Tarek Ziadé

Rozwijanie mikrousług w Pythonie

Budowa, testowanie, instalacja i skalowanie

Helion

Packt›

Tytuł oryginału: Python Microservices Development

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-4596-6

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/rozmik
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/rozmik.zip

Printed in Poland.

Spis treści

1

ROZDZIAŁ

Czym są mikrousługi?

Od zawsze staraliśmy się usprawniać proces tworzenia oprogramowania, a od czasów kart perforowanych usprawnienia te — mówiąc oględnie — są ogromne.

Jednym z takich usprawnień jest trend mikrousług, który pojawił się kilka lat temu. Jego źródłem była między innymi potrzeba przyspieszenia cyklu udostępniania oprogramowania w firmach informatycznych. Nowe produkty i funkcje musiały być oferowane użytkownikom możliwie jak najszybciej. Kolejne wersje miały powstawać często, aby można je było sprzedawać, sprzedawać i jeszcze raz sprzedawać.

W przypadku usługi, z której korzystają tysiące, a nawet miliony użytkowników, dobrą praktyką okazało się udostępnianie im eksperymentalnej funkcjonalności i wycofywanie jej, jeżeli nie działała poprawnie, zamiast cyzelowania jej miesiącami przed udostępnieniem.

Firmy takie jak Netflix promują techniki ciągłego dostarczania oprogramowania, polegające na częstym wprowadzaniu małych zmian w środowisku produkcyjnym i testowaniu ich na wybranej grupie użytkowników. W ten sposób powstało narzędzie Spinnaker (*http://www.spinnaker.io*) automatyzujące proces aktualizacji oprogramowania i udostępniania funkcjonalności w chmurze w postaci niezależnych mikrousług.

Jeżeli korzystasz z serwisów Hacker News lub Reddit, na pewno trudno Ci jest ocenić, które informacje mogą być dla Ciebie ważne, a które są tylko medialnym szumem.

> *Napisz artykuł obiecujący zbawienie, uczyń go czymś uporządkowanym, wirtualnym, abstrakcyjnym i niejasnym na wyższym poziomie, i możesz prawie być pewnym, że rozpoczniesz w ten sposób nowy kult.*
>
> *— Edsger W. Dijkstra*

Celem tego rozdziału jest wyjaśnienie, czym są mikrousługi. Skupimy się w nim na różnych sposobach tworzenia mikrousług w języku Python. Rozdział składa się z następujących części:

- Wprowadzenie do architektury SOA
- Metodyka tworzenia aplikacji monolitycznych
- Metodyka tworzenia aplikacji opartych na mikrousługach
- Zalety mikrousług
- Wady mikrousług
- Kodowanie mikrousług w języku Python

Po przeczytaniu tego rozdziału będziesz dokładnie wiedział, czym są, a czym nie są mikrousługi, i będziesz gotów na poznanie zawiłości ich tworzenia. Dowiesz się również, jak używać języka Python.

Geneza architektury SOA

Nie istnieje oficjalny standard mikrousług, dlatego funkcjonuje kilka ich definicji. Często przy próbie określenia, czym są mikrousługi, używane jest pojęcie architektury **SOA** (ang. *Service-Oriented Architecture*, architektura zorientowana na usługi).

> *SOA — koncepcja tworzenia systemów informatycznych, w której główny nacisk stawia się na definiowanie usług, spełniających wymagania użytkownika. (…) Mianem usługi określa się tu każdy element oprogramowania, mogący działać niezależnie od innych.*
>
> *— Wikipedia*

Element w powyższej definicji oznacza samodzielną usługę, która implementuje pewną funkcjonalność biznesową i udostępnia ją poprzez określony interfejs.

Choć architektura SOA określa, że każda usługa musi być niezależnym procesem, nie narzuca jednak rodzaju protokołów wykorzystywanych przez procesy do komunikacji między sobą. Ogólnie definiuje też budowę i sposób wdrażania aplikacji.

W manifeście SOA (*http://www.soa-manifesto.org*), opublikowanym ok. 2009 r. przez grupę ekspertów, nie ma też ani słowa o tym, jak usługi powinny komunikować się między sobą za pośrednictwem sieci. Usługi SOA mogą komunikować się za pomocą protokołu **IPC** (ang. *Inter-Process Communication*, komunikacja międzyprocesowa), gniazd, współdzielonej pamięci, pośrednich kolejek komunikatów, a nawet protokołu **RPC** (ang. *Remote Procedure Calls*, zdalne wywoływanie procedur). Liczba dostępnych opcji jest bardzo duża. Krótko mówiąc, architektura SOA pasuje do każdej aplikacji, która nie jest pojedynczym procesem.

Często słyszy się, że mikrousługa jest wyspecjalizowaną odmianą architektury SOA. Jest tak dlatego, ponieważ mikrousługi realizują kilka założeń tej architektury, m.in. umożliwiają tworzenie aplikacji złożonych z niezależnych komponentów komunikujących się ze sobą.

Aby sformułować pełną definicję mikrousługi, należy wcześniej przyjrzeć się, jak zbudowana jest większość aplikacji.

Podejście monolityczne

Rozważmy bardzo prosty przykład tradycyjnej monolitycznej aplikacji: serwisu WWW do rezerwowania miejsc w hotelach. Taka strona, oprócz statycznej treści HTML, zawiera funkcjonalność umożliwiającą użytkownikom rezerwowanie miejsc w hotelu w dowolnym mieście na całym świecie. Użytkownik wyszukuje odpowiedni hotel, a następnie rezerwuje w nim pokój i płaci kartą kredytową.

Gdy użytkownik szuka hotelu na stronie WWW, aplikacja wykonuje następujące operacje:

1. Wysyła do bazy danych kilka zapytań SQL.
2. Wysyła zapytanie HTTP do sąsiedniej usługi dodającej hotel do listy.
3. Generuje na podstawie szablonu wynikową stronę HTML.

Gdy użytkownik wybierze odpowiedni hotel i kliknie go w celu zarezerwowania miejsca, aplikacja wykonuje następujące operacje:

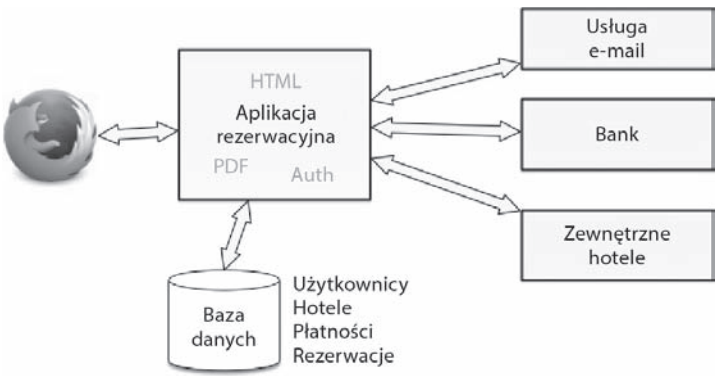
1. Tworzy konto użytkownika, jeżeli jest to konieczne, a następnie uwierzytelnia go.
2. Komunikuje się z usługą bankowości internetowej w celu dokonania płatności.
3. W procesie płatności wykorzystuje dane użytkownika zapisane w bazie.
4. Przygotowuje potwierdzenie za pomocą generatora plików PDF.
5. Za pomocą usługi e-mail wysyła potwierdzenie do użytkownika.
6. Do hotelu wysyła wiadomość e-mail z potwierdzeniem.
7. W bazie danych tworzy rekord rezerwacji.

