Technologia i rozwiązania

Spring MVC
Przewodnik dla początkujących

Wykorzystaj możliwości Spring MVC!

Amuthan G

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Spring MVC: Beginner's Guide

Tłumaczenie: Andrzej Bobak

ISBN: 978-83-283-0517-5

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)

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sprimv
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 9

Apache Tiles oraz Spring Web Flow w praktyce

Podczas implementacji aplikacji internetowej musisz mieć na uwadze możliwość ponownego użycia oraz utrzymywania kodu. Spring Web Flow jest niezależnym szkieletem ułatwiającym pisanie wysoce konfigurowalnych aplikacji internetowych opartych na przepływach. Z kolei Apache Tiles jest popularnym otwartym szkieletem usprawniającym tworzenie aplikacji opartych na szablonach wielokrotnego użytku.

W tym rozdziale nauczysz się, jak włączyć wspomniane szkielety do aplikacji Springa MVC. Apache Tiles znacząco poprawia możliwość ponownego użycia szablonów interfejsu użytkownika. Z kolei Spring Web Flow zmniejsza koszty utrzymania logiki aplikacji. Pamiętaj, że oba szkielety są niezależne. Nie musisz używać ich jednocześnie. Apache Tiles jest zazwyczaj używany, by zmniejszyć rozmiar nadmiarowego kodu po stronie interfejsu użytkownika. Natomiast Spring Web Flow umożliwia implementację stanowych aplikacji internetowych z kontrolowanym przepływem nawigacji. Po zapoznaniu się z zawartością rozdziału będziesz miał pojęcie o poniższych zagadnieniach:

- jak budować aplikacje oparte na przepływie za pomocą Spring Web Flow;
- jak dekomponować strony, używając szablonów wielokrotnego użycia Apache Tiles.

Praca ze Spring Web Flow

Spring Web Flow ułatwia implementację aplikacji internetowych opartych na przepływie. Przepływ w aplikacji internetowej jest serią kroków prowadzących użytkownika przez biznesową operację, taką jak meldunek w hotelu, podanie o pracę lub zamówienie zawartości koszyka. Zazwyczaj przepływ ma jasny punkt początkowy oraz końcowy, obsługuje wiele żądań i odpowiedzi, a użytkownik musi odwiedzić zbiór ekranów w określonej kolejności, by zrealizować przepływ.

We wszystkich poprzednich rozdziałach odpowiedzialność za zdefiniowanie przepływu na stronie (interfejsie użytkownika) spoczywała na kontrolerze. Przepływ był podzielony na pojedyncze kontrolery i widoki. Na przykład odwzorowywałeś żądanie na metodę kontrolera, a ona zwracała nazwę odpowiedniego widoku generowanego jako odpowiedź. Takie podejście jest proste i wystarczające dla podstawowych przepływów stron. Jednak gdy przepływ w aplikacji staje się coraz bardziej skomplikowany, utrzymanie dużego i złożonego przepływu na stronie staje się kłopotliwe.

Jeśli planujesz napisanie złożonej aplikacji wykorzystującej przepływy, **Spring Web Flow** (**SWF**) może się okazać bardzo pomocny. Za jego pomocą możesz zdefiniować i wykonać przepływy w **interfejsie użytkownika** (*User Interface* — **UI**) swojej aplikacji internetowej. Pora zająć się Spring Web Flow i zdefiniować kilka przepływów stron w projekcie.

W poprzednim rozdziale udało Ci się zaimplementować funkcjonalność koszyka. Jednak bez możliwości przeprowadzenia zakupu i wysyłki do klienta jest ona bezużyteczna. Teraz zaimplementujesz brakujące funkcjonalności w dwóch etapach. Najpierw utworzysz po stronie serwera wymagane usługi, obiekty domenowe oraz implementację repozytorium. Nie są one bezpośrednio powiązane ze Spring Web Flow, jednak potrzebujesz ich do przetwarzania zamówienia. W drugim etapie ustalisz definicję Spring Web Flow, używającą zaimplementowanych usług serwerowych, by wykonać ustalony przepływ. Ustawisz też konfigurację oraz definicję przepływu.

Ćwiczenie praktyczne — implementacja usługi obsługującej zamówienie

Rozpoczniesz od implementacji usługi serwerowej przetwarzającej zamówienie. Wykonaj poniższe kroki:

1. Utwórz klasę o nazwie `Address` w pakiecie `com.packt.webstore.domain` w katalogu *src/main/java*. Następnie umieść w niej zawartość listingu 9.1. Zauważ, że nie ma w nim implementacji getterów, setterów oraz metod `hashCode` i `equals`. Dodaj je podczas tworzenia klasy.

Listing 9.1. Implementacja klasy Address

```
package com.packt.webstore.domain;
import java.io.Serializable;
public class Address implements Serializable{
   private static final long serialVersionUID = -530086768384258062L;
   private String doorNo;
   private String streetName;
   private String areaName;
   private String state;
   private String country;
   private String zipCode;
```

```
    // Tu dodaj gettery i settery dla wszystkich pól.
    // Następnie nadpisz metody equals i hashCode, by opierały się na wszystkich polach.
    // Kod możesz pobrać ze strony: www.helion.pl/ksiazki/sprimv.html.
}
```

2. Utwórz w tym samym pakiecie klasę `Customer` i umieść w niej zawartość listingu 9.2.

Listing 9.2. Implementacja klasy Customer

```
package com.packt.webstore.domain;
import java.io.Serializable;
public class Customer implements Serializable{
   private static final long serialVersionUID = 2284040482222162898L;
   private String customerId;
   private String name;
   private Address billingAddress;
   private String phoneNumber;
   public Customer() {
      super();
      this.billingAddress = new Address();
   }
   public Customer(String customerId, String name) {
      this();
      this.customerId = customerId;
      this.name = name;
   }
   // Tu dodaj gettery i settery dla wszystkich pól.
   // Następnie nadpisz metody equals i hashCode, by opierały się na wszystkich polach.
   // Kod możesz pobrać ze strony: www.helion.pl/ksiazki/sprimv.html.
}
```

3. W tym samym pakiecie utwórz klasę domenową `ShippingDetail` i umieść w niej zawartość listingu 9.3.

Listing 9.3. Implementacja klasy ShippingDetail

```
package com.packt.webstore.domain;
import java.io.Serializable;
import java.util.Date;
import org.springframework.format.annotation.DateTimeFormat;
public class ShippingDetail implements Serializable{
   private static final long serialVersionUID = 6350930334140807514L;
   private String name;
   @DateTimeFormat(pattern = "dd/MM/yyyy")
   private Date shippingDate;
   private Address shippingAddress;
   public ShippingDetail() {
      this.shippingAddress = new Address();
```

```
    }
    // Dodaj gettery i settery dla powyższych pól.
}
```

4. W bieżącym pakiecie utwórz jeszcze jedną klasę domenową, o nazwie `Order`, i umieść w niej zawartość listingu 9.4.

Listing 9.4. Implementacja klasy Order

```
package com.packt.webstore.domain;
import java.io.Serializable;
public class Order implements Serializable{
   private static final long serialVersionUID = -3560539622417210365L;
   private Long orderId;
   private Cart cart;
   private Customer customer;
   private ShippingDetail shippingDetail;
   public Order() {
      this.customer = new Customer();
      this.shippingDetail = new ShippingDetail();
   }
   // Tu dodaj gettery i settery dla wszystkich pól.
   // Następnie nadpisz metody equals i hashCode, by opierały się na polu orderId.
   // Kod możesz pobrać ze strony: www.helion.pl/ksiazki/sprimv.html.
}
```

5. Spraw, żeby klasy domenowe `Product`, `CartItem` oraz `Cart` były serializowalne, implementując w nich interfejs `Serializable` (`java.io.Serializable`). Dodaj w nich również pole `serialVersionUID`.
6. Utwórz interfejs o nazwie `OrderRepository` w pakiecie `com.packt.webstore.domain.repository` w katalogu *src/main/java* i umieść w nim przedstawioną poniżej deklarację metody:

```
Long saveOrder(Order order);
```

7. Utwórz klasę o nazwie `InMemoryOrderRepositoryImpl` w pakiecie `com.packt.webstore.domain.repository.impl` w katalogu *src/main/java* i umieść w niej zawartość listingu 9.5.

Listing 9.5. Implementacja klasy InMemoryOrderRepositoryImpl

```
package com.packt.webstore.domain.repository.impl;
import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Repository;
import com.packt.webstore.domain.Order;
import com.packt.webstore.domain.repository.OrderRepository;
@Repository
```

```
public class InMemoryOrderRepositoryImpl implements OrderRepository{
   private Map<Long, Order> listOfOrders;
   private long nextOrderId;
   public InMemoryOrderRepositoryImpl() {
      listOfOrders = new HashMap<Long, Order>();
      nextOrderId = 1000;
   }
   public Long saveOrder(Order order) {
      order.setOrderId(getNextOrderId());
      listOfOrders.put(order.getOrderId(), order);
      return order.getOrderId();
   }
   private synchronized long getNextOrderId() {
      return nextOrderId++;
   }
}
```

8. Otwórz interfejs `OrderService` w pakiecie `com.packt.webstore.service` w katalogu *src/main/java* i umieść w nim deklarację metody przedstawioną w listingu 9.6.

Listing 9.6. Interfejs OrderService

```
package com.packt.webstore.domain.repository;
import com.packt.webstore.domain.Order;
public interface OrderRepository {
   void processOrder(String  productId, long quantity);
   Long saveOrder(Order order);
}
```

9. Otwórz klasę `OrderServiceImpl` znajdującą się w pakiecie `com.packt.webstore.service.impl` w katalogu *src/main/java* i umieść w niej referencje przedstawione w listingu 9.7.

Listing 9.7. Referencje do dodania w klasie OrderServiceImpl

```
@Autowired
private OrderRepository orderRepository;
@Autowired
private CartService cartService;
```

10. Umieść przedstawioną w listingu 9.8 implementację metody `saveOrder` w klasie `OrderServiceImpl`.

Listing 9.8. Implementacja metody saveOrder

```
public Long saveOrder(Order order) {
   Long orderId = orderRepository.saveOrder(order);
```

```
    cartService.delete(order.getCart().getCartId());
    return orderId;
}
```

11. Utwórz klasę wyjątku o nazwie `InvalidCartException` w pakiecie `com.packt.webstore.exception` w katalogu *src/main/java* i umieść w niej zawartość listingu 9.9.

