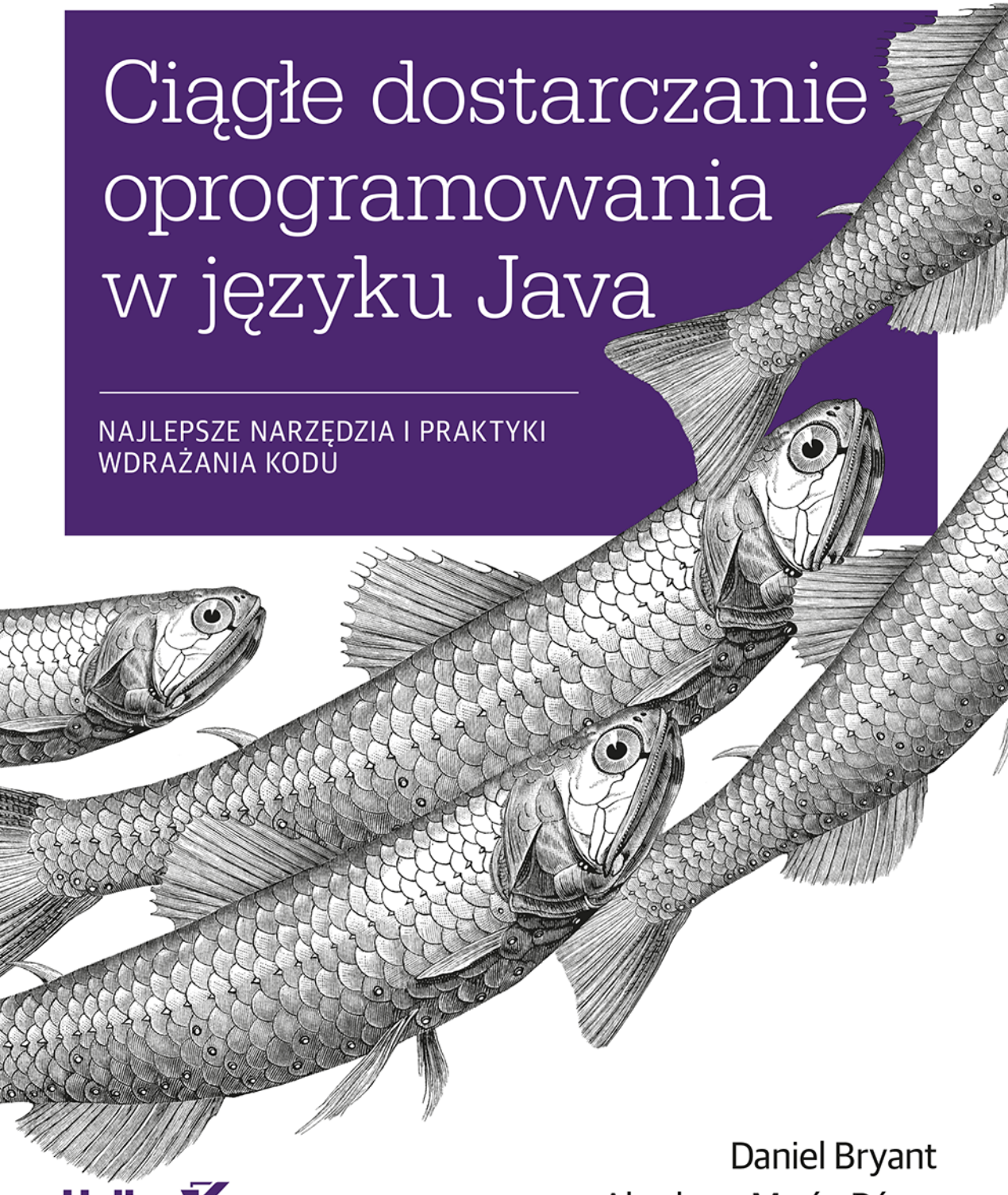
O'REILLY®

Ciągłe dostarczanie oprogramowania w języku Java

NAJLEPSZE NARZĘDZIA I PRAKTYKI WDRAŻANIA KODU



Daniel Bryant
Abraham Marín-Pérez

Helion

Tytuł oryginału: Continuous Delivery in Java: Essential Tools and Best Practices for Deploying Code to Production

Tłumaczenie: Krzysztof Bąbol (rozdz. 1 – 5), Andrzej Watrak (wstęp, rozdz. 8 – 13), Lech Lachowski (rozdz. 6, 7, 14, 15)

ISBN: 978-83-283-5633-7

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 8.

Praca w lokalnym odpowiedniku środowiska produkcyjnego

Zanim zaczniesz budować proces ciągłego dostarczania oprogramowania, musisz sprawdzić, czy jesteś w stanie skutecznie oraz wydajnie pracować nad kodem i systemami na lokalnym komputerze. W tym rozdziale opisano kilka związanych z tym problemów, pojawiających się szczególnie w nowoczesnych systemach rozproszonych o architekturze opartej na usługach. Przedstawione zostały również techniki, takie jak imitowanie (ang. *mocking*) usług, wirtualizacja usług, wirtualizacja infrastruktury (zarówno za pomocą maszyn wirtualnych, jak i kontenerów) oraz lokalne rozwijanie aplikacji FaaS.

Wyzwania związane z lokalnym tworzeniem oprogramowania

Jako programista Java zapewne tradycyjną monolityczną aplikację WWW zazwyczaj rozwijasz w lokalnym, prosto skonfigurowanym środowisku. Jego przygotowanie najczęściej polega na zainstalowaniu systemu operacyjnego, pakietu JDK, narzędzi kompilacyjnych (Maven lub Gradle) oraz środowiska IDE, np. IntelliJ IDEA lub Eclipse. Czasami potrzebne jest dodatkowo oprogramowanie pośredniczące, baza danych lub serwer aplikacyjny. Takie środowisko sprawdza się w przypadku pojedynczej aplikacji Java, ale co robić, gdy trzeba zbudować system składający się z wielu usług, przeznaczony do uruchomienia w chmurze, na platformie kontenerowej lub bezserwerowej?

Kiedy przymierzasz się do tworzenia aplikacji opartej na wielu usługach, najbardziej logiczną decyzją jest zaadaptowanie lokalnych praktyk programistycznych do pracy nad każdą nową usługą. Jednak — jak to zwykle bywa w IT — ręcznie replikując środowisko, nie zajdziesz daleko. Największy problem z takim podejściem jest związany z kosztami integracji i testów oprogramowania. Jeżeli nawet każdą usługę można przetestować pod kątem integracji z innymi komponentami, to trudno zainicjować i skoordynować konfigurację testową, gdy usług jest więcej niż kilka. W takim przypadku trzeba utworzyć lokalną replikę zewnętrznej usługi (klonując ją z repozytorium systemu kontroli wersji, kompilując i uruchamiając), manipulować jej stanem, wykonywać testy i na koniec weryfikować jej stan po uruchomieniu w zewnętrznym systemie.

Problemy z lokalnymi skryptami konfiguracyjnymi

W przeszłości często spotykaliśmy programistów, którzy starali się rozwiązywać problemy z lokalną inicjacją środowiska poprzez tworzenie prostych skryptów (Bash, Groovy itp.) wiążących wszystkie narzędzia i inicjujące dane testowe. Z naszego doświadczenia wynika, że takie skrypty szybko stają się prawdziwą udręką, ponieważ trudno je utrzymywać. Dlatego nie zalecamy takiego podejścia. Wspominamy o nim tylko dlatego, ponieważ stanowi punkt wyjścia do dalszej dyskusji.

Imitacje, atrapy i wirtualizacja usług

Podstawowym podejściem, jakie możesz zastosować do skalowania lokalnego środowiska roboczego, jest imitowanie usług. To technika znana wielu programistom. W tym podrozdziale dowiesz się, jak ją najlepiej wykorzystać. Poznasz również inną, nie tak szeroko stosowaną technikę, ale bardzo przydatną podczas pracy z wieloma zewnętrznymi usługami i interfejsami API, czyli wirtualizację.

Wzorzec 1.: profile, imitacje i atrapy

Jeżeli potrafisz tworzyć kod za pomocą platformy Spring opartej na maszynach JVM, na pewno nieobce jest Ci pojęcie *profilu*. Profil to zestaw różnych ustawień konfiguracyjnych, które aplikuje się podczas kompilowania lub uruchamiania kodu. Za pomocą profili nim można tworzyć imitacje lub atrapy interfejsów zewnętrznych usług i wykorzystywać je w lokalnym środowisku. W zależności od potrzeb profile można przełączać i korzystać z usług lokalnych lub produkcyjnych. Takie podejście stosuje się np. podczas rozwijania usługi *witryna-sklepu* uzależnionej od usługi *wyszukiwanie-produktu*. Interfejs tej ostatniej usługi jest ściśle zdefiniowany, więc można utworzyć kilka opisanych niżej profili i wykorzystywać je do automatycznego testowania kodu za pomocą narzędzia Maven.

bez-wyszukiwania
: Taki profil może zawierać prostą imitację usługi *wyszukiwanie-produktu*. Imitacja ta (utworzona np. za pomocą biblioteki Mockito) nie wykonuje żadnych operacji i zwraca puste wyniki. Przydaje się wtedy, gdy lokalnie rozwijany kod korzysta z usługi *wyszukiwanie-produktu*, ale zwracane przez nią wyniki nie są ważne.

wyszukiwanie-parametryzowane
: Taki profil może zawierać atrapę usługi *wyszukiwanie-produktu*, przy czym atrapę tę można parametryzować tak, aby podczas testów zwracała określone wyniki (np. jeden produkt, dwa produkty, produkt o określonych właściwościach, produkt uszkodzony itp.). Atrapa może być zwykłą klasą Java, która odczytuje z zewnętrznego pliku JSON uprzednio przygotowane wyniki wyszukiwania. Jest to bardzo przydatne podejście, gdy jednak atrapa stanie się bardziej skomplikowana (będzie zawierała mnóstwo instrukcji warunkowych), wtedy warto przyjrzeć się innemu wzorcowi — *wirtualizacji usługi*.

środowisko-produkcyjne
: Ten profil umożliwia korzystanie z rzeczywistej, produkcyjnej usługi *wyszukiwanie-produktu*, implementowanie przetwarzania obiektów, obsługi błędów itp.

Opisany tu wzorzec obejmuje też korzystanie z wbudowanych w proces tymczasowych magazynów danych oraz oprogramowania pośredniczącego, jednak nie są to imitacje ani atrapy. Uruchamiając taki proces, można korzystać z komponentu w taki sam sposób, jak z jego rzeczywistej instancji, przy czym nakład pracy związanej z jego zainicjowaniem i konfiguracją jest znacznie mniejszy.

Zalety wbudowanych baz danych i oprogramowania pośredniczącego

Za pomocą imitacji i atrap można skutecznie testować kod, natomiast w przypadku korzystania z magazynów danych i oprogramowania pośredniczącego trzeba często tworzyć skomplikowane imitacje lub symulować skomplikowane działanie usługi. Jeżeli napotkasz tego rodzaju problemy, zalecamy sprawdzenie, czy magazyn lub oprogramowanie można uruchomić w trybie „wbudowanym" w proces lub pamięć. Można wtedy korzystać z wszystkich funkcjonalności rzeczywistej usługi, ale obciążenie systemu jest znacznie mniejsze niż po uruchomieniu pełnej wersji aplikacji.

Aplikacja w tym trybie zazwyczaj uruchamia się szybciej, a jej czasy odpowiedzi są krótsze (ponieważ dane znajdują się w pamięci i nie trzeba ich odczytywać z dysku). Konfigurację aplikuje się przy każdorazowym uruchamianiu procesu. Wadą tego trybu jest niewielki zbiór danych (który musi zmieścić się w pamięci), a wprowadzane w nim zmiany podczas każdego testu nie są zapisywane w sposób trwały.

Do testowania kodu i tworzenia zautomatyzowanych pakietów testowych z powodzeniem stosowaliśmy następujące wbudowane aplikacje.

- H2 i HSQL jako testowy zamiennik bazy MySQL (pamiętaj o różnicach pomiędzy implementacjami).
- Stubbed Cassandra jako zamiennik Apache Cassandra.
- Baza ElasticSearch uruchomiona jako pojedynczy, wbudowany węzeł.
- Apache Qpid jako wbudowany zamiennik kolejek RabbitMQ i ActiveMQ.
- Platforma Localstack zawierająca wbudowane wersje różnych usług baz danych AWS, m.in. DynamoDB i Kinesis.

Jeżeli wybranej bazy danych lub oprogramowania pośredniczącego nie można uruchomić w trybie wbudowanym w proces lub pamięć albo do testów potrzebne są duże ilości danych, wtedy można zastosować projekt *testcontainers* (*https://www.testcontainers.org*), umieścić wykorzystywane komponenty w kontenerach i uruchamiać je za pomocą narzędzia JUnit.

Imitowanie usług za pomocą biblioteki Mockito

Jedną z najpopularniejszych bibliotek imitacyjnych dla języka Java jest Mockito. Jej najnowsza wersja 2.0+ oferuje elastyczną platformę do weryfikowania interakcji z zależnymi komponentami oraz tworzenia atrap metod.

