



Technologia i rozwiązania

Mikroustługi w Javie

Poradnik eksperta



Sourabh Sharma

[PACKT] open source*
PUBLISHING community experience distilled

Tytuł oryginału: Mastering Microservices with Java

Tłumaczenie: Krzysztof Rychlicki-Kicior

ISBN: 978-83-283-3218-8

Copyright © Packt Publishing 2016

First published in the English language under the title
'Mastering Microservices with Java – (9781785285172)'.

Polish edition copyright © 2017 by Helion SA. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

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

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorach	9
O recenzencie	10
Przedmowa	11
Rozdział 1. Koncepcja rozwiązania	15
Rozwój mikrousług	16
Omówienie architektury monolitycznej	17
Problemy architektury monolitycznej i ich rozwiązania w mikrousługach	17
Jednowymiarowa skalowalność	20
Wycofanie wersji produkcyjnej w razie problemów	21
Problemy związane z wdrażaniem nowych technologii	21
Mikrousługi a metodyki zwinne	22
Łatwość tworzenia oprogramowania — da się lepiej!	23
Budowanie mikrousług — kolejność wykonywania działań	24
Wdrażanie mikrousług w kontenerach na przykładzie Dockera	25
Podsumowanie	28
Rozdział 2. Konfiguracja środowiska programistycznego	29
Konfiguracja Spring Boot	30
Omówienie technologii Spring Boot	30
Dodajemy Spring Boot do przykładowego projektu	31
Dodajemy wbudowany serwer Jetty	33
Przykładowa aplikacja typu REST	34
Piszemy klasę kontrolera REST	35
Uruchamianie przykładowej aplikacji	38
Konfiguracja procesu budowania aplikacji	39
Uruchomienie narzędzia Maven	39
Wykonywanie polecenia w terminalu	40

Testowanie API za pomocą rozszerzenia Postman	40
Prawidłowe przypadki testowe	43
Nieprawidłowe przypadki testowe	44
Instalacja i konfiguracja środowiska NetBeans	45
Źródła	48
Podsumowanie	49
Rozdział 3. Projektowanie sterowane modelem dziedziny	51
Podstawy projektowania sterowanego modelem dziedziny	52
Pojęcia, terminy, definicje	53
Wszelchobecne słownictwo	53
Wielowarstwowa architektura	54
Artefakty związane z projektowaniem sterowanym modelem dziedziny	55
Projektowanie strategiczne i jego założenia	63
Ograniczony kontekst	64
Integracja ciągła	64
Mapa kontekstu	65
Przykładowa usługa dziedziny	68
Implementacja encji	69
Implementacja repozytorium	70
Implementacja usługi	72
Podsumowanie	73
Rozdział 4. Implementujemy mikrousługę	75
Omówienie systemu OTRS	76
Opracowywanie i implementacja mikrousług	77
Mikrousługa Restaurant	78
Usługi użytkowników i rezerwacji	87
Rejestracja i wykrywanie usług (usługa Eureka)	87
Wykonanie	88
Testowanie	88
Źródła	91
Podsumowanie	91
Rozdział 5. Wdrażanie i testowanie	93
Podstawy architektury mikrousług w Netflix OSS	93
Równoważenie obciążenia	95
Równoważenie obciążenia po stronie klienta	95
Równoważenie obciążenia po stronie serwera	98
Wyłącznik automatyczny a monitorowanie	101
Stosowanie metod awaryjnych aplikacji Hystrix	101
Monitorowanie usług	102
Konfiguracja pulpitu Hystrix	103
Konfiguracja aplikacji Turbine	105
Wdrażanie mikrousług za pomocą kontenerów	106
Instalacja i konfiguracja	106
Źródła	115
Podsumowanie	116

Rozdział 6. Mikrousługi a bezpieczeństwo	117
Dodanie obsługi protokołu SSL	117
Uwierzytelnianie i autoryzacja	120
OAuth 2.0	121
Specyfikacja OAuth 2.0 — krótko i na temat	122
Implementacja OAuth z wykorzystaniem Spring Security	138
Źródła	147
Podsumowanie	148
Rozdział 7. Użytkowanie mikrousług za pomocą aplikacji webowej	149
Ogólne założenia frameworka AngularJS	150
MVC	150
MVVM	150
Moduły	151
Dostawcy i usługi	152
Zakresy	153
Kontrolery	153
Filtry	153
Dyrektywy	154
Router interfejsu użytkownika — UI-Router	154
Implementacja funkcji systemu OTRS	155
Lista restauracji/strona domowa	155
Wyszukiwanie restauracji	167
Widok szczegółów restauracji z opcją rezerwacji	168
Strona logowania	169
Potwierdzenie rezerwacji	172
Konfiguracja aplikacji webowej	172
Źródła	183
Podsumowanie	184
Rozdział 8. Dobre praktyki i istotne reguły tworzenia mikrousług	185
Właściwy sposób myślenia	185
Dobre praktyki i przydatne reguły	187
Nanousługa (niezalecana), rozmiar i monolityczność	187
Ciągła integracja i wdrażanie	188
Automatyzacja testów end-to-end	189
Automonitorowanie i logowanie	190
Oddzielny magazyn danych dla każdej mikrousługi	191
Granice transakcji	192
Narzędzia i frameworki do tworzenia mikrousług	193
Netflix Open Source Software (OSS)	193
Źródła	199
Podsumowanie	199

Rozdział 9. Rozwiązywanie problemów	201
Obsługa logów i stos ELK	201
Krótkie wprowadzenie	202
Konfiguracja stosu ELK	204
Zastosowanie skorelowanych ID dla wywołań usług	207
Jak rozwiązać ten problem?	207
Zależności i wersje	207
Zależności cykliczne i ich wpływ	207
Zarządzanie różnymi wersjami	208
Dowiedz się więcej	208
Źródła	209
Podsumowanie	209
Skorowidz	211

Implementujemy mikrousługę

W tym rozdziale przejdziemy od projektu do implementacji naszej przykładowej aplikacji — **internetowego systemu rezerwacji stolików** (Online Table Reservation System — OTRS). Skorzystamy z projektu opracowanego w poprzednim rozdziale i rozszerzymy go, aby stworzyć mikrousługę. Poznasz tu zasady implementacji projektu, ale także inne ważne aspekty tworzenia mikrousług — budowanie, testowanie i tworzenie archiwów aplikacji. Choć skoncentrujemy się na implementacji usługi restauracji, podobne podejście można zastosować również przy tworzeniu innych usług wchodzących w skład OTRS.

W tym rozdziale zajmiemy się następującymi zagadnieniami:

- Omówimy ogólne założenia systemu OTRS.
- Zaimplementujemy mikrousługę.
- Przeprowadzimy testy.