Listing 9.9. Implementacja klasy InvalidCartException

```
package com.packt.webstore.exception;
public class InvalidCartException extends RuntimeException {
    private static final long serialVersionUID = -5192041563033358491L;
    private String cartId;
    public InvalidCartException(String cartId) {
        this.cartId = cartId;
    }
    public String getCartId() {
        return cartId;
    }
}
```

12. Otwórz interfejs `CartService` znajdujący się w pakiecie `com.packt.webstore.service` w katalogu *src/main/java* i umieść w nim poniższą sygnaturę metody:

```
Cart validate(String cartId);
```

13. Następnie otwórz klasę `CartServiceImpl` znajdującą się w pakiecie `com.packt.webstore.service.impl` w katalogu *src/main/java* i umieść w niej implementację metody `validate` zaprezentowaną w listingu 9.10.

Listing 9.10. Implementacja metody validate

```
public Cart validate(String cartId) {
    Cart cart = cartRepository.read(cartId);
    if(cart==null || cart.getCartItems().size()==0) {
        throw new InvalidCartException(cartId);
    }
    return cart;
}
```

Co się właśnie wydarzyło?

Przeprowadzone operacje są Ci już znane. Utworzyłeś kilka klas domenowych (`Address`, `Customer`, `ShippingDetail` oraz `Order`), interfejs `OrderRepository` oraz jego implementację w klasie `InMemoryOrderRepositoryImpl` przechowującą przetworzone obiekty klasy domenowej `Order`. Na koniec utworzyłeś interfejs `OrderService` oraz jego implementację w klasie `OrderServiceImpl`.

Na pierwszy rzut oka wszystkie operacje wyglądają jak zwykle, są jednak pewne drobne detale wymagające wyjaśnienia. Jak zauważyłeś, utworzone w krokach 1. – 4. klasy domenowe implementują interfejs `Serializable`. Oprócz tego w kroku 5. dodałeś wspomniany interfejs do istniejących klas domenowych `Product`, `CartItem` oraz `Cart`. Zrobiłeś to, ponieważ później będziesz używać tych obiektów domenowych w szkielecie Spring Web Flow. On z kolei będzie przechowywać wspomniane obiekty w sesji, by zarządzać stanem pomiędzy przepływami strony. Dane sesji mogą zostać zapisane na dysku lub przesłane do innego serwera WWW w klastrze. Podczas pobierania obiektu sesji z dysku Spring Web Flow zdeserializuje obiekt domenowy (bean formularza), by zarządzać stanem strony. Dlatego też obiekty domenowe i beany formularza muszą być serializowalne. Spring Web Flow używa bytu `Snapshot`, by przechować stany w sesji.

Kroki 6. – 13. nie wymagają szerszego wyjaśnienia. Utworzyłeś interfejsy `OrderRepository` i `OrderService` oraz ich implementacje w klasach `InMemoryOrderRepositoryImpl` i `OrderService` ➥`Impl`. Ich zadaniem jest zapisywanie obiektu domenowego `Order`. Metoda `saveOrder` klasy `OrderServiceImpl` usuwa obiekt `Cart` z `CartRepository` po zapisaniu obiektu domenowego klasy `Order`. Udało Ci się utworzyć wszystkie usługi oraz obiekty domenowe po stronie serwera wymagane do rozpoczęcia prac nad konfiguracją i definicją Spring Web Flow.

Ćwiczenie praktyczne — implementacja przepływu zakupu

W tym ćwiczeniu wzbogacisz aplikację o wsparcie Spring Web Flow. Następnie zdefiniujesz przepływ zamówienia zawartości koszyka. Wykonaj poniższe kroki:

1. Otwórz plik *pom.xml*. Znajdziesz go w katalogu głównym projektu.
2. Na dole okna prezentującego zawartość pliku znajduje się kilka zakładek. Przejdź do tej o nazwie *Dependencies* i użyj przycisku *Add* w sekcji *Dependencies*.
3. Pojawi się okno o nazwie *Select Dependency*. W polu *Group Id* wprowadź `org.springframework.webflow`, a w *Artifact Id* `spring-webflow`. Polu *Version* nadaj wartość `2.3.3.RELEASE`, a polu *Scope* wartość `compile`. Następnie naciśnij przycisk *OK* i zapisz plik *pom.xml*.
4. W katalogu *src/main/webapp/WEB-INF/* utwórz podkatalog *flows/checkout/*. W nim utwórz plik *checkout-flow.xml* i umieść w nim zawartość listingu 9.11.

Listing 9.11. Definicja przepływów w pliku checkout-flow.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/webflow
   http://www.springframework.org/schema/webflow/spring-webflow.xsd">
   <var name="order" class="com.packt.webstore.domain.Order" />
   <action-state id="addCartToOrder">
```

```
        <evaluate expression="cartServiceImpl.validate(requestParameters.cartId)"
        ↪result="order.cart" />
        <transition to="InvalidCartWarning"
        ↪on-exception="com.packt.webstore.exception.InvalidCartException" />
        <transition to="collectCustomerInfo" />
    </action-state>
    <view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
    ↪model="order">
        <transition on="customerInfoCollected" to="collectShippingDetail" />
    </view-state>
    <view-state id="collectShippingDetail" model="order">
        <transition on="shippingDetailCollected" to="orderConfirmation" />
        <transition on="backToCollectCustomerInfo" to="collectCustomerInfo" />
    </view-state>
    <view-state id="orderConfirmation">
        <transition on="orderConfirmed" to="processOrder" />
        <transition on="backToCollectShippingDetail"
        ↪to="collectShippingDetail" />
    </view-state>
    <action-state id="processOrder">
        <evaluate expression="orderServiceImpl.saveOrder(order)"
        ↪result="order.orderId"/>
        <transition to="thankCustomer" />
    </action-state>
    <view-state id="InvalidCartWarning">
        <transition to="endState"/>
    </view-state>
    <view-state id="thankCustomer" model="order">
        <transition to="endState"/>
    </view-state>
    <end-state id="endState"/>
    <end-state id="cancelCheckout" view = "checkOutCancelled.jsp"/>
    <global-transitions>
        <transition on = "cancel" to="endState" />
    </global-transitions>
</flow>
```

5. Otwórz plik *DispatcherServlet-context.xml*, zawierający kontekst konfiguracji aplikacji internetowej, i umieść na początku znacznika `<beans>` poniższy atrybut, definiujący przestrzeń nazw:

```
xmlns:webflow-config="http://www.springframework.org/schema/
↪webflow-config"
```

6. Dodaj przedstawiony poniżej fragment kodu do atrybutu `xsi:schemaLocation` znacznika `<beans>`:

```
http://www.springframework.org/schema/webflow-config http://www.
↪springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd
```

7. Umieść przedstawione w listingu 9.12 znaczniki konfiguracji przepływu w pliku konfiguracji kontekstu aplikacji internetowej (*DispatcherServlet-context.xml*).

Listing 9.12. Znaczniki konfiguracji przepływu w aplikacji

```
<webflow-config:flow-executor id="flowExecutor" flowregistry="flowRegistry" />
<webflow-config:flow-registry id="flowRegistry" base-path="/WEBINF/flows">
   <webflow-config:flow-location path="/checkout/
   ↪checkout-flow.xml"id="checkout"/>
</webflow-config:flow-registry>
```

8. Na koniec w pliku *DispatcherServlet-context.xml* zdefiniuj beany `FlowHandlerMapping` oraz `FlowHandlerAdapter`, przedstawione w listingu 9.13. Następnie zapisz plik.

Listing 9.13. Deklaracja beanów FlowHandlerMapping oraz FlowHandlerAdapter

```
<bean id="flowHandlerMapping"
class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
   <property name="flowRegistry" ref="flowRegistry" />
</bean>
<bean id="flowHandlerAdapter" class="org.springframework.webflow.mvc.
↪servlet.FlowHandlerAdapter">
   <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Co się właśnie wydarzyło?

W krokach 1. – 3. dodałeś do projektu zależność Spring Web Flow za pośrednictwem Mavena. Na nim spoczywa odpowiedzialność za pobranie i skonfigurowanie wymaganych plików JAR powiązanych z przepływem w Twojej aplikacji. W kroku 4. utworzyłeś pierwszą definicję przepływu w pliku *checkout-flow.xml* w katalogu *src/main/webapp/WEB-INF/flows/checkout/*.

Spring Web Flow używa pliku z definicją przepływów, aby nim sterować. Pora zapoznać się z podstawowymi założeniami Spring Web Flow, aby zrozumieć, jakie informacje zostały umieszczone w pliku. Po przyswojeniu sobie założeń wrócisz do analizy zawartości pliku *checkout-flow.xml*.

Zrozumieć definicję przepływu

Definicja przepływu składa się ze zbioru stanów. Każdy stan posiada unikalny identyfikator w definicji przepływu. W Spring Web Flow istnieje sześć typów stanów:

- `start-state`. Każdy przepływ musi posiadać jeden stan początkowy, pomagający w tworzeniu startowego stanu przepływu. Pamiętaj, że jeśli stan startowy nie jest zdefiniowany, staje się nim pierwszy stan umieszczony w definicji przepływu.

- `action-state`. Przepływ może mieć dowolną liczbę stanów akcji. Każdy taki stan wywołuje konkretną akcję. Jest nią zazwyczaj interakcja z usługami serwerowymi, np. wywołanie metody w beanie Springa. Spring Web Flow używa języka wyrażeń Springa (**Spring Expression Language**) do interakcji z serwerowymi beanami usług.
- `view-state`. Stan widoku wskazuje nazwę widoku oraz model wchodzące w interakcje z użytkownikiem. Przepływ może mieć zdefiniowane wiele stanów widoków. Jeżeli atrybut `view` nie jest zdefiniowany, wtedy identyfikator stanu widoku jest używany jako nazwa widoku.
- `decision-state`. Stan decyzyjny jest używany do rozgałęziania przepływu. Na podstawie wyniku logicznego wyrażenia przepływ jest kierowany do następnego stanu.
- `subflow-state`. Podprzepływ jest niezależnym przepływem, który może być użyty wewnątrz innego przepływu. Gdy aplikacja trafia do niego, główny przepływ zostaje wstrzymany do czasu zakończenia pracy w podprzepływie.
- `end-state`. Stan końcowy wskazuje zakończenie wykonywania przepływu. Może występować w przepływie wielokrotnie. Jego atrybut `view` pozwala wskazać widok, który ma zostać wyświetlony użytkownikowi po osiągnięciu stanu końcowego.