Weryfikowanie interakcji

Tworząc kod wykorzystujący zewnętrzne komponenty (zależności), często trzeba sprawdzać, czy aplikacja właściwie się z nimi komunikuje, szczególnie w specyficznych scenariuszach (np. pomyślnego lub problematycznego wykonania pewnej operacji). Listing 8.1 przedstawia przykładowy test.

Listing 8.1. Weryfikacja interakcji z klasą List za pomocą atrapy utworzonej z użyciem biblioteki Mockito

```
import static org.mockito.Mockito.*;
// Utworzenie imitacji komponentu.
List mockedList = mock(List.class);
// Zastosowanie obiektu-imitacji, który nie zgłasza wyjątku „nieoczekiwana interakcja”.
mockedList.add("jeden");
mockedList.clear();
// Selektywna, jawna i czytelna weryfikacja.
verify(mockedList).add("jeden");
verify(mockedList).clear();
```

Użyte w kodzie asercje dotyczą działania aplikacji (tj. sprawdzają, czy aplikacja reaguje poprawnie w określonej sytuacji lub w zadanych warunkach).

Tworzenie atrap metod

Oprócz testowania działania aplikacji często trzeba weryfikować wyniki, stany lub dane zwracane przez zewnętrzne usługi lub rozwijane (testowane) metody. Taka sytuacja pojawia się zazwyczaj podczas implementowania skomplikowanego algorytmu lub korzystania z wielu zewnętrznych usług, przy czym nie tyle ważne są poszczególne interakcje, co końcowy wynik. Ilustruje to listing 8.2.

Listing 8.2. Atrapa klasy LinkedList zwracająca wynik

```
// Można tworzyć atrapy nie tylko interfejsów, ale również klas.
LinkedList mockedList = mock(LinkedList.class);
// Atrapę tworzy się przed wywołaniem metody.
when(mockedList.get(0)).thenReturn("pierwszy");
// Poniższa instrukcja zwraca wartość true.
assertThat(mockedList.get(0), is("pierwszy");
// Poniższa instrukcja wyświetla wynik "null", ponieważ metoda get(999) nie ma atrapy.
System.out.println(mockedList.get(999));
```

Jest to prosty przykład asercji weryfikującej wartość (tutaj zwracaną bezpośrednio przez atrapę metody), a nie interakcję.

Kontroluj złożoność imitacji

Jeżeli stwierdzisz, że imitacja coraz gorzej odzwierciedla rzeczywistą aplikację lub usługę albo coraz więcej czasu poświęcasz na jej tworzenie, oznacza to, że jest ona zbyt skomplikowana i powinieneś zastosować inną technikę lub narzędzie. Pamiętaj, że oprogramowanie jest dla człowieka, a nie odwrotnie!

W tym rozdziale pokazaliśmy tylko drobny fragment funkcjonalności oferowanych przez bardzo przydatną bibliotekę, jaką jest Mockito.

Wzorzec 2.: wirtualizacja usług i symulacja interfejsu API

Gdy imitacje lub atrapy zewnętrznych usług stają się coraz bardziej skomplikowane, jest to sygnał, że lepszym rozwiązaniem może być wirtualizacja usługi. Jeżeli atrapa zawiera mnóstwo instrukcji warunkowych, staje się źródłem nieporozumień z osobami modyfikującymi przygotowane dane, powoduje, że testy kończą się niepowodzeniem i coraz trudniej ją utrzymywać, znaczy to, że atrapa jest zbyt skomplikowana. Wirtualizacja jest techniką umożliwiającą symulowanie zewnętrznej usługi bez konieczności jej uruchamiania i korzystania z niej. To podejście różni się od uruchamiania usługi wbudowanej w proces lub pamięć, ponieważ wirtualna usługa zachowuje się w uprzednio zadany sposób lub zgodnie z zarejestrowanymi interakcjami aplikacji z rzeczywistą usługą.

Z wykorzystaniem wirtualizacji można efektywniej niż za pomocą imitacji lub atrapy symulować działanie skomplikowanej usługi. Technika ta z powodzeniem sprawdza się w różnych scenariuszach, np. gdy usługa zwraca skomplikowane dane (lub ich duże ilości), gdy nie ma do niej dostępu (jest świadczona przez zewnętrzny podmiot lub jest to usługa SaaS) lub gdy komunikuje się z nią wiele innych usług i łatwiej udostępniać wirtualną usługę niż tworzyć imitacje lub atrapy kodu.

Do wirtualizowania usług służą następujące narzędzia.

Mountebank
: Jest to aplikacja JavaScript/Node.js stanowiąca według jej twórców „niezależną od systemu operacyjnego, wieloprotokołową platformę do testowania aplikacji w locie". Można ją stosować do wirtualizowania usług wykorzystujących protokoły HTTP, HTTPS, TCP i SNMP. Platforma ma prosty w użyciu interfejs API i choć czasami wymaga napisania dość rozbudowanego kodu, można za jej pomocą łatwo symulować skomplikowane, zwirtualizowane odpowiedzi danej usługi.

WireMock
: To narzędzie jest podobne do Mountebank. Przy jego użyciu można utworzyć serwer (np. HTTP) o szerokim spektrum wirtualnych odpowiedzi. Narzędzie WireMock napisał w języku Java Tom Akehurst, który cały czas je starannie serwisuje.

Stubby4j
: Jest to narzędzie dla języka Java, bardzo podobne do Mountebank i WireMock. Istnieje już od dłuższego czasu i umożliwia symulowanie skomplikowanych komunikatów SOAP (ang. *Simple Object Access Protocol*, prosty protokół udostępniania obiektów) i WSDL (ang. *Web Services Description Language*, język opisu usług WWW) wykorzystywanych przez starsze, zewnętrzne usługi.

VCR i *Betamax*
: Oba narzędzia są przydatnymi aplikacjami do rejestrowania i odtwarzania ruchu sieciowego. Sprawdzają się szczególnie w sytuacjach, gdy nie ma dostępu do kodu zewnętrznej usługi (a więc kiedy można jedynie obserwować odpowiedzi usługi na odbierane zapytania), gdy usługa zwraca duże ilości danych (które można zarejestrować na zewnętrznym nośniku) albo gdy korzystanie z usługi jest ograniczone lub drogie.

Hoverfly

Jest to nowe narzędzie wirtualizacyjne oferujące więcej opcji konfiguracyjnych niż WireMock czy VCR i umożliwiające symulowanie odpowiedzi wysyłanych przez starsze usługi oraz aplikacje o skomplikowanych strukturach wzajemnie od siebie zależnych mikrousług. Narzędzia Hoverfly można również używać do przeprowadzania testów obciążeniowych aplikacji wykorzystującej zewnętrzną krytyczną usługę, np. SaaS, która w przypadku zwiększonej liczby zapytań staje się słabym punktem całego systemu. Narzędzie jest napisane w języku Go, co oznacza, że jest niewielkie i bardzo wydajne. Uruchomione na niewielkim węźle AWS EC2 może z łatwością obsługiwać tysiące zapytań w ciągu sekundy.

Wirtualizacja usług nie jest zbyt popularna wśród programistów, zatem teraz nieco dokładniej opiszemy, jak się ją konfiguruje i z niej korzysta.

Wirtualizacja usług za pomocą narzędzia Hoverfly

W tym punkcie dowiesz się, jak za pomocą narzędzia Hoverfly wirtualizuje się usługi w lokalnym środowisku programistycznym.

Instalacja narzędzia Hoverfly

W systemie macOS narzędzie Hoverfly instaluje się za pomocą menedżera pakietów *brew*. Na stronie *http://bit.ly/2Q8dhVC* dostępne są wersje tego oprogramowania dla systemów Windows i Linux oraz instrukcje instalacyjne.

Możesz również pobrać prostą usługę *flights* opartą na platformie Spring-Boot i użytą niżej do wyjaśnienia idei wirtualizacji.

Rejestrowanie i symulowanie zapytań za pomocą narzędzia Hoverfly

Najpierw uruchom narzędzie Hoverfly w sposób pokazany na listingu 8.3.

Listing 8.3. Uruchomienie narzędzia Hoverfly

```
$ hoverctl start
Hoverfly is now running

+------------+------+
| admin-port | 8888 |
| proxy-port | 8500 |
+------------+------+
```

W każdej chwili z pomocą polecenia `hoverctl status` możesz sprawdzić, czy narzędzie działa i jakie porty wykorzystuje (listing 8.4).

Listing 8.4. Sprawdzenie stanu narzędzia Hoverfly

```
$ hoverctl status

+------------+----------+
| Hoverfly   | running  |
| Admin port |     8888 |
| Proxy port |     8500 |
| Mode       | capture  |
| Middleware | disabled |
+------------+----------+
```

Uruchom teraz usługę *flights* i wyślij do niej zapytanie, aby sprawdzić, czy działa. W poniższym przykładzie wyszukiwane są wszystkie loty w następnym dniu (pamiętaj, że uzyskany przez Ciebie wynik użycia polecenia `curl` może być inny niż pokazany w listingu 8.5, ponieważ usługa zwraca losowe dane).

Listing 8.5. Uruchomienie przykładowej usługi flights

```
$ ./run-flights-service.sh
waiting for service to start
waiting for service to start
waiting for service to start
service started

$ curl localhost:8081/api/v1/flights?plusDays=1 | jq
[
  {
    "origin": "Berlin",
    "destination": "New York",
    "cost": "617.31",
    "when": "03:45"
  },
  {
    "origin": "Amsterdam",
    "destination": "Dubai",
    "cost": "3895.49",
    "when": "21:20"
  },
  {
    "origin": "Milan",
    "destination": "New York",
    "cost": "4950.31",
    "when": "08:49"
  }
]
```

Teraz przełącz narzędzie Hoverfly w tryb rejestrowania zapytań, tak jak w listingu 8.6.

Listing 8.6. Rejestrowanie zapytań za pomocą narzędzia Hoverfly

```
$ hoverctl mode capture
Hoverfly has been set to capture mode
```

Wyślij zapytanie do interfejsu usługi *flights*, ale tym razem użyj narzędzia Hoverfly jako serwera proxy, zgodnie z listingiem 8.7 (pamiętaj, że Twoje wyniki mogą być inne niż tutaj pokazane).

Listing 8.7. Rejestrowanie odpowiedzi usługi za pomocą narzędzia Hoverfly

```
$ curl localhost:8081/api/v1/flights?plusDays=1 --proxy localhost:8500 | jq
[
  {
    "origin": "Berlin",
    "destination": "Dubai",
    "cost": "3103.56",
    "when": "20:53"
  },
  {
    "origin": "Amsterdam",
    "destination": "Boston",
```

```
    "cost": "2999.69",
    "when": "19:45"
  }
]
```

Argument `-proxy` użyty w powyższym przykładzie powoduje, że zapytanie jest wysyłane do serwera proxy (tutaj narzędzia Hoverfly), a następnie do docelowej usługi. Odpowiedź podąża odwrotną drogą. W ten sposób narzędzie Hoverfly może rejestrować całą komunikację realizowaną poprzez sieć. Zawsze wtedy, gdy nie będziesz wiedział, co się dzieje z narzędziem (czy np. zarejestrowało zapytanie i odpowiedź), możesz sprawdzić dziennik, którego fragment pokazuje listing 8.8.

Listing 8.8. Przeglądanie dziennika narzędzia Hoverfly

```
$ hoverctl logs
INFO[2019-03-30T08:12:32+01:00] Mode has been changed
mode=capture
INFO[2019-03-30T08:14:01+01:00] request and response captured mode=capture request=
↪&map[headers:map[Accept:[*/*] Proxy-Connection:[Keep-Alive] User-Agent:[curl/7.54.0]] body:
↪method:GET scheme:http destination:localhost:8081 path:/api/v1/flights query:map[plusDays:
↪[1]]] response=&map[error:nil response]
...
```

Przyjrzyjmy się teraz utworzonym danym symulacyjnym. Aby to zrobić, należy je wyeksportować i otworzyć w edytorze tekstowym. W listingu 8.9 widoczny jest program `atom`, ale równie dobrze można użyć innego polecenia (np. `vim` lub `emacs`).

Listing 8.9. Eksport danych symulacyjnych z narzędzia Hoverfly

```
$ hoverctl export module-two-simulation.json
Successfully exported simulation to module-two-simulation.json
$ atom module-two-simulation.json
```

Przejrzyj utworzony plik i sprawdź, czy zawiera zarejestrowane dane. Każde zapytanie powinno stanowić osobny element w tabeli. Tak uzyskane dane możesz wykorzystać do symulowania działania usługi. Najpierw zatrzymaj usługę *flights* i sprawdź, czy nie można już z nią nawiązać komunikacji, tak jak pokazuje listing 8.10.

Listing 8.10. Zatrzymanie usługi flights

```
$ ./stop-flights-service.sh
service successfully shut down
$ curl localhost:8081/api/v1/flights?plusDays=1
curl: (7) Failed to connect to localhost port 8081: Connection refused
```

Przełącz narzędzie Hoverfly w tryb symulacji, zgodnie z listingiem 8.11.