Skorzystamy z kluczowych konceptów DDD, które omówiliśmy w poprzednim rozdziale. Zastosowaliśmy je wtedy do stworzenia modelu dziedzinowego w języku Java. Teraz przejdziemy od prostej implementacji dziedziny do implementacji opartej na frameworku Spring. Skorzystamy z technologii Spring Boot, aby zaimplementować koncepcje wynikające z projektowania sterowanego modelem dziedziny, a także przekształcimy je z języka Java do modelu opartego na frameworku Spring.

Użyjemy także frameworka Spring Cloud, który pozwoli nam na stworzenie rozwiązania chmurowego. Spring Cloud również wykorzystuje Spring Boot, dzięki któremu możemy zastosować wbudowany kontener aplikacji (Tomcat lub Jetty) wewnątrz Twojej usługi, opakowanej

jako JAR lub WAR. Archiwum jest uruchamiane jako osobny proces, a mikrousługa zajmie się dostarczaniem odpowiedzi dla wszystkich żądań, które zostały opisane na liście końcówek w danej usłudze.

Spring Cloud można zintegrować także z usługą Netflix Eureka — rejestrem usług. OTRS skorzysta z tej usługi do rejestracji i wykrywania mikrousług.

Omówienie systemu OTRS

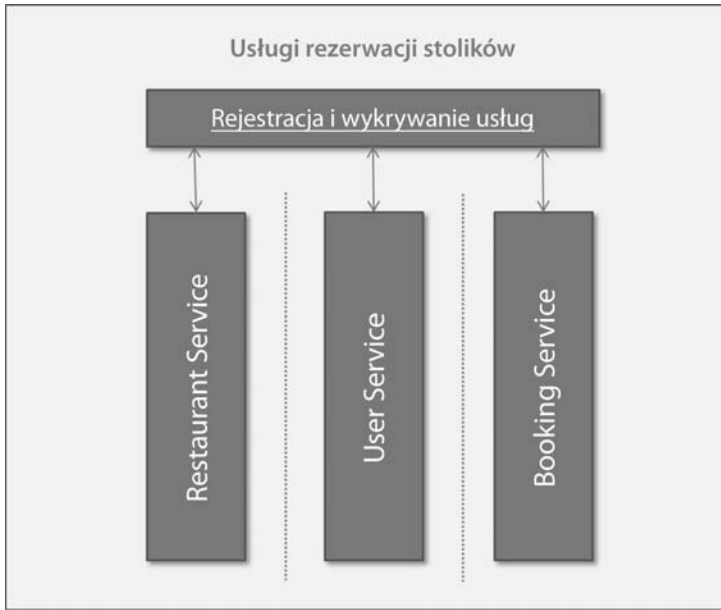
Znając zasady tworzenia mikrousług, możemy stwierdzić, że każda część aplikacji, która jest w stanie funkcjonować niezależnie, powinna być odrębną mikrousługą. W przypadku systemu OTRS możemy wyróżnić trzy główne mikrousługi — restaurację, rezerwację i użytkownika. Oczywiście, można definiować również inne mikrousługi, jednak my skupimy się na tych trzech. Aby były one w pełni niezależne, utworzymy dla nich odrębne bazy danych.

Zakres funkcjonalny wymienionych usług można opisać następująco:

- Usługa restauracji (Restaurant): umożliwia podstawowe zarządzanie zasobami restauracji — tzw. **CRUD** (ang. *Create, Read, Update, Delete* — dodawanie, odczyt, modyfikacja i usunięcie), a dodatkowo ich przeszukiwanie z wykorzystaniem rozmaitych kryteriów. Pozwala również na wiązanie restauracji i stolików. Restauracja udostępni także encje stolika (Table).
- Usługa użytkownika (User): pozwoli na wykonanie operacji typu CRUD dla encji User.
- Usługa rezerwacji (Booking): pozwoli na dokonywanie rezerwacji (operacje typu CRUD) na podstawie usług Restaurant i User. Przeszukiwanie dostępne w usłudze Restaurant pozwoli na znalezienie wybranej restauracji; lista powiązanych z restauracją stolików umożliwi wybór konkretnego stolika na podstawie informacji o dostępności stolików. Usługa ta utworzy związek pomiędzy encją Restaurant/Table a User.

Rysunek 4.1 podkreśla fakt, że każda z mikrousług działa niezależnie. To właśnie dlatego mikrousługi mogą być tworzone, rozwijane i zarządzane niezależnie, bez wpływu na inne. Każda z usług ma odrębną architekturę warstwową i bazę danych. Nie ma ograniczeń co do stosowanych do tworzenia technologii, frameworków czy języków programowania. W dowolnym momencie możesz też tworzyć nowe mikrousługi, np. do celów księgowych, które byłyby wykorzystywane przez usługę Restaurant. To samo dotyczy usług analitycznych i do raportowania.

Do celów demonstracyjnych zajmiemy się implementacją ograniczoną do trzech wymienionych powyżej usług.



Rysunek 4.1. Mikrousługi a rejestracja i wykrywanie

Opracowywanie i implementacja mikrousług

Teraz zajmiemy się implementacją sterowaną modelem dziedziny, a także podejściem przedstawionym w poprzednim rozdziale w celu implementowania mikrousług za pomocą frameworka Spring Cloud. Przypomnijmy sobie kluczowe artefakty naszego projektu:

- **Encje:** są to obiekty identyfikowalne i niezmiennie przez czas życia produktu/usługi. Obiekty te nie są definiowane za pomocą swoich atrybutów — mają odrębną tożsamość i zachowaną ciągłość istnienia.

Encje mają swoją tożsamość i zachowują ciągłość, a także atrybuty, które jednak nie mają wpływu na ciągłość. **Obiekty wartości** (ang. *Value Objects* — VO) mają za to tylko atrybuty, nie dysponując własną tożsamością. Dobrą praktyką jest unikanie zmian w obiektach wartości. We frameworku encje stanowią zwykłe obiekty języka Java (POJO), dlatego będziemy ich używać także jako obiektów wartości.

- **Usługi:** stanowią typowy element wielu projektów. Są używane w warstwie dziedziny w projektowaniu sterowanym modelem dziedziny. Obiekt usługi nie zawiera wewnętrznego stanu — jego jedynym celem jest udostępnianie zachowania dziedziny. Obiekty usług udostępniają zachowania, które nie są związane z konkretnymi encjami lub obiektami wartości. Obiekty usług udostępniają jedno lub wiele zachowań jednej lub większej liczbie encji lub obiektów wartości. Najlepiej jest definiować usługi jawnie w modelu dziedziny.