Dowiedziałeś się, że opis przepływu składa się ze zbioru stanów. Jednak aby przenieść się pomiędzy stanami, musisz zdefiniować przejścia między nimi. Każdy stan w przepływie (poza początkowym i końcowym) wskazuje przejścia pozwalające na przeniesienie do innego stanu. Przejście może zostać wywołane przez zdarzenie zasygnalizowane w stanie.

Zrozumieć przepływ zakupu

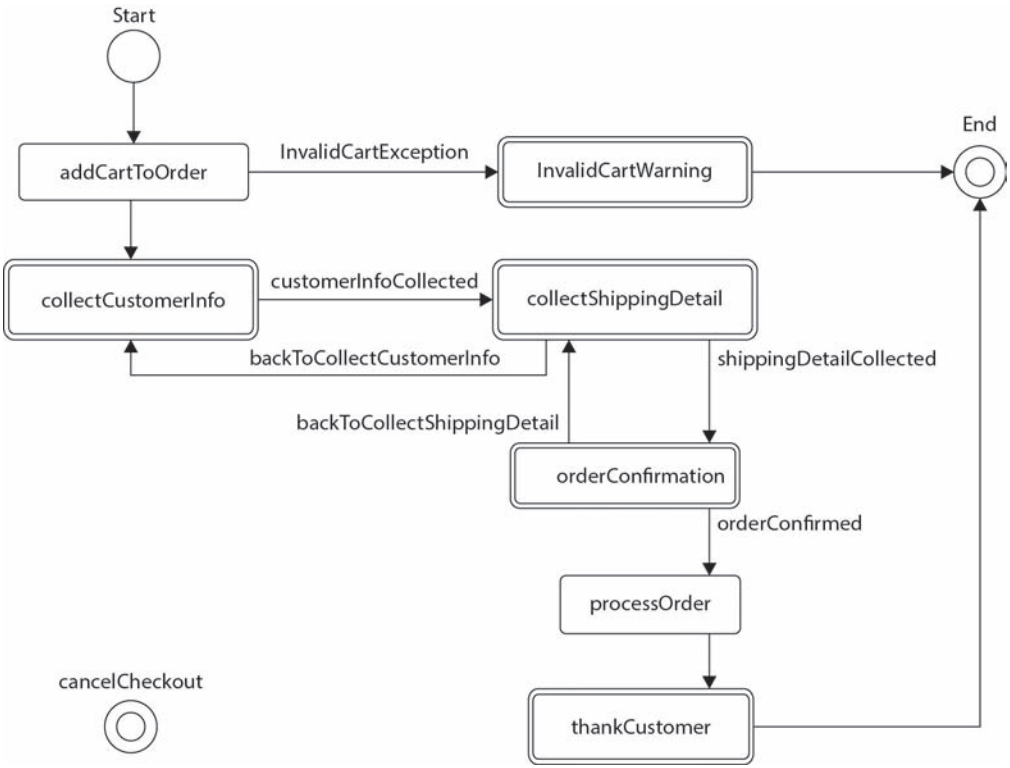
Zapoznałeś się z podstawowym wprowadzeniem do Spring Web Flow. Jest to zaledwie ułamek zagadnień powiązanych z tym narzędziem, jednak zapoznanie się z nimi wszystkimi jest dobrym pomysłem na osobną książkę. Wiesz już wystarczająco dużo, by móc przeanalizować zawartość pliku *checkout-flow.xml*. Zanim to jednak zrobisz, przyjrzyj się diagramowi przepływu zakupu zaprezentowanemu na rysunku 9.1.

Diagram przepływu zakupu ma stan początkowy oraz stan końcowy. Każdy prostokąt z zaokrąglonymi rogami jest stanem akcji, a każdy prostokąt z zaokrąglonymi rogami i podwójną ramką jest stanem widoku. Strzałki oznaczają przejścia, a ich nazwy wskazują na zdarzenia wywołujące przejście. Plik *checkout-flow.xml* zawiera przedstawiony przepływ w formacie XML.

Po otwarciu pliku zauważysz, że pierwszym znacznikiem wewnątrz znacznika `<flow>` jest przedstawiony poniżej znacznik `<tag>`:

```
<var name="order" class="com.packt.webstore.domain.Order" />
```

Znacznik `<var>` tworzy zmienną w przepływie. Jest ona dostępna we wszystkich jego stanach. Oznacza to, że możesz się do niej odwoływać i używać jej w dowolnym stanie przepływu. We wspomnianym znaczniku `<var>` utworzyłeś nową instancję klasy `Order` i umieściłeś ją w zmiennej o nazwie `order`.



Rysunek 9.1. Diagram przepływu zakupu

Kolejnym elementem w pliku *checkout-flow.xml* jest deklaracja `<action-state>`, używana zazwyczaj do wywoływania usług serwera. W omawianym stanie wywołujesz metodę `validate` obiektu `cartServiceImpl` i przechowujesz jej wynik w obiekcie `order.cart`, tak jak przedstawia to listing 9.14.

Listing 9.14. Implementacja stanu addCartToOrder

```
<action-state id="addCartToOrder">
   <evaluate expression ="cartServiceImpl.validate(requestParameters.cartId)"
   ↪result="order.cart" />
   <transition to="InvalidCartWarning" on-exception ="com.packt.webstore.
   ↪exception.InvalidCartException" />
   <transition to="collectCustomerInfo" />
</action-state>
```

Jak już wiesz, zmienna `order` jest dostępna we wszystkich stanach przepływu. Dlatego też użyłeś jej w znaczniku `<evaluate>`, by przechowywać wynik operacji, w tym przypadku wywołania metody `cartServiceImpl.validate(requestParameters.cartId)`.

Metoda `validate` obiektu `cartServiceImpl` próbuje odczytać obiekt `cart` na podstawie otrzymanego parametru `cartId`. Jeżeli odnajdzie poprawny obiekt `cart`, zwraca go. W przeciwnym razie zgłasza wyjątek `InvalidCartException`. W takim wypadku przepływ zostaje przekierowany do stanu o identyfikatorze `InvalidCartWarning`:

```
<transition to="InvalidCartWarning" on-exception =
➥"com.packt.webstore.exception.InvalidCartException" />
```

Jeśli wyjątek nie został zgłoszony, przepływ zostaje przekierowany ze stanu `addCartToOrder` do stanu `collectCustomerInfo`:

```
<transition to="collectCustomerInfo" />
```

Jak zapewne zauważyłeś, `collectCustomerInfoState` jest stanem widoku zdefiniowanym w pliku *checkout-flow.xml*. Listing 9.15 prezentuje, w jaki sposób określiłeś widok, który ma zostać wygenerowany użytkownikowi, oraz model, który musi zostać dołączony do obiektu `model`.

Listing 9.15. Deklaracja stanu collectCustomerInfo

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
➥model="order">
   <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Po przejściu do tego stanu Spring Web Flow wygeneruje widok `collectCustomerInfo` i zatrzyma wykonywanie przepływu do czasu podjęcia przez użytkownika akcji. Gdy użytkownik wprowadzi informacje i prześle formularz, przepływ zostanie przekierowany do stanu widoku `collectShip` ➥`pingDetail`. Jak już wiesz, przekierowanie może zostać wywołane za pomocą zdarzenia. W tym przypadku przekierowanie do stanu `collectShippingDetail` zostanie wywołane po wywołaniu zdarzenia `customerInfoCollected`. Pojawia się pytanie: jak wywołać to zdarzenie (`customer` ➥`InfoCollected`) z poziomu widoku? Odpowiedź na nie znajdziesz w dalszej części rozdziału. Przyjrzyj się poniższemu fragmentowi kodu:

```
<transition on="customerInfoCollected" to="collectShippingDetail" />
```

Następnym zdefiniowanym stanem w przepływie zakupu jest stan widoku `collectShipping` ➥`Detail` zaprezentowany w listingu 9.16. Ma zdefiniowane dwa przepływy: pierwszy cofający do stanu `collectCustomerInfo`, drugi przekierowujący do następnego stanu (o nazwie `orderCon` ➥`firmation`).

Listing 9.16. Definicja stanu collectShippingDetail

```
<view-state id="collectShippingDetail" model="order">
   <transition on="shippingDetailCollected" to="orderConfirmation" />
   <transition on="backToCollectCustomerInfo" to="collectCustomerInfo"/>
</view-state>
```

Zauważ, że w stanie `collectShippingDetail` nie zdefiniowałeś atrybutu `view`. W związku z tym Spring Web Flow potraktuje nazwę stanu jako nazwę widoku.

Definicja stanu `orderConfirmation`, przedstawionego w listingu 9.17, nie wymaga wyjaśnienia. Przypomina stan `collectShippingDetail`, w którym aplikacja prezentuje wszystkie informacje powiązane z zamówieniem i oczekuje na potwierdzenie użytkownika. Po potwierdzeniu zostajesz przekierowany do stanu `processOrder`.

Listing 9.17. Definicja stanu orderConfirmation

```
<view-state id="orderConfirmation">
   <transition on="orderConfirmed" to="processOrder" />
   <transition on="backToCollectShippingDetail" to="collectShippingDetail" />
</view-state>
```

Stan akcji `processOrder` komunikuje się z obiektem `orderServiceImpl` w celu zapisania obiektu `order`. Po pomyślnym zapisaniu obiektu jego identyfikator zostaje zapisany w zmiennej (`order.orderId`). Następnie przepływ zostaje przekierowany do zaprezentowanego w listingu 9.18 stanu `thankCustomer`.

Listing 9.18. Definicja stanu thankCustomer

```
<view-state id="thankCustomer" model="order">
   <transition to="endState"/>
</view-state>
```

Twój przepływ obsługi zakupu ma dwa stany końcowe, zaprezentowane w listingu 9.19. Przepływ dociera do pierwszego z nich, gdy przejdzie przez wszystkie zakładane stany. Drugi z nich jest osiągany, jeśli użytkownik w dowolnym momencie naciśnie przycisk *Anuluj*.

