Глава 3

В этой краткой главе речь пойдет о настройке конфигурации механизма навигации в веб-приложении. В частности, будет показано, как в приложении обеспечить переход с одной страницы на другую в зависимости от действий пользователя и выходных данных, получаемых в результате решений, которые принимаются на уровне бизнес-логики.

Статическая навигация

Прежде всего рассмотрим, что происходит, когда пользователь приложения заполняет веб-страницу. При этом пользователь может вводить данные в текстовые поля, выбирать переключатели или выделять записи в списках.

Все эти операции редактирования осуществляются в браузере пользователя. После того как пользователь щелкнет на кнопке, отвечающей за отправку данных формы, все внесенные им изменения передаются на сервер.

Затем веб-приложение анализирует входные данные пользователя и принимает решение о том, какую из JSF-страниц ему следует использовать для подготовки ответа к отображению. За выбор этой очередной JSF-страницы отвечает *обработчик навигации* (navigation handler).

В простом веб-приложении навигация является статической. Это означает, что щелчок на определенной кнопке всегда приводит к выбору для подготовки к отображению конкретной (заранее заданной) JSF-страницы. В разделе "Простой пример" главы 1 на стр. 20 показан наиболее простой механизм обеспечения статической навигации между страницами JSF.

К каждой кнопке добавляется атрибут `action`:

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

На заметку! Как будет показано в главе 4, действия по навигации могут быть также закреплены за гиперссылками.

Значение атрибута `action` называется *результатом*. Ниже, в разделе "Отображение результатов в идентификаторы представлений" на стр. 81 указано, что результат может быть, по желанию, сопоставлен идентификатору представления. (В спецификации JSF *представлением* называется страница JSF.)

Если подобное отображение для какого-то конкретного результата не предусмотрено, то результат преобразуется в идентификатор представления с использованием следующих шагов.

1. Если в результате не предусмотрено расширение имени файла, то добавляется расширение текущего представления.
2. Если результат не начинается со знака /, то задается префикс в виде пути текущего представления.

В качестве примера можно указать, что результат с именем `welcome` в представлении `/index.xhtml` приводит к получению идентификатора целевого представления `/welcome.xhtml`.

На заметку! Начиная с версии JSF 2.0 применение отображений из результатов в идентификаторы представлений является необязательным. До выхода версии JSF 2.0 необходимо было задавать явные правила навигации для каждого результата.

Динамическая навигация

В большинстве веб-приложений навигация не является статической. Поток страниц зависит не только от того, на какой кнопке выполнен щелчок, но и от того, какие входные данные предоставлены. Например, отправка страницы входа в систему может иметь два результата: успешный и неудачный. Результат вычисляется в коде и в данном случае указывает, являются ли допустимыми имя пользователя и пароль.

Для обеспечения динамической навигации кнопка отправки должна иметь *выражение метода* (method expression), такое, как

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

В данном примере `loginController` ссылается на бин некоторого класса, а этот класс должен иметь метод `verifyUser`.

Выражение метода в атрибуте `action` не имеет параметров. Возвращаемый им тип может быть любым. Возвращаемое значение преобразуется в строку путем вызова метода `toString`.

На заметку! В версии JSF 1.1 метод действия должен был иметь только возвращаемый тип `String`. А начиная с версии 1.2 разрешается использовать любой возвращаемый тип. Удобной альтернативой, в частности, являются перечисления, поскольку это позволяет использовать компилятор для перехвата опечаток в именах действий.

Ниже показан пример метода действия.

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

На заметку! Метод действия может также возвращать и значение `null`, указывающее, что должно быть снова отображено то же представление. В этом случае сохраняется область действия представления, которая рассматривалась в главе 2. Получение любого результата, отличного от `null`, приводит к очистке области действия представления, даже если в конечном итоге должно быть получено представление, совпадающее с текущим.

Этот метод возвращает строку результата, такую как "success" или "failure", используемую для определения следующего представления.