Jest to oczywiście model uproszczony, ale oddający rzeczywistość.

Aplikacja komunikuje się z bazą danych zawierającą informacje o hotelu, szczegóły rezerwacji, płatności, dane użytkownika itp. Komunikuje się również z zewnętrznymi usługami wysyłającymi wiadomości e-mail, realizującymi płatności i dostarczającymi informacji o innych hotelach.

Zgodnie ze starą, dobrą architekturą **LAMP** (ang. *Linux-Apache-MySQL-Perl/PHP/Python*) każde odebrane zapytanie HTTP generuje lawinę zapytań SQL do bazy danych, przez sieć przesyłanych jest wiele zapytań do zewnętrznych usług, po czym serwer, wykorzystując szablon, tworzy kod HTML strony wynikowej.

Poniższy diagram ilustruje scentralizowaną architekturę aplikacji:



Jest to typowa, monolityczna aplikacja, mająca wiele niewątpliwych zalet.

Podstawową zaletą jest jeden kod. Dzięki temu rozpoczęcie projektu tworzenia aplikacji jest łatwiejsze. Zakodowanie testów jest łatwe, a samemu kodowi można nadać przejrzystą strukturę. Umieszczenie wszystkich danych w jednej bazie również upraszcza kodowanie aplikacji. Można modyfikować model danych i sposób pobierania ich za pomocą kodu.

Wdrożenie takiej aplikacji to też żaden problem: kod trzeba skompilować do pakietu i uruchomić go na jakimś serwerze. Aby wyskalować aplikację, wystarczy uruchomić jej kolejne instancje i kilka baz danych razem z jakimś mechanizmem replikacyjnym.

Dopóki aplikacja jest mała, ten model się sprawdza. Poza tym aplikację może z łatwością utrzymywać jeden zespół administratorów.

Projekty jednak mają to do siebie, że rozrastają się do rozmiarów większych niż zakładane na początku. Jeżeli jest jeden kod aplikacji, wtedy zaczynają się pojawiać poważne problemy. Gdy trzeba wprowadzić dużą zmianę, na przykład zmienić usługę bankową lub bazę danych, wtedy aplikacja wchodzi w bardzo niestabilny stan. Taka zmiana jest poważnym problemem w cyklu życia projektu, a wdrożenie nowej wersji aplikacji wymaga jej dodatkowego przetestowania. A takie zmiany zdarzają się często.

Małe zmiany też mogą być przyczyną kłopotów, ponieważ każda część systemu ma inne wymagania dotyczące niezawodności i stabilności. Narażenie procesów rezerwacji i płatności na ryzyko z powodu wadliwej funkcji generującej pliki PDF to jeden z tego rodzaju problemów.

Kolejną kwestią jest niekontrolowany rozrost aplikacji. Wciąż rozszerza się ją o nowe funkcjonalności, a gdy z projektu odchodzą jacyś programiści i dołączają nowi, wtedy kod robi się coraz bardziej zagmatwany, co spowalnia jego testowanie. W efekcie powstaje kod poplątany jak spaghetti, trudny w utrzymaniu, korzystający ze skomplikowanej bazy danych wymagającej stosowania złożonych procedur migracyjnych za każdym razem gdy programista zmieni model danych.

Duże projekty informatyczne dojrzewają kilka lat i powoli pojawia się w nich niesamowity bałagan, z którym trudno jest sobie poradzić. I to nie z powodu niekompetencji programistów, tylko dlatego, że kod jest coraz bardziej złożony, a osób, które w pełni rozumieją skutki wprowadzanych niewielkich zmian, jest coraz mniej. Każdy członek zespołu pracuje osobno na niewielkim fragmencie kodu, a gdy spojrzy się na niego z szerszej perspektywy, widać jeden wielki bałagan.

Wszyscy to przerabialiśmy.

Programiści pracujący nad takimi projektami marzą o tym, aby napisać kod aplikacji od podstaw, wykorzystując przy tym najnowsze platformy. W ten sposób jednak wpadają w tę samą pułapkę i historia się powtarza.

Poniższa lista podsumowuje zalety i wady monolitycznego podejścia podczas tworzenia aplikacji:

- Rozpoczęcie projektu jest proste.
- Scentralizowana baza ułatwia projektowanie i porządkowanie danych.
- Wdrożenie aplikacji jest proste.
- Wszelkie zmiany w kodzie mogą mieć wpływ na pozornie niezwiązane z nimi funkcjonalności. Jeżeli któraś z nich przestanie działać, awarii może ulec cała aplikacja.
- Możliwości skalowania aplikacji są ograniczone: można uruchamiać kilka instancji, ale jeżeli jedna z funkcjonalności zajmie wszystkie dostępne zasoby, wtedy problem obejmuje cały system.
- W miarę powiększania się kodu coraz trudniej jest utrzymywać w nim porządek.

Są oczywiście sposoby uniknięcia niektórych opisanych powyżej problemów.

Oczywistym rozwiązaniem jest podzielenie aplikacji na osobne części, nawet jeżeli kod wynikowy będzie uruchamiany w ramach jednego procesu. Do tego celu programiści wykorzystują zewnętrzne biblioteki i platformy, rozwijane wewnętrznie lub przez społeczność **OSS** (ang. *Open Source Software*, otwarte oprogramowanie).

Jeżeli do tworzenia aplikacji w języku Python użyje się platformy takiej jak Flask, wtedy można skupić się na algorytmie. Ponadto bardzo atrakcyjna jest możliwość wydzielenia fragmentów kodu w postaci rozszerzeń Flask lub pakietów Python. Dodatkowo, podzielenie kodu na małe pakiety jest często dobrym sposobem kontrolowania rozrostu aplikacji.

Małe jest piękne.

— Filozofia systemu UNIX

Na przykład generator plików PDF w opisanej wcześniej aplikacji rezerwacyjnej może być osobnym pakietem wykorzystującym bibliotekę Reportlab i kilka szablonów. Taki pakiet mógłby być wykorzystywany również w innych aplikacjach, a nawet udostępniony społeczności programistów w serwisie PyPI (Python Package Index).

Jednak w przypadku tworzenia jednoczęściowej aplikacji wciąż występuje kilka problemów, na przykład brak możliwości różnego skalowania jej poszczególnych fragmentów lub błędy w wykorzystywanych zależnościach.

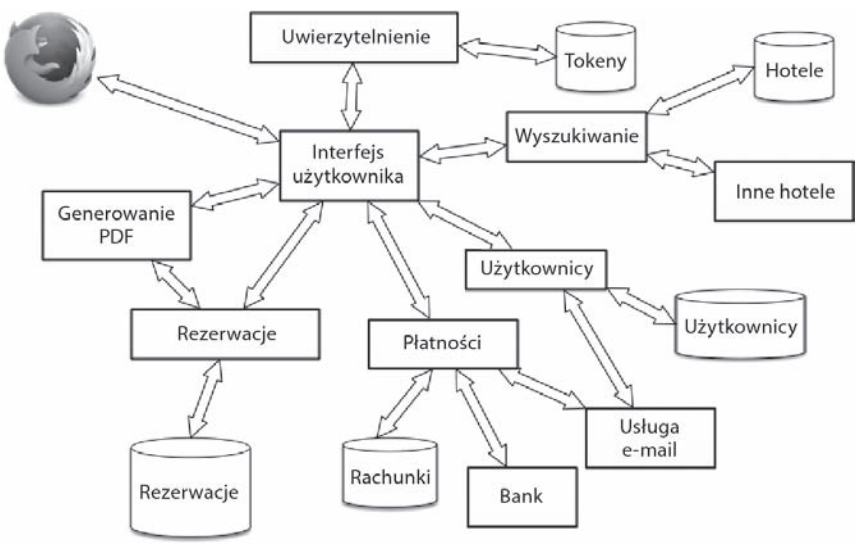
Korzystanie z zależności wiąże się z kolejnymi wyzwaniami, które określa się mianem **piekła zależności**. Jeżeli jakiś fragment aplikacji korzysta z biblioteki, ale na przykład generator plików PDF wymaga stosowania tylko jej określonej wersji, wtedy zazwyczaj trzeba szukać jakiegoś prowizorycznego rozwiązania zastępczego albo tworzyć własną wersję zależności.