- **Obiekt repozytorium:** obiekt ten stanowi element modelu dziedziny, który obsługuje pamięć trwałą, taką jak bazy danych, zewnętrzne źródła itd., aby pozyskiwać utrwalone obiekty. Po otrzymaniu żądania przez repozytorium w celu uzyskania referencji do obiektu, jeśli obiekt jest dostępny bezpośrednio w repozytorium, zwracana jest do niego referencja. W przeciwnym razie najpierw jest on wczytywany z zewnętrznego źródła.

Pobieranie przykładowego kodu

Przykładowy kod jest dostępny na serwerze FTP wydawnictwa Helion, pod adresem <ftp://ftp.helion.pl/przyklady/mikjav.zip>.

- API każdej mikrousługi stanowi usługę sieciową typu REST. API naszego systemu wykorzystuje metody protokołu HTTP, takie jak GET i POST, a także typową strukturę końcówki typu REST. Żądania i odpowiedzi są zdefiniowane za pomocą formatu JSON. Można również stosować język XML.

Mikrousługa Restaurant

Mikrousługa Restaurant zostanie udostępniona za pomocą końcówek typu REST. Poniższe końcówki są dostępne w mikrousłudze Restaurant. Oczywiście, liczba końcówek nie jest ograniczona:

Końcówka	GET /v1/restaurants/<Restaurant_ID>	
Parametry		
Nazwa	Opis	
Restaurant_ID	Parametr ścieżki, który reprezentuje unikalną restaurację skojarzoną z tym ID.	
Żądanie		
Właściwość	Typ	Opis
Brak		
Odpowiedź		
Właściwość	Typ	Opis
Restaurant	Obiekt typu Restaurant	Obiekt restauracji skojarzony z danym ID

Końcówka	GET /v1/restaurants	
Parametry		
Nazwa	Opis	
Brak		
Żądanie		
Właściwość	Typ	Opis
Name	String	Parametr zapytania, który określa nazwę lub fragment nazwy restauracji.
Odpowiedź		
Właściwość	Typ	Opis
Restaurants	Tablica obiektów restauracji	Zwraca wszystkie restauracje, których nazwy zawierają wartość parametru.

Końcówka	POST /v1/restaurants	
Parametry		
Nazwa	Opis	
Brak		
Żądanie		
Właściwość	Typ	Opis
Restaurant	Obiekt restauracji	Reprezentacja obiektu restauracji w formacie JSON
Odpowiedź		
Właściwość	Typ	Opis
Restaurant	Obiekt restauracji	Nowo utworzony obiekt typu Restaurant

Na tej samej zasadzie możemy dodawać rozmaite końcówki i ich implementacje. Do celów demonstracyjnych powyższe końcówki zostaną zaimplementowane za pomocą technologii Spring Cloud.

Klasa kontrolera

Kontroler usługi Restaurant wykorzystuje adnotację `@RestController`, aby wygenerować końcówki usługi. Szczegóły działania tego mechanizmu omówiliśmy w rozdziale 2. `@RestController` to adnotacja na poziomie klasy, która jest używana wobec klas zasobów. Jest to połączenie adnotacji `@Controller` i `@ResponseBody`. Zwraca ona obiekt dziedziny.

Wersjonowanie API

Idąc naprzód, chciałbym wyjaśnić obecność prefiksu v1 w końcówce typu REST. Oznacza ona wersję API. Jest to niezwykle ważna kwestia. Wersjonowanie API jest kluczowe, ponieważ API z czasem ulegają zmianom. Twoja wiedza i doświadczenie zmieniają się z upływem czasu, co prowadzi do wprowadzania zmian w API. Każda zmiana w API może spowodować problemy z działaniem aplikacji klienckich.

Zarządzanie wersjami API można zrealizować na kilka sposobów. Jednym z nich jest podawanie wersji w ścieżce lub w nagłówku HTTP. Nagłówek HTTP może zawierać specjalny nagłówek żądania lub nagłówek Accept, dzięki czemu można określić wersję API. Więcej informacji znajdziesz w książce *REST. Najlepsze praktyki i wzorce w języku Java* Bhaktiego Mehty (Helion, 2015).

```
@RestController
@RequestMapping("/v1/restaurants")
public class RestaurantController {
    protected Logger logger = Logger.getLogger(RestaurantController.class.getName());
    protected RestaurantService restaurantService;
    @Autowired
    public RestaurantController(RestaurantService restaurantService) {
        this.restaurantService = restaurantService;
    }
    /**
     * Pobiera restauracje o określonej nazwie. Niewrażliwość na wielkość znaków jest częściowo obsługiwana.
     * Wywołanie <code>http://.../restaurants/rest</code> znajdzie więc dowolne restauracje zawierające
     * człon 'rest' lub 'REST' w nazwie.
     *
     * @param name
     * @return Niepusta kolekcja restauracji.
     */
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Collection<Restaurant>> findByName(@RequestParam("name")
        ↪String name) {
        logger.info(String.format("Wywołanie metody restaurant-service findByName():{ }
        ↪dla { } ", restaurantService.getClass().getName(), name));
        name = name.trim().toLowerCase();
        Collection<Restaurant> restaurants;
        try {
            restaurants = restaurantService.findByName(name);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Wyjątek metody findByNameREST", ex);
            return new ResponseEntity<Collection<Restaurant>>(HttpStatus.INTERNAL_
            ↪SERVER_ERROR);
        }
        return restaurants.size() > 0 ? new ResponseEntity<Collection<Restaurant>>
        ↪(restaurants, HttpStatus.OK): new ResponseEntity<Collection<Restaurant>>
        ↪(HttpStatus.NO_CONTENT);
    }
    /**
     * Pobiera restaurację dla zadanego ID.
     * <code>http://.../v1/restaurants/{restaurant_id}</code> zwróci restaurację o podanym ID.
     */
}
```

```

*
* @param restaurant_id
* @return Obiekt restauracji.
*/
@RequestMapping(value =("/{restaurant_id}", method = RequestMethod.GET)
public ResponseEntity<Entity> findById(@PathVariable("restaurant_id") String id) {
    logger.info(String.format("Wywołanie restaurant-service findById():{} dla
↳{} ", restaurantService.getClass().getName(), id));
    id = id.trim();
    Entity restaurant;
    try {
        restaurant = restaurantService.findById(id);
    } catch (Exception ex) {
        logger.log(Level.SEVERE, "Wyjątek w wywołaniu findById REST", ex);
        return new ResponseEntity<Entity>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
    return restaurant != null ? new ResponseEntity<Entity>(restaurant,
↳HttpStatus.OK): new ResponseEntity<Entity>(HttpStatus.NO_CONTENT);
}
/**
* Dodaje restaurację na podstawie określonych informacji.
*
* @param Restauracja
* @return Restauracja (bez wartości null)
* @throws RestaurantNotFoundException Jeśli nie udało się znaleźć restauracji.
*/
@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<Restaurant> add(@RequestBody RestaurantVO restaurantVO) {
    logger.info(String.format("Wywołanie restaurant-service add(): %s dla %s",
↳restaurantService.getClass().getName(), restaurantVO.getName()));
    Restaurant restaurant = new Restaurant(null, null);
    BeanUtils.copyProperties(restaurantVO, restaurant);
    try {
        restaurantService.add(restaurant);
    } catch (Exception ex) {
        logger.log(Level.WARNING, "Wyjątek w wywołaniu metody add Restaurant
↳REST "+ ex);
        return new ResponseEntity<Restaurant>(HttpStatus.UNPROCESSABLE_ENTITY);
    }
    return new ResponseEntity<Restaurant>(HttpStatus.CREATED);
}
}
}