Ниже вкратце перечислены шаги, которые выполняются каждый раз, когда пользователь щелкает на командной кнопке, в атрибуте action которой указано выражение метода.

1. Извлечение указанного бина.
2. Вызов метода, указанного в ссылке, и возврат строки результата.
3. Преобразование строки результата в идентификатор представления.
4. Отображение страницы, соответствующей идентификатору представления.

Таким образом, для реализации поведения с ветвлением предоставляется ссылка на метод в соответствующем классе бина. Существует много вариантов размещения этого метода. Наилучший подход состоит в том, чтобы найти класс, имеющий все данные, необходимые для принятия решения.

Отображение результатов в идентификаторы представлений

Одна из ключевых целей проектирования на основе технологии JSF состоит в отделении уровня представления от уровня бизнес-логики. Если решения по навигации принимаются динамически, то в коде, в котором вычисляется результат, не требуются точные сведения об именах веб-страниц. Технология JSF предоставляет механизм отображения логических результатов, таких как успешное или неудачное завершение, в действительные веб-страницы.

Это достигается путем добавления записей navigation-rule (правило навигации) в файл faces-config.xml. Ниже приведен типичный пример.

```
<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>
```

Согласно данному правилу, результат success должен приводить к переходу на страницу /welcome.xhtml, если он возникает во время работы со страницей /index.xhtml.

На заметку! Строки идентификаторов представлений начинаются со знака /. Если используется отображение расширения (такое как суффикс .faces), то расширение должно согласовываться с расширением имени файла (наподобие .xhtml), а не с расширением URL.

Тщательный выбор строк результата позволяет собирать многочисленные правила навигации в группы. Например, может оказаться так, что в приложении на многих страницах имеются кнопки с действием logout. С помощью одного-единственного правила можно сделать так, чтобы щелчок на любой из них приводил к переходу на страницу loggedOut.xhtml:

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

Это правило будет действовать для всех страниц, поскольку не был задан элемент `from-view-id`.

Правила навигации с одинаковым элементом `from-view-id` можно объединять:

```
<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>
```

На заметку! В простых приложениях необходимость в использовании правил навигации может не возникнуть. А по мере усложнения приложений появляется смысл использовать логические результаты в управляемых бинах наряду с правилами навигации для отображения результатов в целевые представления.

Приложение JavaQuiz

В настоящем разделе показано, как ввести в действие средства навигации в программе, которая предъявляет пользователю ряд вопросов викторины (рис. 3.1).

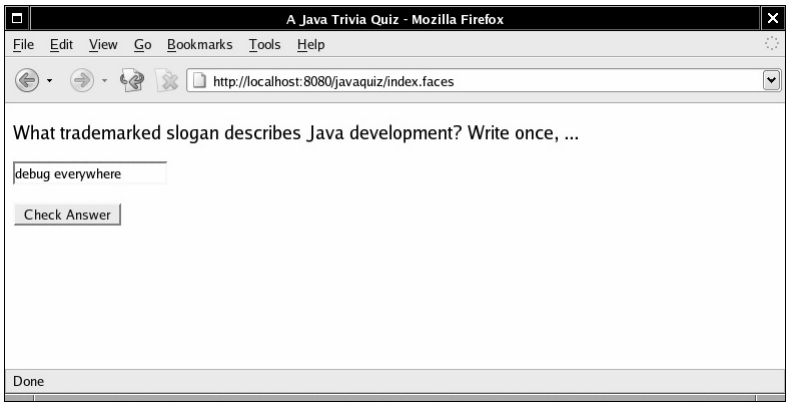

Рис. 3.1. Вопрос викторины

После того как пользователь щелкнет на кнопке Check Answer (Проверить ответ), приложение проверяет, правильным ли является предоставленный им ответ. В случае неправильного ответа пользователю дается еще один дополнительный шанс ответить на тот же вопрос (рис. 3.2).

После двух неправильных ответов отображается следующий вопрос (рис. 3.3).