Oczywiście opisane tu problemy nie ujawniają się dopiero w godzinie zero, gdy projekt już startuje, ale narastają stopniowo w miarę upływu czasu.

Przyjrzyjmy się teraz, jak wyglądałaby aplikacja zbudowana z mikrousług.

Podejście mikrousługowe

Gdybyśmy chcieli zbudować tę samą aplikację z mikrousług, można by było podzielić jej kod na kilka osobnych komponentów uruchamianych w oddzielnych procesach. Zamiast jednej aplikacji realizującej wszystkie funkcjonalności byłoby wiele różnych mikrousług, jak na poniższym rysunku.



Niech Cię nie przeraża liczba komponentów na tym diagramie. Po prostu po podzieleniu monolitycznej aplikacji na części została ujawniona jej wewnętrzna komunikacja. Po nieznacznym uproszczeniu schematu można w nim wyróżnić siedem niezależnych komponentów:

1. **Interfejs użytkownika**: frontalna usługa generująca stronę WWW i komunikująca się z innymi mikrousługami.
2. **Generowanie PDF**: bardzo prosta usługa tworząca potwierdzenia i inne dokumenty w formacie PDF na podstawie szablonów i danych.
3. **Wyszukiwanie**: usługa tworząca listę hoteli w danym mieście. Usługa ta korzysta z własnej bazy danych.
4. **Płatności**: usługa komunikująca się z usługą bankową, korzystająca z bazy danych o płatnościach i wysyłająca wiadomości e-mail potwierdzające dokonanie płatności.
5. **Rezerwacje**: usługa przechowująca informacje o rezerwacjach i tworząca pliki PDF.
6. **Użytkownicy**: usługa przechowująca informacje o użytkownikach i wysyłająca do nich wiadomości e-mail.
7. **Uwierzytelnienie**: usługa stosująca standard OAuth 2.0, generująca tokeny wykorzystywane przez inne usługi do uwierzytelniania użytkowników.

Powyższe mikrousługi wraz z dodatkowymi usługami, na przykład pocztą e-mail, oferują funkcjonalności podobne jak monolityczna aplikacja. Jednak w tym przypadku poszczególne komponenty komunikują się ze sobą za pomocą protokołu HTTP, a funkcjonalności są udostępniane za pomocą protokołu REST.

Nie ma tu jednej centralnej bazy danych. Każda mikrousługa korzysta z własnej wewnętrznej bazy, a dane odbiera i wysyła za pomocą zapytań HTTP i niezależnych od języków programowania formatów JSON, XML lub YAML.

Usługa interfejsu użytkownika różni się nieco od pozostałych, ponieważ generuje stronę WWW. W zależności od wykorzystywanej przez nią platformy wynikiem działania usługi może być zestaw plików HTML i JSON albo wręcz tylko plik JSON, jeżeli po stronie klienta uruchamiany jest statyczny skrypt JavaScript tworzący stronę w przeglądarce.

Poza wyjątkową usługą interfejsu użytkownika wszystkie pozostałe mikrousługi komunikują się ze sobą za pomocą protokołu HTTP, tworząc w ten sposób kompletną aplikację.

W tym kontekście mikrousługi stanowią logiczne jednostki, z których każda wykonuje wyłącznie jedno ściśle określone zadanie. Nasza propozycja definicji mikrousługi brzmi następująco:

> Mikrousługa jest to bardzo prosta aplikacja oferująca wąski zakres funkcjonalności w ściśle określonym kontekście. Jest to *komponent odpowiedzialny za jedną funkcję*, który można rozwijać i wdrażać niezależnie od innych komponentów.

Powyższa definicja nie wspomina o protokole HTTP ani o formacie JSON, ponieważ obejmuje również na przykład małą usługę wykorzystującą protokół UDP do przesyłania danych binarnych.

Jednak w przypadku opisanej aplikacji oraz w przykładowych kodach opisanych w tej książce wszystkie mikrousługi, z wyjątkiem interfejsu użytkownika, są prostymi aplikacjami WWW wykorzystującymi protokół HTTP, wysyłającymi i odbierającymi dane w formacie JSON.

Zalety mikrousług

Choć architektura mikrousługowa wydaje się bardziej skomplikowana od monolitycznej, ma w porównaniu z nią więcej zalet, między innymi:

- pozwala rozdzielać zakresy odpowiedzialności,
- można ją dzielić na mniejsze projekty,
- oferuje więcej opcji skalowania i wdrażania.

W kolejnych częściach powyższe zalety opisane są dokładniej.

Rozdzielenie zakresów odpowiedzialności

Przede wszystkim każdą mikrousługę może rozwijać osobny, niezależny zespół. Na przykład tworzenie usługi rezerwacji może być osobnym projektem. Odpowiedzialny za nią zespół może używać dowolnego języka programowania i bazy danych, o ile tylko dobrze udokumentuje interfejs HTTP API.

Rozdzielenie zakresów odpowiedzialności oznacza również, że rozwój aplikacji można ściślej kontrolować niż w podejściu monolitycznym. Na przykład zmiana komunikacji systemu obsługi płatności z bankiem będzie dotyczyła tylko danej usługi, a pozostała część aplikacji pozostanie nienaruszona i będzie dalej działać stabilnie.

Luźne sprzęganie usług znakomicie przyspiesza rozwój projektu. Filozofia takiego podejścia jest podobna do zasady *pojedynczej odpowiedzialności* zdefiniowanej przez Roberta Martina. Zgodnie z tą zasadą powód zmiany dowolnej klasy może być tylko jeden. Innymi słowy, każda klasa powinna oferować jedną, ściśle określoną funkcjonalność. W odniesieniu do mikrousług zasada ta oznacza, że każda mikrousługa odgrywa w aplikacji tylko jedną rolę.

Mniejsze projekty

Drugą zaletą podejścia mikrousługowego jest rozdzielenie złożoności projektu. Gdy aplikacja jest rozbudowywana o kolejną funkcjonalność, na przykład generowanie plików PDF, jej kod się powiększa, nawet jeżeli robi się to w uporządkowany sposób. Kod staje się bardziej skomplikowany i czasami działa wolniej. Problem rozwiązuje implementowanie nowej funkcjonalności

w postaci osobnej aplikacji. W ten sposób łatwiej jest tworzyć potrzebne funkcjonalności. Ponadto można je często zmieniać, a ponieważ cykle udostępniania nowych wersji są krótsze, aplikacja spełnia bieżące potrzeby użytkowników. Rozwój aplikacji znajduje się pod pełną kontrolą.

Mniejsze projekty redukują również ryzyko wystąpienia problemów podczas usprawniania aplikacji. Jeżeli programiści zechcą wypróbować najnowszą wersję języka programowania lub platformy, mogą szybko przygotować prototyp oferujący ten sam interfejs API, przetestować go i zadecydować o kontynuowaniu prac w tym kierunku.

