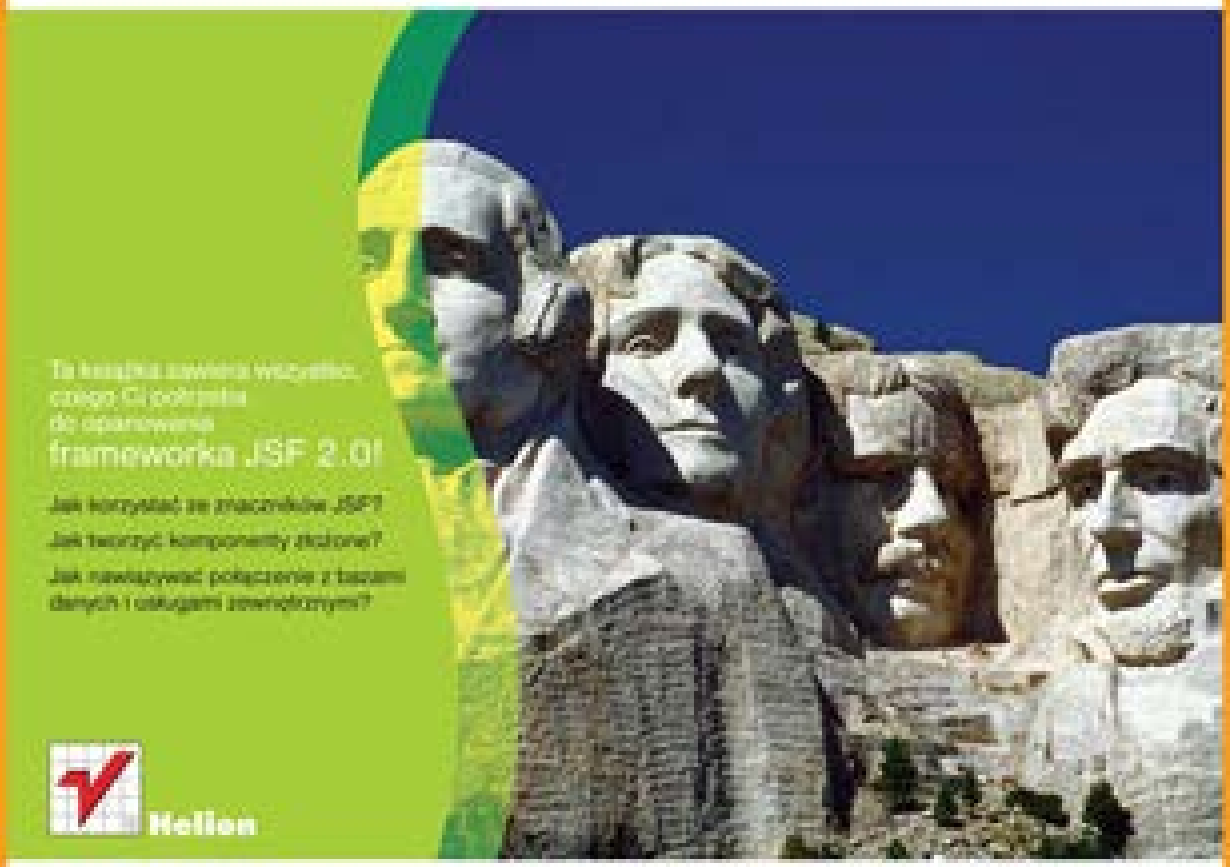
JavaServer™ Faces

WYDANIE III



Ta książka zawiera wszystko, czego Ci potrzeba do opanowania frameworka JSF 2.0!

Jak korzystać ze znaczników JSF?

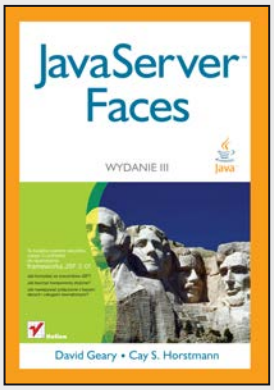
Jak tworzyć komponenty złożone?

Jak nawiązywać połączenie z bazami danych i usługami zewnętrznymi?

Helion

David Geary • Cay S. Horstmann

JavaServer Faces. Wydanie III



Autorzy: David Geary, Cay S. Horstmann
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-2904-6
Tytuł oryginału: Core JavaServer Faces (3rd Edition)
Format: 172×245, stron: 584

Ta książka zawiera wszystko, czego Ci potrzeba do opanowania frameworka JSF 2.0!

- Jak korzystać ze znaczników JSF?
- Jak tworzyć komponenty złożone?
- Jak nawiązywać połączenie z bazami danych i usługami zewnętrznymi?

JavaServer Faces (JSF) to technologia platformy Java EE, ułatwiająca projektowanie i tworzenie interfejsów użytkownika aplikacji internetowych. Umożliwia sprawną pracę nad aplikacjami działającymi po stronie serwera i wprowadzanie jasnego podziału na wizualną prezentację oraz właściwą logikę aplikacji. Specyfikacja JSF 2.0 (inaczej niż poprzednia) jest pochodną wielu rzeczywistych projektów open source. Dzięki temu sam framework jest dużo prostszy i lepiej zintegrowany ze stosem technologii Java EE niż wersja JSF 1.0. Co więcej, jego specyfikacja przewiduje teraz obsługę technologii takich, jak AJAX czy REST. Framework JSF 2.0 jest obecnie jednym z najznamienitszych frameworków aplikacji internetowych tworzonych w Javie. Do jego mocnych stron należą także: uproszczony model programowania poprzez zastosowanie adnotacji i wprowadzenie zasady konwencji ponad konfiguracją oraz rozszerzalny model komponentów.

Książka „JavaServer Faces. Wydanie III" zawiera wszystko, czego trzeba do opanowania rozbudowanych elementów frameworka JSF 2.0. Poznaj tajniki znaczników frameworka JSF oraz obsługi zdarzeń. Dowiedz się, jak budować komponenty złożone, i naucz się implementować własne, niestandardowe. Wykorzystaj w swoich aplikacjach technologię AJAX i opanuj nawiązywanie połączeń z bazami danych czy innymi usługami zewnętrznymi. W ostatnim rozdziale znajdziesz pomocne wskazówki na temat diagnozowania i rejestrowania zdarzeń, a także praktyczne przykłady kodu, rozszerzające technologię JSF.

- Komponenty zarządzane
- Zasięg komponentów
- Nawigacja statyczna i dynamiczna
- Znaczniki standardowe
- Facelety
- Tabele danych
- Konwersja i weryfikacja danych
- Obsługa zdarzeń
- Komponenty złożone
- Technologia AJAX
- Usługi zewnętrzne
- Praca z bazami danych

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

Spis treści

3

Nawigacja

W tym krótkim rozdziale omówimy sposób konfigurowania reguł nawigacji w ramach budowanej aplikacji internetowej. W szczególności wyjaśnimy, jak aplikacja może przechodzić od jednej do drugiej strony internetowej w zależności od czynności wykonywanych przez użytkownika i decyzji podejmowanych na poziomie logiki biznesowej.

Nawigacja statyczna

Przeanalizujmy teraz sytuację, w której użytkownik aplikacji internetowej wypełnia formularz na stronie internetowej. Użytkownik może wpisywać dane w polach tekstowych, klikać przyciski opcji i wybierać elementy list.

Wszystkie te działania są realizowane na poziomie przeglądarki internetowej użytkownika. Kiedy użytkownik klika przycisk akceptujący i wysyłający dane formularza, zmiany są przekazywane na serwer.

Aplikacja internetowa analizuje wówczas dane wpisane przez użytkownika i na ich podstawie decyduje, której strony JSF należy użyć do wizualizacji odpowiedzi. Za wybór kolejnej strony JSF odpowiada **mechanizm obsługujący nawigację** (ang. *navigation handle*).

W przypadku prostych aplikacji internetowych nawigacja pomiędzy stronami ma charakter statyczny. Oznacza to, że kliknięcie określonego przycisku zawsze powoduje przejście do tej samej strony JSF odpowiedzialnej za wizualizację odpowiedzi. W podrozdziale „Prosty przykład" w rozdziale 1. Czytelnicy mieli okazję zapoznać się z najprostszym mechanizmem zapisywania reguł nawigacji statycznej pomiędzy stronami JSF.

Dla każdego przycisku definiujemy atrybut `action` — przykład takiego rozwiązania przedstawiono poniżej:

```
<h:commandButton label="Zaloguj" action="welcome"/>
```


Podczas lektury rozdziału 4. będzie się można przekonać, że akcje nawigacji można przypisywać także do hiperłączy.

Wartość atrybutu `action` określa się mianem **wyniku** (ang. *outcome*). Wrócimy do tematu wyniku w punkcie „Odwzorowywanie wyników na identyfikatory widoków" w następnym podrozdziale — okazuje się, że wynik można opcjonalnie odwzorowywać na **identyfikatory widoków**. W specyfikacji JavaServer Faces mianem **widoku** (ang. *view*) określa się stronę JSF.

