanie skoncentrować się na tym celu. Nie powinieneś za to zajmować się takimi rzeczami, jak wykrywanie usług, logowanie, odporność na błędy i innymi standardowymi zachowaniami; to są doskonali kandydaci do kodu frameworka czy infrastruktury.

Mikrousługi potrzebują warstwy komunikacyjnej przeznaczonej dla komunikatów. To powinno być całkowicie wyabstrahowane od wysyłania i odbierania komunikatów oraz wiedzy o tym, gdzie muszą one iść. Jeśli tylko jedna mikrousługa wie o innej, powstaje zależność, w wyniku której stajesz na równi pochyłej prowadzącej do niestabilnego systemu. Dostarczenie komunikatu powinno więc być niezależne od warstwy transportowej: komunikaty mogą podróżować przez dowolne medium, niezależnie od tego, czy jest to HTTP, magistrala komunikatów, surowe TCP, gniazda internetowe czy cokolwiek innego. Ta abstrakcja jest najważniejszym kawałkiem kodu infrastruktury.

Ponadto mikrousługi muszą mieć jakiś sposób rejestrowania swoich zachowań i błędów. Oznacza to, że muszą mieć sposób na logowanie oraz raportowanie swojego statusu. Zasadniczo jest to ten sam mechanizm i mikrousługa nie powinna zajmować się szczegółami plików dziennika lub raportowania zdarzeń. Mikrousługi muszą być w stanie ulegać awarii na tyle mocno i głośno, aby system i zespół mogli szybko podjąć działania. Abstrakcje do logowania i raportowanie są również niezbędne[38].

Mikrousługi potrzebują również systemu wykonawczego. Powinny informować rejestry systemu o swoim istnieniu, aby można było nimi zarządzać. Powinny też być w stanie zaakceptować zewnętrzne, administracyjne i kontrolne polecenia pochodzące z tego systemu. Chociaż warstwy komunikacji i rejestrowania często mogą być dostarczane wraz z oddzielnymi bibliotekami, które konsolidujesz ze swoimi usługami, to funkcja wykonawcza zależy od bardziej złożonej interakcji ze specjalnymi mechanizmami administracyjnymi i kontrolnymi. Te warstwy muszą również dobrze współgrać z Twoją strategią wdrażania i z oprzyrządowaniem. Zbadamy to bardziej szczegółowo w rozdziale 5.

---

[37] W rzeczywistości zmniejszasz złożoność usługi *artykuł*.

[38] Używanie kontenerów do wdrażania swoich mikrousług to świetny sposób na uzyskanie tego typu narzędzi za darmo.

## 2.12. Podsumowanie

- Jednorodność mikrousług sprawia, że bardzo dobrze sprawdzają się one jako fundamentalne jednostki konstrukcji oprogramowania. Są bardzo praktyczne, jeśli chodzi o planowanie, pomiary, specyfikację i wdrażanie. Ta charakterystyka bierze się stąd, że są jednorodne pod względem rozmiaru i złożoności oraz że przy komunikowaniu się ze światem zewnętrznym są ograniczone do używania komunikatów.
- Ścisła definicja terminu *mikrousługa* jest zbyt ograniczająca. Szybciej wpadniesz na pomysły i poszerzysz przestrzeń potencjalnych rozwiązań, przyjmując bardziej holistyczny punkt widzenia, gdy będziesz miał głębsze zrozumienie.
- Architektury mikrousług można podzielić na dwie szerokie kategorie: synchroniczne (zwykle usługi webowe oparte na REST) i asynchroniczne (zwykle za pośrednictwem kolejki komunikatów). Żadna z nich nie jest pełnym rozwiązaniem, a systemy produkcyjne są często hybrydami.
- Architektury monolityczne tworzą trzy negatywne implikacje. Potrzebują większej koordynacji zespołu, powodując narzut związany z zarządzaniem; cierpią z powodu większego długu technicznego, co powoduje spowolnienie tempa rozwoju; oraz są bardzo ryzykowne, ponieważ wdrożenia wpływają na cały system.
- Niewielki rozmiar mikrousług ma pozytywne skutki. Szacowanie jest dokładniejsze, ponieważ mikrousługi są w większości tego samego rozmiaru; kod jest jednorazowy, co eliminuje egocentryczne zachowania programistyczne; a system jest dynamicznie konfigurowalny, dzięki czemu łatwiej poradzi sobie z tym, co nieoczekiwane.
- Istnieją dwa rodzaje kodu: logika biznesowa i biblioteki infrastruktury. Mają one bardzo różne potrzeby. Mikrousługi są dla logiki biznesowej, ponieważ mogą poradzić sobie z niesprecyzowanymi, ciągle zmieniającymi się wymaganiami.
- Aby zaprojektować system mikrousługowy, zacznij od wymagań, wyraź je poprzez komunikaty, a następnie pogrupuj te komunikaty według usług. Pomyśl o tym, jak komunikaty są obsługiwane przez usługi: synchronicznie czy asynchronicznie? Czy są monitorowane, czy konsumowane?
- Mikrousługi są naturalnymi komponentami oprogramowania. Podlegają enkapsulacji i są wielokrotnego użytku; mają dobrze zdefiniowane interfejsy i, co najważniejsze, mogą być komponowane razem w bardziej funkcjonalne jednostki.

## Pracuj mądrzej i bardziej humanitarnie. Wdrażaj mikrousługi!

Systemy oparte na mikrousługach różnią się od aplikacji monolitycznych. Są skalowalne, efektywniejsze, a także łatwiejsze w implementacji, rozwijaniu i utrzymaniu. Architektura mikrousług pozwala na doskonalenie danego elementu bez zastanawiania się nad działaniem całości aplikacji. Tego rodzaju systemy opierają się na nowoczesnych wzorcach, takich jak asynchroniczna komunikacja za pomocą komunikatów, usługi API i hermetyzacja. Po odpowiedniej optymalizacji dobrze działają zarówno w chmurze, jak i w scentralizowanych środowiskach opartych na kontenerach.

Niniejsza książka jest przeznaczona dla programistów, menedżerów projektów i architektów oprogramowania. Wyjaśniono tu niezbędne pojęcia oraz różnice dzielące systemy oparte na mikrousługach i aplikacje monolityczne, a także zasady ich projektowania. Wyczerpująco omówiono techniki rozwiązywania problemów z mikrousługami oraz sposoby kontrolowania ryzyka wystąpienia awarii. Pokazano, w jaki sposób mikrousługi mogą współpracować z trwałymi danymi i jak wygląda ich współpraca z bazami danych. Sporo miejsca poświęcono technikom oceny kondycji działających systemów mikrousługowych, a także studiom przypadków oraz najlepszym praktykom w zakresie pracy zespołu, planowania zmian i wyboru narzędzi.

### Najważniejsze zagadnienia:

- mikrousługi i ich architektura
- mikrousługi a wymagania biznesowe i korporacyjne
- komunikaty i ich wzorce
- wdrażanie systemów o dużej skali
- wady systemów opartych na mikrousługach

Richard Rodger programuje od 1986 roku (zaczął od Sinclair ZX Spectrum) po dziś dzień. Współtworzył wiele firm produkujących oprogramowanie, przy okazji stał się niekwestionowanym autorytetem w dziedzinie Node.js i mikrousług. Twierdzi, że jego obsesja na punkcie JavaScriptu wzięła się z braku umiejętności kodowania w C++. Napisał kilka uznanych książek, od czasu do czasu występuje na różnych konferencjach technologicznych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl



KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-4807-3

9 788328 348073



Cena: 59,00 zł