Listing 8.11. Przełączenie narzędzia Hoverfly w tryb symulacji

```
$ hoverctl mode simulate
Hoverfly has been set to simulate mode with a matching strategy of 'strongest'
```

W tym trybie narzędzie Hoverfly nie przekazuje zapytań do rzeczywistej usługi, tylko odsyła do klienta zarejestrowaną odpowiedź. Ponownie wyślij zapytanie, wskazując jak poprzednio narzędzie Hoverfly jako serwer proxy. Tym razem jednak zamiast komunikatu o błędzie powinieneś uzyskać poprawną odpowiedź, tak jak w listingu 8.12.

Literał 8.12. Wysyłanie zapytań do narzędzia Hoverfly pełniącego rolę serwera proxy

```
$ curl localhost:8081/api/v1/flights?plusDays=1 --proxy localhost:8500 | jq
[
  {
    "origin": "Berlin",
    "destination": "Dubai",
    "cost": "3103.56",
    "when": "20:53"
  },
  {
    "origin": "Amsterdam",
    "destination": "Boston",
    "cost": "2999.69",
    "when": "19:45"
  }
]
```

To wszystko! Udało Ci się zasymulować odwołanie do interfejsu API usługi! W tym przykładzie użyłeś polecenia `curl`, ale w rzeczywistości zapytania będzie wysłała testowana aplikacja. Jeżeli zapytania i odpowiedzi będą przechowywane w narzędziu Hoverfly, dostęp do oryginalnej usługi nie będzie potrzebny. Ponadto za pomocą tego narzędzia można sterować odpowiedziami. Narzędzie wirtualizacyjne, takie jak Hoverfly, ma m.in. tę zaletę, że w minimalnym stopniu obciąża system i szybko się uruchamia. Za jego pomocą można uruchamiać na laptopie wiele więcej usług wirtualnych niż rzeczywistych w prawdziwym środowisku. Ponadto narzędzie to można wykorzystywać do szybkiego przeprowadzania testów integracyjnych.

Nie implementuj na nowo wirtualnej usługi

Jeżeli okaże się, że funkcjonalności Twoich wirtualnych usług stopniowo odbiegają od rzeczywistych, możesz odczuwać pokusę, aby zaimplementować w nich bardziej zaawansowane algorytmy lub instrukcje warunkowe. To jest wbrew temu wzorcowi! Choć w ten sposób mógłbyś zaoszczędzić sporo czasu, jednak zdecydowanie nie możesz implementować na nowo wirtualnej wersji usługi. Wirtualizacja idealnie nadaje się do tworzenia inteligentnych imitacji lub atrapy usługi, która ma skomplikowaną wewnętrzną strukturę, ale zwraca proste dane. Innymi słowy, powinieneś koncentrować się na wirtualizacji działania, a nie stanu usługi.

Jeżeli zaczniesz modyfikować wirtualną usługę, wprowadzając w niej mnóstwo instrukcji warunkowych określających odpowiedzi, które usługa ma zwracać na odbierane zapytania, albo gdy wirtualna usługa zacznie przypominać rzeczywistą, to znaczy, że stosujesz antywzorzec i powinieneś zastosować inną technikę.

Maszyny wirtualne oraz narzędzia Vagrant i Packer

Często podczas wdrażania aplikacji w chmurze będziesz musiał tworzyć obrazy maszyn wirtualnych zawierających Twoją aplikację Java. W ten sposób będziesz mógł również uruchamiać kilka zależnych od siebie usług (jeżeli będzie ich niewiele, a komputer, którego używasz, będzie odpowiednio wydajny). Teraz dowiesz się, jak za pomocą narzędzia HashiCorp Vagrant tworzy się i inicjuje maszyny wirtualne na lokalnym komputerze.

Instalacja narzędzia Vagrant

Plik instalacyjny narzędzia Vagrant dla systemów macOS, Linux lub Windows można pobrać ze strony *https://www.vagrantup.com/downloads.html*. Potrzebny Ci również będzie hiperwizor, np. Oracle VirtualBox (*https://www.virtualbox.org/wiki/Downloads*) lub VMware Fusion (*https://www.vmware.com/uk/products/fusion.html*).

Utworzenie pliku Vagrantfile

W pliku *Vagrantfile* definiuje się wszystkie maszyny wirtualne tworzące lokalne środowisko programistyczne. Określa się w nim liczbę maszyn, ilości wykorzystywanych przez nie zasobów oraz ustawienia sieciowe. Dodatkowo można umieszczać skrypty instalujące i konfigurujące wymagane zależności. Listing 8.13 przedstawia przykładowy plik *Vagrantfile*.

Listing 8.13. Plik Vagrantfile tworzący pojedynczą maszynę wirtualną z systemem Ubuntu, instalujący program Jenkins i uruchamiający serwer za pomocą kilku prostych poleceń Bash

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
# Poniżej znajduje się pełna konfiguracja narzędzia Vagrant.
# Argument "2" funkcji Vagrant.configure określa wersję konfiguracji
# (zastosowany jest starszy styl w celu zapewnienia wstecznej
# kompatybilności). Nie zmieniaj go, chyba że jesteś pewien, co robisz.
Vagrant.configure("2") do |config|
  # Poniżej są opisane najczęściej stosowane opcje konfiguracyjne.
  # Pełna dokumentacja jest dostępna na stronie https://docs.vagrantup.com.
  # W każdym środowisku Vagrant potrzebna jest maszyna wirtualna. Lista
  # dostępnych maszyn znajduje się na stronie https://atlas.hashicorp.com/search.
  config.vm.box = "ubuntu/xenial64"
  config.vm.box_version = "20170922.0.0"
  config.vm.network "forwarded_port", guest: 8080, host: 8080
  config.vm.provider "virtualbox" do |v|
    v.memory = 2048
  end
  # Konfiguracja środowiska za pomocą skryptu powłoki. Można stosować narzędzia
  # konfiguracyjne, np. Puppet, Chef, Ansible, Salt lub Docker. Informacje
  # na temat składni poleceń i ich użycia znajdują się w dokumentacji.
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    # Instalacja komponentów OpenJDK Java JDK i Maven.
    apt-get install -y openjdk-8-jdk
    apt-get install -y maven
    # Instalacja sbt.
    echo "deb https://dl.bintray.com/sbt/debian /" |
    tee -a /etc/apt/sources.list.d/sbt.list
    apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
    --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
    apt-get update
    apt-get install sbt
    # Instalacja środowiska Docker (może być bardziej skomplikowana,
    # jeżeli są potrzebne specyficzne pakiety).
    apt-get install -y apt-transport-https ca-certificates
    apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

```
    echo deb https://apt.dockerproject.org/repo ubuntu-xenial main >>
    /etc/apt/sources.list.d/docker.list
    apt-get update
    apt-get purge lxc-docker
    apt-get install -y linux-image-extra-$(uname -r) linux-image-extra-virtual
    apt-get install -y docker-engine
    # Instalacja narzędzia Jenkins.
    wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | apt-key add -
    echo deb http://pkg.jenkins-ci.org/debian binary/ >
    /etc/apt/sources.list.d/jenkins.list
    apt-get update
    apt-get install -y jenkins
    # Wyświetlenie klucza narzędzia Jenkins wymaganego podczas inicjacji.
    printf "\n\nKLUCZ JENKINS\n*********************************"
    # Dodanie użytkownika jenkins do grupy docker.
    usermod -aG docker jenkins
    # Oczekiwanie na uruchomienie narzędzia Jenkins i utworzenie
    # pliku initialAdminPassword.
    while [ ! -f /var/lib/jenkins/secrets/initialAdminPassword ]
    do
        sleep 2
    done
    cat /var/lib/jenkins/secrets/initialAdminPassword
    printf "*********************************"
    # Ponowne uruchomienie usługi Jenkins w celu zastosowania zmian
    # wprowadzonych za pomocą polecenia usermod.
    service jenkins restart
    # Instalacja Docker Compose.
    curl -s -L https://github.com/docker/compose/releases/


    download/1.10.0/docker-compose-`uname -s`-`uname -m` >↵


    /usr/local/bin/docker-compose
    chmod +x /usr/local/bin/docker-compose
  SHELL
end
```

Maszyny wirtualne zdefiniowane w pliku *Vagrantfile* uruchamia się za pomocą polecenia `vagrant up`, tak jak w listingu 8.14, a zatrzymuje i usuwa odpowiednio poleceniami `vagrant halt` i `vagrant destroy`.

Listing 8.14. Uruchamianie maszyny wirtualnej za pomocą narzędzie Vagrant

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'ubuntu/xenial64' is up-to-date...
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 8080 (guest) => 8080 (host) (adapter 1)
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
```