Listing 9.19. Definicje stanów końcowych przepływu

```
<end-state id="endState"/>
<end-state id="cancelCheckout" view="checkOutCancelled.jsp"/>
```

Zauważ, że w stanie końcowym `cancelCheckout` wskazałeś widok za pomocą atrybutu `view`. Przekierowanie do niego odbywa się za pomocą konfiguracji `global-transitions` zaprezentowanej w listingu 9.20.

Listing 9.20. Definicja globalnych przepływów

```
<global-transitions>
   <transition on = "cancel" to="cancelCheckout" />
</global-transitions>
```

Globalne przepływy służą do współdzielenia przejść pomiędzy stanami. Zamiast powtarzać definicje przejścia w każdym stanie, możesz ustawić je jako globalne przepływy. Dzięki temu są one dostępne dla każdego stanu w przepływie. Dlatego zdefiniowałeś przejście do stanu `cancelCheckout` w sekcji `global-transitions`.

Wiesz już, jak wygląda definicja przepływu zakupu (*checkout-flow.xml*). Spring MVC powinien odczytać ten plik podczas startu aplikacji, żeby móc przekierować żądania powiązane z przepływem do szkieletu Spring Web Flow. W tym celu w krokach 5. – 8. umieściłeś kilka znaczników konfiguracji przepływu w pliku konfiguracji kontekstu aplikacji (*DispatherServlet-context.xml*).

W krokach 5. i 6. dodałeś w pliku *DispatcherServlet-context.xml* wymaganą przestrzeń nazw `webflow-config` oraz lokalizację schematu. W kroku 7. utworzyłeś znaczniki `flow-executor` oraz `flow-registry`. Pierwszy z nich tworzy przepływ na podstawie definicji pobieranej z `flow-registry`. Możesz utworzyć dowolną liczbę znaczników `flow-registry`. Każdy z nich jest kolekcją definicji przepływów. Gdy żądanie użytkownika trafia do przepływu, zostaje dla tego żądania utworzona instancja przepływu na podstawie definicji (listing 9.21).

Listing 9.21. Konfiguracja przepływu zakupu

```
<webflow-config:flow-executor id="flowExecutor" flowregistry="flowRegistry" />
<webflow-config:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">
   <webflow-config:flow-location path="/checkout/checkout-flow.xml"
   ↪id="checkout"/>
</webflow-config:flow-registry>
```

W przedstawionej w listingu 9.21 konfiguracji utworzyłeś element `flow-registry`, którego parametr `base-path` ma wartość */WEB-INF/flows*. Oznacza to, że definicje przepływów powinny znajdować się w katalogu */WEB-INF/flows*, by były odczytane przez `flow-registry`. Dlatego w kroku 4. utworzyłeś plik *checkout-flow.xml* w katalogu *src/main/webapp/WEB-INF/flows/checkout/*. Jak już zostało wspomniane, `flow-registry` może mieć wiele definicji przepływów. Każdy przepływ jest oznaczony identyfikatorem `flow-registry`. W Twoim przypadku utworzyłeś pojedynczą definicję przepływu o identyfikatorze `checkout` i relatywnej ścieżce */checkout/checkout-flow.xml*. Pamiętaj, że atrybut `path` znacznika `<webflow-config:flow-location>` jest relatywny wobec atrybutu `base-path` znacznika `<webflow-config:flow-registry>`.

Ważnym zagadnieniem jest sposób wzbudzania przepływu. Odbywa się to poprzez przesłanie żądania `GET` pod relatywny adres równy identyfikatorowi przepływu. Tak więc aby wzbudzić przepływ zakupu (`checkout`), należy przesłać żądanie typu `GET` pod adres: *http://localhost:8080/webstore/checkout*. W definicji przepływu (*checkout-flow.xml*) nie zdefiniowałeś żadnego stanu początkowego, dlatego też pierwszy zdefiniowany stan (`addCartToOrder`) jest stanem początkowym. Przedstawiony w listingu 9.22 stan `addCartToOrder` oczekuje, że w żądaniu będzie obecny parametr o nazwie `cartId`.

Listing 9.22. Ilustracja parametrów wymaganych przez stan addCartToOrder

```
<action-state id="addCartToOrder">
   <evaluate expression = "cartServiceImpl.validate(requestParameters.cartId)"
   ↪result="order.cart" />
   <transition to="InvalidCartWarning"
   ↪on-exception="com.packt.webstore.exception.InvalidCartException" />
   <transition to="collectCustomerInfo" />
</action-state>
```

Poprawna postać żądania wzbudzającego przepływ jest mniej więcej podobna do: *http://localhost:8080/webstore/checkout?**cartId=55AD1472D4EC***. Część żądania za znakiem zapytania (*cartId=55AD1472D4EC*) jest uznawana za parametr żądania.

Zdefiniowałeś przepływ zakupu i skonfigurowałeś go w Spring Web Flow. Kolejnym krokiem było utworzenie w kroku 8. (listing 9.23) dwóch beanów w kontekście aplikacji internetowej (*DispatcherServlet-context.xml*), by przekazywać wszystkie żądania związane z przepływem do obiektu `flow-executor`.

Listing 9.23. Deklaracja beanów obsługujących żądania powiązane z przepływem

```
<bean id="flowHandlerMapping" class= "org.springframework.webflow.mvc.
↪servlet.FlowHandlerMapping">
   <property name="flowRegistry" ref="flowRegistry" />
</bean>
<bean id="flowHandlerAdapter" class= "org.springframework.webflow.mvc.
↪servlet.FlowHandlerAdapter">
   <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Parametr `flowHandlerMapping` tworzy i konfiguruje uchwyt odwzorowujący żądania na przepływy na podstawie identyfikatora każdego przepływu skonfigurowanego we `flowRegistry`. Bean `flowHandlerAdapter` zachowuje się jak pomost pomiędzy serwletem przekazującym a Spring Web Flow, wspomagając wykonanie instancji przepływu.

Krótki test — przepływ

Pytanie 1. Przyjmij konfigurację rejestru przepływu zaprezentowaną w listingu 9.24, deklarującą jeden przepływ zdefiniowany w pliku. Jaką postać ma mieć URL wzbudzający przepływ?

Listing 9.24. Przykładowa deklaracja rejestru przepływów

```
<webflow-config:flow-registry id="flowRegistry" base-path="/WEB-INF/flows">
   <webflow-config:flow-location path="/customer/
   ↪validate.xml"id="validateCustomer"/>
</webflow-config:flow-registry>
```

1. *http://localhost:8080/webstore/customer/validate*.
2. *http://localhost:8080/webstore/validate*.
3. *http://localhost:8080/webstore/validateCustomer*.

Pytanie 2. Zakładając, że URL wzbudzający przepływ na postać: *http://localhost:8080/webstore/validate?customerId=C1234*, jak w pliku definiującym przepływ zadeklarujesz pobieranie parametru żądania o nazwie `customerId`?

1. `<evaluate expression = "requestParameters.customerId " result = "customerId" />`.
2. `<evaluate expression = "requestParameters(customerId) " result = "customerId" />`.
3. `<evaluate expression = "requestParameters[customerId ]" result = "customerId" />`.

Ćwiczenie praktyczne — tworzenie widoków dla każdego stanu widoku

Wykonałeś niezbędne operacje, by uruchomić przepływ zakupu. Pozostała do zrobienia jeszcze jedna rzecz. Musisz utworzyć widoki dla wszystkich stanów widoków obecnych w przepływie. Łącznie zdefiniowałeś sześć stanów widoków (`collectCustomerInfo`, `collectShippingDetail`, `orderConfirmation`, `InvalidCartWarning`, `thankCustomer` oraz `cancelCheckout`), więc potrzebujesz sześciu plików JSP. Pora je utworzyć:

1. Utwórz plik widoku JSP o nazwie *collectCustomerInfo.jsp* w katalogu *src/main/webapp/WEB-INF/flows/checkout/* i umieść w nim zawartość listingu 9.25. Zauważ, że nie zostały w nim umieszczone znaczniki `<input>` dla większości pól obiektu domenowego `Customer`. Znajdziesz je w pełnej wersji kodu źródłowego, który możesz pobrać ze strony: *www.helion.pl/ksiazki*/SPRIMV.*html*.

Listing 9.25. Implementacja widoku collectCustomerInfo

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
      <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
      ↪3.0.0/css/bootstrap.min.css">
      <title>Klient</title>
   </head>
   <body>
      <section>
         <div class="jumbotron">
            <div class="container">
               <h1>Klient</h1>
```

```
                <p>Dane klienta</p>
            </div>
        </div>
    </section>
    <section class="container">
        <form:form modelAttribute="order.customer" class="formhorizontal">
            <fieldset>
                <legend>Dane klienta</legend>
                <div class="form-group">
                    <label class="control-label col-lg-2 col-lg-2"
                    ↪for="customerId" />Identyfikator klienta</label>
                    <div class="col-lg-10">
                        <form:input id="customerId" path="customerId" type="text"
                        ↪class="form:input-large" />
                    </div>
                </div>
                <!-- Tutaj umieść pominięte pola formularza dla pozostałych pól obiektu
                ↪domenowego klasy Customer. -->
                <input type="hidden" name="_flowExecutionKey"
                ↪value="${flowExecutionKey}"/>
                <div class="form-group">
                    <div class="col-lg-offset-2 col-lg-10">
                        <input type="submit" id="btnAdd" class="btn btnprimary"
                        ↪value="Utwórz" name="_eventId_customerInfoCollected" />
                        <button id="btnCancel" class="btn btn-default"
                        ↪name="_eventId_cancel">Anuluj</button>
                    </div>
                </div>
            </fieldset>
        </form:form>
    </section>
</body>
</html>
```

2. Następnie utwórz w tym samym katalogu plik widoku JSP o nazwie *collectShippingDetail.jsp* i umieść w nim zawartość listingu 9.26. Zauważ, że nie zostały w nim umieszczone znaczniki `<input>` dla większości pól obiektu domenowego typu `Address` o nazwie `shippingAddress`. Znajdziesz je w pełnej wersji kodu źródłowego, który możesz pobrać ze strony: *www.helion.pl/ksiazki/sprimv.html*.