Praktycznym przykładem takiego podejścia jest mikrousługa Firefox Sync. Obecnie prowadzone są pewne eksperymenty nad przejściem z aktualnie wykorzystywanego języka Python i bazy MySQL na język Go i bazę SQLite. Prototyp jest w fazie bardzo eksperymentalnej, ale dzięki temu, że wydzieliliśmy osobną mikrousługę przechowującą dane i zdefiniowaliśmy interfejs HTTP API, łatwo będzie przetestować ją na wybranej małej grupie użytkowników.

Skalowanie i wdrażanie

Wreszcie dzięki podzieleniu aplikacji na mniejsze komponenty łatwiej jest ją skalować w zależności od warunków. Załóżmy, że z aplikacji korzysta każdego dnia bardzo wielu użytkowników, a mikrousługa generująca pliki PDF znacznie obciąża procesor. W takim przypadku wystarczy uruchomić tę jedną mikrousługę na kilku serwerach wyposażonych w bardziej wydajne procesory.

Innym przykładem są mikrousługi wykorzystujące pamięć RAM, na przykład komunikujące się z bazą *Redis* lub *Memcache*. Tego rodzaju usługi można uruchamiać na serwerach wyposażonych w słabsze procesory, ale mających znacznie większą pamięć.

Zalety mikrousług można podsumować następująco:

- Zespoły programistów mogą rozwijać usługi niezależnie od siebie i używać dowolnych technologii. Mogą definiować własne cykle udostępniania nowych wersji usług. Wymagane jest tylko udostępnienie niezależnego od języka programowania interfejsu HTTP API.
- Programiści mogą dzielić skomplikowaną aplikację na logiczne komponenty. Każda mikrousługa wykonuje tylko jedno zadanie, ale za to wykonuje je dobrze.
- Niektóre usługi mogą być niezależnymi aplikacjami. Dzięki temu można lepiej kontrolować wdrażanie aplikacji i łatwiej ją skalować.

Architektura mikrousługowa pozwala rozwiązać mnóstwo problemów pojawiających się w miarę rozbudowy aplikacji. Należy jednak mieć świadomość związanych z nimi innych praktycznych kwestii.

Wady mikrousług

Jak wcześniej wspomniałem, budowanie aplikacji z mikrousług ma wiele zalet, ale nie jest złotym środkiem na wszystkie problemy.

Należy pamiętać o następujących najważniejszych problemach, jakie mogą pojawić się podczas kodowania mikrousług:

- nielogiczny podział aplikacji,
- więcej interakcji sieciowych,
- powielanie danych,
- problemy z kompatybilnością,
- skomplikowane testy.

W kolejnych częściach powyższe problemy opisane są dokładniej.

Nielogiczny podział aplikacji

Podstawową kwestią w podejściu mikrousługowym jest zaprojektowanie architektury aplikacji. Jest rzeczą niemożliwą, aby zespół programistów za pierwszym razem zdefiniował idealną architekturę. Niektóre mikrousługi, na przykład generator plików PDF, są oczywiste, ale w zależności od algorytmu istnieje duże prawdopodobieństwo, że zanim podzieli się go na odpowiednie mikrousługi, kod już będzie na tyle rozwinięty, że właściwy podział okaże się niemożliwy do przeprowadzenia.

Architektura musi dojrzewać w cyklach prób i błędów. Dodawanie i usuwanie mikrousług może być bardziej uciążliwe niż tworzenie monolitycznej aplikacji. Ten problem można rozwiązać, unikając dzielenia aplikacji na mikrousługi, jeżeli powody nie są oczywiste.

Pochopne dzielenie aplikacji jest źródłem wszelkiego zła.

Jeżeli masz jakiekolwiek wątpliwości, czy dany podział jest właściwy, bezpieczniej będzie pozostawić kod w jednej części. Zawsze łatwiej jest później podzielić go na nowe mikrousługi, niż łączyć kod dwóch mikrousług w jedną całość, gdy decyzja o podziale okaże się błędna.

Jeżeli na przykład zawsze trzeba instalować dwie określone usługi razem lub jeżeli zmiana wprowadzona w jednej z nich wpływa na model danych w innej, oznacza to, że prawdopodobnie mikrousługi nie zostały wyodrębnione poprawnie i należy je z powrotem ze sobą połączyć.

Więcej interakcji sieciowych

Innym problemem jest zwiększona ilość interakcji sieciowych wewnątrz aplikacji. W wersji monolitycznej, nawet jeżeli kod jest zagmatwany, wszystkie operacje wykonywane są przez jeden proces, a wyniki są wysyłane do użytkownika bez konieczności wcześniejszego odwoływania się do licznych usług w tle.

Trzeba zwrócić szczególną uwagę na sposób odwoływania się do usług w tle i odpowiedzieć sobie na następujące pytania:

- Co się stanie, gdy interfejs użytkownika nie będzie mógł korzystać z usługi generującej pliki PDF z powodu jej przeciążenia lub z powodu przerwy w komunikacji sieciowej?
- Czy interfejs użytkownika odwołuje się do innych usług w sposób synchroniczny, czy też asynchroniczny?
- Jak sposób odwoływania się od usług wpływa na czas odpowiedzi interfejsu użytkownika?

Aby odpowiedzieć na powyższe pytania, potrzebna jest przemyślana strategia rozwoju aplikacji, o czym będzie mowa w rozdziale 5, „Interakcje z innymi usługami".

Powielanie danych

Kolejnym problemem jest przechowywanie i współdzielenie danych. Skuteczna mikrousługa powinna być niezależna od innych mikrousług i w idealnym przypadku nie powinna współdzielić z nimi swojej bazy danych. Co to oznacza w przypadku aplikacji rezerwacyjnej?

Tak jak poprzednio, pojawia się w tym momencie wiele pytań, na przykład:

- Czy we wszystkich bazach danych identyfikator danego użytkownika powinien być taki sam, czy też każda usługa musi definiować go niezależnie jako ukryty szczegół implementacyjny?
- Gdy do systemu zostanie dodany nowy użytkownik, czy należy replikować jego dane w innych bazach, stosując odpowiednią strategię, na przykład pompowanie danych, czy też jest to zbędna operacja?
- W jaki sposób dane mają być usuwane z bazy?

Nie jest łatwo znaleźć odpowiedzi na powyższe pytania, a ponadto istnieje wiele sposobów rozwiązywania tego rodzaju problemów, o czym dowiesz się z tej książki.

> Maksymalne redukowanie duplikacji danych przy jednoczesnym izolowaniu usług jest jednym z największych wyzwań towarzyszących tworzeniu aplikacji opartych na mikrousługach.

Problemy z kompatybilnością

Inne problemy pojawiają się w przypadku, gdy zmiana określonej funkcjonalności wpływa na kilka mikrousług. Jeżeli zmiana nie zapewnia wstecznej kompatybilności usługi i zakłóca przepływ danych między mikrousługami, wtedy pojawiają się kłopoty.

Czy należy wdrażać nową usługę współpracującą ze starszymi wersjami innych usług? Czy też trzeba zmieniać kilka usług jednocześnie? Czy to oznacza, że takie usługi powinny zostać z powrotem scalone?

Dokumentowanie wersji kodu usługi i utrzymywanie dyscypliny w jej interfejsie API pozwala uniknąć tego typu problemów, o czym przekonasz się w drugiej części książki poświęconej kodowaniu przykładowej aplikacji.