```

Klasy usług

Klasa `RestaurantController` wykorzystuje interfejs `RestaurantService`, który wprowadza operacje typu CRUD i przeszukiwanie, zgodnie z poniższą deklaracją:

```

public interface RestaurantService {
    public void add(Restaurant restaurant) throws Exception;
    public void update(Restaurant restaurant) throws Exception;
    public void delete(String id) throws Exception;
    public Entity findById(String restaurantId) throws Exception;
}

```

```

    public Collection<Restaurant> findByName(String name) throws Exception;
    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>>
        ↪name) throws Exception;
}

```

Teraz możemy zaimplementować usługę zgodnie z powyższym interfejsem. Dziedziczy ona również po klasie BaseService, utworzonej w poprzednim rozdziale. Skorzystamy z adnotacji @Service, aby została utworzona usługa:

```

@Service("restaurantService")
public class RestaurantServiceImpl extends BaseService<Restaurant, String>
    ↪implements RestaurantService {
    private RestaurantRepository<Restaurant, String> restaurantRepository;
    @Autowired
    public RestaurantServiceImpl(RestaurantRepository<Restaurant, String>
        ↪restaurantRepository) {
        super(restaurantRepository);
        this.restaurantRepository = restaurantRepository;
    }
    public void add(Restaurant restaurant) throws Exception {
        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Nazwa restauracji nie może mieć wartości null
                ↪ani pustej nazwy");
        }
        if (restaurantRepository.containsName(restaurant.getName())) {
            throw new Exception(String.format("Istnieje już restauracja o tej
                ↪nazwie: - %s", restaurant.getName()));
        }
        super.add(restaurant);
    }
    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        return restaurantRepository.findByName(name);
    }
    @Override
    public void update(Restaurant restaurant) throws Exception {
        restaurantRepository.update(restaurant);
    }
    @Override
    public void delete(String id) throws Exception {
        restaurantRepository.remove(id);
    }
    @Override
    public Entity findById(String restaurantId) throws Exception {
        return restaurantRepository.get(restaurantId);
    }
    @Override
    public Collection<Restaurant> findByCriteria(Map<String, ArrayList<String>>
        ↪name) throws Exception {
        throw new UnsupportedOperationException("Metoda nie została
            ↪zaimplementowana. ");
        // Aby zmienić treść generowanych metod, wybierz opcję Tools/Templates.
    }
}

```

Klasy repozytorium

Interfejs `RestaurantRepository` definiuje dwie nowe metody: `containsName` i `findByName`. Dziedziczy on także po interfejsie `Repository`:

```
public interface RestaurantRepository<Restaurant, String> extends
↳Repository<Restaurant, String> {
    boolean containsName(String name) throws Exception;
    Collection<Restaurant> findByName(String name) throws Exception;
}
```

Interfejs `Repository` wprowadza trzy nowe metody: `add`, `remove` i `update`. Dziedziczy on także po interfejsie `ReadOnlyRepository`:

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {
    void add(TE entity);
    void remove(T id);
    void update(TE entity);
}
```

Definicja interfejsu `ReadOnlyRepository` zawiera metody `get` i `getAll`, które zwracają wartości logiczne, encję i kolekcję encji. Ma to sens, gdy chcesz udostępnić jedynie mechanizmy tylko do odczytu:

```
public interface ReadOnlyRepository<TE, T> {
    boolean contains(T id);
    Entity get(T id);
    Collection<TE> getAll();
}
```

Framework Spring pozwala na zastosowanie adnotacji `@Repository` do wskazywania klasy ziarna, która pełni funkcję repozytorium. W przypadku klasy `RestaurantRepository` stosujemy mapę (słownik), która została użyta w miejscu bazy danych. W ten sposób wszystkie encje są przechowywane w pamięci. W związku z tym po uruchomieniu usługi w pamięci będą tylko dwie restauracje. Możemy skorzystać z technologii JPA jako metody utrwalania danych w bazie danych. Tak na ogół postępuje się w aplikacjach produkcyjnych:

```
@Repository("restaurantRepository")
public class InMemRestaurantRepository implements RestaurantRepository<Restaurant,
↳String> {
    private Map<String, Restaurant> entities;
    public InMemRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Restauracja z burgerami", "1", null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("Restauracja burgerowa", "2", null);
        entities.put("2", restaurant);
    }
    @Override
    public boolean containsName(String name) {
```

```

        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
            //Obsługa wyjątków
        }
        return false;
    }
    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }
    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }
    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getId())) {
            entities.put(entity.getId(), entity);
        }
    }
    @Override
    public Collection<Restaurant> findByName(String name) throws Exception {
        Collection<Restaurant> restaurants = new ArrayList();
        int noOfChars = name.length();
        entities.forEach((k, v) -> {
            if (v.getName().toLowerCase().contains(name.subSequence(0, noOfChars))) {
                restaurants.add(v);
            }
        });
        return restaurants;
    }
    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Metoda nie została
        ↪zaimplementowana");
        // Aby zmienić treść generowanych metod, wybierz opcję Tools/Templates.
    }
    @Override
    public Entity get(String id) {
        return entities.get(id);
    }
    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
}

```


Klasy encji

Encja `Restaurant`, która dziedziczy po klasie `BaseEntity`, ma następującą definicję:

```
public class Restaurant extends BaseEntity<String> {
    private List<Table> tables = new ArrayList<>();
    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }
    public void setTables(List<Table> tables) {
        this.tables = tables;
    }
    public List<Table> getTables() {

        return tables;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, nazwa: {}, liczba miejsc: {}",
            ↪this.getId(), this.getName(), this.getCapacity()));
        return sb.toString();
    }
}
```

Skoro w definicjach klas korzystamy z klas POJO, nie musimy tworzyć wielu obiektów wartości. Trzeba tylko pamiętać, aby nie modyfikować stanu encji.

Encja `Table` dziedziczy po encji `BaseEntity`:

```
public class Table extends BaseEntity<BigInteger> {
    private int capacity;
    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }
    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
    public int getCapacity() {
        return capacity;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, nazwa: {}", this.getId(), this.getName()));
        sb.append(String.format("Stoliki: {}" + Arrays.asList(this.getTables())));
        return sb.toString();
    }
}
```

Treść klasy abstrakcyjnej Entity jest przedstawiona poniżej:

```
public abstract class Entity<T> {
    T id;
    String name;
    public T getId() {
        return id;
    }
    public void setId(T id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Klasa abstrakcyjna BaseEntity jest zdefiniowana w poniższy sposób. Dziedziczy ona po klasie Entity:

```
public abstract class BaseEntity<T> extends Entity<T> {
    private T id;
    private boolean isModified;
    private String name;
    public BaseEntity(T id, String name) {
        this.id = id;
        this.name = name;
    }
    public T getId() {
        return id;
    }
    public void setId(T id) {
        this.id = id;
    }
    public boolean isIsModified() {
        return isModified;
    }
    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Usługi użytkowników i rezerwacji

Implementacja `RestaurantService` stanowi podstawę do dalszej pracy nad implementacją usług `Booking` i `User`. Usługa `User` może udostępniać końcówkę związaną z operacjami typu CRUD dla użytkowników, zaś usługa `Booking`, poza operacjami typu CRUD, może weryfikować dostępność stolików. Cały kod źródłowy znajdziesz na serwerze FTP wydawnictwa Helion.

Rejestracja i wykrywanie usług (usługa Eureka)

Framework Spring Cloud oferuje wzorcowe wsparcie dla usługi Netflix Eureka, rejestru usług i narzędzia do ich wykrywania. Wszystkie usługi, które uruchamiasz, są dołączane do usługi Eureka i mogą być przez nią wykrywane. Dokonuje się to dzięki konfiguracji klienta usługi Eureka w projekcie usługi.

Najpierw musimy dodać zależność frameworka Spring Cloud w pliku `pom.xml` i klasę startową wraz z adnotacją `@EnableEurekaApplication`:

Zależność Mavena:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

Klasa startowa:

Klasa startowa `App` wywoła automatycznie usługę Eureka tylko i wyłącznie dzięki adnotacji `@EnableEurekaApplication`:

```
package com.packtpub.mmj.eureka.service;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.
EnableEurekaServer;
@SpringBootApplication
@EnableEurekaServer
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Możesz też skorzystać ze znacznika `<start-class>com.packtpub.mmj.eureka.service.App</start-class>` w znaczniku `<properties>` w pliku `pom.xml`.

Konfiguracja Spring:

Usługa Eureka wymaga dodania niezbędnej konfiguracji serwerowej do pliku `src/main/resources/application.yml`:

```

server:
  port: ${vcap.application.port:8761} # port HTTP
eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0

```

Każda z usług systemu OTRS powinna zawierać konfigurację klienta Eureka, dzięki czemu można nawiązać połączenie pomiędzy klientem a serwerem Eureka. Bez tego nie będą możliwe wykrywanie i rejestracja usług.

Klient Eureka: wszystkie Twoje usługi powinny zawierać następującą sekcję konfiguracyjną:

```

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

```

Wykonanie

Aby sprawdzić nasz kod w praktyce, musisz go najpierw zbudować. W tym celu wykonamy polecenie `clean package` dostępne w Mavenie, co doprowadzi do zbudowania archiwum.

Aby uruchomić archiwum z usługą, wykonaj poniższe polecenie:

```
java -jar target/<plik_uslugi_jar>
```

Na przykład:

```

java -jar target/restaurant-service.jar
java -jar target/eureka-service.jar

```

Testowanie

Aby móc korzystać z testów jednostkowych, dodaj poniższą zależność do pliku `pom.xml`:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>

```

Przetestowanie klasy `RestaurantController` wymaga dodania następujących plików:

- Klasa `RestaurantControllerIntegrationTests`, która wykorzystuje adnotację `@SpringApplicationConfiguration`, aby zastosować tę samą konfigurację, co Spring Boot:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = RestaurantApp.class)
public class RestaurantControllerIntegrationTests extends
↳AbstractRestaurantControllerTests {
}
```

- Abstrakcyjna klasa, w której napiszemy nasze testy:

```
public abstract class AbstractRestaurantControllerTests {
    protected static final String RESTAURANT = "1";
    protected static final String RESTAURANT_NAME = "Restauracja Test";
    @Autowired
    RestaurantController restaurantController;
    @Test
    public void validResturantById() {
        Logger.getGlobal().info("Start testu validResturantById");
        ResponseEntity<Entity> restaurant = restaurantController.findById(RESTAURANT);
        Assert.assertEquals(HttpStatus.OK, restaurant.getStatusCode());
        Assert.assertTrue(restaurant.hasBody());
        Assert.assertNotNull(restaurant.getBody());
        Assert.assertEquals(RESTAURANT, restaurant.getBody().getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.getBody().getName());
        Logger.getGlobal().info("Koniec testu validResturantById");
    }
    @Test
    public void validResturantByName() {
        Logger.getGlobal().info("Start testu validResturantByName");
        ResponseEntity<Collection<Restaurant>> restaurants =restaurantController.
↳findByName(RESTAURANT_NAME);
        Logger.getGlobal().info("Wewnątrz testu validAccount");
        Assert.assertEquals(HttpStatus.OK, restaurants.getStatusCode());
        Assert.assertTrue(restaurants.hasBody());
        Assert.assertNotNull(restaurants.getBody());
        Assert.assertFalse(restaurants.getBody().isEmpty());
        Restaurant restaurant = (Restaurant) restaurants.getBody().toArray()[0];
        Assert.assertEquals(RESTAURANT, restaurant.getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.getName());
        Logger.getGlobal().info("Koniec testu validResturantByName");
    }
    @Test
    public void validAdd() {
        Logger.getGlobal().info("Start testu validAdd");
        RestaurantVO restaurant = new RestaurantVO();
        restaurant.setId("999");
        restaurant.setName("Test Restaurant");
        ResponseEntity<Restaurant> restaurants = restaurantController.add(restaurant);
        Assert.assertEquals(HttpStatus.CREATED, restaurants.getStatusCode());
        Logger.getGlobal().info("Koniec testu validAdd");
    }
}
```