Listing 9.26. Implementacja widoku collectShippingDetails

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
```

```
<head>
   <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
   <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
   ↪3.0.0/css/bootstrap.min.css">
   <title>Klient</title>
</head>
<body>
   <section>
      <div class="jumbotron">
         <div class="container">
            <h1>Wysyłka</h1>
            <p>Dane do wysyłki</p>
         </div>
      </div>
   </section>
   <section class="container">
      <form:form modelAttribute="order.shippingDetail" class=
      ↪"form-horizontal">
         <fieldset>
            <legend>Dane do wysyłki</legend>
            <div class="form-group">
               <label class="control-label col-lg-2 col-lg-2" for="name"
               ↪/>Nazwa</label>
               <div class="col-lg-10">
                  <form:input id="name" path="name" type="text"
                  ↪class="form:input-large" />
               </div>
            </div>
            <div class="form-group">
               <label class="control-label col-lg-2 col-lg-2"
               ↪for="shippingDate" />Data wysyłki (dd/mm/yyyy)</label>
               <div class="col-lg-10">
                  <form:input id="shippingDate" path="shippingDate"
                  ↪type="text" class="form:input-large" />
               </div>
            </div>
            <div class="form-group">
               <label class="control-label col-lg-2" for="doorNo">Numer
               ↪mieszkania</label>
               <div class="col-lg-10">
                  <form:input id="doorNo" path="shippingAddress.doorNo"
                  ↪type="text" class="form:input-large" />
               </div>
            </div>
            <!-- Tutaj umieść pominięte pola formularza dla pozostałych pól obiektu
            ↪domenowego shippingAddress klasy Address. -->
            <input type="hidden" name="_flowExecutionKey"
            ↪value="${flowExecutionKey}"/>
            <div class="form-group">
```

```
                    <div class="col-lg-offset-2 col-lg-10">
                        <button id="back" class="btn btn-default" name="
                        ➥_eventId_backToCollectCustomerInfo">wstecz</button>
                        <input type="submit" id="btnAdd" class="btn btnprimary"
                        ➥value="Utwórz" name="_eventId_shippingDetailCollected"/>
                        <button id="btnCancel" class="btn btn-default"
                        ➥name="_eventId_cancel">Anuluj</button>
                    </div>
                </div>
            </fieldset>
        </form:form>
    </section>
</body>
</html>
```

3. Nadal pracując w tym samym katalogu, utwórz kolejny plik widoku JSP, o nazwie *orderConfirmation.jsp*, i umieść w nim zawartość listingu 9.27. Zadaniem widoku będzie potwierdzenie zamówienia złożonego przez klienta. Zauważ, że w listingu dla większości pól obiektu domenowego typu `Order` nie zostały umieszczone znaczniki `<input>`. Kompletną postać pliku widoku *orderConfirmation.jsp* znajdziesz w pełnej wersji kodu źródłowego, który możesz pobrać ze strony: *www.helion.pl/ksiazki/sprimv.html*.

Listing 9.27. Implementacja widoku orderConfirmation

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
        <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
        ➥3.0.0/css/bootstrap.min.css">
        <title>Potwierdzenie zamówienia</title>
    </head>
    <body>
        <section>
            <div class="jumbotron">
                <div class="container">
                    <h1>Zamówienie</h1>
                    <p>Potwierdzenie zamówienia</p>
                </div>
            </div>
        </section>
        <div class="container">
            <div class="row">
```

```
            <form:form modelAttribute="order" class="form-horizontal">
               <input type="hidden" name="_flowExecutionKey"
               ↪value="${flowExecutionKey}" />
                  <div class="well col-xs-10 col-sm-10 col-md-6 col-xsoffset-1
                  ↪col-sm-offset-1 col-md-offset-3">
                     <div class="text-center">
                        <h1>Paragon</h1>
                     </div>
                     <div class="row">
                        <div class="col-xs-6 col-sm-6 col-md-6">
                           <address>
                              <strong>Adres do wysyłki</strong> <br>
                              ↪${order.shippingDetail.name}<br>
                              <!-- W podobny sposób umieść wartości pozostałych pól
                              ↪obiektu order.shippingDetail. W tym listingu zostały
                              ↪pominięte. -->
                           </address>
                        </div>
                     </div>
                     <!-- Umieść wartości wszystkich pól obiektu order w tabelce HTML,
                     ↪używając wyrażenia "${}". W tym listingu zostały pominięte. -->
                     <button id="back" class="btn btn-default" name="
                     ↪_eventId_backToCollectShippingDetail">wstecz</button>
                     <button type="submit" class="btn btn-success" name="
                     ↪_eventId_orderConfirmed">Zatwierdź <span class="glyphicon
                     ↪glyphiconchevron-right"></span>
                     </button>
                     <button id="btnCancel" class="btn btn-default" name="
                     ↪_eventId_cancel">Anuluj</button>
                     </div>
               </div>
            </form:form>
         </div>
      </div>
   </body>
</html>
```

4. Potrzebujesz widoku prezentującego komunikat błędu w przypadku, gdy użytkownik spróbuje złożyć zamówienie, nie umieszczając wcześniej żadnych przedmiotów w koszyku. Utwórz plik *InvalidCartWarning.jsp* i umieść w nim zawartość listingu 9.28.

Listing 9.28. Implementacja widoku InvalidCartWarning

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
   <head>
```

```
      <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
      <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
      ↪3.0.0/css/bootstrap.min.css">
      <title>Niepoprawny koszyk</title>
   </head>
   <body>
      <section>
         <div class="jumbotron">
            <div class="container">
               <h1 class="alert alert-danger">Niepoprawny koszyk</h1>
            </div>
         </div>
      </section>
      <section>
         <div class="container">
            <p>
               <a href="<spring:url value="/products" />" class="btn btn-
               ↪primary">
                  <span class="glyphicon-hand-left glyphicon"></span> produkty
               </a>
            </p>
         </div>
      </section>
   </body>
</html>
```

5. Potrzebujesz widoku prezentującego podziękowanie użytkownikowi po pomyślnym zrealizowaniu przepływu zakupu. Utwórz plik widoku JSP o nazwie *thankCustomer.jsp* i umieść w nim zawartość listingu 9.29.

Listing 9.29. Implementacja widoku thankCustomer

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
      <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
      ↪3.0.0/css/bootstrap.min.css">
      <title>Dziękujemy! </title>
   </head>
   <body>
      <section>
         <div class="jumbotron">
            <div class="container">
               <h1 class="alert alert-danger"> Dziękujemy!</h1>
```

```
                <p>Dziękujemy za złożenie zamówienia. Zostanie przesłane
                   <fmt:formatDate type="date" value="${order.shippingDetail.
                   ↪shippingDate}" pattern="dd.MM.yyyy"/>!
                </p>
                <p>Numer Twojego zamówienia to ${order.orderId}.</p>
             </div>
          </div>
       </section>
       <section>
          <div class="container">
             <p>
                <a href="<spring:url value="/products" />" class=
                ↪"btn btn-primary">
                   <span class="glyphicon-hand-left glyphicon"></span> produkty
                </a>
             </p>
          </div>
       </section>
    </body>
</html>
```

6. Jeśli użytkownik zdecyduje się na anulowanie składanego zamówienia w dowolnym z widoków, musisz przedstawić mu informację, że ta operacja się powiodła. W tym celu utwórz plik widoku *checkOutCancelled.jsp* i umieść w nim zawartość listingu 9.30.

Listing 9.30. Implementacja widoku checkOutCancelled

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset="utf-8">
      <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/
      ↪3.0.0/css/bootstrap.min.css">
      <title>Składanie zamówienia anulowane! </title>
   </head>
   <body>
      <section>
         <div class="jumbotron">
            <div class="container">
               <h1 class="alert alert-danger">Składanie zamówienia
               ↪anulowano.</h1>
               <p>Proces składania zamówienia został anulowany! Możesz wrócić
               ↪do zakupów ...</p>
            </div>
         </div>
      </section>
```

```
        <section>
            <div class="container">
                <p>
                    <a href="<spring:url value="/products" />" class="btn
                    ↪btn-primary">
                        <span class="glyphicon-hand-left glyphicon"></span> produkty
                    </a>
                </p>
            </div>
        </section>
    </body>
</html>
```

7. Otwórz plik *cart.jsp* znajdujący się w katalogu *src/main/webapp/WEB-INF/views* i atrybutowi `href` hiperłącza *Kupuję* nadaj wartość `<spring:url value="/checkout?cartId=${cartId}"/>`, tak jak przedstawia listing 9.31.

Listing 9.31. Implementacja hiperłącza rozpoczynającego przepływ zakupu

```
<a href= "<spring:url value="/checkout?cartId=${cartId}"/>" class="btn
↪btn-success pull-right">
    <span class="glyphicon-shopping-cart glyphicon"></span> Kupuję
</a>
```

8. Uruchom aplikację i odwiedź adres: *http://localhost:8080/webstore/products*. Naciśnij przycisk *Szczegóły* dowolnego produktu, aby przejść na stronę ze szczegółowymi informacjami o produkcie. Tam dodaj go do koszyka, używając przycisku *Zamów teraz*. Przejdź do strony prezentującej zawartość koszyka, używając przycisku *Koszyk*. Znajdziesz na niej przycisk *Złóż zamówienie*. Naciśnij go, by wyświetliła się zaprezentowana na rysunku 9.2 strona służąca do wprowadzania informacji o kliencie.
9. Jeśli po wprowadzeniu danych klienta naciśniesz przycisk *Dodaj*, Spring Web Flow przekieruje Cię do następnego stanu. W nim zostanie wyświetlony formularz służący do wprowadzenia danych do dostawy, a następnie zostaniesz poproszony o potwierdzenie zamówienia. Na koniec Spring Web Flow zaprezentuje Ci stronę z podziękowaniem, będącą jednocześnie stanem końcowym przepływu.

Co się właśnie wydarzyło?

W krokach 1. – 6. utworzyłeś pliki widoków JSP dla każdego stanu widoku. Jak zapewne pamiętasz, zdefiniowałeś atrybut `model`, dostępny dla każdego stanu widoku w pliku *checkout-flow.xml*.

Model zdefiniowany w listingu 9.32 został użyty w znaczniku `<form:form>` widoku w sposób przedstawiony w listingu 9.33.

Rysunek 9.2. Formularz wprowadzania danych klienta

Listing 9.32. Definicja modelu w stanie collectCustomerInfo

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
➥model="order">
   <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Listing 9.33. Użycie modelu stanu przepływu w widoku