Skomplikowane testy

Na koniec, gdy trzeba wykonać całościowe testy i wdrożyć gotową aplikację, należy rozważyć kilka kwestii. Aby proces rozwoju aplikacji był sprawny, musi być spójny i elastyczny. Musi istnieć możliwość korzystania z całej aplikacji w trakcie jej rozwijania. Testując jeden jej fragment, nie można mieć pewności, że całość będzie działać poprawnie.

Na szczęście istnieje wiele narzędzi ułatwiających wdrażanie aplikacji złożonych z wielu komponentów, o czym dowiesz się w dalszej części książki. Rozwój tego rodzaju narzędzi prawdopodobnie przyczynił się do wzrostu popularności mikrousług i vice versa.

> Architektura mikrousługowa napędza rozwój narzędzi do wdrażania aplikacji, a narzędzia te ułatwiają adaptację architektury mikrousługowej.

Wady mikrousług można podsumować następująco:

- Nieprzemyślany podział aplikacji na mikrousługi może być przyczyną problemów z architekturą aplikacji.
- Komunikacja sieciowa pomiędzy mikrousługami dodatkowo obciąża system i jest kolejnym potencjalnym słabym punktem aplikacji.
- Testowanie i wdrażanie mikrousług może być skomplikowanym procesem.
- Największym wyzwaniem są trudności we współdzieleniu danych przez mikrousługi.

Na razie nie przejmuj się zbytnio wadami opisanymi w tej części rozdziału.

Powyższe wnioski mogą przytłaczać, a tradycyjne monolityczne podejście wydawać się bezpiecznym rozwiązaniem. Jednak w dłuższej perspektywie dzięki podzieleniu aplikacji na mikrousługi praca programistów i administratorów będzie łatwiejsza.

Implementacja mikrousług w języku Python

Python jest wyjątkowo wszechstronnym językiem. Jak zapewne wiesz, jest on wykorzystywany do tworzenia różnego rodzaju aplikacji, od prostych skryptów systemowych wykonywanych na serwerze po skomplikowane obiektowe aplikacje obsługujące miliony użytkowników.

Badania przeprowadzone w 2014 r. przez Philipa Guo, opublikowane na stronie stowarzyszenia Association for Computing Machinery (ACM), wykazały, że na amerykańskich uniwersytetach Python jest językiem znacznie częściej wykorzystywanym niż Java i jest najczęściej stosowany w nauczaniu informatyki.

Ten trend jest również widoczny w branży rozwoju oprogramowania. Python znajduje się w pierwszej piątce języków w rankingu TIOBE (*http://www.tiobe.com/tiobe-index*), a w świecie aplikacji WWW jest prawdopodobnie jeszcze bardziej popularny, ponieważ coraz rzadziej tworzy się aplikacje w językach takich jak C.

> W tej książce przyjęte jest założenie, że znasz język Python. Jeżeli nie masz jeszcze w tej dziedzinie doświadczenia, polecam Ci książkę *Profesjonalne programowanie w Pythonie. Poziom ekspert*, z której nauczysz się zaawansowanych technik programowania w języku Python.

Niektórzy programiści mają jednak krytyczną opinię o języku Python, ponieważ nie jest on przystosowany do tworzenia wydajnych usług WWW, a napisany w nim kod działa powoli. Bezsprzecznie kod utworzony w tym języku działa powoli, niemniej jednak jest to właściwy język do tworzenia mikrousług i wiele dużych firm programistycznych z powodzeniem go w tym celu wykorzystuje.

Ta część rozdziału zawiera podstawowe informacje o różnych sposobach kodowania mikrousług w języku Python, o synchronicznym i asynchronicznym wywoływaniu procedur oraz kilka szczegółowych uwag dotyczących wydajności kodu.

Ta sekcja rozdziału składa się z pięciu części:

- Standard WSGI
- Biblioteki Greenlet i Gevent
- Platformy Twisted i Tornado
- Moduł asyncio
- Wydajność kodu

Standard WSGI

Większość programistów stron WWW znajduje się pod wrażeniem łatwości, z jaką tworzy się i uruchamia aplikacje WWW w języku Python.

Społeczność użytkowników tego języka inspirowana standardem CGI (ang. *Common Gateway Interface*, wspólny interfejs bramowy) opracowała standard WSGI (ang. *Web Server Gateway Interface*, interfejs bramowy serwera WWW), który znacznie uprościł tworzenie aplikacji obsługujących zapytania HTTP. Kod wykorzystujący ten standard można uruchamiać na serwerach WWW takich jak Apache lub Nginx wyposażonych w rozszerzenia `uwsgi` lub `mod_wsgi`.

Aplikacja WWW musi również odbierać żądania i wysyłać odpowiedzi w formacie JSON. Operacje te są zaimplementowane w standardowej bibliotece języka Python.

Kod w pełni funkcjonalnej mikrousługi zwracającej lokalny czas serwera, wykorzystujący zwykłe biblioteki Python, składa się z niecałych 10 wierszy (patrz niżej:

```
import json
import time
def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    return [bytes(json.dumps({'time': time.time()}), 'utf8')]
```

Interfejs WSGI już od momentu opracowania stał się podstawowym standardem szeroko stosowanym przez społeczność użytkowników języka Python. Programiści wykorzystują go do tworzenia oprogramowania pośredniczącego i oferującego funkcje, które mogą być wykorzystywane przez aplikacje do realizowania różnych operacji w swoich środowiskach.

Niektóre platformy do tworzenia stron WWW, np. Bottle (*http://bottlepy.org*), zostały specjalnie oparte na tym standardzie i wkrótce za pomocą interfejsu WSGI będzie można korzystać ze wszystkich platform.

Mankamentem interfejsu WSGI jest jednak jego synchroniczna natura. Funkcję w powyższym kodzie można wywoływać dokładnie tylko raz po odebraniu zapytania, a odpowiedź na nie można wysłać dopiero wtedy, gdy funkcja zakończy działanie. Oznacza to, że przy każdym wywołaniu tej funkcji wykonywanie głównego kodu aplikacji jest wstrzymywane do czasu, aż funkcja zwróci wynik.

W podejściu mikrousługowym kod musi cały czas oczekiwać na odpowiedzi od różnych zasobów sieciowych. Innymi słowy, aplikacja pozostaje bezczynna i blokuje obsługę użytkownika do czasu przygotowania pełnej odpowiedzi.

Jest to zupełnie poprawne podejście w przypadku interfejsu HTTP API. Nie mówimy tu o aplikacjach realizujących dwukierunkową komunikację, opartą na przykład na gniazdach sieciowych. Co się jednak stanie, gdy aplikacja odbierze kilka zapytań jednocześnie?

Serwer WSGI pozwala uruchamiać kilka wątków jednocześnie, co umożliwia równoległe obsługiwanie zapytań. Ale nie można uruchamiać tysięcy wątków, ponieważ ich pula się wyczerpie i kolejne zapytanie zostanie wstrzymane, mimo że mikrousługa nie wykonuje żadnych

operacji i czeka na odpowiedź innej usługi w tle. Ten problem jest jedną z przyczyn ogromnej popularności platform, które nie wykorzystują standardu WSGI. Są to w pełni asynchroniczne platformy Twisted i Tornado, a w świecie JavaScript jest to Node.js.