- Na zakończenie tworzymy klasę `RestaurantControllerTests`, która dziedziczy po utworzonej przed chwilą abstrakcyjnej klasie, a także tworzy obiekty typów `RestaurantService` i `RestaurantRepository`:

```

public class RestaurantControllerTests extends AbstractRestaurantControllerTests {
    protected static final Restaurant restaurantStaticInstance = new
        ↪Restaurant(RESTAURANT, RESTAURANT_NAME, null);
    protected static class TestRestaurantRepository implements
        ↪RestaurantRepository<Restaurant, String> {
        private Map<String, Restaurant> entities;
        public TestRestaurantRepository() {
            entities = new HashMap();
            Restaurant restaurant = new Restaurant("Restauracja Test", "1", null);
            entities.put("1", restaurant);
            restaurant = new Restaurant("Test restauracja", "2", null);
            entities.put("2", restaurant);
        }
        @Override
        public boolean containsName(String name) {
            try {
                return this.findByName(name).size() > 0;
            } catch (Exception ex) {
                // Obsługa wyjątku
            }
            return false;
        }
        @Override
        public void add(Restaurant entity) {
            entities.put(entity.getId(), entity);
        }
        @Override
        public void remove(String id) {
            if (entities.containsKey(id)) {
                entities.remove(id);
            }
        }
        @Override
        public void update(Restaurant entity) {
            if (entities.containsKey(entity.getId())) {
                entities.put(entity.getId(), entity);
            }
        }
        @Override
        public Collection<Restaurant> findByName(String name) throws Exception {
            Collection<Restaurant> restaurants = new ArrayList();
            int noOfChars = name.length();
            entities.forEach((k, v) -> {
                if (v.getName().toLowerCase().contains(name.subSequence(0,
                    ↪noOfChars))) {
                    restaurants.add(v);
                }
            });
        }
    }
}

```

```

        return restaurants;
    }
    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Metoda nie została
        ↪zaimplementowana.");
    }
    // Aby zmienić treść generowanych metod, wybierz opcję Tools/Templates.
    }
    @Override
    public Entity get(String id) {
        return entities.get(id);
    }
    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
}
protected TestRestaurantRepository testRestaurantRepository =new
    ↪TestRestaurantRepository();
protected RestaurantService restaurantService = new
    ↪RestaurantServiceImpl(testRestaurantRepository);
@Before
public void setup() {
    restaurantController = new RestaurantController(restaurantService);
}
}

```

Źródła

- Bhakti Mehta, *REST. Najlepsze praktyki i wzorce w języku Java*, Helion, Gliwice 2015
- Spring Cloud: <http://cloud.spring.io>
- Netflix Eureka: <https://github.com/netflix/eureka>

Podsumowanie

W tym rozdziale zapoznałeś się z praktycznym zastosowaniem projektu sterowanego modelem dziedziny w implementacji mikrousługi. Po uruchomieniu demonstracyjnej aplikacji wiesz już, jak tworzyć, wdrażać i testować mikrousługi niezależnie. Z pewnością jesteś w stanie tworzyć mikrousługi za pomocą Spring Cloud. Dowiedziałeś się także, jak zastosować usługę Eureka, aby rejestrować i wykrywać mikrousługi dzięki technologii Spring Cloud.

W następnym rozdziale dowiesz się, jak wdrażać mikrousługi w kontenerach, takich jak Docker. Nauczysz się także testować mikrousługi za pomocą klientów typu REST pisanych w Javie i innych narzędzi.

Skorowidz

A

- adnotacja
 - @ComponentScan, 39
 - @Configuration, 38
 - @Controller, 35, 79
 - @EnableAutoConfiguration, 38
 - @EnableCircuitBreaker, 101
 - @EnableDiscoveryClient, 98
 - @EnableEurekaApplication, 87
 - @EnableWebMvc, 38
 - @EnableZuulProxy, 98
 - @HystrixCommand, 101
 - @PathVariable, 36
 - @Repository, 83
 - @RequestMapping, 35
 - @RequestParam, 36
 - @ResponseBody, 35, 79
 - @RestController, 35
 - @SpringBootApplication, 38
- agregat, 58, 59
- Amazon, 17, 51
- Amazon Machine Image, 193
- Amazon Web Services, 193
- AMI, 193
- Aminator, 194
- AMQP, 103
- AngularJS, 150, 151, 152
 - dyrektywa, 154
 - zakres, 153
- Ant, 30
- Apache Cassandra, 93
- Apache Mesos, 198
- API, 20
 - brama, *Patrz:* brama API
 - matematyczne, 34
 - wersja, 80
- API gateway, *Patrz:* brama API
- aplikacja
 - agentowa, 126
 - agentowe, 127
 - implementacja, 185
 - jednostronicowa, 154, 155
 - monolityczna, *Patrz:* architektura monolityczna
 - natywna, 126, 127
 - produkcyjna, 83
 - publiczna, 127
 - webowa, 126
 - strona domowa, 155
- Archaius, 197
- architektura
 - bezkontenerowa, 30
 - mikrouslug, 17, 93, 186, 187, 191, 207
 - monolityczna, 17, 18, 21, 186, 188, 207
 - kod, 23
 - metodyka zwinna, 22
 - skalowalnosc, 20
 - wdrazanie, 21, 22
 - z uslugami, 18
 - wielowarstwowa, 54
 - zorientowana na uslugi, *Patrz:* SOA
- artefakt, 25, 55
- atak brute force, 129
- Atlas, 195

autoryzacja, 119, 120
AWS, 193

B

baza danych, 18, 60, 76, 78, 191, 203
współdzielenie, 192
zarządzanie, *Patrz:* MDM
biblioteka, 55
Angular, 151
Netflix Spectator, 195
boot2docker, 27
Bower, 177
brama API, 19

C

Camel, 192
certyfikat, 119, 120
classpath, *Patrz:* ścieżka klas
Cloud Foundry, 194
Cockcroft Adrian, 16, 190
CRUD, 76, 81

D

DDD, 51, 52, 53, 57, 68, 75, 188
destylacja, 63, 68
Docker, 25, 26, 106, 202
architektura, 27
daemon, 27
dziennik, 202
dziennik obrazu, 115
klient, 27
kontener, 28
zależny, 114
zarządzanie, *Patrz:* Docker Compose
łączenie z Mavenem, 107, 108, 110, 113
menedżer kontenerów, *Patrz:* Docker Compose
obraz, 28, 113, 114
rejestr lokalny, 114
uruchamianie, 27
Docker Compose, 113, 115
dziennik, 202
Docker Hub, 27
Docker Toolbox, 106
DOM, 151
Domain-Driven Design, *Patrz:* DDD
dostawca, 152

drzewo dokumentu, *Patrz:* DOM
dyrektywa links, 114
dziennik logów, 201

E

eBay, 17
Edda, 196, 197
Elasticsearch, 203, 204
encja, 55, 56, 57, 76, 77, 83, 124
implementacja, 69
kolekcja, 83
korzeń, *Patrz:* korzeń
przechowywanie w pamięci, 83
Enterprise Service Bus, *Patrz:* ESB
ESB, 16
Etsy, 20
Eureka, *Patrz:* Netflix Eureka
Evans Eric, 52