```
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: ubuntu
    default: SSH auth method: password
==> default: Machine booted and ready!
```

Jeżeli przyjrzysz się plikowi *Vagrantfile* przedstawionemu w listingu 8.13, zauważysz wiersz `config.vm.network "forwarded_port", guest: 8080, host: 8080` wiążący port nr 8080 wykorzystywany przez maszynę wirtualną z portem o takim samym numerze używanym przez lokalny system operacyjny. Oznacza to, że po wpisaniu w przeglądarce adresu *http://localhost:8080* pojawi się strona narzędzia Jenkins uruchomionego na maszynie utworzonej za pomocą narzędzia Vagrant.

Jak pamiętasz z podrozdziału „Tworzenie obrazów maszyn dla wielu chmur za pomocą programu Packer", możesz utworzyć za pomocą wymienionego narzędzia obrazy maszyn, które następnie zainicjujesz przy użyciu opcji konfiguracyjnej `config.vm.box` w pliku *Vagrantfile*.

Wzorzec 3.: pudełkowe środowisko produkcyjne

Za pomocą narzędzia wirtualizacyjnego, takiego jak HashiCorp Vagrant, można łatwo uruchamiać na lokalnym komputerze pobrane z internetu obrazy maszyn wirtualnych, zawierające gotowe usługi i wykorzystywać je do tworzenia aplikacji i uruchamiania automatycznych testów. W ten sposób można też utworzyć tzw. pudełkowe środowisko produkcyjne, czyli prostszą wersję rzeczywistego środowiska, którą można udostępnić innym członkom zespołu i zapewniać w ten sposób spójną współpracę. W tym celu należy utworzyć obraz maszyny zawierającej system operacyjny, kod źródłowy i pliki binarne aplikacji, jej konfigurację oraz niezbędne magazyny danych.

Czy tworzenie pudełkowego środowiska produkcyjnego jest antywzorcem

Pudełkowe środowisko produkcyjne najbardziej przydaje się w zespołach korzystających z niewielkiego, stabilnego środowiska produkcyjnego, w którym uruchomionych jest niewiele usług. Gdy jednak aplikacja zacznie się powiększać, będzie korzystać z więcej niż pięciu usług lub ich struktura i konfiguracja zaczną się komplikować, wtedy lokalne replikowanie środowiska produkcyjnego stanie się niepraktyczne, a utrzymywanie spójności obu środowisk będzie zajmowało dużo czasu. Gdy zauważysz, że lokalna replika nie odzwierciedla działania rzeczywistego środowiska lub zbyt wiele wysiłku i czasu wymaga utrzymywanie repliki, będzie to znak, że wzorzec stał się antywzorcem.

Wraz z pojawieniem się narzędzia HashiCorp Packer proces tworzenia obrazów maszyn wirtualnych stał się jeszcze prostszy. Za pomocą tego narzędzia można określać parametry obrazu i stosować je do tworzenia różnych środowisk (np. produkcyjnego w chmurze Azure, testowego w OpenStack i programistycznego na lokalnym komputerze). Niewątpliwie do popularności tego stylu pakowania aplikacji przyczyniła się platforma Docker (opisana dalej w tym rozdziale) z narzędziem Fig niczym wisienką na torcie. Z czasem narzędzie to przekształciło się w Docker Compose

i obecnie umożliwia tworzenie deklaratywnych specyfikacji aplikacji i usług, związanych z nimi zależności oraz magazynów danych. Opisany tu wzorzec pozwala w elastyczny sposób uruchamiać na lokalnym komputerze zestaw uzależnionych od siebie usług, a jedynym ograniczeniem są zasoby sprzętowe (szczególnie w przypadku korzystania z platform wirtualizacyjnych).

Stosując wzorzec pudełkowy, można tworzyć znacznie prostsze lokalne środowiska programistyczne i unikać potencjalnych konfliktów konfiguracyjnych (spowodowanych np. różnymi wersjami języka Java), ponieważ skonfigurowane usługi wraz z zależnościami stanowią jedną całość. Obrazy można parametryzować (za pomocą opcji inicjacyjnych i zmiennych środowiskowych) i tworzyć profile, dzięki czemu usługi mogą działać zgodnie z oczekiwaniami. Wtyczki Docker dla narzędzia Maven pozwalają integrować kontenery z procedurami testowymi. Potencjalnie opisany wzorzec można rozszerzyć o rozwijanie kodu samych obrazów, np. poprzez zamontowanie lokalnego kodu źródłowego w uruchomionej instancji obrazu. Jeżeli zrobi się to umiejętnie, można praktycznie zrezygnować z instalowania na lokalnym komputerze jakichkolwiek narzędzi (oprócz ulubionego środowiska IDE), co znacznie upraszcza cały proces tworzenia aplikacji.

Programowanie chmurowe (z dużym pudełkowym środowiskiem produkcyjnym)?

Na rynku pojawiają się chmurowe środowiska IDE, np. Eclipse Che i Amazon Cloud9. Część analityków uważa, że przyszłością programowania są narzędzia instalowane w chmurze, a nie na lokalnym komputerze. Czas pokaże, czy to prawda. Sam jednak wielokrotnie przekonasz się, że środowiska chmurowe umożliwiają uruchamianie repliki środowiska produkcyjnego (lub jego części) i dołączanie go do „lokalnego" chmurowego środowiska IDE, jak to ma miejsce w przypadku bezserwerowych aplikacji FaaS. Niezależnie od tego, czy zamierzasz używać takiego lokalnego środowiska, czy nie, warto, abyś poznał ten wzorzec i przyszłościowe metody programowania.

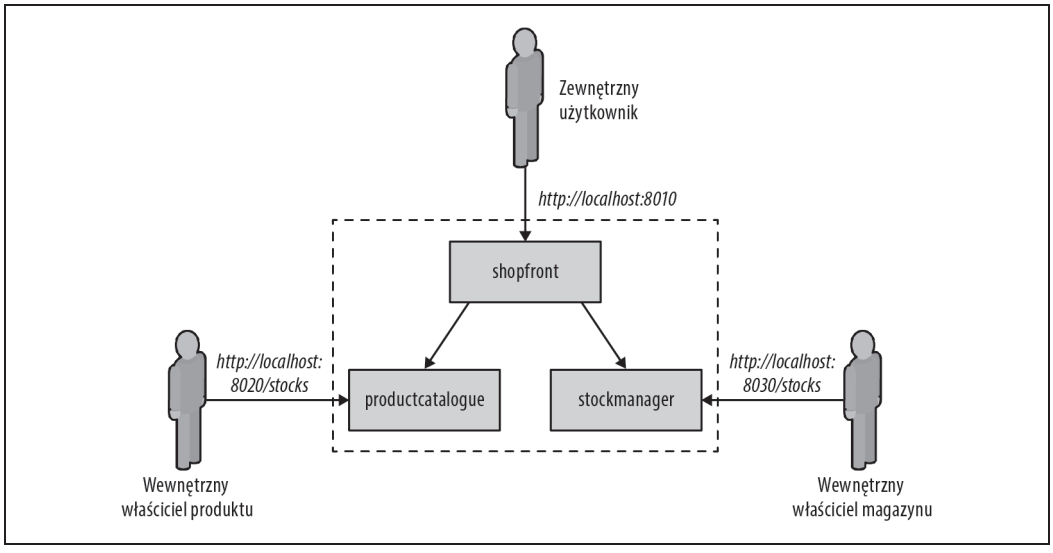
Kontenery: Kubernetes, minikube i Telepresence

W tym podrozdziale dowiesz się, jak lokalnie korzystać z kontenerów Docker i platformy instrumentacyjnej Kubernetes.

Przykładowa aplikacja Docker Java Shop

Do uruchamiania kontenerów na produkcyjną skalę potrzebna jest platforma instrumentacyjna i planistyczna. Istnieje kilka takich platform, m.in. Docker Swarm, Apache Mesos i AWS ECS, ale najpopularniejszą jest Kubernetes (*https://kubernetes.io*), stosowana w wielu środowiskach produkcyjnych. Obecnie platformę tę rozwija fundacja CNCF (ang. *Cloud Native Computing Foundation*, *https://www.cncf.io*). Poniżej opisany jest przykład wykorzystania jej do uruchomienia w kontenerach Docker aplikacji Java obsługującej sklep internetowy.

Rysunek 8.1 przedstawia architekturę aplikacji Docker Java Shopfront, którą umieścimy w kontenerach i wdrożymy za pomocą platformy Kubernetes.



Rysunek 8.1. Architektura aplikacji Docker Java Shopfront

Tworzenie aplikacji Java i obrazów kontenerów

Zanim utworzysz kontener i odpowiedni plik konfiguracyjny dla platformy Kubernetes, musisz wykonać następujące operacje.

Zainstalować środowisko Docker w systemie macOS (https://dockr.ly/2zwBIqz), Windows (https://dockr.ly/2NL7dWn) lub Linux (https://dockr.ly/2xUSIV5)
: Środowisko to służy do tworzenia, uruchamiania i testowania kontenerów Docker na lokalnym komputerze niezależnie od platformy Kubernetes.

Zainstalować narzędzie minikube (http://bit.ly/2xNk8w4)
: Jest to narzędzie ułatwiające uruchamianie jednowęzłowego klastra kontenerów jako maszyny wirtualnej na lokalnym komputerze.

*Utworzyć konto w serwisie GitHub (https://github.com) i zainstalować program Git (*https://git-scm.com*)*
: Przykładowe kody są zapisane w serwisie GitHub, a za pomocą narzędzia Git można tworzyć odgałęzienia repozytorium i zatwierdzać zmiany wprowadzane w lokalnej kopii aplikacji.

Utworzyć konto w serwisie Docker Hub (https://hub.docker.com)
: Jeżeli zamierzasz postępować zgodnie z podanymi niżej wskazówkami, będziesz potrzebować konta w serwisie Docker Hub, aby w nim zapisywać kopie utworzonych obrazów.

Zainstalować pakiet Java SDK w wersji 8. lub 9. (http://bit.ly/2xO16pw) i narzędzie Maven (https://maven.apache.org)
: W kodzie wykorzystasz funkcjonalności języka Java 8. Skompilujesz go wraz z zależnościami za pomocą narzędzia Maven.

Sklonuj repozytorium zapisane w serwisie GitHub (ewentualnie utwórz odgałęzienie repozytorium i jego lokalną kopię) zgodnie z listingiem 8.15. Następnie odszukaj aplikację mikrousługową Shopfront (*http://bit.ly/2Og0JOP*).

Listing 8.15. Klonowanie przykładowego repozytorium

```
$ git clone git@github.com:danielbryantuk/oreilly-docker-java-shopping.git
$ cd oreilly-docker-java-shopping/shopfront
```

W swoim ulubionym środowisku, np. IntelliJ IDEA lub Eclipse, otwórz kod aplikacji, przejrzyj go i skompiluj za pomocą narzędzia Maven, zgodnie z listingiem 8.16. Wykonywalny plik JAR aplikacji zostanie utworzony w katalogu *./target*.

Listing 8.16. Kompilowanie aplikacji Spring Boot

```
$ mvn clean install
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15.140 s
[INFO] Finished at: 2019-03-30T18:01:49+01:00
[INFO] Final Memory: 41M/328M
[INFO] ------------------------------------------------------------------------
```

Teraz utwórz obraz kontenera Docker. Dane systemu operacyjnego, konfigurację i instrukcje tworzące obraz zazwyczaj umieszcza się w pliku *Dockerfile*. Listing 8.17 przedstawia przykładowy plik zapisany w katalogu *shopfront*.

Listing 8.17. Przykładowy plik Dockerfile dla aplikacji Spring Boot Java

```
FROM openjdk:8-jre
ADD target/shopfront-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8010
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Pierwszy wiersz zawiera informację, że obraz zostanie utworzony na podstawie obrazu bazowego `openjdk:8-jre`. Obraz ten (dostępny na stronie *https://hub.docker.com/_/openjdk*) jest utrzymywany przez zespół OpenJDK i zawiera wszystko, co jest potrzebne do uruchamiania w kontenerze Docker aplikacji napisanych w języku Java 8 (m.in. system operacyjny i skonfigurowane środowisko OpenJDK 8 JRE). Drugi wiersz zawiera instrukcję odczytującą plik JAR i umieszczającą go w obrazie. W trzecim wierszu wskazany jest port nr 8010 wykorzystywany przez aplikację. Port ten musi być dostępny z zewnątrz kontenera. Czwarty wiersz zawiera punkt wejścia do aplikacji lub polecenie wykonywane podczas inicjacji kontenera. Listing 8.18 przedstawia proces tworzenia kontenera.

Listing 8.18. Proces tworzenia kontenera Docker

```
$ docker build -t danielbryantuk/djshopfront:1.0 .
Successfully built 87b8c5aa5260
Successfully tagged danielbryantuk/djshopfront:1.0
```

Utworzony obraz umieść w serwisie Docker Hub zgodnie z listingiem 8.19. Wcześniej, za pomocą wiersza poleceń musisz zalogować się do serwisu, używając swojego loginu i hasła.

Listing 8.19. Wysłanie obrazu do serwisu Docker Hub

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub.
If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username:
Password:
Login Succeeded
$
$ docker push danielbryantuk/djshopfront:1.0
The push refers to a repository [docker.io/danielbryantuk/djshopfront]
9b19f75e8748: Pushed
...
cf4ecb492384: Pushed
1.0: digest: sha256:8a6b459b0210409e67bee29d25bb512344045bd84a262ede80777edfcff3d9a0
size: 2210
```

Wdrożenie kontenera na platformie Kubernetes

Teraz uruchom kontener na platformie Kubernetes. Najpierw przejdź do głównego katalogu projektu *kubernetes*.

```
$ cd ../kubernetes
```

Otwórz plik wdrożeniowy *shopfront-service.yaml*. Listing 8.20 przedstawia przykładową zawartość tego pliku.

Listing 8.20. Plik wdrożeniowy shopfront-service.yaml usługi Shopfront

```
---
apiVersion: v1
kind: Service
metadata:
  name: shopfront
  labels:
    app: shopfront
spec:
  type: ClusterIP
  selector:
    app: shopfront
  ports:
  - protocol: TCP
    port: 8010
    name: http
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: shopfront
  labels:
    app: shopfront
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopfront
  template:
    metadata:
```

```
    labels:
      app: shopfront
  spec:
    containers:
    - name: djshopfront
      image: danielbryantuk/djshopfront:1.0
      ports:
      - containerPort: 8010
      livenessProbe:
        httpGet:
          path: /health
          port: 8010
        initialDelaySeconds: 30
        timeoutSeconds: 1
```

W pierwszej sekcji pliku tworzona jest usługa `shopfront`, która będzie kierowała wysyłane do portu 8010 pakiety TCP do kontenera z etykietą `app: shopfront`. W drugiej sekcji tworzone jest środowisko, w którym będzie uruchomiona jedna replika (instancja) kontenera zdefiniowanego w pierwszej sekcji pliku. Oprócz tego otwierany jest tu port 8010 wykorzystywany przez kontener Docker oraz deklarowane są identyfikatory `livenessProbe` i `healthcheck`, niezbędne dla platformy Kubernetes do sprawdzania, czy aplikacja działa poprawnie i może przetwarzać odbierane pakiety. Wpisz teraz polecenie `minikube`, tak jak na listingu 8.21, aby zainstalować usługę (zwróć uwagę, że w zależności od zasobów dostępnych na Twoim komputerze może być konieczna zmiana parametrów określających liczbę procesorów i wielkość pamięci).

Listing 8.21. Instalacja usługi za pomocą polecenia minikube

```
$ minikube start --cpus 2 --memory 4096
Starting local Kubernetes v1.7.5 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
$ kubectl apply -f shopfront-service.yaml
service "shopfront" created
deployment "shopfront" created
```

Listę wszystkich usług uruchomionych na platformie Kubernetes możesz w każdej chwili wyświetlić za pomocą polecenia `kubectl get svc`, tak jak na listingu 8.22.

Listing 8.22. Polecenie kubectl get svc

```
$ kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes   10.0.0.1     <none>        443/TCP          18h
shopfront    10.0.0.216   <nodes>       8010:31208/TCP   12s
$ kubectl get pods
NAME              READY     STATUS              RESTARTS   AGE
shopfront-0w1js   0/1       ContainerCreating   0          18s
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
shopfront-0w1js   1/1       Running   0          2m
```

Wdrożyłeś na platformie Kubernetes swoją pierwszą usługę!

Prosty test usługi

Za pomocą polecenia `curl` użytego ze ścieżką `health`, tak jak na listingu 8.23, możesz pobrać dane z usługi. W ten prosty sposób możesz sprawdzić, czy wszystko działa zgodnie z oczekiwaniami.

Listing 8.23. Prosty test usługi

```
$ curl $(minikube service shopfront --url)/health
{"status":"UP"}
```

Wynik zwrócony przez polecenie `curl` użyte ze ścieżką `/health` potwierdza, że usługa działa poprawnie. Jednak do pełnego uruchomienia aplikacji niezbędne jest wdrożenie pozostałych kontenerów z mikrousługami.

Utworzenie pozostałych usług

Teraz, gdy główny kontener działa prawidłowo, utwórz zgodnie z listingiem 8.24 pozostałe dwa kontenery z dodatkowymi mikrousługami.

Listing 8.24. Utworzenie pozostałych usług

```
$ cd ..
$ cd productcatalogue/
$ mvn clean install
…
$ docker build -t danielbryantuk/djproductcatalogue:1.0 .
...
$ docker push danielbryantuk/djproductcatalogue:1.0
...
$ cd ..
$ cd stockmanager/
$ mvn clean install
...
$ docker build -t danielbryantuk/djstockmanager:1.0 .
...
$ docker push danielbryantuk/djstockmanager:1.0
```

Utworzyłeś i zapisałeś w serwisie Docker Hub wszystkie mikrousługi i obrazy kontenerów. Pora wdrożyć usługi `productcatalogue` i `stockmanager` na platformie Kubernetes.

Wdrożenie całej usługi Java na platformie Kubernetes

W sposób podobny do poprzedniego musisz wdrożyć na platformie Kubernetes dwie pozostałe usługi (listing 8.25).

Listing 8.25. Wdrożenie całej aplikacji Java na platformie Kubernetes