Разумеется, следующий вопрос отображается также после правильного ответа. Наконец, после последнего вопроса отображается итоговая страница с результатами, которая предлагает пользователю попытаться еще раз пройти викторину (рис. 3.4).

В рассматриваемом приложении имеются два класса. Класс `Problem`, показанный в листинге 3.1, описывает одну задачу с вопросом, ответом и методом проверки того, является ли предъявленный ответ правильным.

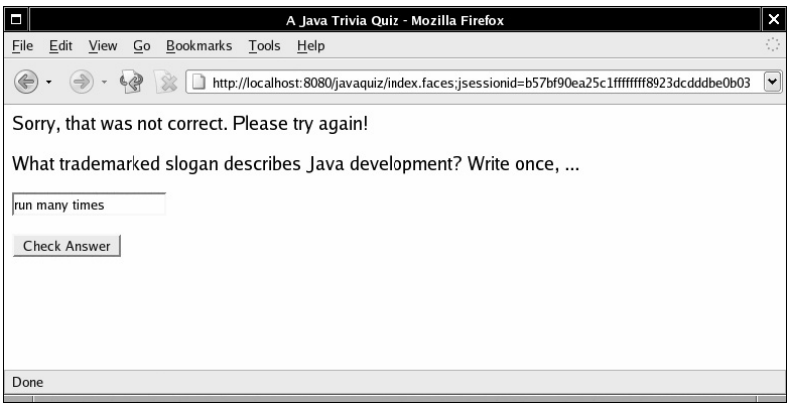

Рис. 3.2. Один неправильный ответ: сделать еще одну попытку

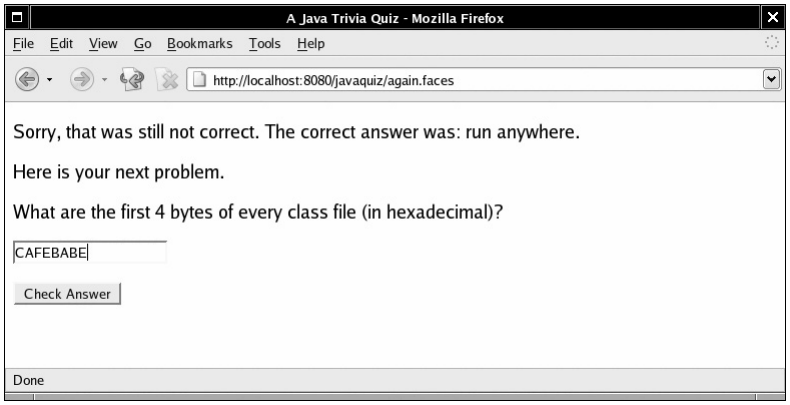

Рис. 3.3. Два неправильных ответа: перейти к следующему

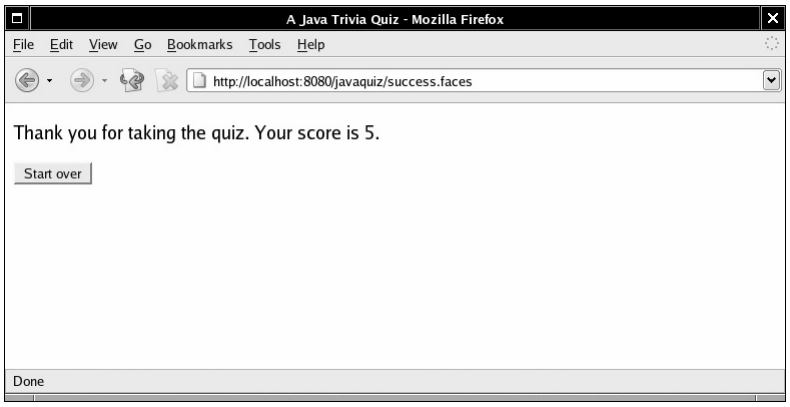

Рис. 3.4. Завершение викторины

Класс `QuizBean` содержит описание викторины, которая состоит из ряда вопросов. Кроме того, экземпляр `QuizBean` дополнительно отслеживает текущую задачу и общее количество очков, заработанное пользователем. Полный код приложения приведен в листинге 3.2.