F

fabryka, 61
FeignClient, 95
Fenzo, 198
FIDO, 191, 198, 199
Fielding Roy, 35
filtr, 153, 166
format
jar, 31
JSON, 78
formularz, 171
wyszukiwania, 167
funkcja
Math.pow, 35
Math.sqrt, 35

G

Gancarz Mike, 16, 187
Google Computer Engine, 194
Gradle, 30, 47
grant
danych uwierzytelniających klienta, 130, 137
hasła właściciela zasobu, 130, 135, 136, 146
kodu autoryzacji, 130, 142
niejawny, 130, 134, 145
uprawnień klienckich, 146
Groupon, 17

H

histogram czasu odpowiedzi, 190
 historyjka użytkownika, 22
 HTTP, 117, 194
 nagłówek, 80
 HTTPS, 118
 Hudson, 65
 Hystrix, *Patrz:* Netflix Hystrix
 Hystrix Dashboard, 94

I

Ice, 198
 identyfikator, 56
 integracja ciągła, 63, 64, 65
 Interface Segregation Principle, *Patrz:* ISP
 interfejs
 cname, 26
 DiscoveryEnabledNIWSServerList, 95
 IPing, 95
 jądra systemu operacyjnego, 26
 użytkownika, 20, 23, 27, 54
 invariant, *Patrz:* niezmiennik
 ISP, 69

J

jądro współdzielone, 63, 66
 Jenkins, 24, 188
 Jenkins CI, 65
 język modelowania zunifikowany, *Patrz:* UML

K

Kibana, 204, 206
 klasa
 BaseEntity, 85
 DiscoveryClient, 95
 Entity, 86
 POJO, 85
 resource, 35
 startowa, 87
 ścieżka, *Patrz:* ścieżka klas
 klient
 identyfikator, 128
 poufny, 125
 profil, 126

 publiczny, 125
 uwierzytelnienie, 128
 klucz, 33
 dostępowy, 125
 główny, 56
 magazyn, *Patrz:* magazyn kluczy
 obcego, 59
 kod
 scalanie, 65
 testowanie, 65
 kolejka komunikatów, 103
 kontekst, 64
 mapa, *Patrz:* mapa kontekstu
 ograniczenie, 63, 64
 kontener, 25, 106
 Docker, *Patrz:* Docker
 intermodalny, 25
 oprogramowania, 25
 zależny, 114
 kontroler, 150, 153, 164, 165
 REST, 35
 końcówka, 78, 129, 159
 autoryzacji, 129
 prefiks v1, 80
 przekierowania, 129
 tokena, 129
 wywołanie, 207
 korzeń, 59

L

latencja, 190
 logika
 aplikacji, 18, 55, 150
 biznesowa, 18, 55, 150
 logowanie, 120, 122, 169, 170, 191
 rejestrwanie zdarzeń, 191
 Logstash, 203, 205
 Long Josh, 31

M

magazyn kluczy, 33, 119
 magistrała usług korporacyjna, *Patrz:* ESB
 mała
 Chaos, 196
 Conformity, 196
 Janitor, 196
 Security, 196

mapa kontekstu, 63, 65
 Master Data Management, *Patrz:* MDM
 maszyna wirtualna, *Patrz:* VM
 Javy, *Patrz:* JVM
 Maven, 29, 30, 33, 39, 47, 87, 88, 103, 105, 107, 108, 110, 113
 metoda
 awaryjna, 102
 config, 152
 fabryka, 153
 get, 83
 GET, 78
 getAll, 83
 getLocalServiceInstance, 95
 main, 34, 151
 POST, 78
 run, 152
 sqrt, 37
 stała, 152
 usługa, 153
 wartość, 152
 middleware, *Patrz:* oprogramowanie
 pośredniczące
 mikrousluga, 16, 17, 18, 20, 187, *Patrz też:* usługa
 debugowanie, 207
 granice transakcji, 192
 monitorowanie, 190, 191
 w czasie rzeczywistym, 191
 skalowanie, 20, *Patrz też:* skalowanie
 testowanie, 22, 23, 24
 tworzenie, 24, 75, 76, 193
 wdrażanie, 21, 22, 23, 24, 93
 Docker, 25, 28
 wielkość, 187
 zależność cykliczna, 207, 208
 model, 150
 dziedziny, 52, 57, 69, 78
 implementacja, 52
 integralność, 63
 korporacyjny, 63
 refaktoring, 64
 widoku, 150
 moduł, 63, 151
 MVC, 150
 MVVM, 150

N

nanousługa, 187
 NetBeans, 45, 46, 47
 NetBeans IDE, 29
 Netflix, 23, 93
 Netflix Atlas, 191, 195
 Netflix Edda, 196, 197
 Netflix Eureka, 76, 87, 88, 94, 194
 Netflix Hystrix, 94, 101, 102, 103, 194
 Netflix Nebula, 193
 Netflix OSS, 193
 Netflix Ribbon, 94, 95, 194
 Netflix Spectator, 195
 Netflix Turbine, 94, 103, 105
 niezmiennik, 59
 Node.js, 172, 177
 npm, 172, 177

O

OAuth, 120
 grant autoryzacji, *Patrz:* grant
 implementacja, 138
 końcówka, 129
 specyfikacja, 122
 wersja, 121
 zastosowania, 121
 obiekt
 cykl życia, 58, 60
 dziedziny, 60, 79
 kolekcja, 59
 repozytorium, 78
 separacja, 63
 specjalizowany, 152
 usług, 57
 usuwanie, 59
 wartości, 56, 57, 77
 Object-Oriented Programming, *Patrz:* OOP
 obsługa błędów, 102
 ograniczenie, 59
 OOP, 57
 Open Source Software Center, *Patrz:* OSS
 oprogramowanie pośredniczące, 16
 OSS, 93