Jeśli programista nie zdefiniuje takiego odwzorowania dla określonego wyniku, wynik jest przekształcany w identyfikator widoku według następujących reguł:

1. Jeśli wynik nie obejmuje rozszerzenia pliku, dopisuje się do niego rozszerzenie bieżącego widoku.
2. Jeśli wynik nie rozpoczyna się od prawego ukośnika (*/*), poprzedza się go ścieżką do bieżącego widoku.

Na przykład wynik `welcome` w widoku */index.xhtml* zostanie automatycznie przekształcony w identyfikator widoku docelowego */welcome.xhtml*.


JSF 2.0 Odwzorowania wyników na identyfikatory widoków są opcjonalne, począwszy od wersji JSF 2.0. We wcześniejszych wersjach programista musiał wprost określać reguły nawigacji dla każdego wyniku.

Nawigacja dynamiczna

W przypadku większości aplikacji internetowych nawigacja nie ma charakteru statycznego. Ścieżka przechodzenia pomiędzy stronami nie zależy tylko od klikanych przycisków, ale także od danych wejściowych wpisywanych przez użytkownika. Na przykład wysyłając dane ze strony logowania (zwykle nazwę użytkownika i hasło), możemy oczekiwać dwóch odpowiedzi ze strony serwera: akceptacji tych danych bądź ich odrzucenia. Odpowiedź serwera zależy więc od pewnych obliczeń (w tym konkretnym przypadku od poprawności nazwy użytkownika i hasła).

Aby zaimplementować nawigację dynamiczną, przycisk akceptacji formularza należy związać z **wyrażeniem odwołującym się do metody**:

```
<h:commandButton label="Zaloguj" action="#{loginController.verifyUser}"/>
```

W naszym przypadku wyraz `loginController` odwołuje się do komponentu pewnej klasy, która musi definiować metodę nazwaną `verifyUser`.

Wyrażenie odwołujące się do metody, które przypisujemy atrybutowi `action`, nie otrzymuje żadnych parametrów, ale może zwracać wartość jakiegoś typu. Zwracana wartość jest konwertowana na łańcuch za pomocą wywołania metody `toString`.


W standardzie JSF 1.1 metody będące przedmiotem odwołań w tego rodzaju wyrażeniach musiały zwracać wartości typu `String`. W technologii JSF 1.2 zwracana wartość może być obiektem dowolnego typu. Taka możliwość jest szczególnie ważna w przypadku typów wyliczeniowych, ponieważ eliminuje ryzyko przeoczenia literówek w nazwach akcji (wychwytywanych przez kompilator).

Przykład metody akcji przedstawiono poniżej:

```
String verifyUser() {
   if (...)
      return "success";
   else
      return "failure";
}
```

Powyższa metoda zwraca jeden z dwóch łańcuchów wynikowych: `"success"` lub `"failure"`. Na podstawie tej wartości identyfikuje się następny widok.


Metoda akcji może zwrócić wartość `null`, aby zasygnalizować konieczność ponownego wyświetlenia tej samej strony. W takim przypadku zasięg widoku (który omówiliśmy w rozdziale 2.) jest zachowywany. Każdy wynik inny niż `null` powoduje wyczyszczenie tego zasięgu, nawet jeśli widok wynikowy jest taki sam jak widok bieżący.

Krótko mówiąc, za każdym razem, gdy użytkownik klika przycisk polecenia, dla którego zdefiniowano wyrażenie odwołujące się do metody w atrybucie `action`, implementacja JSF podejmuje następujące kroki:

1. uzyskuje dostęp do wskazanego komponentu;
2. wywołuje wskazaną metodę;
3. przekształca łańcuch wynikowy w identyfikator widoku;
4. wyświetla odpowiednią stronę (na podstawie identyfikatora widoku).

Oznacza to, że implementacja rozgałęzień (ang. *branching*) wymaga przekazania referencji do metody w klasie odpowiedniego komponentu. Mamy dużą dowolność w kwestii miejsca umieszczenia tej metody. Najlepszym rozwiązaniem jest znalezienie klasy obejmującej wszystkie dane potrzebne do podejmowania właściwych decyzji.

Odwzorowywanie wyników na identyfikatory widoków

Jednym z najważniejszych celów projektowych technologii JSF jest oddzielenie prezentacji od logiki biznesowej. Jeśli decyzje nawigacyjne mają charakter dynamiczny, kod obliczający wynik z natury rzeczy nie powinien być zobligowany do znajomości precyzyjnych nazw stron internetowych. Technologia JSF oferuje mechanizm odwzorowywania **wyników logicznych**, na przykład `success` i `failure`, na właściwe strony internetowe.

Takie odwzorowanie można zdefiniować, dodając wpisy `navigation-rule` do pliku *faces-config.xml*. Typowy przykład takiej konstrukcji pokazano poniżej:

```
<navigation-rule>
   <from-view-id>/index.xhtml</from-view-id>
   <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/welcome.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

Przytoczona reguła określa, że wynik `success` na stronie */index.xhtml* ma kierować użytkownika na stronę */welcome.xhtml*.

Łańcuchy identyfikatorów widoków rozpoczynają się od prawego ukośnika (*/*). Jeśli programista stosuje odwzorowania rozszerzeń (na przykład przyrostka *.faces*), rozszerzenia muszą odpowiadać rozszerzeniom plików (na przykład *.xhtml*), nie rozszerzeniom adresów URL.

Jeśli programista odpowiednio dobierze łańcuchy wyników, będzie mógł zebrać wiele reguł nawigacji w jednym miejscu. Programista może na przykład stworzyć jedną regułę dla wszystkich przycisków skojarzonych z akcją `logout` (dostępnych na wielu stronach danej aplikacji). Wszystkie te przyciski mogą powodować przejście na stronę *loggedOut.xhtml* — wystarczy zdefiniować jedną regułę:

```
<navigation-rule>
   <navigation-case>
      <from-outcome>logout</from-outcome>
      <to-view-id>/loggedOut.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

Ta reguła jest stosowana dla wszystkich stron, ponieważ nie zdefiniowano elementu `from-view-id`.

Istnieje też możliwość scalania reguł nawigacji z tym samym elementem `from-view-id`. Przykład takiego rozwiązania pokazano poniżej:

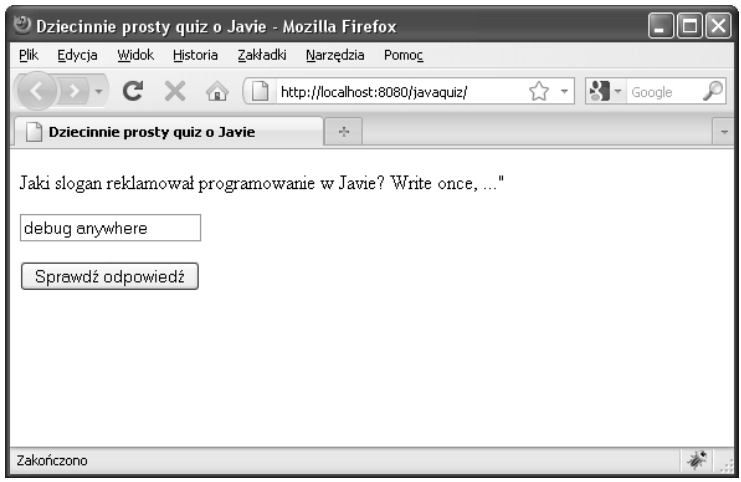
```
<navigation-rule>
   <from-view-id>/index.xhtml</from-view-id>
   <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/welcome.xhtml</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/newuser.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

W przypadku prostych aplikacji reguły nawigacji zwykle w ogóle nie są potrzebne. Kiedy jednak budowane aplikacje stają się coraz bardziej złożone, warto stosować wyniki logiczne w ramach komponentów zarządzanych oraz reguły nawigacji odwzorowujące te wyniki na widoki docelowe.

Aplikacja JavaQuiz

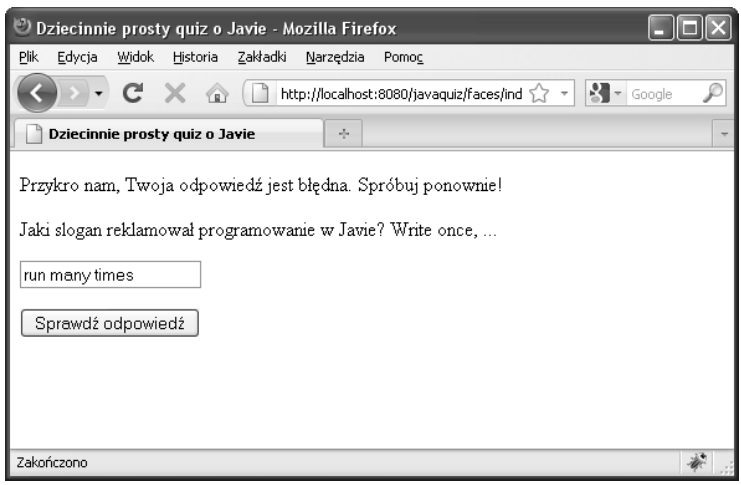
W tym punkcie umieścimy zapisy dotyczące nawigacji w przykładowym programie prezentującym użytkownikowi sekwencję pytań quizu (patrz rysunek 3.1).

Rysunek 3.1. *Pytanie quizu*



Kiedy użytkownik klika przycisk *Sprawdź odpowiedź*, aplikacja sprawdza, czy podana odpowiedź jest prawidłowa. Jeśli nie, użytkownik otrzymuje jeszcze jedną szansę rozwiązania tego samego problemu (patrz rysunek 3.2).

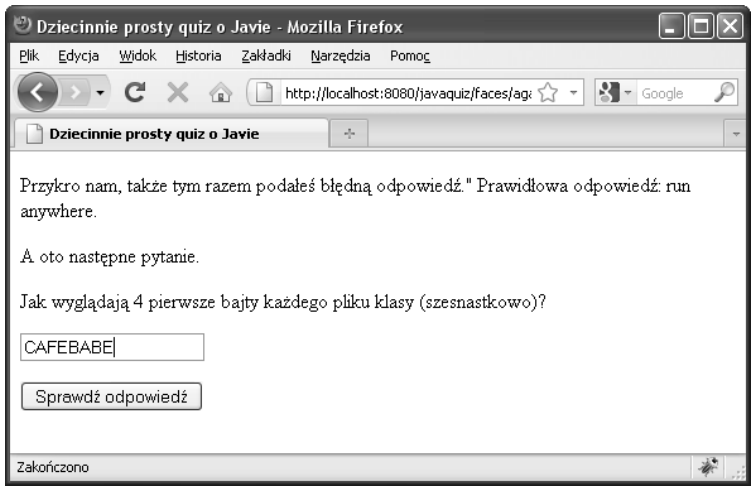
Rysunek 3.2. *Jedna błędna odpowiedź — spróbuj ponownie*



Po dwóch błędnych odpowiedziach aplikacja prezentuje użytkownikowi kolejny problem do rozwiązania (patrz rysunek 3.3).

Oczywiście także w razie podania prawidłowej odpowiedzi aplikacja przechodzi do następnego problemu. I wreszcie po rozwiązaniu ostatniego problemu następuje wyświetlenie strony podsumowania z uzyskaną liczbą punktów i propozycją ponownego przystąpienia do quizu (patrz rysunek 3.4).

Rysunek 3.3. *Dwie błędne odpowiedzi — kontynuuj*



Rysunek 3.4. *Quiz zakończony — punktacja*

Nasza aplikacja składa się z dwóch klas. Klasa `Problem` (przedstawiona na listingu 3.1) opisuje pojedynczy problem, czyli pytanie, odpowiedź oraz metodę weryfikacji, czy dana odpowiedź jest prawidłowa.

Listing 3.1. *Zawartość pliku javaquiz/src/java/com/corejsf/Problem.java*

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. public class Problem implements Serializable {
6.    private String question;
7.    private String answer;
8.
9.    public Problem(String question, String answer) {
10.      this.question = question;
11.      this.answer = answer;
12.   }
```

```
13.
14.    public String getQuestion() { return question; }
15.
16.    public String getAnswer() { return answer; }
17.
18.    // należy przykryć tę metodę bardziej wyszukanym kodem sprawdzającym
19.    public boolean isCorrect(String response) {
20.       return response.trim().equalsIgnoreCase(answer);
21.    }
22. }
```

Klasa `QuizBean` opisuje quiz obejmujący szereg pytań. Egzemplarz tej klasy dodatkowo śledzi bieżące pytanie i łączną punktację uzyskaną przez użytkownika. Kompletny kod tej klasy przedstawiono na listingu 3.2.

Listing 3.2. *Zawartość pliku javaquiz/src/java/com/corejsf/QuizBean.java*

```
 1. package com.corejsf;
 2.
 3. import java.io.Serializable;
 4.
 5. import java.util.ArrayList;
 6. import java.util.Arrays;
 7. import java.util.Collections;
 8.
 9. import javax.inject.Named;
10.    // lub import javax.faces.bean.ManagedBean;
11. import javax.enterprise.context.SessionScoped;
12.    // lub import javax.faces.bean.SessionScoped;
13.
14. @Named // lub @ManagedBean
15. @SessionScoped
16. public class QuizBean implements Serializable {
17.    private int currentProblem;
18.    private int tries;
19.    private int score;
20.    private String response = "";
21.    private String correctAnswer;
22.
23.    // Poniżej zakodowano problemy na stałe. W rzeczywistej aplikacji
24.    // najprawdopodobniej odczytywalibyśmy je z bazy danych.
25.    private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
26.    new Problem(
27.       "Jaki slogan reklamował programowanie w Javie? Write once, ...",
28.       "run anywhere"),
29.    new Problem(
30.       "Jak wyglądają 4 pierwsze bajty każdego pliku klasy (szesnastkowo)?",
31.       "CAFEBABE"),
32.    new Problem(
33.       "Co zostanie wyświetlone przez to wyrażenie? System.out.println(1+\"2\");",
34.       "12"),
35.    new Problem(
36.       "Które słowo kluczowe Javy służy do definiowania podklasy?",
37.       "extends"),
38.    new Problem(
39.       "Jak brzmiała oryginalna nazwa języka programowania Java?",
40.       "Oak"),
```

```
41.    new Problem(
42.       "Która klasa pakietu java.util opisuje punkt w czasie?",
43.       "Date")));
44.
45.    public String getQuestion() { return problems.get(currentProblem).getQuestion(); }
46.
47.    public String getAnswer() { return correctAnswer; }
48.
49.    public int getScore() { return score; }
50.
51.    public String getResponse() { return response; }
52.    public void setResponse(String newValue) { response = newValue; }
53.
54.    public String answerAction() {
55.       tries++;
56.       if (problems.get(currentProblem).isCorrect(response)) {
57.          score++;
58.          nextProblem();
59.          if (currentProblem == problems.size()) return "done";
60.          else return "success";
61.       }
62.       else if (tries == 1) return "again";
63.       else {
64.          nextProblem();
65.          if (currentProblem == problems.size()) return "done";
66.          else return "failure";
67.       }
68.    }
69.
70.    public String startOverAction() {
71.       Collections.shuffle(problems);
72.       currentProblem = 0;
73.       score = 0;
74.       tries = 0;
75.       response = "";
76.       return "startOver";
77.    }
78.
79.    private void nextProblem() {
80.       correctAnswer = problems.get(currentProblem).getAnswer();
81.       currentProblem++;
82.       tries = 0;
83.       response = "";
84.    }
85. }
```

W analizowanym przykładzie właśnie klasa `QuizBean` jest właściwym miejscem dla metod odpowiedzialnych za nawigację. Wspomniany komponent dysponuje pełną wiedzą o działaniach użytkownika i może bez trudu określić, która strona powinna być wyświetlona jako następna.

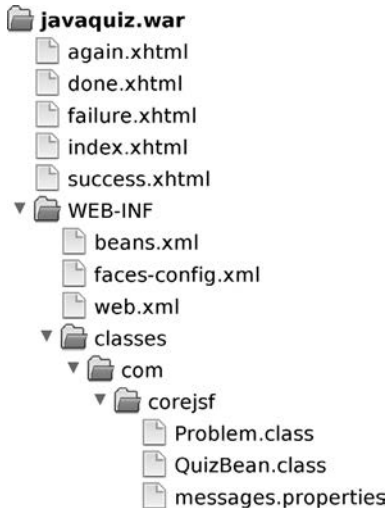
Logikę nawigacji zaimplementowano w metodzie `answerAction` klasy `QuizBean`. Metoda `answerAction` zwraca jeden z kilku możliwych łańcuchów: `"success"` lub `"done"` (jeśli użytkownik prawidłowo odpowiedział na pytanie), `"again"` (jeśli użytkownik po raz pierwszy udzielił błędnej odpowiedzi) oraz `"failure"` lub `"done"` (jeśli po raz drugi padła zła odpowiedź).

```
public String answerAction() {
   tries++;
   if (problems.get(currentProblem).isCorrect(response)) {
      score++;
      nextProblem();
      if (currentProblem == problems.size()) return "done";
      else return "success";
   }
   else if (tries == 1) return "again";
   else {
      nextProblem();
      if (currentProblem == problems.size()) return "done";
      else return "failure";
   }
}
```

Z przyciskami na każdej z tych stron wiążemy wyrażenie odwołujące się do metody `answer➥Action`. Na przykład strona *index.xhtml* zawiera następujący element:

```
<h:commandButton value="#{msgs.checkAnswer}" action="#{quizBean.answerAction}"/>
```

Na rysunku 3.5 przedstawiono strukturę katalogów naszej aplikacji. Na listingu 3.3 przedstawiono kod strony głównej aplikacji quizu: *index.xhtml*. Kodem stron *success.xhtml* i *failure.xhtml* nie będziemy się zajmować, ponieważ różni się od kodu strony *index.xhtml* tylko komunikatem wyświetlanym w górnej części.