```
<form:form modelAttribute="order.customer" class="form-horizontal">
   <fieldset>
      <legend>Dane klienta</legend>
         <div class="form-group">
            <label class="control-label col-lg-2 col-lg-2"
            ➥for="customerId">Identyfikator klienta</label>
            <div class="col-lg-10">
               <form:input id="customerId" path="customerId" type="text"
               ➥class="form:input-large" />
            </div>
         </div>
```

Listing 9.33 zawiera fragment pliku widoku *collectCustomerInfo.jsp*, w którym znacznik `<form:input>` został powiązany z polem `customerId` obiektu `customer` pochodzącego z obiektu modelu (`order.customer`). W podobny sposób powiązałeś obiekt `shippingDetail` z plikiem widoku *collectShippingDetail.jsp* oraz obiekt `order` z plikiem widoku *orderConfirmation.jsp*.

Połączyłeś obiekty typu `Order`, `Customer` i `ShippingDetail` z widokami. Pora odpowiedzieć na pytanie: co się stanie, gdy użytkownik naciśnie przycisk zatwierdzający formularz, cofający do poprzedniego kroku lub anulujący zamówienie? Przyjrzyj się poniższemu fragmentowi kodu, pochodzącemu z pliku *collectCustomerInfo.jsp*:

```
<input type="submit" id="btnAdd" class="btn btn-primary" value="Utwórz"
➥name="_eventId_customerInfoCollected" />
```

Na pierwszy rzut oka znacznik `<input>` wygląda jak przycisk zatwierdzający formularz. Wyróżnia go jednak atrybut `name` (`name="_eventId_customerInfoCollected"`). Nadałeś atrybutowi `name` znacznika `<input>` wartość `_eventId_customerInfoCollected` w celu poinstruowania Spring Web Flow, by zainicjował zdarzenie na podstawie atrybutu `name`. Ponieważ przedrostkiem atrybutu `name` jest `_eventId_` (`_eventId_customerInfoCollected`), Spring Web Flow rozpoznał go jako nazwę zdarzenia i zainicjował nowe zdarzenie, o nazwie `customerInfoCollected`.

Jak już wiesz, przejście między stanami jest wyzwalane za pomocą zdarzenia. Listing 9.34 ilustruje, jak po przesłaniu formularza znajdującego się w widoku `collectCustomerInfo` Spring Web Flow przekieruje Cię do następnego stanu, o nazwie `collectShippingDetail`.

Listing 9.34. Definicja przejścia ze stanu collectCustomerInfo do stanu collectShippingDetail

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
➥model="order">
   <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

W podobny sposób możesz przyporządkować nazwy zdarzeń do pozostałych przycisków. Listing 9.35 prezentuje przyporządkowanie zdarzeń do przycisków *Wstecz* oraz *Anuluj*.

Listing 9.35. Przyporządkowanie zdarzeń do przycisków Wstecz oraz Anuluj w pliku widoku collectShippingDetail.jsp

```
<button id="back" class="btn btn-default"
➥name="_eventId_backToCollectCustomerInfo">Wstecz</button>
<button id="btnCancel" class="btn btn-default"
➥name="_eventId_cancel">Anuluj</button>
```

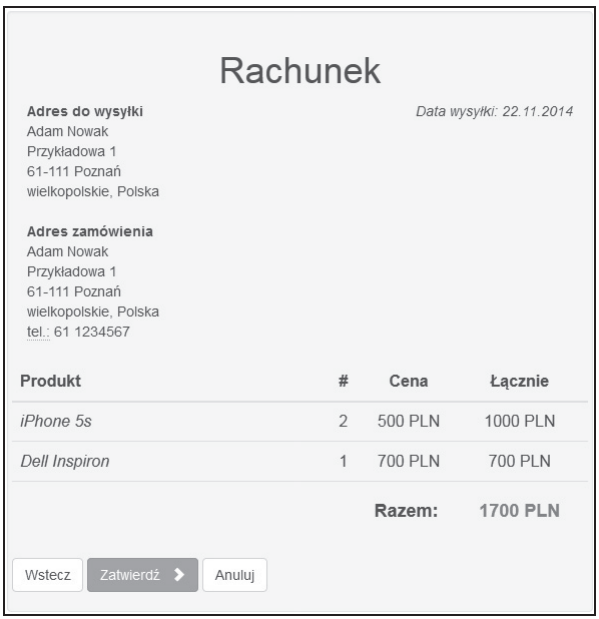
Wiesz już, jak zainicjować zdarzenie z poziomu widoku, by przekierować przepływ pomiędzy stanami. Pozostało do omówienia jeszcze jedno ważne zagadnienie związane z przepływami, tj. identyfikacja przepływów w uruchomionej aplikacji. Kiedy przepływ znajdzie się w stanie widoku, jego wykonanie zostaje wstrzymane do czasu wykonania akcji przez użytkownika, np. wypełnienia i przesłania formularza. Gdy użytkownik prześle formularz lub zdecyduje się na anulowanie go, wraz z danymi formularza zostaje przesłany klucz identyfikujący przepływ. Dzięki temu właściwy przepływ będzie wznowiony. Klucz jest przesyłany za pomocą ukrytego pola, zaprezentowanego poniżej:

```
<input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}"/>
```

Jeśli przyjrzysz się plikom widoków odpowiadającym stanom widoków przepływów, takim jak *collectCustomerInfo.jsp* czy *collectShippingDetail.jsp*, w każdym z nich znajdziesz przytoczony powyżej znacznik. Spring Web Flow przechowuje unikalny klucz wywołania przepływu w atrybucie modelu o nazwie `flowExecutionKey` w każdym widoku powiązanym z przepływem. Musisz przechowywać tę wartość w zmiennej o nazwie `_flowExecutionKey`, by poprawnie identyfikować przepływ w Spring Web Flow.

To już wszystko, co musisz wiedzieć o widokach powiązanych z definicją przepływu zakupu. Pozostało dowiedzieć się, jak wywołać przepływ, naciskając przycisk *Złóż zamówienie*. Jak już wiesz, aby wywołać przepływ, musisz przesłać żądanie zawierające identyfikator koszyka. Dlatego też w kroku 7. zmieniłeś atrybut `href` hiperłącza *Złóż zamówienie*, by generowało adres podobny do: *http://localhost:8080/webstore/checkout?cartId=55AD1472D4EC*.

Jeśli dodasz kilka produktów do koszyka, a następnie naciśniesz przycisk *Złóż zamówienie*, zainicjujesz przepływ zakupu. Gdy osiągniesz stan `orderConfirmation`, zostanie wyświetlona strona zaprezentowana na rysunku 9.3.



Rysunek 9.3. Strona potwierdzenia zamówienia

Dla ambitnych — dodawanie stanu decyzyjnego

Wprawdzie skończyłeś pracę z przepływem, ale możesz jeszcze dodać do niego kilka usprawnień. Przykładowo: za każdym razem, gdy rozpoczyna się przepływ zakupu, zbierasz dane o kliencie. A jeśli zamówienie składa klient, który robił już zakupy w sklepie? Jest bardzo prawdopodobne,

że nie ma ochoty po raz kolejny wprowadzać swoich danych. Możesz automatycznie wypełnić jego dane, używając zapisanych informacji. Oto kilka pomysłów na usprawnienie obsługi regularnych klientów:

- Utwórz repozytorium klientów oraz warstwę usług pozwalającą na przechowywanie, odczytywanie i odnajdywanie obiektów typu `Customer`. Zastanów się nad utworzeniem w interfejsach `CustomerRepository` i `CustomerService` oraz ich implementacjach poniższych metod:
  - `public void saveCustomer(Customer customer)`,
  - `public Customer getCustomer(String customerId)`,
  - `public Boolean isCustomerExist(String customerId)`.
- Zdefiniuj w pliku *checkout-flow.xml* stan widoku pozwalający na wprowadzenie identyfikatora użytkownika. Nie zapomnij o utworzeniu pliku widoku JSP.
- Utwórz stan decyzyjny w pliku *checkout-flow.xml*, sprawdzający za pośrednictwem `CustomerService`, czy w `CustomerRepository` istnieje klient o wskazanym identyfikatorze. Na podstawie zwróconej wartości logicznej przekieruj przepływ do stanu widoku zbierającego dane użytkownika lub wypełnij obiekt `order.customer` danymi z `CustomerRepository`. Przykładowy stan decyzyjny został zaprezentowany w listingu 9.36.

Listing 9.36. Przykładowy stan decyzyjny