```
$ cd ..
$ cd kubernetes/
$ kubectl apply -f productcatalogue-service.yaml
service "productcatalogue" created
deployment "productcatalogue" created
```

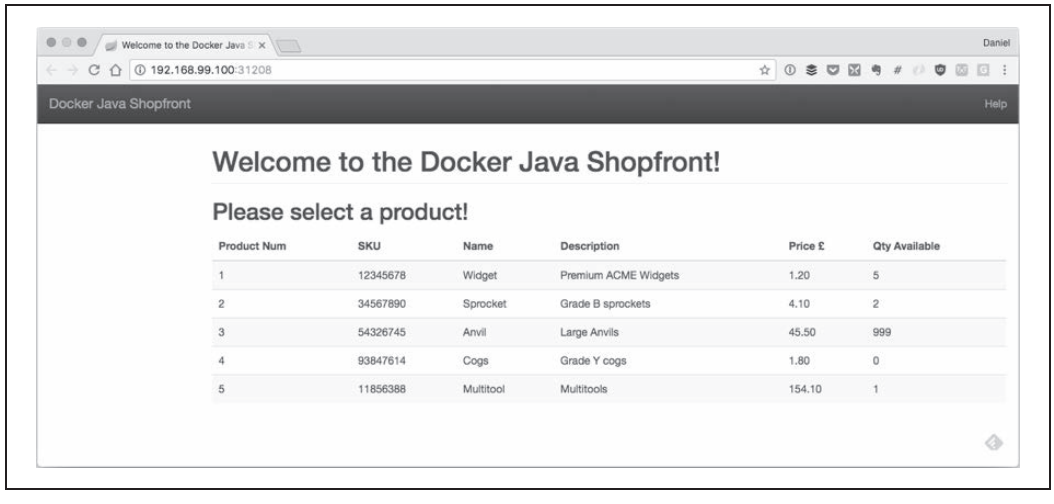
```
$ kubectl apply -f stockmanager-service.yaml
service "stockmanager" created
deployment "stockmanager" created
$ kubectl get svc
NAME              CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes        10.0.0.1     <none>        443/TCP          19h
productcatalogue  10.0.0.37    <nodes>       8020:31803/TCP   42s
shopfront         10.0.0.216   <nodes>       8010:31208/TCP   13m
stockmanager      10.0.0.149   <nodes>       8030:30723/TCP   16s
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
productcatalogue-79qn4    1/1       Running   0          55s
shopfront-0w1js           1/1       Running   0          13m
stockmanager-lmgj9        1/1       Running   0          29s
```

Jeżeli zbyt wcześnie wpiszesz polecenie `kubectl get pods`, może się okazać, że nie wszystkie kontenery zostały uruchomione. Zanim przejdziesz do następnego punktu, zaczekaj, aż wszystkie usługi zaczną działać (jest to dobry moment, aby napić się kawy!).

Kontrola wdrożonej aplikacji

Po wdrożeniu wszystkich usług i uruchomieniu kontenerów powinieneś uzyskać dostęp do interfejsu graficznego aplikacji. Kiedy wpiszesz poniższe polecenie `minikube`, powinna się otworzyć w przeglądarce strona pokazana na rysunku 8.2.

```
$ minikube service shopfront
```



Rysunek 8.2. Prosty interfejs graficzny aplikacji Shopfront uruchomionej w kontenerze Docker

Platformę Kubernetes można uruchamiać nie tylko lokalnie, ale również na zewnętrznym klastrze serwerów. Aplikację można wtedy rozwijać za pomocą narzędzia Datawire Telepresence. Przyjrzyjmy się teraz temu rozwiązaniu.

Telepresence: praca zdalna i lokalna

Telepresence jest to bezpłatne narzędzie umożliwiające uruchamianie na lokalnym komputerze usługi, która komunikuje się z zewnętrznym klastrem serwerów uruchomionych na platformie Kubernetes. Dzięki temu programista tworzący aplikację opartą na mikrousługach może:

- szybko tworzyć jedną usługę na lokalnym komputerze, nawet jeżeli jest ona uzależniona od usług działających w klastrze, a po wprowadzeniu i zapisaniu zmian natychmiast zobaczyć swoją usługę w działaniu,
- testować, diagnozować i edytować usługę za pomocą narzędzi zainstalowanych na lokalnym komputerze, np. dostępnych w środowisku IDE,
- używać swojego komputera, tak jakby był częścią klastra platformy Kubernetes, i w prosty sposób lokalnie uruchamiać usługę, tak jak w zewnętrznym klastrze.

Najpierw zainstaluj aplikację Telepresence (*http://bit.ly/2N6wAwJ*). Instalacja w systemie macOS i Linux jest prosta. Pełna instrukcja instalacyjna dla pozostałych systemów jest dostępna na ww. stronie. Listing 8.26 przedstawia przebieg instalacji tego narzędzia w systemie macOS.

Listing 8.26. Instalacja narzędzia Telepresence w systemie macOS

```
$ brew cask install osxfuse
$ brew install socat datawire/blackbird/telepresence
...
$ telepresence --version
0.77
```

Szczegóły techniczne narzędzia Telepresence

Narzędzie Telepresence jest dwukierunkowym serwerem proxy działającym w kontenerze uruchomionym na platformie Kubernetes. Serwer przekazuje dane (połączenia TCP, zmienne środowiskowe) z platformy do lokalnego procesu. Komunikacja sieciowa jest przezroczysta i zapytania wysyłane do serwera DNS oraz połączenia TCP są przekazywane przez serwer proxy do klastra Kubernetes.

Oto zalety tego rozwiązania:

- lokalna usługa ma pełny dostęp do usług działających w klastrze,
- lokalna usługa ma pełny dostęp do zmiennych środowiskowych platformy Kubernetes, haseł i zasobów ConfigMap,
- usługi działające w klastrze mają pełny dostęp do lokalnej usługi.

Działanie narzędzia Telepresence jest szczegółowo opisane na stronie *http://bit.ly/2IkkXl7*.

Teraz utwórz klaster Kubernetes. Listing 8.27 przedstawia przykład uruchomienia pełnego klastra za pomocą usługi Google Cloud Platform (GCP) GKE. Jeżeli chcesz przeprowadzić ten proces, utwórz konto w serwisie GCP i zainstaluj na lokalnym komputerze narzędzie `gclouds`. Po zainstalowaniu musisz skonfigurować poświadczenia, których użyłeś podczas zakładania konta (dokładny opis, jak to zrobić, znajdziesz na stronie Google Cloud SDK *http://bit.ly/2NKqjfm*). W czasie pisania tej książki wymagana była również instancja komponentów narzędzia `gcloud` (instrukcja jest dostępna na stronie *http://bit.ly/2OTEknF*).

Listing 8.27. Tworzenie w platformie GCP GKE klastra współdzielonych instancji

```
$ gcloud container clusters create telepresence-demo
--machine-type n1-standard-2 --preemptible
Creating cluster telepresence-demo...done.
Created [https://container.googleapis.com/v1beta1/projects/k8s-leap-forward/zones/
↪us-central1-a/clusters/telepresence-demo].

To inspect the contents of your cluster, go to:
https://console.cloud.google.com/kubernetes/workload_/gcloud/us-central1-a/
↪telepresence-demo?project=k8s-leap-forward

kubeconfig entry generated for telepresence-demo.
NAME  LOCATION  MASTER_VERSION  MASTER_IP  MACHINE_TYPE  NUM_NODES  STATUS
telepresence-demo  us-central1-a 1.8.8-gke.0 35.193.55.23 n1-standard-2  3 RUNNING
```

Utworzony klaster będzie składał się z kilku instancji *n1-standard-2*, które są nieco większe niż domyślne, ponieważ niektóre aplikacje Java wymagają większej ilości pamięci, niż jest dostępna w mniejszych instancjach. Aby zmniejszyć koszty klastra, możesz utworzyć go ze *współdzielonych* instancji, które są znacznie tańsze niż standardowe, ale ich działanie może być spowolnione, gdy Google będzie potrzebować większych mocy obliczeniowych. Zdarza się to dość rzadko, jednak zawsze istnieje takie ryzyko. W takich sytuacjach platforma Kubernetes samoczynnie diagnozuje i ponownie uruchamia spowolnioną aplikację.

Po utworzeniu klastra możesz wdrożyć w nim przykładowe usługi. Zwróć uwagę, że po zainicjowaniu narzędzia Telepresence możesz używać polecenia `curl` z parametrami `shopfront` i `health` tak, jakbyś był zalogowany do klastra (listing 8.28). Nie musisz w tym celu używać zewnętrznego adresu IP (ani udostępniać usługi w Internecie).

Listing 8.28. Korzystanie za pomocą polecenia curl z usługi w klastrze, tak jak z lokalnej usługi

```
$ cd oreilly-docker-java-shopping/kubernetes
$ kubectl apply -f .
service "productcatalogue" created
deployment "productcatalogue" created
service "shopfront" created
deployment "shopfront" created
service "stockmanager" created
deployment "stockmanager" created
$
$ telepresence
Starting proxy with method 'vpn-tcp', which has the following limitations:
All processes are affected, only one telepresence can run per machine,  and you can't use other
VPNs. You may need to add cloud hosts with --also-proxy. For a full list of method limitations see
https://telepresence.io/reference/methods.html
Volumes are rooted at $TELEPRESENCE_ROOT. See https://telepresence.io/howto/volumes.html for details.

No traffic is being forwarded from the remote Deployment to your local machine.
You can use the --expose option to specify which ports you want to forward.

Password:
Guessing that Services IP range is 10.63.240.0/20. Services started after
this point will be inaccessible if are outside this range; restart telepresence if you can't access
a new Service.

@gke_k8s-leap-forward_us-central1-a_demo| $ curl shopfront:8010/health
{"status":"UP"}
@gke_k8s-leap-forward_us-central1-a_demo| kubernetes $ exit
```

To jest jedynie drobny fragment możliwości oferowanych przez narzędzie Telepresence. Najciekawszą funkcjonalnością jest diagnozowanie lokalnej usługi komunikującej się z usługami działającymi w klastrze (*http://bit.ly/2OjmPQg*). Na stronie WWW narzędzia dostępna jest szczegółowa instrukcja, jak to się robi.

Porządkowanie klastra GKE

Pamiętaj, aby na koniec usunąć klaster. Jeżeli tego nie zrobisz, zaskoczy Cię rachunek za jego użytkowanie! Klaster usuwa się za pomocą następującego polecenia:

```
$ gcloud container clusters delete telepresence-demo
```

Wzorzec 4.: dzierżawa środowiska

W skrócie mówiąc, wzorzec ten umożliwia tworzenie i automatyczne konfigurowanie własnego środowiska zawierającego dowolne usługi i dane. Wzorzec ten jest dość podobny do pudełkowego środowiska produkcyjnego. Różni się tym, że replika środowiska nie jest uruchamiana lokalnie, tylko w chmurze. Usługi i dane (wraz z powiązanymi z nimi komponentami infrastruktury) definiuje się programistycznie za pomocą narzędzia IaC (ang. *Infrastructure as Code* — infrastruktura jako kod), np. Terraform (*https://terraform.io*) lub jednego z narzędzi do automatycznego tworzenia i konfigurowania systemów, np. Ansible (*http://www.ansible.com*). Aby skutecznie stosować ten wzorzec, zespół użytkowników musi posiadać wiedzę o konfigurowaniu i użytkowaniu środowiska.

Środowisko po zdefiniowaniu i zainicjowaniu jest „dzierżawione" przez programistów. Komputer każdego z nich musi być skonfigurowany tak, aby z usługami zainstalowanymi w zewnętrznym środowisku komunikował się w taki sam sposób jak z lokalnymi usługami. Wzorzec ten wykorzystuje się do wdrażania aplikacji na platformach chmurowych. Pozwala również szybko tworzyć i usuwać środowisko na żądanie.

Dzierżawienie środowiska wymaga programowanej infrastruktury i wiedzy utrzymaniowej

Dzierżawa środowiska jest zaawansowanym wzorcem. Aby go stosować, musi istnieć możliwość tworzenia i skalowania środowiska na żądanie (np. w chmurze prywatnej lub publicznej). Ponadto zespół programistów musi dobrze znać charakterystykę operacyjnej platformy produkcyjnej. Oprócz tego komputer programisty musi nawiązywać stabilne połączenie z środowiskiem. Automatyzację baz danych i aktualizację lokalizacji środowiska każdego programisty ułatwiają lokalne serwery proxy, takie jak Datawire Telepresence, NGINX lub HAProxy w połączeniu z HashiCorp Consul (*https://www.consul.io*) oraz consul-template (*https://github.com/hashicorp/consul-template*), jak również platforma Spring Cloud (*http://bit.ly/2Q76njB*) połączona z Netflix Eureka (*https://github.com/Netflix/eureka*).

Funkcja jako usługa: AWS Lambda i SAM Local

W 2016 r. firma Amazon wprowadziła do swojej oferty model SAM (ang. *Serverless Application Model* — bezserwerowy model aplikacji), aby ułatwić programistom wdrażanie usług FaaS. Rdzeń modelu jest otwartym oprogramowaniem opartym na usłudze AWS CloudFormation ułatwiającej konfigurowanie i utrzymywanie infrastruktury bezserwerowej.

Narzędzie SAM Local wykorzystuje najbardziej przydatne elementy modelu SAM i umożliwia wykonywanie na lokalnym komputerze operacji, takich jak:

- tworzenie i testowanie funkcji AWS Lambda z wykorzystaniem środowiska Docker,
- symulowanie wywoływania funkcji za pomocą znanych źródeł zdarzeń, przykładowo Amazon Simple Storage Service (S3), Amazon DynamoDB, Amazon Kinesis, Amazon Simple Notification Service (SNS) oraz wielu innych usług Amazon,
- uruchamianie lokalnego interfejsu Amazon API Gateway za pomocą szablonu SAM oraz szybkie iterowanie i gorące przeładowywanie funkcji,
- szybkie weryfikowanie szablonów SAM, również za pomocą środowiska IDE i programów analizujących poprawność kodu,
- interaktywne diagnozowanie funkcji AWS Lambda.

Przyjrzyjmy się teraz narzędziu AWS SAM Local.

Instalacja narzędzia SAM Local

Narzędzie SAM Local można zainstalować na kilka sposobów. Najprostszy polega na użyciu menedżera pakietów Python `pip` (*https://pypi.org/project/pip*). Szczegółowy opis procesu instalacji wykracza poza zakres tej książki, jednak szczegółowe informacje można znaleźć na ww. stronie, jak również na stronie narzędzia SAM Local (*https://github.com/awslabs/aws-sam-cli*).

Po zainstalowaniu menedżera `pip` na lokalnym komputerze należy zainstalować narzędzie SAM Local, wpisując w terminalu polecenie pokazane w listingu 8.29.

Listing 8.29. Instalacja narzędzia SAM Local