W kodzie opartym na platformie Twisted wykorzystywane są funkcje zwrotne wstrzymujące i wznawiające pracę podczas przygotowywania odpowiedzi na zapytanie. Oznacza to, że aplikacja może odbierać i przetwarzać kolejne zapytania. Ten model radykalnie skraca okresy bezczynności procesu, a aplikacja może obsługiwać tysiące jednoczesnych zapytań. Oczywiście, nie oznacza to, że szybciej wtedy odpowiada na zapytania, ale że jeden proces może odbierać więcej jednoczesnych zapytań i odpowiadać na nie w miarę otrzymywania niezbędnych danych.

Nie istnieje jeden prosty sposób zaimplementowania podobnej funkcjonalności z wykorzystaniem standardu WSGI. Społeczność programistów bez powodzenia debatowała latami nad znalezieniem konsensusu. Prawdopodobnie standard ten zostanie ostatecznie porzucony na rzecz jakiegoś innego rozwiązania.

Na razie wciąż można tworzyć mikrousługi wykorzystujące platformy synchroniczne. Jest to całkowicie poprawne podejście, jeżeli aplikacja dopuszcza warunek *jedno zapytanie = jeden wątek*, typowy dla standardu WSGI.

Można jednak zastosować pewną sztuczkę przyspieszającą działanie aplikacji synchronicznych: użyć biblioteki Greenlet opisanej w następnej części rozdziału.

Biblioteki Greenlet i Gevent

Podstawową zasadą programowania asynchronicznego jest symulowanie równoległości procesów poprzez uruchamianie ich w wielu kontekstach jednocześnie.

Aplikacja asynchroniczna wykorzystuje pętlę zdarzeń, która wstrzymuje i wznawia konteksty wykonawcze, gdy pojawi się jakieś zdarzenie. W danej chwili aktywny może być tylko jeden kontekst. Jawne instrukcje w kodzie określają, kiedy pętla ma wstrzymać działanie i proces może podjąć zawieszone wcześniej zadania. Przechodzenie z jednego kontekstu wykonawczego do innego nosi nazwę **przełączania kontekstów**.

Biblioteka Greenlet (*https://github.com/python-greenlet/greenlet*) jest pakietem utworzonym w języku Stackless (szczególnej implementacji CPython) oferującym tzw. **greenlety**.

Greenlety są to pseudowątki, których instancje — w odróżnieniu od prawdziwych wątków — tworzy się bardzo łatwo i które można wykorzystywać do wywoływania funkcji w języku Python. Funkcje te mogą przełączać konteksty i przekazywać sterowanie innym funkcjom. Przełączanie odbywa się w pętli zdarzeń. W ten sposób można tworzyć asynchroniczne aplikacje podobne do aplikacji wielowątkowych.

Poniżej przedstawiony jest przykładowy kod z dokumentacji biblioteki Greenlet:

```
from greenlet import greenlet
def test1(x, y):
    z = gr2.switch(x+y)
    print(z)
def test2(u):
    print (u)
    gr1.switch(42)
gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch("Witaj,", " świecie!")
```

Ten kod jawnie przełącza między sobą dwa greenlety.

Kod mikrousługi wykorzystujący standard WSGI i greenlety może obsługiwać wiele żądań jednocześnie i przełączać się pomiędzy nimi, podobnie jak w przypadku zapytań wejścia/wyjścia, jeżeli obsługa któregoś z zapytań zostanie zablokowana.

Jednak przełączanie pomiędzy greenletami musi odbywać się jawnie, a to powoduje, że kod szybko się komplikuje i staje się nieczytelny. W tym momencie bardzo przydaje się biblioteka Gevent.

Biblioteka Gevent (*http://www.gevent.org*) jest oparta na bibliotece Greenlet i oferuje między innymi możliwość automatycznego przełączania kodu między greenletami. Oferuje również specjalny moduł `socket` wykorzystujący greenlety do automatycznego wstrzymywania i wznawiania kodu, gdy w gnieździe pojawią się dane. Dostępna jest również funkcjonalność automatycznego zastępowania standardowego gniazda jego analogiczną wersją. Po wpisaniu jednego dodatkowego wiersza wykorzystujący gniazda kod synchroniczny w magiczny sposób zamienia się w kod asynchroniczny (jak poniżej):

```
from gevent import monkey; monkey.patch_all()
def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    # ...Kod obsługujący gniazdo...
    return result
```

Ta magia ma jednak swoją cenę. Aby biblioteka Gevent działała poprawnie, cały wykorzystujący ją kod musi być kompatybilny z jej wersją. Z tego powodu niektóre pakiety rozwijane przez społeczność programistów blokują się, a nawet zwracają nieoczekiwane wyniki. Dotyczy to w szczególności pakietów wykorzystujących rozszerzenia w języku C omijające standardowe funkcjonalności biblioteki Gevent.

Jednak w większości przypadków biblioteka ta działa poprawnie. Projekty na niej oparte są oznaczane jako zielone. Jeśli biblioteka nie działa poprawnie, wtedy zazwyczaj autor poprawia ją na życzenie społeczności użytkowników. Tak było na przykład w przypadku skalowania usługi Firefox Sync w Mozilli.

Platformy Twisted i Tornado

Jeżeli tworzysz aplikację złożoną z mikrousług i ważną kwestią jest duża liczba jednoczesnych zapytań do obsłużenia, warto rozważyć odejście od standardu WSGI na rzecz asynchronicznej platformy, na przykład Tornado (*http://www.tornadoweb.org*) lub Twisted (*https://twistedmatrix.com/trac*).

Platforma Twisted istnieje od dawna. Kod implementujący tę samą mikrousługę co poprzednio jest nieco bardziej rozbudowany (jak poniżej):

```
import time
import json
from twisted.web import server, resource
from twisted.internet import reactor, endpoints
class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
        request.responseHeaders.addRawHeader(b"content-type",
                                             b"application/json")
        return bytes(json.dumps({'time': time.time()}), 'utf8')
    site = server.Site(Simple())
    endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
    endpoint.listen(site)
    reactor.run()
```

Platforma Twisted jest wyjątkowo spójna i wydajna, jednak ma kilka mankamentów ujawniających się podczas tworzenia aplikacji WWW, m.in.:

- Każdy punkt końcowy mikrousługi trzeba implementować za pomocą klasy pochodnej od `Resource` i kodować wszystkie jej metody. W przypadku prostej mikrousługi trzeba z tego powodu napisać mnóstwo zbędnego kodu interfejsu API.
- Kod wykorzystujący tę platformę jest mało czytelny i trudny w diagnozowaniu ze względu na jego asynchroniczną naturę.
- W przypadku długiego łańcucha funkcji zwrotnych wywoływanych jedna po drugiej łatwo jest wpaść w tzw. **piekło wywołań zwrotnych**, a ponadto kod staje się zagmatwany.
- Trudno jest przetestować aplikację i do tego celu trzeba używać specjalnego modułu do testów jednostkowych.

Platforma Tornado jest oparta na podobnym modelu, jednak w niektórych obszarach sprawdza się lepiej. Jest wyposażona w lepszy system kierowania zapytań, a jej kod jest bardziej podobny do zwykłego kodu Python. Platforma Tornado wykorzystuje jednak model wywołań zwrotnych, co utrudnia diagnostykę kodu.

Jednak zespoły rozwijające obie platformy intensywnie pracują nad usunięciem niedoskonałości, aby można było korzystać z funkcjonalności asynchronicznych wprowadzonych w języku Python 3.