Листинг 3.1. Файл `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.    // Переопределить в целях более сложной проверки
19.    public boolean isCorrect(String response) {
20.       return response.trim().equalsIgnoreCase(answer);
21.    }
22. }
```

В этом примере для размещения методов, связанных с навигацией, в наибольшей степени подходит класс `QuizBean`. В этом бине имеется вся информация о действиях пользователя, с помощью которой можно определить, какая страница должна быть отображена следующей.

Метод `answerAction` класса `QuizBean` реализует логику навигации. Этот метод возвращает строку `"success"` или `"done"`, если пользователь ответил на вопрос правильно, строку `"again"`, если пользователь ответил на вопрос неправильно первый раз, и строку `"failure"` или `"done"` после второй неправильной попытки.

```
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";
 }
}
```

Выражение метода `answerAction` присоединяется к кнопкам на каждой странице. Например, страница `index.xhtml` содержит следующий элемент:

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

На рис. 3.5 показана структура каталогов данного приложения, а в листинге 3.3 приведен код главной страницы викторины `index.xhtml`. Код страниц `success.xhtml` и `failure.xhtml` мы решили не приводить в данной книге, поскольку он отличается от кода `index.xhtml` только сообщением, отображаемым в верхней части.

На странице done.xhtml, которая рассматривается в листинге 3.4, отображаются окончательная оценка и приглашение пользователю вернуться к началу игры. На этой странице заслуживает внимания командная кнопка. Создается впечатление, будто можно было бы применить статическую навигацию, поскольку щелчок на кнопке Start Over (Начать сначала) всегда приводит к возврату пользователя к странице index.xhtml. Однако мы использовали выражение метода:

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

Метод startOverAction выполняет полезную работу, необходимую для того, чтобы игру можно было начать сначала, а именно — сбрасывает значение количества очков и случайным образом перемешивает элементы ответов:

```
public String startOverAction() {
  Collections.shuffle(problems);
  currentProblem = 0;
  score = 0;
  tries = 0;
response = "";
  return "startOver";
}
```

В целом методы действия применяются для достижения двух целей.

- Внесение в модель обновлений, являющихся следствием действий пользователя.
- Передача указания обработчику навигации, какая страница должна отображаться далее.

На заметку! Как будет показано в главе 8, к кнопкам можно также присоединять прослушиватели действий. После того как пользователь щелкнет на кнопке, будет выполнен код метода processAction прослушивателя действий. Однако прослушиватели действий не взаимодействуют с обработчиком навигации.

В листинге 3.5 приведен файл конфигурации приложения с правилами навигации. Чтобы лучше понять эти правила, рассмотрите переходы со страницы на страницу, показанные на рис. 3.6.

Для трех из применяемых результатов ("success", "again" и "done") правила навигации не предусмотрены. Эти результаты всегда приводят к страницам /success.xhtml, /again.xhtml и /done.xhtml. А результат "startOver" мы отображаем на странице /index.xhtml. С другой стороны, результат failure требует большего объема работы. Он первоначально приводит к странице /again.xhtml, с помощью которой пользователь может сделать вторую попытку. Но если результат failure возникает и на этой странице, то следующей страницей становится /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>
```

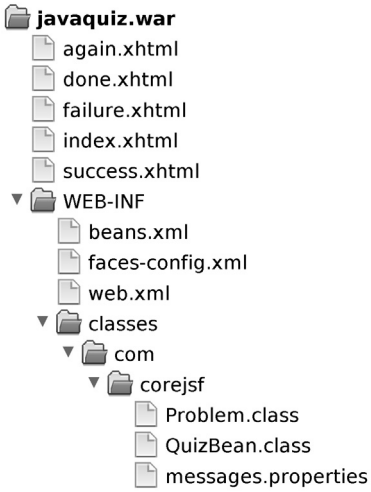

Рис. 3.5. Структура каталогов приложения JavaQuiz

Обратите внимание на то, что имеет значение порядок правил. Второе правило согласуется, если текущей страницей не является `/again.xhtml`.