Rysunek 3.5. *Struktura katalogów aplikacji quizu o Javie*

Listing 3.3. *Kod strony javaquiz/web/index.xhtml*

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3.    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml"
5.       xmlns:h="http://java.sun.com/jsf/html">
6.    <h:head>
7.       <title>#{msgs.title}</title>
8.    </h:head>
```

```
 9.    <h:body>
10.       <h:form>
11.          <p>#{quizBean.question}</p>
12.          <p><h:inputText value="#{quizBean.response}"/></p>
13.          <p>
14.             <h:commandButton value="#{msgs.checkAnswer}"
15.                              action="#{quizBean.answerAction}"/>
16.          </p>
17.       </h:form>
18.    </h:body>
19. </html>
```

Strona *done.xhtml*, której kod przedstawiono na listingu 3.4, prezentuje użytkownikowi ostateczny wynik i zachęca go do ponownej gry. Warto zwrócić uwagę na jedyny przycisk dostępny na tej stronie. Na pierwszy rzut oka może się wydawać, że mamy do czynienia z nawigacją statyczną, ponieważ każde kliknięcie przycisku *Rozpocznij od nowa* powoduje powrót na stronę *index.xhtml*. Okazuje się jednak, że element definiujący ten przycisk wykorzystuje wyrażenie odwołujące się do metody:

```
<h:commandButton value="#{msgs.startOver}" action="#{quizBean.startOverAction}"/>
```

Listing 3.4. *Zawartość pliku javaquiz/web/done.xhtml*

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3.    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4. <html xmlns="http://www.w3.org/1999/xhtml"
 5.       xmlns:f="http://java.sun.com/jsf/core"
 6.       xmlns:h="http://java.sun.com/jsf/html">
 7.    <h:head>
 8.       <title>#{msgs.title}</title>
 9.    </h:head>
10.    <h:body>
11.       <h:form>
12.          <p>
13.             #{msgs.thankYou}
14.             <h:outputFormat value="#{msgs.score}">
15.                <f:param value="#{quizBean.score}"/>
16.             </h:outputFormat>
17.          </p>
18.          <p>
19.             <h:commandButton value="#{msgs.startOver}"
20.                              action="#{quizBean.startOverAction}"/>
21.          </p>
22.       </h:form>
23.    </h:body>
24. </html>
```

Metoda `startOverAction` wykonuje wszelkie działania niezbędne do przywrócenia oryginalnego stanu gry. Działanie tej metody polega na przypadkowym uporządkowaniu pytań i wyzerowaniu wyniku:

```
public String startOverAction() {
   Collections.shuffle(problems);
   currentProblem = 0;
```

```
        score = 0;
        tries = 0;
        response = "";
        return "startOver";
    }
```

Ogólnie metody akcji pełnią dwie funkcje:

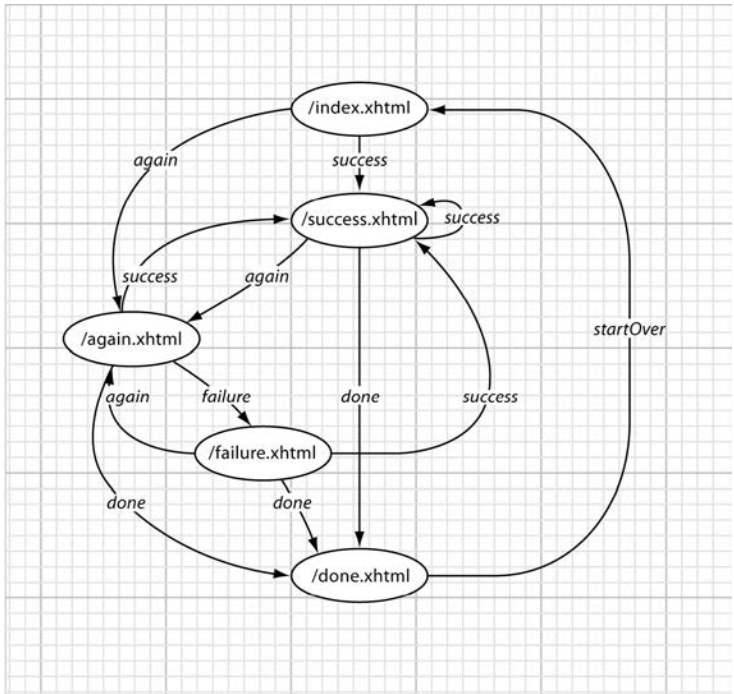
- aktualizują model w reakcji na działania podejmowane przez użytkownika;
- sygnalizują mechanizmowi nawigacji konieczność skierowania użytkownika w określone miejsce.

Podczas lektury rozdziału 8. Czytelnicy przekonają się, że istnieje możliwość dołączania do przycisków także obiektów nasłuchujących akcji. Kiedy użytkownik klika tak zdefiniowany przycisk, następuje wykonanie kodu metody `processAction` obiektu nasłuchującego. Warto jednak pamiętać, że obiekt nasłuchujący akcji nie komunikuje się z mechanizmem odpowiedzialnym za nawigację.

Na listingu 3.5 przedstawiono plik konfiguracyjny tej aplikacji obejmujący między innymi reguły nawigacji. Aby lepiej zrozumieć te reguły, warto rzucić okiem na diagram przechodzenia pomiędzy stronami, pokazany na rysunku 3.6.

Listing 3.5. *Zawartość pliku javaquiz/web/WEB-INF/faces-config.xml*

```
 1. <?xml version="1.0"?>
 2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
 3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 5.       http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
 6.    version="2.0">
 7.    <navigation-rule>
 8.       <navigation-case>
 9.          <from-outcome>startOver</from-outcome>
10.          <to-view-id>/index.xhtml</to-view-id>
11.       </navigation-case>
12.    </navigation-rule>
13.    <navigation-rule>
14.       <from-view-id>/again.xhtml</from-view-id>
15.       <navigation-case>
16.          <from-outcome>failure</from-outcome>
17.          <to-view-id>/failure.xhtml</to-view-id>
18.       </navigation-case>
19.    </navigation-rule>
20.    <navigation-rule>
21.       <navigation-case>
22.          <from-outcome>failure</from-outcome>
23.          <to-view-id>/again.xhtml</to-view-id>
24.       </navigation-case>
25.    </navigation-rule>
26.
27.    <application>
28.       <resource-bundle>
29.          <base-name>com.corejsf.messages</base-name>
30.          <var>msgs</var>
```



Rysunek 3.6. *Diagram przejść aplikacji quizu o Javie*

```
31.       </resource-bundle>
32.    </application>
33. </faces-config>
```

Dla trzech spośród naszych wyników (`"success"`, `"again"` i `"done"`) nie zdefiniowano reguł nawigacji. Wymienione wyniki zawsze kierują użytkownika odpowiednio na strony */success.xhtml*, */again.xhtml* oraz */done.xhtml*. Wynik `"startOver"` odwzorowujemy na stronę */index.xhtml*. Nieco trudniejsza jest obsługa wyniku `failure`, który początkowo kieruje użytkownika na stronę */again.xhtml*, stwarzając mu drugą okazję do udzielenia odpowiedzi. Jeśli jednak także odpowiedź wpisana na tej stronie okazuje się błędna, ten sam wynik kieruje użytkownika na stronę */failure.xhtml*:

```
<navigation-rule>
   <from-view-id>/again.xhtml</from-view-id>
   <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/failure.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
<navigation-rule>
   <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/again.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

Warto pamiętać o tym, że kolejność reguł nie jest bez znaczenia. Druga reguła jest uwzględniana w sytuacji, gdy bieżącą stroną nie jest */again.xhtml*.

I wreszcie na listingu 3.6 pokazano łańcuchy komunikatów.

Listing 3.6. *Zawartość pliku javaquiz/src/java/com/corejsf/messages.properties*