Moduł asyncio

Gdy Guido van Rossum rozpoczął pracę nad wzbogaceniem języka Python 3 o asynchroniczność, część społeczności użytkowników zaczęła nawoływać do stosowania rozwiązań takich jak biblioteka Gevent, ponieważ o wiele rozsądniej było tworzyć aplikacje działające w synchroniczny, sekwencyjny sposób, niż stosować jawne wywołania zwrotne, jak w bibliotekach Tornado i Twisted.

Guido zdecydował się jednak na jawną technikę i rozpoczął eksperymentalny projekt Tulip inspirowany platformą Twisted. W ten sposób powstał moduł `asyncio`, który został dodany do języka Python.

Patrząc z perspektywy czasu, implementacja jawnej pętli zdarzeń jest znacznie lepszym rozwiązaniem od mechanizmu oferowanego przez bibliotekę Gevent. Sposób zakodowania przez twórców języka Python modułu `asyncio` i eleganckie rozszerzenie go o instrukcje `async` i `await` implementujące koprocedury sprawiły, że asynchroniczny kod napisany w podstawowym języku Python 3.5 jest bardzo czytelny i podobny do kodu synchronicznego.

> Koprocedury są to funkcje wstrzymujące i wznawiające swoje działanie. Rozdział 12, „Co dalej?" szczegółowo opisuje ich implementację i wykorzystanie w kodzie Python.

Dzięki temu doskonałemu rozwiązaniu można uniknąć bałaganu w asynchronicznym kodzie, jaki czasami pojawia się w aplikacjach opartych na platformach Node.js lub Twisted (Python 2).

Oprócz koprocedur moduł `asyncio` oferuje pełny zestaw funkcji i klas pomocniczych do tworzenia aplikacji asynchronicznych. Ich opis jest dostępny na stronie *https://docs.python.org/3/library/asyncio.html*.

Obecnie Python jest równie ekspresyjnym językiem jak Lua, umożliwiającym tworzenie aplikacji opartych na koprocedurach. Ponadto pojawiło się kilka platform wykorzystujących nowe funkcjonalności, z których można korzystać tylko w wersji języka Python 3.5 lub nowszej. Jedną z nich jest `aiohttp` (*http://aiohttp.readthedocs.io*). W pełni asynchroniczny kod tej samej mikrousługi co poprzednio składa się zaledwie z kilku eleganckich wierszy:

```
from aiohttp import web
import time
async def handle(request):
    return web.json_response({'time': time.time()})
if __name__ == '__main__':
    app = web.Application()
    app.router.add_get('/', handle)
    web.run_app(app)
```

Ten krótki kod jest bardzo podobny do swojej synchronicznej wersji. Jedyną różnicą jest słowo kluczowe `async` definiujące funkcję jako koprocedurę.

W ten właśnie sposób należy kodować w języku Python każdą asynchroniczną aplikację. Poniżej przedstawiony jest przykładowy kod z dokumentacji projektu, wykorzystujący biblioteki `aiopg` i `asyncio` do obsługi bazy danych PostgreSQL:

```
import asyncio
import aiopg
dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'
async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]
loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

Poza słowami kluczowymi `async` i `await` funkcja wysyłająca zapytanie SQL i zwracająca wynik jest bardzo podobna do swojej synchronicznej wersji.

Asynchroniczne platformy i biblioteki dla języka Python 3 znajdują się wciąż na etapie rozwoju. Do zakodowania potrzebnych funkcjonalności trzeba stosować konkretne implementacje modułów `asyncio` i `aiohttp`.

W przypadku użycia w asynchronicznym kodzie biblioteki synchronicznej trzeba wykonać dodatkową i niełatwą pracę, aby uniknąć blokowania pętli zdarzeń. Jeżeli mikrousługa korzysta z niewielu zasobów, takie rozwiązanie jest do przyjęcia. Jednak na razie bezpieczniej jest korzystać z synchronicznych platform, które istnieją już od dłuższego czasu, niż z platform asynchronicznych. Lepiej jest używać dojrzałych pakietów i zaczekać, aż moduł `asyncio` będzie bardziej zaawansowany. A obecnie do tworzenia mikrousług w języku Python dostępnych jest wiele doskonałych synchronicznych platform, m.in. Bottle, Pyramid z Cornice i Flask.

> Istnieje duże prawdopodobieństwo, że w drugim wydaniu tej książki zostanie opisana platforma asynchroniczna. W tym wydaniu jednak jest opisana platforma Flask, która istnieje od dłuższego czasu, jest bardzo spójna i dojrzała. Pamiętaj jednak, że przedstawione tu przykłady można dostosować do dowolnej wersji języka Python i platformy. Jest tak dlatego, ponieważ mikrousługi są w większości napisane w kodzie bardzo podobnym do zwykłego kodu w języku Python. Platforma Flask jest najczęściej wykorzystywana do kierowania zapytań, a ponadto oferuje kilka funkcji pomocniczych.

Wydajność kodu

W poprzedniej części rozdziału poznałeś dwa różne sposoby kodowania mikrousług: synchroniczny i asynchroniczny. Niezależnie od użytego sposobu bezpośredni wpływ na wydajność mikrousługi ma szybkość języka Python.

Oczywiście wszyscy wiedzą, że kod Python jest wolniejszy niż kod Java lub Go, ale szybkość nie zawsze jest kwestią priorytetową. Mikrousługa jest często niewielkim kodem, który przez większość czasu czeka na odpowiedzi od innych usług. Szybkość takiego kodu ma zazwyczaj mniejsze znaczenie od szybkości przetwarzania zapytań SQL przez bazę PostgreSQL, ponieważ większość czasu potrzebnego na przygotowanie odpowiedzi zajmują operacje w bazie danych. Niemniej jednak ważne jest, aby aplikacja działała możliwie jak najszybciej.

Jednym z najbardziej gorących tematów dyskutowanych w społeczności użytkowników języka Python jest blokada GIL (ang. *Global Interpreter Lock*, globalna blokada interpretera), która może zniweczyć wydajność kodu ze względu na brak możliwości wykorzystania kilku procesów w aplikacji wielowątkowej.

Istnienie blokady GIL jest uzasadnione. Chroni ona te części interpretera CPython, które nie są wielowątkowe. Takie blokady są stosowane również w innych językach, na przykład Ruby. Jak dotąd wszystkie próby usunięcia tej blokady, mające na celu przyspieszenie interpretera CPython, zakończyły się niepowodzeniem.

> Larry Hasting pracuje nad projektem Gilectomy (*https://github.com/larryhastings/gilectomy*). Jest to odmiana projektu CPython pozbawiona blokady GIL. Podstawowym celem jest zaimplementowanie języka bez blokady, w którym kod będzie działał równie szybko jak w interpreterze CPython. W chwili pisania tej książki implementacja jest wciąż wolniejsza od CPython. Warto jednak śledzić ten projekt i czekać na dzień, w którym kod osiągnie wymaganą szybkość. Wersja interpretera CPython bez blokady GIL byłaby bardzo atrakcyjna.

Blokada GIL nie tylko uniemożliwia korzystanie z wielu rdzeni w jednym procesie, ale również obniża wydajność kodu obciążonego częstymi wywołaniami systemowymi.

Jednak ogólna ocena blokady GIL jest korzystna. W ostatnich latach włożono wiele pracy w zmniejszenie jej wpływu na interpreter i w niektórych obszarach wydajność kodu Python radykalnie wzrosła.

Należy pamiętać, że nawet gdy twórcy języka Python usuną blokadę GIL, dalej będzie to język interpretowany, wyposażony w mechanizm porządkowania pamięci. Te dwie cechy powodują, że jego wydajność będzie zawsze mniejsza niż w przypadku innych języków.

Dla użytkowników zainteresowanych dekomponowaniem kodu przez interpreter dostępny jest moduł `dis`. Poniższy listing pokazuje, jak interpreter zdekomponował prostą funkcję zwiększającą wartość zmiennej w 29 krokach:

```
>>> def myfunc(data):
...     for value in data:
...         yield value + 1
...
>>> import dis
>>> dis.dis(myfunc)
  2           0 SETUP_LOOP              23 (to 26)
              3 LOAD_FAST                0 (data)
              6 GET_ITER
        >>    7 FOR_ITER                15 (to 25)
             10 STORE_FAST               1 (value)
  3          13 LOAD_FAST                1 (value)
             16 LOAD_CONST               1 (1)
             19 BINARY_ADD
             20 YIELD_VALUE
             21 POP_TOP
             22 JUMP_ABSOLUTE        7
        >>   25 POP_BLOCK
        >>   26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

Funkcja napisana w kompilowanym języku i zwracająca dokładnie ten sam wynik wymagałaby wykonania znacznie mniejszej liczby operacji.

Istnieją jednak pewne metody przyspieszenia kodu Python. Jednym z nich jest napisanie rozszerzeń kodu w języku C lub wykorzystanie statycznych rozszerzeń, np. Cython (*http://cython.org*). W ten sposób jednak kod bardziej się skomplikuje.

Innym, bardziej obiecującym sposobem jest uruchamianie aplikacji w środowisku PyPy (*http://pypy.org*). Jest to kompilator JIT (ang. *just-in-time*, dokładnie na czas), który na bieżąco zamienia fragmenty kodu w języku Python na kod maszynowy wykonywany bezpośrednio przez procesor. Cała magia tego środowiska polega na wykrywaniu na bieżąco, kiedy i jak należy kompilować kod przed jego wykonaniem.

Wprawdzie kompilator PyPy jest bardziej przystosowany do starszych wersji języka Python niż interpreter CPython, osiągnął jednak etap rozwoju pozwalający na wykorzystanie go w środowisku produkcyjnym, a jego wydajność robi wrażenie. Jeżeli u nas, w firmie Mozilla, jakiś kod wymaga szybkiego wykonania, korzystamy ze środowiska PyPy, które jest niemal tak szybkie jak kompilator Go. Z tego powodu zamiast Go używamy języka Python.

> Strona *PyPy Speed Center* (*http://speed.pypy.org*) zawiera doskonałe porównanie szybkości kompilatora PyPy i interpretera CPython.

Jeżeli jednak korzystasz z rozszerzeń napisanych w języku C, będziesz musiał je rozkodować, co może być problemem, szczególnie w przypadku gdy stosują je inni programiści. Jeżeli jednak utworzysz mikrousługę, wykorzystując standardowe biblioteki, wtedy z dużym prawdopodobieństwem będzie ona działała w środowisku PyPy bez wprowadzania zmian. Warto zatem spróbować.

W większości projektów zalety języka Python i całego jego ekosystemu przeważają nad opisanymi tu kwestiami wydajnościowymi, ponieważ obciążenie mikrousługi rzadko jest problemem. Jeżeli problemem jest wydajność, wtedy dzięki podejściu mikrousługowemu można krytyczne elementy aplikacji napisać w innym języku, co nie wpłynie na działanie pozostałej części aplikacji.

Podsumowanie

Ten rozdział zawiera porównanie monolitycznego i mikrousługowego podejścia w tworzeniu aplikacji WWW. Jak się przekonałeś, wybór modelu, od którego należy zacząć i przy nim trwać, nie jest oczywisty.

Traktuj podejście mikrousługowe jako sposób na usprawnienie działania monolitycznej aplikacji. W miarę dojrzewania projektu jego fragmenty powinny migrować w kierunku mikrousług. Jest to praktyczne podejście, o czym przekonałeś się w tym rozdziale, ale należy stosować je ostrożnie, aby nie wpaść w jedną z typowych pułapek.

Ważny wniosek jest również taki, że Python jest uważany za jeden z najlepszych języków do tworzenia aplikacji WWW, a więc również mikrousług, tym bardziej że dostępnych jest mnóstwo platform i bibliotek ułatwiających to zadanie.

W tym rozdziale opisano ogólnie kilka platform synchronicznych i asynchronicznych. W pozostałej części książki będzie wykorzystywana platforma Flask. Następny rozdział stanowi wprowadzenie do tej fantastycznej platformy. Gdy ją poznasz, na pewno ją polubisz.

Python jest powolnym językiem, co w pewnych szczególnych przypadkach może być problemem. Jednak znajomość przyczyn jego powolności i różnych sposobów radzenia sobie z nią zazwyczaj wystarcza do rozwiązania problemu.

Skorowidz

L

M

N

O

P

R

S

T

U

W

Z

Mikrousługi w języku Python: integracja doskonała!

Mikrousługi są bardzo ciekawym trendem w tworzeniu kodu. Pojawił się on kilka lat temu z uwagi na potrzebę przyspieszenia cyklu udostępniania oprogramowania. Nowe produkty i funkcje musiały być oferowane użytkownikom możliwie najszybciej. Wkrótce okazało się, że tworzenie architektury aplikacji składającej się z małych, funkcjonalnych jednostek — właśnie mikrousług — jest bardzo obiecującym sposobem pracy. Pozwala na zwiększenie elastyczności oraz szybkości wprowadzania innowacji, gdyż programista może zająć się jednym elementem bez zastanawiania się nad całością aplikacji. W świecie, w którym rządzą wydajność i krótki czas dostarczania kodu, jest to duża wartość!

Dzięki tej książce dowiesz się, w jaki sposób niewielkie, standardowe elementy kodu mogą złożyć się na kompletną, działającą aplikację. Nauczysz się tworzyć takie mikrousługi, rozwiązywać pojawiające się problemy i wykształcisz nawyk stosowania dobrych praktyk. Szybko zaczniesz pisać aplikacje w Pythonie za pomocą całego wachlarza dostępnych narzędzi, włączając w to Flask czy Tox. Przy okazji nauczysz się zasad programowania zorientowanego na testy. Dowiesz się, jak zabezpieczać komunikację pomiędzy usługami i kodować funkcjonalności zapory aplikacyjnej w języku Lua dla serwera Nginx. Poznasz też możliwości instalowania mikrousług w chmurze AWS z wykorzystaniem kontenerów Docker.

W tej książce między innymi:

- mikrousługi i ich rola w tworzeniu nowoczesnych aplikacji WWW
- cykl tworzenia kodu pod kątem testów i ciągłej integracji
- monitorowanie i zabezpieczanie mikrousług
- tworzenie mikrousług w JavaScripie
- budowa mikrousług niezależnie od operatorów chmurowych i technologii wirtualizacyjnych
- unikanie problemów wynikających z centralizacji aplikacji

Tarek Ziadé specjalizuje się w programowaniu w Pythonie. Pracuje w firmie Mozilla, w zespole zajmującym się usługami. Jest założycielem grupy AFPy zrzeszającej francuskich programistów Pythona, napisał również kilka książek poświęconych temu językowi. Mieszka we Francji, niedaleko Dijon. Kiedy nie tworzy kodu i nie spędza czasu ze swoją rodziną, oddaje się dwóm pasjom: bieganiu i grze na trąbce.

ISBN 978-83-283-4596-6
9 788328 345966
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł
Packt