```
<decision-state id="checkCustomerExist">
   <if test="customerServiceImpl.isCustomerExist(order.customer.customerId)"
      then=" collectShippingDetail"
      else=" collectCustomerInfo"/>
</decision-state>
```

- Nie zapomnij o zapisaniu zebranych danych klienta w `CustomerRepository` za pomocą stanu akcji. Po przejściu stanu decyzyjnego wypełnij obiekt `order.customer`.

Zwiększanie możliwości ponownego użycia kodu interfejsu użytkownika za pomocą Apache Tiles

W poprzednich rozdziałach utworzyłeś w aplikacji zbiór stron internetowych (widoków), takich jak strona prezentująca produkty, strona pozwalająca na dodawanie produktów itd. Mimo że każdy z widoków realizuje inne zadanie, wszystkie wyglądają podobnie. Przykładowo: każdy widok ma nagłówek, obszar przeznaczony na treść strony itp. Do tej pory wszystkie te elementy

były powtórzone we wszystkich plikach widoków JSP. Jest to złe rozwiązanie m.in. dlatego, że jeśli w przyszłości chciałbyś zmienić szatę graficzną strony, musiałbyś nanieść zmiany we wszystkich plikach widoków.

Współczesne aplikacje internetowe rozwiązują ten problem, używając mechanizmu szablonów. Apache Tiles jest jednym ze szkieletów wspomagających tworzenie szablonów. Tiles pozwala programiście zdefiniować fragmenty wielokrotnego użytku (kafelki) pozwalające na złożenie kompletnych stron w trakcie działania aplikacji. Wspomniane fragmenty mogą mieć parametry pozwalające na generowanie dynamicznej zawartości. Dzięki temu zwiększa się możliwość ponownego użycia kodu i zmniejsza jego duplikację.

Ćwiczenie praktyczne — tworzenie widoków dla każdego stanu widoku

Tyle tytułem wstępu. Pora rozpocząć pracę z Apache Tiles, tworząc wspólną szatę graficzną dla aplikacji internetowej, która będzie rozszerzana przez strony. Wykonaj poniższe kroki:

1. Otwórz plik *pom.xml*. Znajdziesz go w katalogu głównym projektu.
2. Na dole okna prezentującego zawartość pliku znajduje się kilka zakładek. Przejdź do tej o nazwie *Dependencies* i użyj przycisku *Add* w sekcji *Dependencies*.
3. Pojawi się okno o nazwie *Select Dependency*. W polu *Group Id* wprowadź `org.apache.tiles`, polu *Artifact Id* nadaj wartość `tiles-extras`, polu *Version* wartość `3.0.3`, a polu *Scope* wartość `compile`. Naciśnij przycisk *OK* i zapisz plik *pom.xml*.
4. Dodaj kolejną zależność o parametrze *Group Id* równym `org.slf4j`, *Artifact Id* równym `slf4j-api`, *Version* równym `3.0.3` oraz *Scope* równym `compile`. Następnie naciśnij przycisk *OK* i zapisz plik *pom.xml*.
5. Utwórz podkatalog *tiles/definition* w katalogu *src/main/webapp/WEB-INF/*. W nim utwórz plik XML o nazwie *tile-definition.xml*, a następnie umieść w nim zawartość listingu 9.37. Zauważ, że wspomniany listing nie jest kompletny. Pełną definicję nazw widoków i pól znajdziesz w kodzie źródłowym, który możesz pobrać ze strony: *www.helion.pl/ksiazki/troyaid.html*.

Listing 9.37. Definicje plików widoków w Apache Tiles

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles
➥Configuration 3.0//EN" "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
   <definition name="baseLayout" template="/WEB-INF/tiles/template/
   ➥baseLayout.jsp">
      <put-attribute name="title" value="Przykładowy tytuł" />
      <put-attribute name="heading" value="" />
```

```
        <put-attribute name="tagline" value="" />
        <put-attribute name="navigation" value="/
        ↪WEB-INF/tiles/template/navigation.jsp" />
        <put-attribute name="content" value="" />
        <put-attribute name="footer" value="/WEB-INF/tiles/template/
        ↪footer.jsp" />
    </definition>
    <definition name="welcome" extends="baseLayout">
        <put-attribute name="title" value="Produkty" />
        <put-attribute name="heading" value="Produkty" />
        <put-attribute name="tagline" value="Dostępne produkty" />
        <put-attribute name="content" value="/WEB-INF/views/products.jsp" />
    </definition>
    <definition name="products" extends="baseLayout">
        <put-attribute name="title" value="Produkty" />
        <put-attribute name="heading" value="Produkty" />
        <put-attribute name="tagline" value="Dostępne produkty" />
        <put-attribute name="content" value="/WEB-INF/views/products.jsp" />
    </definition>
    <definition name="product" extends="baseLayout">
        <put-attribute name="title" value="Produkt" />
        <put-attribute name="heading" value="Produkt" />
        <put-attribute name="tagline" value="Produkt" />
        <put-attribute name="content" value="/WEB-INF/views/product.jsp" />
    </definition>
    <!--W podobny sposób dodaj znaczniki definition dla każdej nazwy widoku.
    Dla oszczędności miejsca zostały pominięte w tym listingu. Znajdziesz
    je w paczce z kodem źródłowym przykładów do książki.-->
</tiles-definitions>
```

6. Utwórz podkatalog o nazwie *template* w katalogu *src/main/webapp/WEB-INF/tiles*. Następnie utwórz w nim plik JSP o nazwie *baseLayout.jsp* i umieść w nim zawartość listingu 9.38.

Listing 9.38. Implementacja szablonu widoku baseLayout.jsp

```
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tagstiles"%>
<!DOCTYPE html>
<html lang="pl">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initialscale=1.0">
        <title><tiles:insertAttribute name="title" /></title>
```

```
        <link href="http://getbootstrap.com/dist/css/bootstrap.css"
        ↪rel="stylesheet">
        <link href="http://getbootstrap.com/examples/jumbotron/jumbotron.css"
        ↪rel="stylesheet">
    </head>
    <body>
        <div class="container">
            <div class="header">
                <ul class="nav nav-pills pull-right">
                    <tiles:insertAttribute name="navigation" />
                </ul>
                <h3 class="text-muted">Sklep internetowy</h3>
            </div>
            <div class="jumbotron">
                <h1>
                    <tiles:insertAttribute name="heading" />
                </h1>
                <p>
                    <tiles:insertAttribute name="tagline" />
                </p>
            </div>
            <div class="row">
                <tiles:insertAttribute name="content" />
            </div>
            <div class="footer">
                <tiles:insertAttribute name="footer" />
            </div>
        </div>
    </body>
</html>
```

7. W katalogu *template* utwórz szablon JSP o nazwie *navigation.jsp* i umieść w nim zawartość listingu 9.39.

Listing 9.39. Implementacja szablonu navigation.jsp

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<li><a href="<spring:url value="/products"/>">Strona główna</a></li>
<li><a href="<spring:URL value="/products/"/>">Produkty</a></li>
<li><a href="<spring:url value="/products/add"/>">Dodaj produkt</a></li>
<li><a href="<spring:url value="/cart/"/>">Koszyk</a></li>
```

8. Utwórz ostatni plik szablonu JSP, o nazwie *footer.jsp*, i umieść w nim poniższą linię kodu:

```
<p>&copy; Firma 2014</p>
```

9. Utworzyłeś podstawowy szablon strony i definicje kafelków wszystkich swoich stron. Teraz musisz usunąć wspólne elementy z plików widoków JSP. Przykładowo: jeśli

z widoku *products.jsp* usuniesz sekcję `jumbotron` i zostawisz wyłącznie sekcję `container`, Twój plik widoku będzie wyglądał tak, jak przedstawia to listing 9.40. Pamiętaj, że nie powinieneś usuwać odwołań `taglib` oraz odnośników do plików JavaScript.

Listing 9.40. Widok products.jsp po usunięciu części wspólnej

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<section class="container">
   <div class="row">
      <c:forEach var="product" items="${products}">
         <div class="col-sm-6 col-md-3" style="padding-bottom: 15px">
            <div class="thumbnail">
               <img src="<c:url value="/resource/images/${product.
               ↪productId}.png"></c:url>" alt="image" style="width: 100%" />
               <div class="caption">
                  <h3>${product.name}</h3>
                  <p>${product.description}</p>
                  <p>${product.unitPrice} PLN</p>
                  <p>Liczba dostępnych sztuk: ${product.unitsInStock} </p>
                  <p>
                     <a href=" <spring:url value="/products/product?id=
                     ↪${product.productId}" /> " class="btn btn-primary">
                        <span class="glyphicon-info-sign glyphicon"
                        ↪/></span> Szczegóły
                     </a>
                  </p>
               </div>
            </div>
         </div>
      </c:forEach>
   </div>
</section>
```

10. Analogicznie usuń sekcję `jumbotron` z każdego pliku widoku JSP znajdującego się w katalogu *src/main/webapp/WEB-INF/views*. Pamiętaj, aby nie usuwać odwołań `taglib` oraz odnośników do plików JavaScript. Przykładowo: w pliku *cart.jsp* powinieneś zachować kod zaprezentowany w listingu 9.41.

Listing 9.41. Odwołania taglib oraz odnośniki do plików JavaScript pozostawione w pliku cart.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.1/
↪angular.min.js"></script>
<script src="/webstore/resource/js/controllers.js"></script>
```

11. W pliku konfiguracji kontekstu aplikacji (*DispatcherServlet-context.xml*) umieść zaprezentowanego w listingu 9.42 beana `UrlBasedViewResolver` posiadającego parametr `TilesView`.

Listing 9.42. Definicja beana tilesViewResolver

```
<bean id="tilesViewResolver" class="org.springframework.web.servlet.
➥view.UrlBasedViewResolver">
   <property name="viewClass" value="org.springframework.web.servlet.
   ➥view.tiles3.TilesView" />
   <property name="order" value="-2" />
</bean>
```

12. Utwórz w pliku *DispatcherServlet-context.xml* zaprezentowaną w listingu 9.43 definicję beana klasy `TilesConfigurer`. Jego zadaniem jest wskazanie pliku z definicjami kafelków.

Listing 9.43. Definicja beana klasy TilesConfigurer

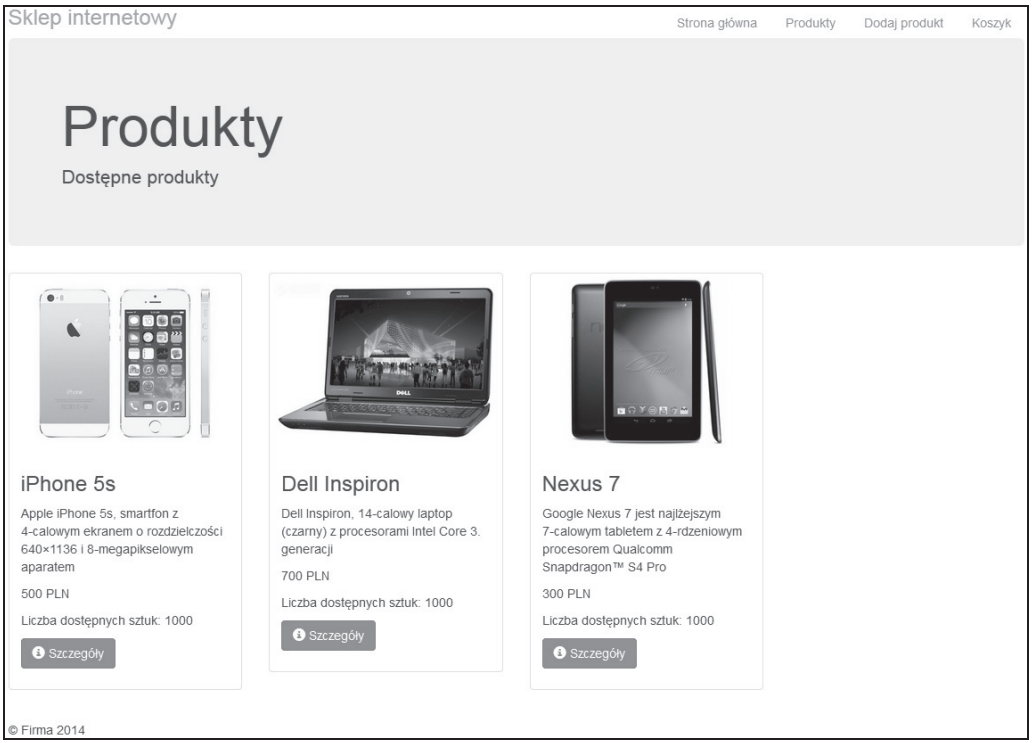
```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.
➥view.tiles3.TilesConfigurer">
   <property name="definitions">
      <list>
         <value>/WEB-INF/tiles/definitions/tile-definition.xml</value>
      </list>
   </property>
</bean>
```

13. Uruchom aplikację i wprowadź adres: *http://localhost:8080/webstore/products*. Zostanie wyświetlona strona prezentująca listę produktów, widoczna na rysunku 9.4. Jej nowymi elementami są pasek nawigacji na górze oraz stopka na dole. Dzięki paskowi nawigacji możesz dodać nowy produkt do oferty sklepu, używając hiperłącza *Dodaj produkt*.

Co się właśnie wydarzyło?

W krokach 1. – 4. za pomocą Mavena dodałeś do projektu biblioteki JAR niezbędne w celu rozpoczęcia pracy z Apache Tiles. Krok 5. jest istotny. Utworzyłeś w nim plik zawierający definicję kafelków (*tile-definition.xml*). Zrozumienie zawartości wspomnianego pliku jest bardzo ważne w perspektywie dalszej pracy z Apache Tiles. Pora przyjrzeć mu się szczegółowo.

Plik definicji kafelków jest kolekcją definicji. Każda z nich może być powiązana z szablonem za pośrednictwem atrybutu `template`, tak jak przedstawia to listing 9.44.



Rysunek 9.4. Lista produktów wykonana za pomocą Apache Tiles

Listing 9.44. Definicja szablonu baseLayout