```
1. title=Dziecinnie prosty quiz o Javie
2. checkAnswer=Sprawdź odpowiedź
3. startOver=Rozpocznij od nowa
4. correct=Gratulacje, prawidłowa odpowiedź.
5. notCorrect=Przykro nam, Twoja odpowiedź jest błędna. Spróbuj ponownie!
6. stillNotCorrect=Przykro nam, także tym razem podałeś błędną odpowiedź.
7. correctAnswer=Prawidłowa odpowiedź: {0}.
8. score=Twój wynik punktowy: {0}.
9. nextProblem=A oto następne pytanie.
10. thankYou=Dziękujemy za udział w quizie.
```

Przekierowania

Programista może wymusić na implementacji technologii JavaServer Faces **przekierowanie** (ang. *redirection*) użytkownika do nowego widoku. Implementacja JSF wysyła następnie przekierowanie protokołu HTTP do klienta. Odpowiedź przekierowania wskazuje klientowi adres URL kolejnej strony — na tej podstawie klient generuje żądanie `GET` na odpowiedni adres URL.

Przekierowywanie jest czasochłonne, ponieważ wymaga powtórzenia całego cyklu komunikacji z udziałem przeglądarki. Przekierowanie ma jednak tę zaletę, że stwarza przeglądarce możliwość aktualizacji jej pola adresu.

Na rysunku 3.7 pokazano, jak zmienia się zawartość pola adresu wskutek użycia mechanizmu przekierowania.

Bez przekierowania oryginalny adres URL (*localhost:8080/javaquiz/faces/index.xhtml*) pozostałby niezmieniony przy okazji przejścia użytkownika ze strony */index.xhtml* na stronę */success.xhtml*. Przekierowanie powoduje, że przeglądarka wyświetla nowy adres URL (*localhost:8080/javaquiz/faces/success.xhtml*).

Jeśli nie stosujemy reguł nawigacji, powinniśmy uzupełnić łańcuch wyniku o następujący zapis:

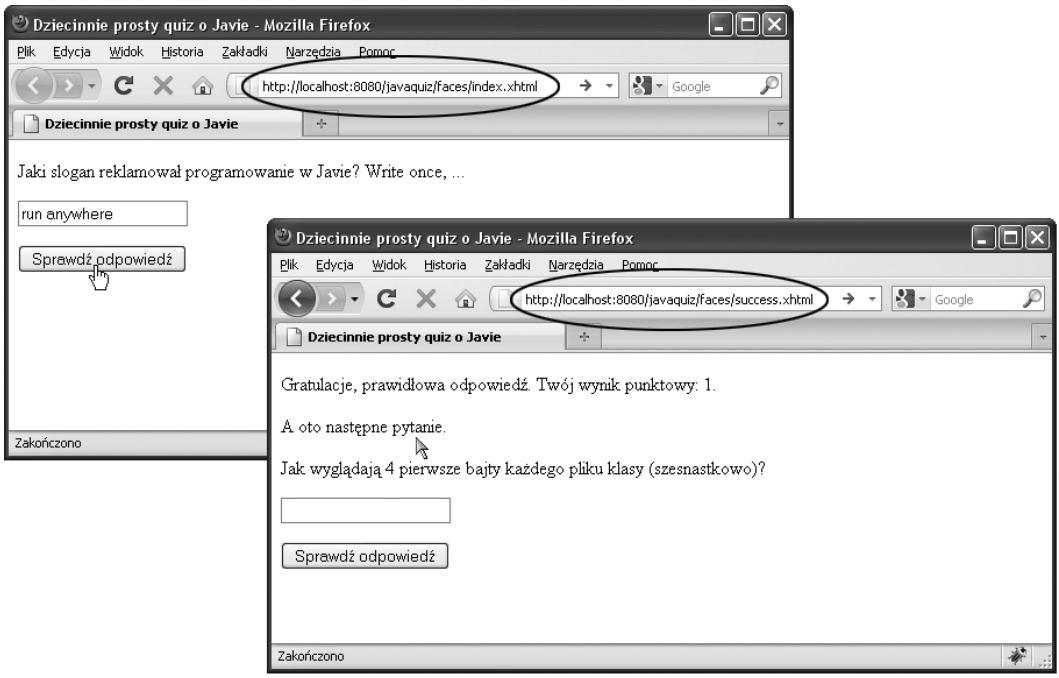
```
?faces-redirect=true
```

Po wprowadzeniu tej zmiany przykładowy łańcuch wyniku może mieć następującą postać:

```
<h:commandButton label="Zaloguj" action="welcome?faces-redirect=true"/>
```

W regule nawigacji należy umieścić element `redirect` bezpośrednio za elementem `to-view-id`:

```
<navigation-case>
   <from-outcome>success</from-outcome>
   <to-view-id>/success.xhtml</to-view-id>
   <redirect/>
</navigation-case>
```



Rysunek 3.7. *Przekierowanie aktualizujące adres URL w przeglądarce*

Przekierowanie i obiekt flash JSF 2.0

Aby zminimalizować rozmiary zasięgu sesji, warto możliwie często rezygnować z tego zasięgu na rzecz zasięgu żądania. Bez elementu `redirect` istnieje możliwość stosowania komponentów o zasięgu żądania dla danych prezentowanych w następnym widoku.

Przeanalizujmy teraz, jak będzie działała nasza aplikacja w przypadku użycia elementu `redirect`:

1. Klient wysyła żądanie na serwer.
2. Mapa zasięgu żądania jest wypełniana komponentami o zasięgu żądania.
3. Serwer wysyła do klienta status HTTP 302 (przekierowanie tymczasowe) wraz z miejscem docelowym przekierowania. Ten krok kończy bieżące żądanie, zatem komponenty o zasięgu żądania są usuwane.
4. Klient generuje żądanie `GET` i wysyła pod nowy adres.
5. Serwer wizualizuje następny widok. Okazuje się jednak, że komponenty należące do zasięgu poprzedniego żądania nie są już dostępne.

Aby obejść ten problem, technologia JSF 2.0 oferuje obiekt **flash**, który można wypełnić podczas obsługi jednego żądania i użyć w ramach kolejnego żądania. (Koncepcję obiektu flash

zaczerpnięto z frameworku internetowego Ruby on Rails). Typowym zastosowaniem obiektów flash są komunikaty. Na przykład metoda obsługująca przycisk może umieścić jakiś komunikat właśnie w obiekcie flash:

```
ExternalContext.getFlash().put("message", "Twoje hasło niedługo straci ważność");
```

Metoda `ExternalContext.getFlash()` zwraca obiekt klasy `Flash` implementującej interfejs `Map<String, Object>`.

W kodzie strony JSF można odwołać się do obiektu flash za pomocą zmiennej `flash`. Na przykład powyższy komunikat można wyświetlić, stosując konstrukcję:

```
#{flash.message}
```

Po wizualizacji komunikatu i dostarczeniu przekierowania do klienta łańcuch komunikatu jest automatycznie usuwany z obiektu flash.

Okazuje się, że wartość w obiekcie flash można utrzymywać dłużej niż podczas przetwarzania jednego żądania. Na przykład wyrażenie w postaci:

```
#{flash.keep.message}
```

nie tylko zwraca wartość klucza `message` przechowywaną w obiekcie flash, ale też dodaje tę wartość ponownie z myślą o kolejnym cyklu żądania.

Jeśli z czasem ilość danych przerzucanych pomiędzy obiektem flash a komponentem staje się naprawdę duża, warto rozważyć użycie zasięgu konwersacji.

Nawigacja zgodna ze stylem REST i adresy URL zapewniające możliwość stosowania zakładek JSF 2.0

Aplikacja JSF domyślnie generuje sekwencję żądań `POST` wysyłanych na serwer. Każde żądanie `POST` zawiera dane formularza. Takie rozwiązanie jest uzasadnione w przypadku aplikacji gromadzących dużą ilość danych wejściowych wpisywanych przez użytkownika. Okazuje się jednak, że znaczna część aplikacji internetowych działa w zupełnie inny sposób. Wyobraźmy sobie na przykład sytuację, w której użytkownik przegląda katalog sklepu internetowego, klikając łącza do kolejnych produktów. Poza wyborem klikanych łączy trudno w tym przypadku mówić o jakichkolwiek danych wejściowych użytkownika. Łącza powinny zapewniać **możliwość stosowania zakładek** (ang. *bookmarkable*), aby użytkownik mógł wrócić na tę samą stronę po wpisaniu tego samego adresu URL. Co więcej, strony powinny być **przystosowane do przechowywania w pamięci podręcznej** (ang. *cacheable*). Przechowywanie stron w pamięci podręcznej jest ważnym czynnikiem decydującym o efektywności aplikacji internetowych. Stosowanie zakładek ani przechowywanie w pamięci podręcznej oczywiście nie jest możliwe w przypadku żądań `POST`.

Zgodnie z zaleceniami stylu architektury nazwanego REST (od ang. *Representational State Transfer*) aplikacje internetowe powinny posługiwać się protokołem HTTP według oryginalnych

reguł przyjętych przez jego twórców. Na potrzeby operacji wyszukiwania należy stosować żądania GET. Żądania PUT, POST i DELETE powinny być wykorzystywane odpowiednio do tworzenia, modyfikowania i usuwania.

Zwolennicy stylu REST najbardziej cenią sobie adresy URL w następującej formie:

```
http://serwer.pl/catalog/item/1729
```

Architektura REST nie narzuca nam jednak jednego stylu. Na przykład dla żądania GET z parametrem można by użyć następującego adresu:

```
http://serwer.pl/catalog?item=1729
```

Także ten adres jest w pełni zgodny ze stylem REST.

Należy pamiętać, że żądań GET nigdy nie powinno się używać do aktualizowania informacji. Na przykład żądanie GET dodające element do koszyka w tej formie:

```
http://serwer.pl/addToCart?cart=314159&item=1729
```

nie byłoby właściwe. Żądania GET powinny być **idempotentne**. Oznacza to, że dwukrotne użycie tego samego żądania nie powinno prowadzić do rezultatów innych niż jednorazowe żądanie. Właśnie od tego zależy możliwość przechowywania żądań w pamięci podręcznej. Żądanie *dodaj do koszyka* z natury rzeczy nie jest idempotentne — jego dwukrotne wysłanie spowoduje dodanie dwóch takich samych towarów do koszyka. W tym kontekście dużo odpowiedniejsze byłoby żądanie POST. Oznacza to, że nawet aplikacje internetowe zgodne ze stylem REST muszą korzystać z żądań POST.

Technologia JSF nie oferuje obecnie standardowego mechanizmu generowania ani przetwarzania tzw. **dobrych adresów URL**, jednak począwszy od wersji JSF 2.0, możemy liczyć na obsługę żądań GET. Omówimy ten aspekt w kolejnych punktach tego podrozdziału.

Parametry widoku

Przeanalizujmy teraz żądanie GET, które ma spowodować wyświetlenie informacji o konkretnym produkcie:

```
http://serwer.pl/catalog?item=1729
```

Identyfikator produktu jest przekazywany w formie parametru zapytania. Po otrzymaniu żądania wartość tego parametru musi zostać przekazana do właściwego komponentu. Programista może użyć do tego celu **parametrów widoku**.

Na początku strony należy dodać znaczniki podobne do poniższych:

```
<f:metadata>
   <f:viewParam name="item" value="#{catalog.currentItem}"/>
</f:metadata>
```

W czasie przetwarzania żądania wartość parametru zapytania item jest przekazywana do metody setCurrentItem komponentu catalog.

Strona JSF może zawierać dowolną liczbę parametrów widoku. Jak wszystkie parametry żądania, parametry widoku można konwertować i sprawdzać pod kątem poprawności. (Zagadnienia związane z konwersją i weryfikacją zostaną szczegółowo omówione w rozdziale 7.).

Często niezbędne jest uzyskiwanie dodatkowych danych już po ustawieniu parametrów widoku. Na przykład po ustawieniu parametru widoku `item` może zaistnieć potrzeba uzyskania właściwości tego produktu z bazy danych, aby wygenerować stronę opisującą ten produkt. W rozdziale 8. omówimy sposób implementacji tego rodzaju mechanizmów w ramach metody obsługującej zdarzenie `preRenderView`.

Łącza żądań GET

W poprzednim punkcie omówiliśmy sposób przetwarzania przez implementację technologii JSF żądania `GET`. Aplikacje zgodne z architekturą REST powinny zapewniać użytkownikom możliwość nawigacji przy użyciu tego rodzaju żądań. Oznacza to, że programista powinien umieszczać na swoich stronach przyciski i łącza, których klikanie generuje właśnie żądania `GET`. Przyciski i łącza tego typu można definiować odpowiednio za pomocą znaczników `h:button` i `h:link`. (Do definiowania przycisków i łączy generujących żądania `POST` służą odpowiednio znaczniki `h:commandButton` i `h:commandLink`).

Programista powinien mieć kontrolę nad identyfikatorami widoków docelowych i parametrami zapytań tego rodzaju żądań. Identyfikator widoku docelowego można określić za pośrednictwem atrybutu `outcome`. Wartość tego atrybutu może mieć albo postać stałego łańcucha:

```
<h:button value="Gotowe" outcome="done"/>
```

albo **wyrażenia reprezentującego wartość**:

```
<h:button value="Pomiń" outcome="#{quizBean.skipOutcome}"/>
```

Drugi zapis powoduje wywołanie metody `getSkipOutcome`. Metoda ta musi zwrócić łańcuch wyniku, który trafia do mechanizmu obsługi nawigacji, gdzie jest przetwarzany w tradycyjny sposób w celu określenia identyfikatora widoku docelowego.

Istnieje zasadnicza różnica dzieląca atrybut `outcome` znacznika `h:button` od atrybutu `action` znacznika `h:commandButton`. Atrybut `outcome` jest przetwarzany przed wizualizacją strony, zatem odpowiednie łącze może być osadzone w kodzie tej strony. Atrybut `action` jest natomiast przetwarzany dopiero po kliknięciu przycisku polecenia przez użytkownika. Właśnie dlatego w specyfikacji JSF można spotkać termin **nawigacji wyprzedzającej** (ang. *preemptive navigation*) stosowany w kontekście wyznaczania identyfikatorów widoków docelowych dla żądań `GET`.

Wyrażenie języka wyrażeń (EL) przypisane atrybutowi `outcome` jest wyrażeniem reprezentującym wartość, **nie** wyrażeniem odwołującym się do metody. Ogólnie, akcja skojarzona z przyciskiem polecenia może w ten czy inny sposób zmieniać stan aplikacji. Należy jednak pamiętać, że przetwarzanie wyniku łącza żądania `GET` nie powinno skutkować żadnymi zmianami — takie łącze jest przecież umieszczane na stronie z myślą o potencjalnym wykorzystaniu w przyszłości.

Określanie parametrów żądania

Często stajemy przed koniecznością dołączania parametrów do łączy żądań GET. Takie parametry mogą pochodzić z trzech różnych źródeł:

- łańcucha wyniku,
- parametrów widoku,
- zagnieżdżonych znaczników f:param.

Jeśli ten sam parametr zostanie określony więcej niż raz, pierwszeństwo ma wartość zdefiniowana przez późniejsze źródło z poniższej listy (najwyższy priorytet mają więc zagnieżdżone znaczniki f:param).

Przeanalizujmy teraz szczegóły każdego z tych rozwiązań.

Parametry można określać w ramach łańcucha wyniku, na przykład:

```
<h:link outcome="index?q=1" value="Skip">
```

Mechanizm obsługujący nawigację wyodrębnia parametry z łańcucha wyniku, określa identyfikator widoku docelowego i dopisuje wyodrębnione wcześniej parametry do tego identyfikatora. Oznacza to, że w tym przypadku identyfikatorem widoku docelowego będzie */index.xhtml?q=1*.

W razie zdefiniowania wielu parametrów koniecznie należy zastąpić separator & odpowiednią sekwencją specjalną (ucieczki):

```
<h:link outcome="index?q=1&amp;score=0" value="Skip">
```

W łańcuchu wyniku oczywiście zawsze można użyć wyrażenia reprezentującego wartość, na przykład:

```
<h:link outcome="index?q=#{quizBean.currentProblem + 1}" value="Skip">
```

Istnieje jeszcze wygodny zapis skrócony umożliwiający włączenie wszystkich parametrów widoku do jednego łańcucha zapytania. Wystarczy dodać do znacznika h:link następujący atrybut:

```
<h:link outcome="index" includeViewParams="true" value="Skip">
```

W ten sposób można łatwo przenosić wszystkie parametry widoku z jednej strony na drugą, co jest dość typowym wymaganiem stawianym aplikacjom zgodnym z architekturą REST.

Parametry widoku można nadpisać, stosując znacznik f:param. Przykład takiego rozwiązania pokazano poniżej:

```
<h:link outcome="index" includeViewParams="true" value="Skip">
   <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
</h:link>
```

Możliwość włączania parametrów widoku znacznie ułatwia także definiowanie łączy przekierowań, które także mają postać żądań GET. Zamiast jednak ustawiać atrybut w znaczniku, należy dodać odpowiedni parametr do wyniku:

```
<h:commandLink action="index?faces-redirect=true&amp;includeViewParams=true"
   value="Skip"/>
```

Żądanie w tej formie **nie** uwzględnia jednak zagnieżdżonych znaczników f:param.

Jeśli reguły nawigacji są definiowane w pliku konfiguracyjnym w formacie XML, należy użyć atrybutu include-viewparams i zagnieżdżonych znaczników view-param, na przykład:

```
<redirect include-view-params=true>
   <view-param>
      <name>q</name>
      <value>#{quizBean.currentProblem + 1}</value>
   </view-param>
</redirect>
```

Nie należy przywiązywać zbyt dużej wagi do drobnych niekonsekwencji widocznych w powyższej konstrukcji składniowej. Czytelnik powinien je traktować raczej jako środek do podniesienia poziomu czujności i — tym samym — udoskonalenia swoich umiejętności programistycznych.

Dodanie łączy umożliwiających stosowanie zakładek do aplikacji quizu

Wróćmy na chwilę do aplikacji quizu, którą omówiliśmy we wcześniejszej części tego rozdziału (przy okazji demonstrowania zasad nawigacji). Czy można tę aplikację w większym stopniu dostosować do wymogów architektury REST?

Żądanie GET nie byłoby właściwym sposobem wysyłania odpowiedzi, ponieważ w tym kontekście nie spełnia warunku idempotentności. Wysłanie odpowiedzi modyfikuje wynik użytkownika. Okazuje się jednak, że możemy dodać do naszej aplikacji zgodne z architekturą REST łącza do nawigowania pomiędzy pytaniami.

Aby zachować prostotę naszej aplikacji, umieszczamy na stronie pojedyncze łącze umożliwiające stosowanie zakładek i powodujące przejście do następnego pytania. Używamy do tego celu parametru widoku:

```
<f:metadata>
   <f:viewParam name="q" value="#{quizBean.currentProblem}"/>
</f:metadata>
```

Samo łącze definiujemy w następujący sposób:

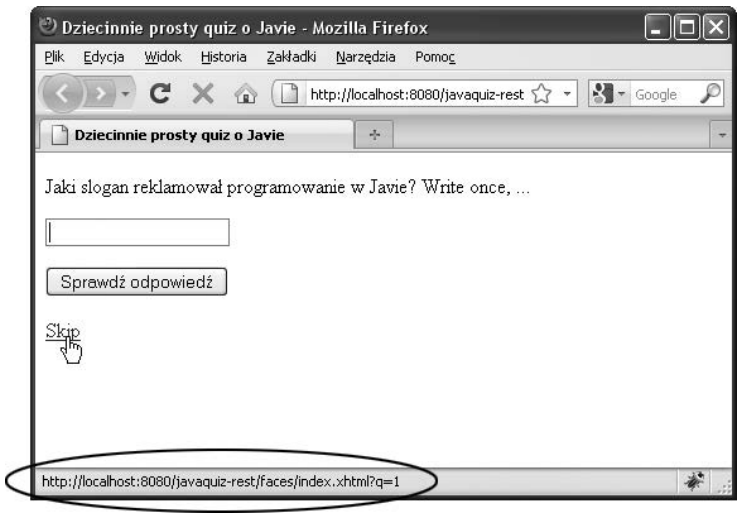
```
<h:link outcome="#{quizBean.skipOutcome}" value="Skip">
   <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
</h:link>
```

Metoda getSkipOutcome klasy QuizBean zwraca łańcuch "index" lub "done" w zależności od tego, czy są dostępne jeszcze jakieś pytania:

```
public String getSkipOutcome() {
   if (currentProblem < problems.size() - 1) return "index";
   else return "done";
}
```

Tak zdefiniowane łącze będzie miało następującą postać (patrz rysunek 3.8):

```
http://localhost:8080/javaquiz-rest/faces/index.xhtml?q=1
```



Rysunek 3.8. *Przykład łącza zgodnego z architekturą REST*

Użytkownik może dodać to łącze do zakładek i wrócić do dowolnego pytania quizu.

Na listingu 3.7 pokazano kod strony *index.xhtml* zawierający parametr widoku i znacznik `h:link`. Na listingu 3.8 pokazano zmodyfikowany kod komponentu `QuizBean`. Uzupełniliśmy ten komponent o metodę `setCurrentProblem` i zmodyfikowaliśmy mechanizm obliczania wyniku punktowego. Ponieważ aplikacja oferuje teraz możliwość wielokrotnego odwiedzenia strony tego samego pytania, musimy mieć pewność, że użytkownik nie otrzymuje punktów za więcej niż jedną odpowiedź na to samo pytanie.

Listing 3.7. *Zawartość pliku javaquiz-rest/web/index.xhtml*

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4. <html xmlns="http://www.w3.org/1999/xhtml"
 5.       xmlns:f="http://java.sun.com/jsf/core"
 6.       xmlns:h="http://java.sun.com/jsf/html">
 7.    <f:metadata>
 8.       <f:viewParam name="q" value="#{quizBean.currentProblem}"/>
 9.    </f:metadata>
10.    <h:head>
11.       <title>#{msgs.title}</title>
12.    </h:head>
13.    <h:body>
14.       <h:form>
15.          <p>#{quizBean.question}</p>
16.          <p><h:inputText value="#{quizBean.response}"/></p>
```

```
17.          <p><h:commandButton value="#{msgs.checkAnswer}"
18.             action="#{quizBean.answerAction}"/></p>
19.          <p><h:link outcome="#{quizBean.skipOutcome}" value="Skip">
20.             <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
21.          </h:link>
22.          </p>
23.       </h:form>
24.    </h:body>
25. </html>
```

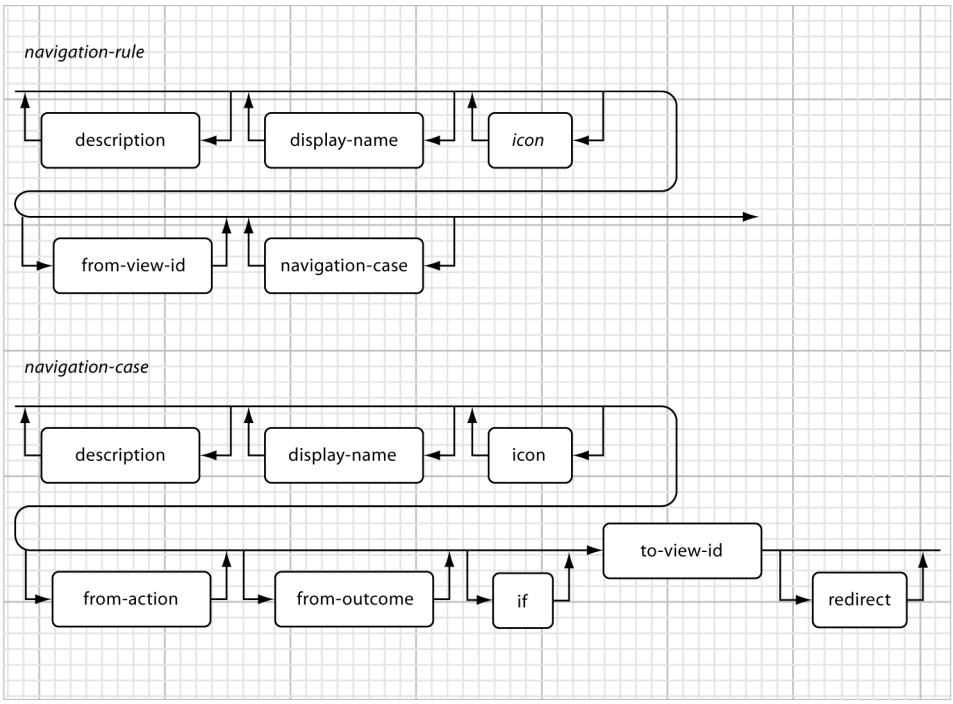
Listing 3.8. *Zawartość pliku javaquiz-rest/src/java/com/corejsf/QuizBean.java*

```
 1. package com.corejsf;
 2.
 3. import java.io.Serializable;
 4.
 5. import java.util.ArrayList;
 6. import java.util.Arrays;
 7. import java.util.Collections;
 8. import javax.inject.Named;
 9.    // lub import javax.faces.bean.ManagedBean;
10. import javax.enterprise.context.SessionScoped;
11.    // lub import javax.faces.bean.SessionScoped;
12.
13. @Named // lub @ManagedBean
14. @SessionScoped
15. public class QuizBean implements Serializable {
16.    private int currentProblem;
17.    private int tries;
18.    private String response = "";
19.    private String correctAnswer;
20.
21.    // Poniżej zakodowano problemy na stałe. W rzeczywistej aplikacji
22.    // najprawdopodobniej odczytywalibyśmy je z bazy danych.
23.    private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
24.       new Problem(
25.       "Jaki slogan reklamował programowanie w Javie? Write once, ...",
26.       "run anywhere"),
27.    new Problem(
28.       "Jak wyglądają 4 pierwsze bajty każdego pliku klasy (szesnastkowo)?",
29.       "CAFEBABE"),
30.    new Problem(
31.       "Co zostanie wyświetlone przez to wyrażenie? System.out.println(1+\"2\");",
32.       "12"),
33.    new Problem(
34.       "Które słowo kluczowe Javy służy do definiowania podklasy?",
35.       "extends"),
36.    new Problem(
37.       "Jak brzmiała oryginalna nazwa języka programowania Java?",
38.       "Oak"),
39.    new Problem(
40.       "Która klasa pakietu java.util opisuje punkt w czasie?",
41.       "Date")));
42.
43.    private int[] scores = new int[problems.size()];
44.
45.    public String getQuestion() {
46.       return problems.get(currentProblem).getQuestion();
```

```
47.    }
48.
49.    public String getAnswer() { return correctAnswer; }
50.
51.    public int getScore() {
52.       int score = 0;
53.       for (int s : scores) score += s;
54.       return score;
55.    }
56.
57.    public String getResponse() { return response; }
58.    public void setResponse(String newValue) { response = newValue; }
59.
60.    public int getCurrentProblem() { return currentProblem; }
61.    public void setCurrentProblem(int newValue) { currentProblem = newValue; }
62.
63.    public String getSkipOutcome() {
64.       if (currentProblem < problems.size() - 1) return "index";
65.       else return "done";
66.    }
67.
68.    public String answerAction() {
69.       tries++;
70.       if (problems.get(currentProblem).isCorrect(response)) {
71.          scores[currentProblem] = 1;
72.          nextProblem();
73.          if (currentProblem == problems.size()) return "done";
74.          else return "success";
75.       }
76.       else {
77.          scores[currentProblem] = 0;
78.          if (tries == 1) return "again";
79.          else {
80.             nextProblem();
81.             if (currentProblem == problems.size()) return "done";
82.             else return "failure";
83.          }
84.       }
85.    }
86.
87.    public String startOverAction() {
88.       Collections.shuffle(problems);
89.       currentProblem = 0;
90.       for (int i = 0; i < scores.length; i++)
91.          scores[i] = 0;
92.       tries = 0;
93.       response = "";
94.       return "startOver";
95.    }
96.
97.    private void nextProblem() {
98.       correctAnswer = problems.get(currentProblem).getAnswer();
99.       currentProblem++;
100.      tries = 0;
101.      response = "";
102.   }
103. }
```

Zaawansowane techniki nawigacji

Techniki opisane w poprzednich podrozdziałach powinny w zupełności wystarczyć do implementacji mechanizmów nawigacji w większości aplikacji. W niniejszym podrozdziale skoncentrujemy się na pozostałych regułach, które można konfigurować za pośrednictwem elementów nawigacji w pliku *faces-config.xml*. Diagram składni tych elementów przedstawiono na rysunku 3.9.