Наконец, в листинге 3.6 показаны строки сообщений.

Листинг 3.2. Файл `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.    // или import javax.faces.bean.ManagedBean;
11. import javax.enterprise.context.SessionScoped;
12.    // или import javax.faces.bean.SessionScoped;
13.
14. @Named // или @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.     // Здесь формулировки задач приведены в коде. В реальном приложении
24.     // они должны считываться из базы данных
25.     private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
26.        new Problem(
27.           "What trademarked slogan describes Java development? Write once, ...",
28.           "run anywhere"),
29.        new Problem(
30.           "What are the first 4 bytes of every class file (in hexadecimal)?",
```

```
31.            "CAFEBABE"),
32.         new Problem(
33.            "What does this statement print? System.out.println(1+\"2\");",
34.            "12"),
35.         new Problem(
36.            "Which Java keyword is used to define a subclass?",
37.            "extends"),
38.         new Problem(
39.            "What was the original name of the Java programming language?",
40.            "Oak"),
41.         new Problem(
42.            "Which java.util class describes a point in time?",
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. }
```

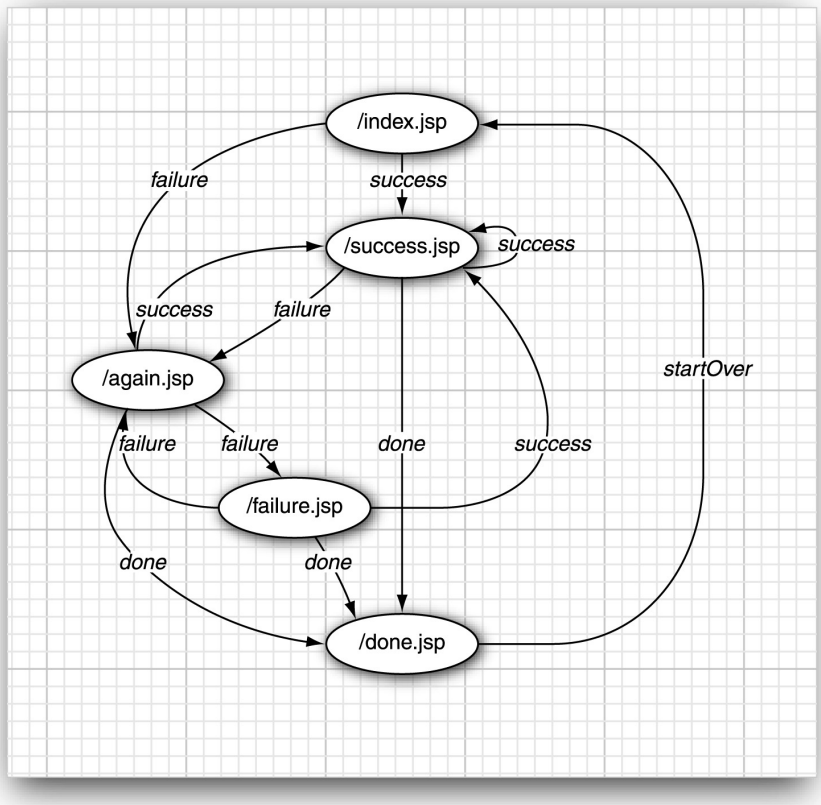

Рис. 3.6. Схема перехода со страниц на страницы в приложении JavaQuiz

Листинг 3.3. Файл `javaquiz/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>#{quizBean.question}"/></p>
         <p><h:inputText value="#{quizBean.response}"/></p>
         <p>
            <h:commandButton value="#{msgs.checkAnswer}"
                             action="#{quizBean.answerAction}"/>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.4. Файл `javaquiz/web/done.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>
            #{msgs.thankYou}
            <h:outputFormat value="#{msgs.score}">
                <f:param value="#{quizBean.score}"/>
            </h:outputFormat>
         </p>
         <p>
            <h:commandButton value="#{msgs.startOver}"
                             action="#{quizBean.startOverAction}"/>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.5. Файл `javaquiz/web/WEB-INF/faces-config.xml`

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
   version="2.0">
   <navigation-rule>
      <navigation-case>
         <from-outcome>startOver</from-outcome>
         <to-view-id>/index.xhtml</to-view-id>
      </navigation-case>
   </navigation-rule>
   <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>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
```

```
32.    </application>
33. </faces-config>
```