```
<definition name="baseLayout" template = "/WEB-INF/tiles/template/
↪baseLayout.jsp" >
   <put-attribute name="title" value="Przykładowy tytuł" />
   <put-attribute name="heading" value="Przykładowy nagłówek" />
   <put-attribute name="tagline" value="Przykładowy slogan" />
   <put-attribute name="navigation" value = "/WEB-INF/tiles/template/
   ↪navigation.jsp" />
   <put-attribute name="content" value="" />
   <put-attribute name="footer" value = "/WEB-INF/tiles/template/footer.jsp" />
</definition>
```

Możesz utworzyć wiele atrybutów wewnątrz każdej definicji. Każdy z atrybutów może być zarówno zmienną tekstową, jak i sporym plikiem widoku. Wszystkie są dostępne w szablonie za pośrednictwem znacznika `<tiles:insertAttribute>`. Przykładowo: jeśli otworzysz plik *baseLayout.jsp*, zauważysz w nim zawartość listingu 9.45 wewnątrz znacznika `<div>` klasy `jumbotron`.

Listing 9.45. Wyświetlanie w widoku atrybutów zdefiniowanych w kafelkach

```
<h1>
   <tiles:insertAttribute name="heading" />
</h1>
<p>
   <tiles:insertAttribute name="tagline" />
</p>
```

Podczas pracy aplikacji Apache Tiles wstawi tekst `Przykładowy nagłówek` w miejsce znacznika `<tiles:insertAttribute name="heading" />`. Analogicznie w miejsce znacznika `<tiles:insert ↪Attribute name="tagline" />` zostanie wstawiony tekst `Przykładowy slogan`. Kafelek `baseLayout` jest powiązany z szablonem */WEB-INF/tiles/template/baseLayout.jsp*. Możesz w nim umieścić za pomocą znacznika `<tiles:insertAttribute>` zdefiniowane atrybuty, takie jak `title`, `heading` lub `tagline`.

Apache Tiles pozwala na rozszerzanie definicji analogiczne do rozszerzania klasy Javy. Istniejące atrybuty są dostępne dla pochodnej definicji. Mogą być również przez nią zmieniane. Przyjrzyj się listingowi 9.46, prezentującemu fragment pliku *tile-definition.xml*.

Listing 9.46. Definicja kafelka products

```
<definition name="products" extends="baseLayout">
   <put-attribute name="title" value="Produkty" />
   <put-attribute name="heading" value="Produkty" />
   <put-attribute name="tagline" value="Dostępne produkty" />
   <put-attribute name="content" value="/WEB-INF/views/products.jsp" />
</definition>
```

Przedstawiona w listingu 9.46 definicja jest rozszerzeniem definicji `baseLayout`. Nadpisuje atrybuty `title`, `heading`, `tagline` oraz `content`, a ponieważ nie wskazałeś dla niej szablonu, używa szablonu skonfigurowanego dla definicji `baseLayout`.

W podobny sposób utworzyłeś definicje wszystkich nazw widoków zwracanych przez kontrolery. Zauważ, że każda nazwa definicji, z wyjątkiem `baseLayout`, jest jednocześnie nazwą widoku.

Następnie w krokach 6. – 8. utworzyłeś szablony, których możesz używać w definicjach kafelków. Na początku utworzyłeś szablon widoku (*baseLayout.jsp*), następnie szablon nawigacji (*navigation.jsp*), a na koniec szablon stopki (*footer.jsp*).

W krokach 9. – 10. dowiedziałeś się, jak usunąć z wszystkich plików JSP istniejący nadmiarowy kod, np. znacznik `<div> jumbotron`. Bądź ostrożny podczas usuwania kodu. Nie usuń przez przypadek odwołań `taglib` ani odnośników do plików JavaScript.

Podczas kroku 11. zdefiniowałeś `UrlBasedViewResolver` dla `TilesView` (`org.springframework.web. ↪servlet.view.tiles3.TilesView`), aby odwzorowywać nazwy widoków na widoki kafelkowe.

Na koniec w kroku 12. skonfigurowałeś `TilesConfigurer` (`org.springframework.web.servlet.view.tiles3.TilesConfigurer`), mający za zadanie lokalizację plików definiujących kafelki szkieletu Apache Tiles.

To tyle! Jeśli uruchomisz aplikację i wprowadzisz w przeglądarce adres: *http://localhost:8080/webstore/products*, wyświetli się strona z listą produktów, wzbogacona, jak zostało wspomniane w kroku 13., o pasek nawigacji na górze oraz stopkę na dole. Możesz przechodzić do strony dodawania produktów, używając hiperłącza *Dodaj produkt*. Do tej pory, gdy kontroler zwracał nazwę widoku, `InternalResourceViewResolver` odnajdywał odpowiadający jej widok JSP. Teraz `UrlBasedViewResolver`, korzystając z definicji szablonu, tworzy widok na podstawie jego nazwy.

Krótki test — Apache Tiles

Pytanie 1. Które zdania są prawdziwe w kontekście Apache Tiles?

1. Logiczna nazwa widoku zwracana przez kontroler musi być taka sama jak nazwa znacznika `<definition>`.
2. Znacznik `<tiles:insertAttribute>` jest wypełniaczem w szablonie.
3. Znacznik `<definition>` może rozszerzać inny znacznik `<definition>`.

Podsumowanie

Spring Web Flow oraz Apache Tiles są dwoma odrębnymi szkieletami. Przyswoiłeś sobie minimum informacji pozwalające na szybkie zapoznanie się z oboma. Na początek poznałeś podstawowe elementy szkieletu Spring Web Flow, pozwalające na utworzenie przepływu zakupu w aplikacji sklepu internetowego. W drugiej części rozdziału dowiedziałeś się, jak wykorzystać szkielet Apache Tiles, aby zapewnić maksimum wielokrotnego wykorzystania plików widoków i zapewnić spójny wygląd interfejsu użytkownika na wszystkich stronach aplikacji.

W następnym rozdziale dowiesz się, jak testować aplikację internetową, używając rozmaitych API dostarczanych przez Springa MVC.

Skorowidz

A

B

D

E

F

G

H

I

J

K

L

M

O

P

R

S

T

U

V

W

Z

Ż

Spring MVC Przewodnik dla początkujących

Spring MVC to szkielet dostarczający architekturę model-widok-kontroler (ang. Model-View-Controller framework). Za jego pomocą można zbudować wiele zaawansowanych aplikacji internetowych oraz REST-owe API. Spring MVC zapewnia niezwykłą elastyczność oraz gwarantuje programistom wygodę. To leży u podstaw jego ogromnej popularności. Wokół tego szkieletu powstała też duża społeczność, zawsze chętna do udzielania pomocy.

Jeżeli chcesz poznać możliwości Spring MVC i zbudować z nim Twoją pierwszą aplikację, nie mogłeś trafić lepiej. Ta książka wprowadzi Cię w najważniejsze zagadnienia oraz w niezwykle przejrzysty sposób przedstawi dostępne możliwości. Na samym początku zainstalujesz środowisko JDK oraz przydatne narzędzia, takie jak Maven i Spring Tool Suite. Budowana tu przykładowa aplikacja to sklep internetowy działający na serwerze Apache Tomcat. Z kolejnych rozdziałów nauczysz się, jak korzystać z kontrolerów, bibliotek znaczników oraz walidatorów. Książka ta jest obowiązkową lekturą dla wszystkich początkujących użytkowników Spring MVC.

Zbuduj aplikację internetową ze szkieletem Spring MVC!

Dzięki tej książce:

- przygotujesz swoje środowisko pracy
- poznasz rolę kontrolerów, modelu oraz widoków
- wykorzystasz walidatory
- przygotujesz automatyczne testy dla Twojej aplikacji
- opanujesz możliwości Spring MVC

Amuthan G — ma wieloletnie doświadczenie programistyczne. Dysponuje szeroką wiedzą w zakresie języka Java, szkieletu Spring oraz standardu JPA. Pracuje dla dużej firmy tworzącej chmurę obliczeniową. W wolnych chwilach prowadzi bloga.

ISBN 978-83-283-0517-5

cena: 49,00 zł