Rysunek 3.9. *Diagram składni elementów nawigacji*

Z lektury podrozdziału „Konfigurowanie komponentów" w rozdziale 2. wiemy, że informacje o nawigacji można umieszczać także w innych plikach konfiguracyjnych niż plik standardowy *faces-config.xml*.

Z diagramu przedstawionego na rysunku 3.9 wynika, że każdy element `navigation-rule` i `navigation-case` może obejmować dowolną liczbę elementów `description`, `display-name` i `icon`. Wymienione elementy w założeniu mają być stosowane na poziomie narzędzi do wizualnego projektowania interfejsów, zatem nie będziemy ich poddawać szczegółowej analizie.

Symbole wieloznaczne

W elementach `from-view-id` wykorzystywanych w ramach reguł nawigacji można stosować **symbole wieloznaczne**. Poniżej przedstawiono odpowiedni przykład:

```
<navigation-rule>
   <from-view-id>/secure/*</from-view-id>
   <navigation-case>
      ...
   </navigation-case>
</navigation-rule>
```

Tak zdefiniowana reguła będzie stosowana dla wszystkich stron rozpoczynających się od przedrostka */secure/*. Dopuszczalny jest tylko jeden znak gwiazdki (*), który musi się znajdować na końcu łańcucha identyfikatora.

Jeśli będzie istniało wiele dopasowań do wzorca zdefiniowanego z użyciem symbolu wieloznacznego, zostanie wykorzystane dopasowanie najdłuższe.

Zamiast rezygnować z elementu `from-view-id`, można zastosować jedną z dwóch poniższych konstrukcji definiujących regułę stosowaną dla wszystkich stron:

```
<from-view-id>/*</from-view-id>
```

lub

```
<from-view-id>*</from-view-id>
```

Stosowanie elementu from-action

Struktura elementu `navigation-case` jest bardziej złożona od tej, do której ograniczaliśmy się w naszych dotychczasowych przykładach. Oprócz elementu `from-outcome` mamy do dyspozycji także element `from-action`. Taka elastyczność okazuje się szczególnie przydatna w sytuacji, gdy dysponujemy dwiema odrębnymi akcjami z identycznym łańcuchem wyniku.

Przypuśćmy na przykład, że w naszej aplikacji quizu o Javie metoda `startOverAction` zwraca łańcuch `"again"` zamiast łańcucha `"startOver"`. Ten sam łańcuch może zostać zwrócony przez metodę `answerAction`. Do rozróżnienia obu przypadków (z perspektywy mechanizmu nawigacji) można wykorzystać element `from-action`. Zawartość tego elementu musi jednak być identyczna jak łańcuch wyrażenia odwołującego się do metody zastosowany w atrybucie `action`:

```
<navigation-case>
   <from-action>#{quizBean.answerAction}</from-action>
   <from-outcome>again</from-outcome>
   <to-view-id>/again.xhtml</to-view-id>
</navigation-case>
<navigation-case>
   <from-action>#{quizBean.startOverAction}</from-action>
   <from-outcome>again</from-outcome>
   <to-view-id>/index.xhtml</to-view-id>
</navigation-case>
```

Mechanizm odpowiedzialny za nawigację **nie** wywołuje metody otoczonej konstrukcją `#{...}`. Odpowiednia metoda jest wywoływana, zanim jeszcze wspomniany mechanizm przystępuje do pracy. W tej sytuacji mechanizm nawigacji ogranicza się do wykorzystania łańcucha zdefiniowanego w elemencie `from-action` w roli klucza umożliwiającego odnalezienie pasującego przypadku.

Warunkowe przypadki nawigacji JSF 2.0

W wersji JSF 2.0 wprowadzono dodatkowy element `if`, który umożliwia aktywowanie przypadku nawigacji tylko w razie spełnienia określonego warunku. W roli tego warunku należy użyć wyrażenia reprezentującego wartość. Przykład takiego rozwiązania pokazano poniżej:

```
<navigation-case>
   <from-outcome>previous</from-outcome>
   <if>#{quizBean.currentQuestion != 0}</if>
   <to-view-id>/main.xhtml</to-view-id>
</navigation-case>
```

Dynamiczne identyfikatory widoków docelowych JSF 2.0

Element `to-view-id` może mieć postać wyrażenia reprezentującego wartość — w takim przypadku wymaga dodatkowego przetworzenia. Wynik tego wyrażenia jest używany w roli identyfikatora widoku. Przykład takiej konstrukcji przedstawiono poniżej:

```
<navigation-rule>
   <from-view-id>/main.xhtml</from-view-id>
   <navigation-case>
      <to-view-id>#{quizBean.nextViewID}</to-view-id>
   </navigation-case>
</navigation-rule>
```

W tym przykładzie uzyskanie identyfikatora widoku docelowego wymaga wywołania metody `getNextViewID` komponentu `quiz`.

Podsumowanie

W tym rozdziale omówiliśmy wszystkie elementy technologii JavaServer Faces w zakresie zarządzania nawigacją. Należy pamiętać, że nawigacja w najprostszej formie jest wyjątkowo łatwa do zaimplementowania — akcje przycisków poleceń i łączy mogą po prostu zwracać wynik wskazujący następną stronę. Jeśli jednak programista potrzebuje większej kontroli, framework JSF udostępnia mu niezbędne narzędzia.

W następnym rozdziale będziemy koncentrowali się wyłącznie na standardowych komponentach frameworku JSF.

JavaServer Faces

JavaServer Faces (JSF) to technologia platformy Java EE, ułatwiająca projektowanie i tworzenie interfejsów użytkownika aplikacji internetowych. Umożliwia sprawną pracę nad aplikacjami działającymi po stronie serwera i wprowadzanie jasnego podziału na wizualną prezentację oraz właściwą logikę aplikacji. Specyfikacja JSF 2.0 (inaczej niż poprzednia) jest pochodną wielu rzeczywistych projektów open source. Dzięki temu sam framework jest **dużo** prostszy i lepiej zintegrowany ze stosem technologii Java EE niż wersja JSF 1.0. Co więcej, jego specyfikacja przewiduje teraz obsługę technologii takich, jak AJAX czy REST. Framework JSF 2.0 jest obecnie jednym z najznamienitszych frameworków aplikacji internetowych tworzonych w Javie. Do jego mocnych stron należą także: uproszczony model programowania poprzez zastosowanie adnotacji i wprowadzenie zasady **konwencji ponad konfigurację** oraz rozszerzalny model komponentów.

Książka „JavaServer Faces. Wydanie III" zawiera wszystko, czego trzeba do opanowania rozbudowanych elementów frameworka JSF 2.0. Poznaj tajniki znaczników frameworka JSF oraz obsługi zdarzeń. Dowiedz się, jak budować komponenty złożone, i naucz się implementować własne, niestandardowe. Wykorzystaj w swoich aplikacjach technologię AJAX i opanuj nawiązywanie połączeń z bazami danych czy innymi usługami zewnętrznymi. W ostatnim rozdziale znajdziesz pomocne wskazówki na temat diagnozowania i rejestrowania zdarzeń, a także praktyczne przykłady kodu, rozszerzające technologię JSF.

- Komponenty zarządzane
- Zasięg komponentów
- Nawigacja statyczna i dynamiczna
- Znaczniki standardowe
- Facelety
- Tabele danych
- Konwersja i weryfikacja danych
- Obsługa zdarzeń
- Komponenty złożone
- Technologia AJAX
- Usługi zewnętrzne
- Praca z bazami danych

David Geary jest prezesem Clarity Training Inc, firmy organizującej szkolenia i oferującej usługi konsultingowe, oraz autorem ośmiu książek poświęconych technologiom Javy. Jest też członkiem grupy ekspertów odpowiedzialnej za rozwój specyfikacji JSF oraz częstym prelegentem na wielu konferencjach poświęconych oprogramowaniu. Ma też tytuł Java Champion i trzykrotnie został wyróżniony tytułem JavaOne Rock Star.

Cay S. Horstmann jest głównym autorem książek „Java. Podstawy. Wydanie VIII" i „Java. Techniki zaawansowane. Wydanie VIII". Cay jest profesorem informatyki na Uniwersytecie Stanowym w San José, ma tytuł Java Champion i często wygłasza odczyty na konferencjach związanych z branżą komputerową.

Nr katalogowy: 5743

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

Informatyka w najlepszym wydaniu

Cena 99,00 zł

ISBN 978-83-246-2904-6

9 788324 629046