Листинг 3.6. Файл `javaquiz/src/java/com/corejsf/messages.properties`

```
 1. title=A Java Trivia Quiz
 2. checkAnswer=Check Answer
 3. startOver=Start over
 4. correct=Congratulations, that is correct.
 5. notCorrect=Sorry, that was not correct. Please try again!
 6. stillNotCorrect=Sorry, that was still not correct.
 7. correctAnswer=The correct answer was: {0}.
 8. score=Your score is {0}.
 9. nextProblem=Here is your next problem.
10. thankYou=Thank you for taking the quiz.
```

Перенаправление

Предусмотрена возможность передать реализации JSF запрос на перенаправление к новому представлению. Затем реализация JSF передает клиенту код HTTP перенаправления. Ответ перенаправления `redirect` сообщает клиенту, какой URL должен использоваться для следующей страницы. После этого клиент выполняет запрос `GET` к указанному URL.

Перенаправление происходит медленно, поскольку требует дополнительного цикла обмена данными между сервером и клиентом (браузером). Однако перенаправление дает браузеру шанс обновить свое поле адреса.

На рис. 3.7 показано, как изменяется поле адреса при использовании перенаправления.

Если перенаправление не используется, то исходный URL-адрес (`localhost:8080/javaquiz/faces/index.xhtml`) остается неизменным, когда пользователь переходит со страницы `/index.xhtml` на страницу `/success.xhtml`. А в случае перенаправления браузер отображает новый URL-адрес (`localhost:8080/javaquiz/faces/success.xhtml`).

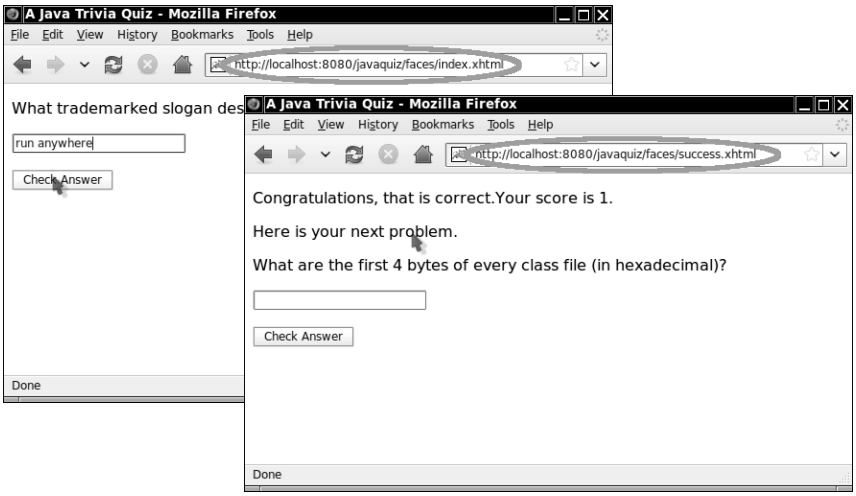

Рис. 3.7. Перенаправление с помощью обновления URL в браузере

Если правила навигации не используются, то необходимо добавить строку

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

к строке результата:

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

В правиле навигации элемент `redirect` добавляется после элемента `to-view-id` следующим образом:

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

Перенаправление и флеш-память JSF 2.0

Чтобы уменьшить до предела заполнение ненужными элементами области действия сеанса, имеет смысл в максимально возможной степени использовать область действия запроса. Не прибегая к использованию элемента `redirect`, можно применять бины с областью действия запроса для данных, отображаемых в следующем представлении.