```
$ pip install aws-sam-cli
```

Jeżeli na komputerze jest zainstalowany język Go, wówczas najnowszą wersję narzędzia można utworzyć, kompilując jego kod źródłowy poleceniem `go get github.com/awslabs/aws-sam-local`.

Tworzenie funkcji AWS Lambda

Listing 8.30 przedstawia prostą funkcję Java implementującą usługę Product Catalogue w opisanej wcześniej przykładowej aplikacji Shopping. Pełny kod usługi jest dostępny w repozytorium GitHub (*https://github.com/continuous-delivery-in-java/product-catalogue-aws-lambda*). Poniższy listing zawiera główną klasę usługi.

Listing 8.30. Przykładowa funkcja AWS Lambda w języku Java

```
package uk.co.danielbryant.djshoppingserverless.productcatalogue;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.google.gson.Gson;
import uk.co.danielbryant.djshoppingserverless.productcatalogue.services.ProductService;
import java.util.HashMap;
import java.util.Map;

/**
* Funkcja Lambda obsługująca zapytania.
*/
public class ProductCatalogueFunction
  implements RequestHandler<Map<String, Object>, GatewayResponse> {
    private static final int HTTP_OK = 200;
    private static final int HTTP_INTERNAL_SERVER_ERROR = 500;
    private ProductService productService = new ProductService();
    private Gson gson = new Gson();

    public GatewayResponse handleRequest(final Map<String, Object> input,
    final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        String output = gson.toJson(productService.getAllProducts());
        return new GatewayResponse(output, headers, HTTP_OK);
    }
}
```

Metoda `handleRequest` jest wywoływana przez platformę AWS Lambda w lokalnym lub zewnętrznym (produkcyjnym) środowisku. W bibliotece *aws-lambda-java-core* importowanej za pomocą narzędzia Maven dostępnych jest wiele predefiniowanych interfejsów `RequestHandler` (*https://amzn.to/2QbAgiF*) i powiązanych z nimi metod `handleRequest`. W powyższym przykładzie wykorzystany został interfejs `RequestHandler<Map<String, Object>, GatewayResponse>` przetwarzający umieszczone w argumencie funkcji dane JSON (metodę HTTP, nagłówki, parametry i ciało zapytania). Interfejs zwraca obiekt `GatewayResponse`, który jest następnie przesyłany do użytkownika lub odpowiedniej usługi.

Listing 8.31 przedstawia plik *pom.xml* projektu. Zwróć uwagę, że plik JAR przeznaczony do wdrożenia jest pakowany za pomocą opisanej w poprzednim rozdziale wtyczki Maven Shade.

Listing 8.31. Plik pom.xml funkcji AWS Lambda ProductCatalogue

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>uk.co.danielbryant.djshoppingserverless</groupId>
    <artifactId>ProductCatalogue</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <name>Prosta usługa Product Catalogue utworzona za pomocą narzędzia SAM CLI sam-ini.</name>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
```

```
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.1.0</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.5</version>
        </dependency>
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
          <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-shade-plugin</artifactId>
          <version>3.1.1</version>
          <configuration>
          </configuration>
          <executions>
            <execution>
              <phase>package</phase>
              <goals>
                <goal>shade</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
</project>
```

Aby lokalnie skompilować i przetestować kod, potrzebny jest plik manifestu *template.yaml* (listing 8.32) specyfikujący konfigurację funkcji Lambda i tworzący prosty interfejs API Gateway.

Listing 8.32. Plik AWS Lambda template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: >
    Funkcja Lambda Product Catalogue
    (oparta na przykładowym szablonie SAM sam-app)
Globals:
    Function:
        Timeout: 20
Resources:
    ProductCatalogueFunction:
        Type: AWS::Serverless::Function
```

```
        Properties:
            CodeUri: target/ProductCatalogue-1.0.jar
            Handler: uk.co.danielbryant.djshoppingserverless.productcatalogue.ProductCatalogue
            ↪Function::handleRequest
            Runtime: java8
            Environment: # Więcej informacji o zmiennych środowiskowych: https://github.com/awslabs/
                         # serverless-application-model/blob/master/versions/2016-10-31.md#environment-object
                Variables:
                    PARAM1: VALUE
            Events:
                HelloWorld:
                    Type: Api # Więcej informacji o interfejsie API źródła zdarzeń: https://github.com/awslabs/
                              # serverless-application-model/blob/master/versions/2016-10-31.md#api
                    Properties:
                        Path: /products
                        Method: get
Outputs:
    HelloWorldApi:
      Description: "Ścieżka URL API Gateway dla funkcji Lambda Product Catalogue"
      Value: !Sub "https://${ServerlessRestApi}.execute-api
      .${AWS::Region}.amazonaws.com/prod/products/"
    HelloWorldFunction:
      Description: "ARN funkcji Lambda Product Catalogue"
      Value: !GetAtt ProductCatalogueFunction.Arn
    HelloWorldFunctionIamRole:
      Description: "Niejawna rola IAM utworzona dla funkcji Lambda Product Catalogue"
      Value: !GetAtt ProductCatalogueFunction.Arn
```

Testowanie obsługi zdarzeń za pomocą funkcji AWS Lambda

Narzędzie SAM Local oferuje polecenie `sam local generate-event` służące do generowania zdarzeń testowych. Szczegółowe informacje na temat tej operacji można uzyskać, używając powyższego polecenia z argumentem `--help` umieszczonym w różnych miejscach. W tym przykładzie trzeba wygenerować przykładowe zdarzenie API Gateway. Reprezentuje je syntetyczna wersja obiektu JSON wysyłanego w chwili, gdy usługa lub użytkownik wysyłają zapytanie do bramy Amazon API Gateway stanowiącej wejście do funkcji. Przeanalizujmy listing 8.33.

Listing 8.33. Generowanie zdarzeń testowych za pomocą narzędzia SAM Local

```
$ sam local generate-event --help
Usage: sam local generate-event [OPTIONS] COMMAND [ARGS]...
Generate an event

Options:
  --help  Show this message and exit.

Commands:
  api       Generates a sample Amazon API Gateway event
  dynamodb  Generates a sample Amazon DynamoDB event
  kinesis   Generates a sample Amazon Kinesis event
  s3        Generates a sample Amazon S3 event
  schedule  Generates a sample scheduled event
  sns       Generates a sample Amazon SNS event
$
$ sam local generate-event api --help
```

```
Usage: sam local generate-event api [OPTIONS]

Options:
  -m, --method TEXT    HTTP method (default: "POST")
  -b, --body TEXT      HTTP body (default: "{ "test": "body"}")
  -r, --resource TEXT  API Gateway resource name (default: "/{proxy+}")
  -p, --path TEXT      HTTP path (default: "/examplepath")
  --debug              Turn on debug logging
  --help               Show this message and exit.
$
$ sam local generate-event api -m GET -b "" -p "/products"
{
    "body": null,
    "httpMethod": "GET",
    "resource": "/{proxy+}",
    "queryStringParameters": {
        "foo": "bar"
    },
    "requestContext": {
        "httpMethod": "GET",
        "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
        "path": "/{proxy+}",
        "extendedRequestId": null,
        "resourceId": "123456",
        "apiId": "1234567890",
        "stage": "prod",
        "resourcePath": "/{proxy+}",
        "identity": {
            "accountId": null,
            "apiKey": null,
            "userArn": null,
            "cognitoAuthenticationProvider": null,
            "cognitoIdentityPoolId": null,
            "userAgent": "Custom User Agent String",
            "caller": null,
            "cognitoAuthenticationType": null,
            "sourceIp": "127.0.0.1",
            "user": null
        },
        "accountId": "123456789012"
    },
    "headers": {
        "Accept-Language": "en-US,en;q=0.8",
        "Accept-Encoding": "gzip, deflate, sdch",
        "X-Forwarded-Port": "443",
        "CloudFront-Viewer-Country": "US",
        "X-Amz-Cf-Id": "aaaaaaaaaae3VYQb9jd-nvCd-de396UhbpO27Y2JvkCPNLmGJHqlaA==",
        "CloudFront-Is-Tablet-Viewer": "false",
        "User-Agent": "Custom User Agent String",
        "Via": "1.1 O8f323deadbeefa7af34d5feb414ce27.cloudfront.net (CloudFront)",
        "CloudFront-Is-Desktop-Viewer": "true",
        "CloudFront-Is-SmartTV-Viewer": "false",
        "CloudFront-Is-Mobile-Viewer": "false",
        "X-Forwarded-For": "127.0.0.1, 127.0.0.2",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
        image/webp,*/*;q=0.8",
        "Upgrade-Insecure-Requests": "1",
        "Host": "1234567890.execute-api.us-east-1.amazonaws.com",
```

```
            "X-Forwarded-Proto": "https",
            "Cache-Control": "max-age=0",
            "CloudFront-Forwarded-Proto": "https"
        },
        "stageVariables": null,
        "path": "/products",
        "pathParameters": {
            "proxy": "/products"
        },
        "isBase64Encoded": false
    }
```

Za pomocą tak wygenerowanego zdarzenia można przetestować funkcję na różne sposoby. Najprostszy polega na połączeniu w potok polecenia generującego zdarzenie z poleceniem `sam local invoke <nazwa_funkcji>`. Listing 8.34 przedstawia przykład zastosowania tego sposobu.

Listing 8.34. Wygenerowanie zdarzenia Amazon API Gateway i przesłanie go do lokalnie wywołanej funkcji Lambda

```
$ sam local generate-event api -m GET -b "" -p "/products" | sam local invoke ProductCatalogue
↪Function
2018-06-10 14:06:04 Reading invoke payload from stdin (you can also pass it from file with --event)
2018-06-10 14:06:05 Invoking uk.co.danielbryant.djshoppingserverless.productcatalogue.Product
↪CatalogueFunction::handleRequest (java8)
2018-06-10 14:06:05 Found credentials in shared credentials file: ~/.aws/credentials
2018-06-10 14:06:05 Decompressing /Users/danielbryant/Documents/dev/daniel-bryant-uk/tmp/
↪aws-sam-java/sam-app/target/ProductCatalogue-1.0.jar

Fetching lambci/lambda:java8 Docker container image......
2018-06-10 14:06:06 Mounting /private/var/folders/1x/81f0qg_50vl6c4gntmt008w40000gn/T/tmp1kC9fo
↪as /var/task:ro inside runtime container
START RequestId: 054d0a81-1fa9-41b9-870c-18394e6f6ea9 Version: $LATEST
END RequestId: 054d0a81-1fa9-41b9-870c-18394e6f6ea9
REPORT RequestId: 054d0a81-1fa9-41b9-870c-18394e6f6ea9 Duration: 82.60 ms Billed Duration: 100 ms
↪Memory Size: 128 MB Max Memory Used: 19 MB

{"body":"[{\"id\":\"1\",\"name\":\"Widget\",\"description\":\"Premium ACME
↪Widgets\",\"price\":1.19},{\"id\":\"2\",\"name\":\"Sprocket\",\"description\":\"Grade B
↪sprockets\",\"price\":4.09},{\"id\":\"3\",\"name\":\"Anvil\",\"description\":\"Large
↪Anvils\",\"price\":45.5},{\"id\":\"4\",\"name\":\"Cogs\",\"description\":\"Grade Y
↪cogs\",\"price\":1.80},{\"id\":\"5\",\"name\":\"Multitool\",\"description\":\"Multitools\",\
↪"price\":154.09}]","headers":{"Content-Type":"application/json"},"statusCode":200}
```

Aby dokładniej dostosować zdarzenie do swoich potrzeb, możesz skierować wynik polecenia do pliku, zmodyfikować jego zawartość, a następnie użyć polecenia `cat` razem z poleceniem wywołującym funkcję, tak jak w listingu 8.35.

Listing 8.35. Zapisanie wygenerowanego zdarzenia w pliku, zmodyfikowanie go i wykorzystanie za pomocą narzędzia SAM Local