P

plik

app.js, 159, 167
 application.yml, 95, 98, 107
 index.html, 151, 156
 JAR, 33, 34
 pom.xml, 25, 31, 87, 88, 108

podmodel, 63

POJO, 34, 85

polecenie

clean package, 88
 logs, 202
 mvn clean, 40

Postman, 40

Postman Chrome, 40

potokowanie, 203

programowanie

sterowane testami, *Patrz:* TDD
 zorientowane obiektowo, *Patrz:* OOP

projektowanie sterowane modelem dziedziny,
Patrz: DDD

protokół

SSL, *Patrz:* SSL
 AMQP, *Patrz:* AMQP
 HTTP, 34, *Patrz:* HTTP
 Secure Socket Layer, *Patrz:* SSL
 SOAP, *Patrz:* SOAP
 SSL, 33
 TCP, 194
 TLS, *Patrz:* TLS
 Transport Layer Security, *Patrz:* TLS
 UDP, 194

przestrzeń nazw, 26

Pujals Tony, 17

Q

Quora, 120

logowanie, 122, 125

R

RabbitMQ, 103

reguła segregacji interfejsów, *Patrz:* ISP

rejestr usług, 76, 87

repozytorium, 60, 78

artifacts, 25
 implementacja, 70

REST, 35, *Patrz też:* usługa REST

RestTemplate, 95

Ribbon, *Patrz:* Netflix Ribbon

równoważenie obciążenia, 94, 95
 po stronie klienta, 95
 po stronie serwera, 98

S

Scumblr, 191, 198

Security Monkey, 191, 198, *Patrz też:* małpaService-Oriented Architecture, *Patrz:* SOA

serwer

Apache Tomcat, 30, 33
 autoryzacji, 124, 125, 128, 129
 Eureka, 95
 graniczny, 98, 99, 159, 195
 Jetty, 30, 33
 pośredniczący, 98, 195
 proxy, *Patrz:* serwer pośredniczący
 wirtualizowany, 106
 zasobów, 124

Simian Army, 195

skalowanie

dwuwymiarowe, 20
 jednowymiarowe, 20
 platformowe, 20
 produktu, 20

sniffer, 117

SOA, 16, 187

SOAP, 16

SPA, 154, 155

Spinnaker, 193, 194

Spring, 83

Spring Boot, 29, 30

konfiguracja, 30

wersja, 31

Spring Cloud, 77, 87, 94, 98, 193

klient, 95

Spring Cloud Ribbon, 95

Spring_INITIALIZER, 30

Spring Security, 138

SSL, 118, 119

stos ELK, 202

Elasticsearch, 203, 204

Kibana, 204, 206

konfiguracja, 204

Logstash, 203, 205

system plików, 60

Ś

ścieżka

- klas, 33
- niezależna, 63, 67
- URI, 36

T

TeamCity, 24, 65, 188

technologia

- AngularJS, *Patrz:* AngularJS
- Apache Cassandra, *Patrz:* Apache Cassandra
- JPA, *Patrz:* JPA
- Node.js, *Patrz:* Node.js

test

- end-to-end, 151, 189, 190
- integracyjny, 111, 189
- jednostkowy, 88, 151, 190

TLS, 118, 129

token, 124

- dostępowy, 124, 129, 131
- końcówka, *Patrz:* końcówka tokena
- odświeżenia, 124
- uwierzytelniania, 131

trasowanie, 154

U

UI, *Patrz:* interfejs użytkownika

UI-Router, 154

UML, 53, 65

Unified Model Language, *Patrz:* UMLusługa, 77, *Patrz też:* mikrousługa

- \$injector, 152
- \$log, 152
- Docker Hub, 113
- działająca transakcyjnie, 192
- implementacja, 72
- Monkey, *Patrz:* małpa
- otwartego hosta, 63, 68
- rejestr, *Patrz:* rejestr usług
- REST, 21, 30, 34, 55, 78, 189
 - końcówka, *Patrz:* końcówka
- testowanie, 30
- tworzenie, 58, 68, 161
- uwierzytelnienie, 119, 120, 169
- użytkownik końcowy, 124

V

Value Object, *Patrz:* obiekt wartości

Vector, 197

VirtualBox, 27, 106

VM, 25

VO, *Patrz:* obiekt wartości

W

warstwa

- aplikacji, 54, 55, 58
- biznesowa, 58
- DAO, 18
- dziedziny, 54, 55, 58
- infrastruktury, 54, 55
- ograniczająca przekłamania, 63, 67
- prezentacji, 18, 54, 150

widok, 150

wstrzykiwanie zależności, 152

wyłącznik automatyczny, 101

wyrażenie regularne, 37

wzorzec

- fabryki, 62
- fasady, 67
- klient-dostawca, 63, 66
- konformisty, 63, 67
- projektowy
 - MVC, *Patrz:* MVC
 - MVVM, *Patrz:* MVVM
- wstrzykiwania zależności, 152
- Wyłącznik Automatyczny, 101

Z

Zuul Server, 94, 98, 119, 195

związek, 59

Ż

żądanie GET, 34, 35

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

Mikrouługi w Javie Poradnik eksperta

Chmury obliczeniowe otworzyły nowe możliwości projektowania aplikacji korporacyjnych. Obecnie konstruuje się je z małych, lekkich i zorientowanych na proces komponentów, nazywanych mikrouługami. Tworzone w ten sposób aplikacje są skalowalne i łatwe do zarządzania, a przy tym niezwykle elastyczne i wykorzystujące dostępne zasoby w wyjątkowo efektywny sposób. Jak łatwo się przekonać, projektowanie i implementacja mikrouług otwierają przed programistami Javy bardzo obiecujące perspektywy!

Niniejsza książka jest przeznaczona dla programistów Javy, którzy znają podstawowe pojęcia dotyczące mikrouług i chcą tworzyć funkcjonalne aplikacje biznesowe w tej technologii. Przedstawiono tu najważniejsze zasady realizacji dużych projektów, zaprezentowano ważne szczegóły dotyczące konfiguracji środowiska programistycznego i ciągłej integracji, które ułatwią wdrażanie mikrouług. Opisano zalety i sposoby wykorzystania biblioteki Spring Security. Zaprezentowano dobre praktyki projektowania mikrouług, a także techniki ich debugowania, tak aby bez problemu można było zaprojektować i wdrożyć aplikację w środowisku korporacyjnym.

Mikrouługi — sprytnie rozwiązania biznesowe dla dużych korporacji!



W książce znajdziesz:

- środowiska programistyczne i narzędzia do projektowania mikrouług
- różne technologie wdrażania mikrouług
- sposoby zabezpieczania mikrouług, w tym uwierzyteliwanie i autoryzacja
- testowanie mikrouług za pomocą klientów REST
- tworzenie interfejsów użytkownika w technologii Angular JS

Sourabh Sharma tworzy aplikacje od ponad 10 lat. Jest cenionym ekspertem w dziedzinie tworzenia, wdrażania i testowania wielowarstwowych aplikacji internetowych. Opracował i wdrożył wiele rozwiązań samodzielnych i chmurowych dla klientów z listy Fortune 500. Jest autorem licznych systemów opartych na mikrouługach. Pasjonat Javy, uwielbia analizowanie skomplikowanych problemów i poszukiwanie niestandardowych rozwiązań.

PACKT open source
PUBLISHING community experience distilled

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
● <http://helion.pl/promocje>
Książki najchętniej czytane:
● <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
● <http://helion.pl/nawosci>

siegnij po WIĘCEJ



KOD KORZYSCI

ISBN 978-83-283-3218-8



9 788328 332188

Informatyka w najlepszym wydaniu

cena: 49,00 zł