Technologia i rozwiązania

Mikrousługi w Javie
Poradnik eksperta

Sourabh Sharma

Helion

[PACKT] PUBLISHING open source 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 jakiejkolwiek 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

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

4

ROZDZIAŁ

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

Kup książkę Poleć książkę

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.

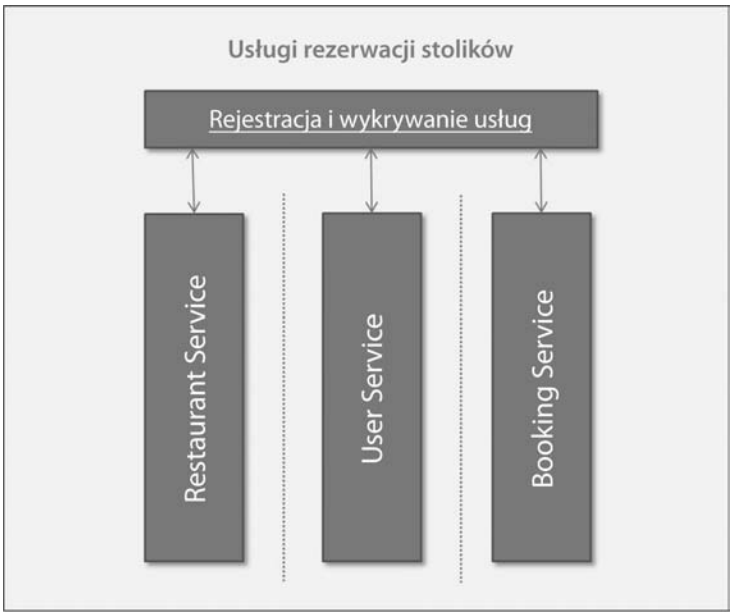
Zakres funkcjonalny wymienionych usług można opisać następująco:

- 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.

Kup książkę Poleć książkę



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 niezmienne 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.

Kup książkę Poleć książkę

- **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

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

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.
```


Kup książkę Poleć książkę

```
     *
     * @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, 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;
```


Kup książkę Poleć książkę

```
    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.
    }
}
```


Kup książkę Poleć książkę

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) {
```


Kup książkę Poleć książkę

```
            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();
        }
}
```


Kup książkę Poleć książkę

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();
       }
   }
```


Kup książkę Poleć książkę

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;
        }
}
```


Kup książkę Poleć książkę

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*.

Kup książkę Poleć książkę

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>
```


Kup książkę Poleć książkę

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");
    }
}
```


Kup książkę Poleć książkę

- 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);
                }
            });
```


Kup książkę Poleć książkę

```
            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.

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Skorowidz

A

Kup książkę Poleć książkę

Kup książkę Poleć książkę

H

I

J

K

L

M

Kup książkę Poleć książkę

N

O

Kup książkę Poleć książkę

P

Q

R

S

Kup książkę Poleć książkę

Ś

T

U

V

W

Z

Ż

Kup książkę Poleć książkę

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

GRUPA WYDAWNICZA
Helion SA

Mikrousługi 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 mikrousługami. 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 mikrousług otwierają przed programistami Javy bardzo obiecujące perspektywy!

Niniejsza książka jest przeznaczona dla programistów Javy, którzy znają podstawowe pojęcia dotyczące mikrousług 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 mikrousług. Opisano zalety i sposoby wykorzystania biblioteki Spring Security. Zaprezentowano dobre praktyki projektowania mikrousług, a także techniki ich debugowania, tak aby bez problemu można było zaprojektować i wdrożyć aplikację w środowisku korporacyjnym.

Mikrousługi — sprytne rozwiązania biznesowe dla dużych korporacji!

W książce znajdziesz:

- środowiska programistyczne i narzędzia do projektowania mikrousług
- różne technologie wdrażania mikrousług
- sposoby zabezpieczania mikrousług, w tym uwierzytelnianie i autoryzacja
- testowanie mikrousług 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 mikrousługach. 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/nowosci

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-3218-8

cena: 49,00 zł