```
$ sam local generate-event api -m GET -b "" -p "/products" > api_event.json
$ # Zmień plik api_event.json, używając ulubionego edytora, a następnie zapisz go.
$ cat api_event.json | sam local invoke ProductCatalogueFunction
...
```

Funkcję wywołaną w środowisku Docker możesz diagnozować, wywołując polecenie z parametrem `--debug-port <numer_portu>` i podłączając do wskazanego portu zewnętrzny debugger (np. dostępny w środowisku IntelliJ IDEA). Po wywołaniu funkcji narzędzie SAM Local wstrzymuje

jej wykonywanie do chwili podłączenia debuggera. Można ustawiać pułapki i podglądy zmiennych, tak jak podczas zwykłego diagnozowania kodu, a po zakończeniu działania funkcji sprawdzać w terminalu zwracane przez nią wyniki.

Testowanie funkcji za pomocą narzędzia SAM Local

Przy użyciu narzędzia SAM Local można również symulować lokalne uruchamianie usługi Amazon API Gateway, którą można zintegrować z funkcją Lambda. Zarówno usługę, jak i funkcję uruchamia się, wpisując polecenie `sam local start-api` po przejściu do katalogu zawierającego plik *template.yaml*. Aby przetestować funkcję Lambda, można użyć polecenia `curl` z lokalną ścieżką, tak jak w listingu 8.36.

Listing 8.36. Uruchomienie usługi API Gateway i funkcji Lambda za pomocą narzędzia SAM Local i sprawdzenie funkcji z wykorzystaniem polecenia curl

```
$ sam local start-api
2018-06-10 14:56:03 Mounting ProductCatalogueFunction
at http://127.0.0.1:3000/products [GET]
2018-06-10 14:56:03 You can now browse to the above endpoints to invoke your functions. You do
↪not need to restart/reload SAM CLI while working on your functions changes will be reflected
↪instantly/automatically. You only need to restart SAM CLI if you update your AWS SAM template
2018-06-10 14:56:03  * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)

[Open new terminal]

$ curl http://127.0.0.1:3000/products
[{"id":"1","name":"Widget","description":"Premium ACME Widgets","price":1.19},...]
```

Jeżeli teraz wrócisz do pierwszego terminala, w którym uruchomiłeś interfejs API, zauważysz, że pojawiły się w nim dodatkowe informacje zawierające nie tylko wpisy dziennika, ale również dane o czasie działania funkcji i ilości zajętej pamięci (listing 8.37). Informacje te są specyficzne dla konfiguracji komputera (liczby procesorów i ilości pamięci), ale można je wykorzystać do oszacowania kosztów uruchomienia funkcji w środowisku produkcyjnym.

Listing 8.37. Wynik użycia narzędzia SAM Local po symulacji uruchomienia usługi Amazon API Gateway

```
$ sam local start-api
2018-06-10 14:56:03 Mounting ProductCatalogueFunction
at http://127.0.0.1:3000/products [GET]
2018-06-10 14:56:03 You can now browse to the above endpoints to invoke your functions. You do
↪not need to restart/reload SAM CLI while working on your functions changes will be reflected
↪instantly/automatically. You only need to restart SAM CLI if you update your AWS SAM template
2018-06-10 14:56:03  * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
2018-06-10 14:56:37 Invoking uk.co.danielbryant.djshoppingserverless.
productcatalogue.ProductCatalogueFunction::handleRequest (java8)
2018-06-10 14:56:37 Found credentials in shared credentials file: ~/.aws/credentials
2018-06-10 14:56:37 Decompressing /Users/danielbryant/Documents/
dev/daniel-bryant-uk/tmp/aws-sam-java/sam-app/target/ProductCatalogue-1.0.jar

Fetching lambci/lambda:java8 Docker container image......
2018-06-10 14:56:38 Mounting /private/var/folders/1x/81f0qg_50vl6c4gntmt008w40000gn/
T/tmp9BwMmf as /var/task:ro inside runtime container
START RequestId: b5afd403-2fb9-4b95-a887-9a8ea5874641 Version: $LATEST
END RequestId: b5afd403-2fb9-4b95-a887-9a8ea5874641
REPORT RequestId: b5afd403-2fb9-4b95-a887-9a8ea5874641 Duration: 94.77 ms
Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 19 MB
2018-06-10 14:56:40 127.0.0.1 - - [10/Jun/2018 14:56:40] "GET /products HTTP/1.1" 200 -
```

Często podczas lokalnych testów funkcja Lambda musi integrować się z usługami Amazon, np. S3 lub DynamoDB. Wtedy mogą pojawiać się problemy i konieczne jest tworzenie imitacji usług lub wirtualnych zależności przy użyciu technik opisanych w tym rozdziale. Jednak zamiast rozwiązywać problem na własną rękę, warto skorzystać z opcji opracowanych przez społeczność programistów (podczas pobierania i lokalnego uruchamiania kodu należy zachować ostrożność, szczególnie wtedy, gdy robi się to z uprawnieniami administratora systemu lub gdy funkcja odwołuje się do środowiska testowego albo produkcyjnego). Jedną z takich opcji jest LocalStack (*https://github.com/localstack/localstack*) — w pełni funkcjonalny stos chmury AWS.

Lokalne uruchamianie usług AWS za pomocą LocalStack

Pakiet LocalStack oferuje narzędzia do testów integracyjnych, umożliwiające uruchamianie lokalnych wersji (na komputerze programisty lub w środowisku testowym) różnych usług, np. DynamoDB, Kinesis lub S3. Usługi te funkcjonują tak samo jak ich rzeczywiste odpowiedniki, tj. zazwyczaj udostępniają interfejsy REST API i charakterystyczne dla nich protokoły. Ponadto można przystosowywać ich działanie i zwracane dane odpowiednio do testów. Lokalne usługi mogą nawet zgłaszać błędy specyficzne dla usług chmurowych działających w środowisku produkcyjnym, dzięki czemu można odpowiednio przystosować testowany kod.

FaaS: usługa Azure Functions i edytor Visual Studio Code

W 2016 r. w chmurze Azure pojawiła się usługa Azure Functions. Za jej pomocą można uruchamiać aplikacje Java dodane do platformy FaaS w 2017 r. Nie ma bezpośredniej analogii pomiędzy konfiguracją infrastruktury AWS SAM a Azure Functions, jednak można tworzyć pliki konfiguracyjne i stosować narzędzia ułatwiające kompilowanie i testowanie funkcji w lokalnym i zewnętrznym środowisku. Niezbędne operacje można wykonywać za pomocą wiersza poleceń, ale zazwyczaj o wiele łatwiej używać do tego celu doskonałego edytora Visual Studio Code (*https://code.visualstudio.com*).

Instalacja najważniejszych komponentów Azure Functions

Aby tworzyć w języku Java aplikacje wykorzystujące Azure Functions, należy na lokalnym komputerze zainstalować następujące komponenty:

- Java Developer Kit, wersja 8.,
- Apache Maven, wersja 3.0 lub nowsza,
- Azure CLI (*http://bit.ly/2xH4raK*),
- Azure Functions Core Tools (*http://bit.ly/2OdOuSR*), wymagający uprzedniego zainstalowania pakietu .NET Core 2.1 SDK,
- Visual Studio Code (opcjonalnie).

Funkcje można łatwo tworzyć w języku Java, wykorzystując generator archetypów Maven. Listing 8.38 przedstawia polecenie `mvn archetype:generate` użyte z przykładowymi parametrami. W listingu widoczne są pytania zadawane w trakcie procesu generowania funkcji.

Listing 8.38. Tworzenie funkcji Java w usłudze Azure Functions za pomocą narzędzia Maven

```
$ mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure
    -DarchetypeArtifactId=azure-functions-archetype
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/
↪maven-release-plugin/2.5.3/maven-release-plugin-2.5.3.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/
↪maven-release-plugin/2.5.3/maven-release-plugin-2.5.3.pom (11 kB at 24 kB/s)
...
Define value for property 'groupId' (should match expression '[A-Za-z0-9_\-\.]+'): helloworld
[INFO] Using property: groupId = helloworld
Define value for property 'artifactId' (should match expression '[A-Za-z0-9_\-\.]+'):
↪ProductCatalogue
[INFO] Using property: artifactId = ProductCatalogue
Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' helloworld: :
uk.co.danielbryant.helloworldserverless.productcatalogue
Define value for property 'appName' productcatalogue-20180923111807725: :
Define value for property 'appRegion' westus: :
Define value for property 'resourceGroup' java-functions-group: :
Confirm properties configuration:
groupId: helloworld
groupId: helloworld
artifactId: ProductCatalogue
artifactId: ProductCatalogue
version: 1.0-SNAPSHOT
package: uk.co.danielbryant.helloworldserverless.productcatalogue
appName: productcatalogue-20180923111807725
appRegion: westus
resourceGroup: java-functions-group
 Y: : Y
...
[INFO] Project created from Archetype in dir: /Users/danielbryant/Documents/dev/
↪daniel-bryant-uk/tmp/ProductCatalogue
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 03:25 min
[INFO] Finished at: 2018-09-23T11:19:12+01:00
[INFO] ------------------------------------------------------------------------
```

Wynikiem powyższego procesu jest utworzenie prostej klasy Java zawierającej metodę `HttpTriggerJava`, którą można wywoływać za pomocą zapytania HTTP GET. Listing 8.39 przedstawia zawartość tej klasy. W ten sposób możesz nauczyć się lokalnie tworzyć i diagnozować funkcje Azure.

Listing 8.39. Przykładowa klasa Function utworzona za pomocą generatora archetypów Maven

```
public class Function {
    @FunctionName("HttpTrigger-Java")
    public HttpResponseMessage HttpTriggerJava(
        @HttpTrigger(name = "req",
                     methods = {HttpMethod.GET, HttpMethod.POST},
                     authLevel = AuthorizationLevel.ANONYMOUS)
        HttpRequestMessage<Optional<String>> request, final ExecutionContext context) {
        context.getLogger().info("Funkcja Java przetwarza zapytanie HTTP.");
        // Analiza parametrów zapytania.
        String query = request.getQueryParameters().get("name");
```

```
        String name = request.getBody().orElse(query);
        if (name == null) {
            return request.createResponseBuilder(HttpStatus.BAD_REQUEST)
                          .body("Umieść nazwę w ścieżce lub ciele zapytania").build();
        } else {
            return request.createResponseBuilder(HttpStatus.OK)
                   .body("Witaj, " + name).build();
        }
    }
}
```

W głównym katalogu projektu Java znajdują się pliki konfiguracyjne *local.settings.json* i *host.json* zawierające odpowiednio ustawienia aplikacji oraz narzędzi Azure Functions Core Tools. Plik *host.json* zawiera globalne opcje konfiguracyjne wszystkich funkcji. Domyślny plik jest dość prosty. Listing 8.40 pokazuje bardziej zaawansowaną konfigurację interfejsu HTTP API, ścieżek, monitora i dziennika.

Listing 8.40. Bardziej zaawansowany plik konfiguracyjny host.json usługi Azure Function

```
{
    "version": "2.0",
    "extensions": {
        "http": {
            "routePrefix": "api",
            "maxConcurrentRequests": 5,
            "maxOutstandingRequests": 30
            "dynamicThrottlesEnabled": false
        }
    },
    "healthMonitor": {
        "enabled": true,
        "healthCheckInterval": "00:00:10",
        "healthCheckWindow": "00:02:00",
        "healthCheckThreshold": 6,
        "counterThreshold": 0.80
    },
    "id": "9f4ea53c5136457d883d685e57164f08",
    "logging": {
        "fileLoggingMode": "debugOnly",
        "logLevel": {
          "Function.MyFunction": "Information",
          "default": "None"
        },
        "applicationInsights": {
            "sampling": {
              "isEnabled": true,
              "maxTelemetryItemsPerSecond" : 5
            }
        }
    },
    "watchDirectories": [ "Shared", "Test" ]
}
```

Tak utworzony projekt można rozwijać w taki sam sposób jak każdy projekt Maven. Można tworzyć artefakty i za pomocą polecenia `mvn clean package` umieszczać je w usłudze Azure Functions.

Lokalne kompilowanie i testowanie funkcji

Za pomocą narzędzi Azure Function Core Tools można kompilować i lokalnie testować funkcje. Służy do tego celu wtyczka Azure-Function Maven uruchamiana za pomocą polecenia `mvn azure-functions:run` pokazanego na listingu 8.41.

Listing 8.41. Lokalne uruchomienie funkcji Java za pomocą wtyczki Azure Maven

```
$ mvn azure-functions:run
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< helloworld:ProductCatalogue >---------------------
[INFO] Building Azure Java Functions 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- azure-functions-maven-plugin:1.0.0-beta-6:run (default-cli) @ ProductCatalogue ---
AI: INFO 1: Configuration file has been successfully found as resource
AI: INFO 1: Configuration file has been successfully found as resource
[INFO] Azure Function App's staging directory found at:
/Users/danielbryant/Documents/dev/daniel-bryant-uk/tmp/ProductCatalogue/target/
↪azure-functions/productcatalogue-20180923111807725
[INFO] Azure Functions Core Tools found.
                  %%%%%%
                 %%%%%%
            @   %%%%%%    @
          @@   %%%%%%      @@
       @@@    %%%%%%%%%%    @@@
     @@      %%%%%%%%%%        @@
       @@         %%%%       @@
         @@      %%%       @@
           @@    %%      @@
                 %%
                 %
Azure Functions Core Tools (2.0.3)
Function Runtime Version: 2.0.12115.0
...
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...
Http Functions:

HttpTrigger-Java: http://localhost:7071/api/HttpTrigger-Java

[23/09/2018 10:25:24] [INFO] {MessageHandler.handle}: Message generated by "StartStream.Builder"
[23/09/2018 10:25:24] Worker initialized
[23/09/2018 10:25:25] "HttpTrigger-Java" loaded
 (ID: 7115f6e7-f5de-475c-b196-089e6a6a2a89,
 Reflection: "/Users/danielbryant/Documents/dev/
 daniel-bryant-uk/tmp/ProductCatalogue/target/
 azure-functions/productcatalogue-20180923111807725/
 ProductCatalogue-1.0-SNAPSHOT.jar"::
 "uk.co.danielbryant.helloworldserverless.productcatalogue.Function.run")
[23/09/2018 10:25:28] Host lock lease acquired by
 instance ID '000000000000000000000000826B7EEE'.
```

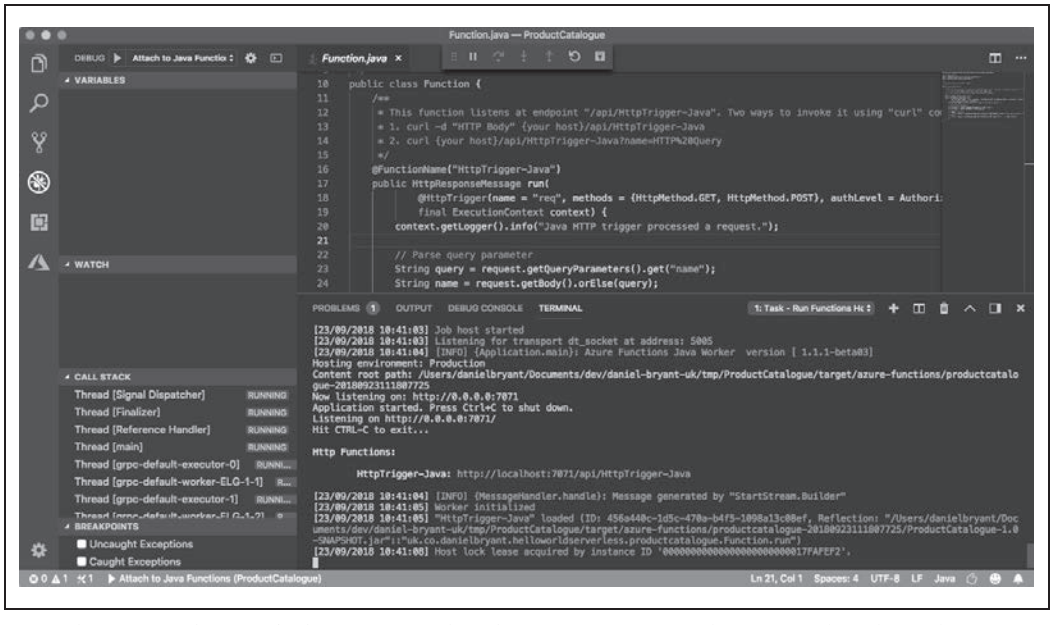
W końcowej części wyniku polecenia `mvn azure-functions:run` są widoczne wszystkie funkcje, które można wywoływać za pomocą zapytań HTTP i przypisanych im ścieżek URL, np. `HttpTrigger-Java: http://localhost:7071/api/HttpTrigger-Java`. W osobnym oknie terminala możesz wywoływać lokalne funkcje za pomocą polecenia `curl`, tak jak w listingu 8.42. Zwróć uwagę, w jaki sposób w argumencie funkcji `HttpTriggerJava` został umieszczony ciąg `Helion`. Dane te, poprzedzone prefiksem `Hello`, zostały następnie zwrócone przez funkcję.

Listing 8.42. Lokalne wywołanie funkcji za pomocą usługi Azure Local i polecenia curl

```
$ curl -w '\n' -d Helion http://localhost:7071/api/HttpTrigger-Java
Witaj, Helion
```

Aby przerwać działanie lokalnej funkcji, przejdź do okna terminala, w którym wywołałeś funkcję, i naciśnij klawisze *Ctrl+C*.

Jeżeli zainstalowałeś edytor Visual Studio Code, możesz dodatkowo zainstalować komponenty Azure Functions Extension (*http://bit.ly/2DwMwsH*) i Java Extension Pack (*http://bit.ly/2xR580l*), dzięki którym będziesz mógł wywoływać funkcje bezpośrednio w edytorze, naciskając klawisz *F5*, tak jak pokazuje rysunek 8.3.



Rysunek 8.3. Wywoływanie funkcji Java w środowisku Azure za pomocą edytora Visual Studio Code

Po wywołaniu funkcji za pomocą edytora możesz ją diagnozować, używając wbudowanego debuggera. W tym celu wystarczy, że ustawisz pułapki na marginesie okna w odpowiednich wierszach kodu, a następnie wywołasz funkcję za pomocą polecenia `curl` lub innego narzędzia testowego.

Testowanie lokalnych i zewnętrznych funkcji za pomocą edytora Visual Studio Code

W niektórych sytuacjach testowanie lokalnych funkcji może być bardzo trudne, gdy np. funkcja jest uzależniona od działającej w chmurze usługi, którą bardzo trudno zastąpić realistyczną imitacją lub atrapą. Korzystając z usługi Azure Functions, można dość łatwo diagnozować funkcje uruchomione w zewnętrznym środowisku chmurowym.

Aby wykonać opisane niżej operacje, musisz utworzyć konto w chmurze Azure i posiadać subskrypcję usług (płatną lub bezpłatną). W celu zalogowania się do chmury Azure za pomocą edytora Visual Studio Code wpisz w polu *Command Palette* (paleta poleceń) polecenie *Sign In* (zaloguj) i postępuj według pojawiających się wskazówek (zazwyczaj otwiera się wtedy w przeglądarce strona logowania do usługi Azure).

Po zalogowaniu kliknij w panelu *Azure: Functions* ikonę *Deploy to Function App* (zainstaluj funkcję) lub wpisz tę nazwę w polu *Command Palette*. Następnie wybierz katalog zawierający projekt, który chcesz wdrożyć, i skonfiguruj go według pojawiających się wskazówek. Po zainstalowaniu funkcji w panelu *OUTPUT* pojawi się skojarzona z funkcją ścieżka, którą możesz wykorzystać z poleceniem `curl`, podobnie jak w przypadku funkcji lokalnej. Listing 8.43 przedstawia przykład wywołania funkcji za pomocą tego polecenia.

Listing 8.43. Wywołanie przy użyciu polecenia curl funkcji wdrożonej za pomocą usługi Azure Functions

```
$ curl -w '\n' https://product-catalogue-5438231.azurewebsites.net/api/httptrigger-java -d Helion
Witaj, Helion
```

Aby diagnozować zewnętrznie uruchomioną funkcję, musisz za pomocą menedżera NPM (ang. *Node Package Manager*, menedżer pakietów Node) zainstalować narzędzie `cloud-debug-tools` w sposób pokazany w listingu 8.44.

Listing 8.44. Instalacja narzędzia cloud-debug-tools za pomocą menedżera NPM

```
$ npm install -g cloud-debug-tools
```

Po zainstalowaniu narzędzia podłącz do uruchomionej funkcji serwer proxy. W tym celu użyj polecenia `dbgproxy` z przypisanym funkcji adresem URL, tak jak pokazuje listing 8.45.

Listing 8.45. Przykład użycia polecenia dbgproxy

```
$ dbgproxy product-catalogue-5438231.azurewebsites.net
Function App:                "product-catalogue-5438231.azurewebsites.net"
Subscription:                "Pay-As-You-Go" (ID = "xxxx")
Resource Group:              "new-java-function-group"
Fetch debug settings:        done
done
done
Set JAVA_OPTS:               done
Set HTTP_PLATFORM_DEBUG_PORT: done
Remote debugging is enabled on "product-catalogue-5438231.azurewebsites.net"
[Server] listening on 127.0.0.1:8898

Now you should be able to debug using "jdb -connect
com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8898"
```

Po podłączeniu serwera proxy skonfiguruj w edytorze Visual Studio Code w pliku *.vscode/launch.json* debugger (listing 8.46) i podłącz go do wskazanego lokalnego portu.

Listing 8.46. Przykładowa konfiguracja debuggera w edytorze Visual Studio Code

```
{
    "name": "Dołączenie debuggera do usługi Azure Funkcje",
    "type": "java",
    "request": "attach",
    "hostName": "localhost",
    "port": 8898
}
```

Teraz możesz ustawiać w edytorze pułapki i podłączać debugger do działającej w chmurze funkcji, wykorzystując panel *DEBUG*. Po wywołaniu zewnętrznej funkcji będziesz mógł ją diagnozować lokalnie.

Podsumowanie

W tym rozdziale dowiedziałeś się, w jaki sposób można najlepiej skonfigurować środowisko programistyczne do lokalnego kompilowania i testowania funkcji. Poznałeś następujące techniki:

- tworzenie atrap i imitacji usług oraz wirtualizowanie usług, do których nie ma dostępu (np. z powodu braku połączenia lub niezbędnych zasobów),
- tworzenie za pomocą narzędzi, takich jak Vagrant, instancji spójnych, powtarzalnych maszyn wirtualnych przeznaczonych do lokalnego rozwijania usług,
- tworzenie za pomocą narzędzi kontenerowych, takich jak minikube i Telepresence, spójnych i łatwo kontrolowanych środowisk przeznaczonych do tworzenia lokalnych i zewnętrznych usług,
- tworzenie kodu FaaS i lokalne zarządzanie infrastrukturą za pomocą narzędzia AWS SAM Local,
- lokalne diagnozowanie aplikacji Java uruchomionych w zewnętrznej usłudze Azure Funkcje za pomocą narzędzia `cloud-debug-tools`.

W następnym rozdziale poznasz ciągłą integrację i pierwsze etapy procesu ciągłego dostarczania oprogramowania.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

Z

O'REILLY®

Java i CD: tak zdobędziesz prawdziwą przewagę!

W ciągu ostatnich lat radykalnie zmieniły się wymagania i oczekiwania biznesowe wobec oprogramowania. Kluczowymi wartościami są innowacyjność, szybkość i czas wejścia na rynek. Do spełnienia tych wymagań konieczne okazały się nowe architektury i modele tworzenia kodu. Metodyka ciągłego dostarczania, zwanego też CD, polega na tworzeniu w krótkich cyklach wartościowych i solidnych produktów. Funkcjonalności są dodawane w małych krokach, a oprogramowanie można wydawać niezawodnie w dowolnej chwili. Dzięki takim działaniom, możemy też szybko otrzymywać informacje zwrotne. Jednak opisywany sposób pracy wymaga odpowiednich ram organizacyjnych, a zespół projektowy musi przyswoić nieco inny od tradycyjnego styl pracy.

Ta książka jest praktycznym przewodnikiem, dzięki któremu programiści Javy opanują techniki potrzebne do pomyślnego zastosowania metody ciągłego dostarczania. Opisano tu najlepsze zasady budowy architektury oprogramowania, automatycznej kontroli jakości, pakowania aplikacji i wdrażania ich w różnych środowiskach produkcyjnych. Szczególną uwagę poświęcono testowaniu oprogramowania: przedstawiono całą gamę metodyk testowania, opisano ich zastosowanie i znaczenie w cyklu życia aplikacji. Ciekawym elementem książki są informacje o złych praktykach i antywzorcach wraz ze wskazówkami dotyczącymi rozwiązywania tego rodzaju problemów.

W tej książce między innymi:

- solidne podstawy ciągłego dostarczania oprogramowania
- migracja do ciągłego dostarczania oprogramowania
- narzędzia: Jenkins, PMD i FindSecBugs
- zasady testowania funkcjonalności i jakości oprogramowania
- techniki obserwacji aplikacji w środowisku produkcyjnym

Daniel Bryant — specjalizuje się we wdrażaniu procesów ciągłego dostarczania, w identyfikowaniu strumieni wartości, tworzeniu procesów kompilacyjnych i implementowaniu strategii testowania. Jest znawcą narzędzi DevOps, platform chmurowych i kontenerowych, mikrousług, a także ekspertem Javy.

Abraham Marín-Pérez — programuje w językach Java i Scala. Jest członkiem społeczności London Java Community i doradcą zawodowym w londyńskiej grupie Meet a Mentor. Lubi dzielić się swoim doświadczeniem z innymi. Mieszka w Londynie, dla relaksu gotuje i wędruje po górach.

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 978-83-283-5633-7
9 788328 356337
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 69,00 zł