Однако рассмотрим, что происходит при перенаправлении.

1. Клиент отправляет запрос серверу.
2. Отображение области действия запроса заполняется бинами, для которых областью действия служит запрос.
3. Сервер отправляет клиенту код состояния HTTP 302 (Moved temporarily), указывающий на временное перемещение страницы наряду со сведениями о месте перенаправления. На этом обработка текущего запроса заканчивается, и происходит удаление бинов с областью действия запроса.
4. Клиент выполняет запрос `GET` к новому местоположению.
5. Сервер подготавливает к отображению следующее представление. Однако бины с применявшейся ранее областью действия запроса становятся недоступными.

В целях преодоления этой проблемы в версии JSF 2.0 предусмотрен объект флеш-памяти, который может заполняться в одном запросе и использоваться в другом. (Понятие *флеш-памяти* заимствовано из терминологии, применяющейся на веб-платформе Ruby on Rails.) Обычно флеш-память используется для сообщений. Например, поместить сообщение во флеш-память может обработчик кнопки:

```
ExternalContext.getFlash().put("message", "Your password is about to expire");
```

Метод `ExternalContext.getFlash()` возвращает объект класса `Flash`, который реализует интерфейс `Map<String, Object>`.

На странице JSF ссылка на объект флеш-памяти осуществляется с помощью переменной флеш-памяти. Например, сообщение можно отобразить следующим образом:

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

После подготовки сообщения к отображению и передачи подготовленного представления клиенту строка сообщения автоматически удаляется из флеш-памяти. Значение во флеш-памяти можно даже сохранить больше чем для одного запроса. Выражение

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

отправляет значение ключа сообщения во флеш-память и добавляет его снова для еще одного цикла запроса.

На заметку! Если в ходе разработки обнаруживается, что между флеш-памятью и бином перебрасываются большие объемы данных, то вместо этого можно рассмотреть возможность использования области действия сеанса.

Навигация с поддержкой метода REST и применение URL, обеспечивающих формирование закладок JSF 2.0

По умолчанию приложение JSF выполняет последовательность запросов POST, передаваемых на сервер. Каждый запрос POST содержит данные формы. Это имеет смысл для приложения, которое собирает большой объем ввода от пользователя. Но большинство веб-приложений не действует подобным образом. Рассмотрим пример, в котором пользователь просматривает каталог товаров для совершения покупок, с помощью щелчков переходя от одной ссылки к другой. Пользователь не вводит никаких данных, а лишь выбирает ссылки для последующего щелчка на них. Эти ссылки должны обеспечивать формирование закладок, чтобы пользователь мог возвращаться к одним и тем же страницам при повторном посещении данного URL. Кроме того, страницы должны быть кэшируемыми. Кэширование — это важная часть создания эффективных веб-приложений. Безусловно, запросы POST не могут применяться для решения задач создания закладок или кэширования.

Архитекторы веб-приложений применяют стиль, получивший название REST (Representational State Transfer — передача представительного состояния), который основан на использовании в приложениях протокола HTTP по такому принципу, который был заложен в нем первоначально. В операциях поиска должны использоваться запросы GET. Запросы PUT, POST и DELETE должны служить для создания, модификации и удаления.

Сторонники подхода на основе REST, как правило, предпочитают применять URL примерно такого типа:

```
http://myserver.com/catalog/item/1729
```

но архитектура REST не требует применения подобных URL, которые принято называть "привлекательными". Запрос GET с параметром

```
http://myserver.com/catalog?item=1729
```

также вполне можно рассматривать как поддерживающий метод REST.

Следует учитывать, что запросы GET ни в коей мере не должны использоваться для обновления информации. Например, запрос GET для добавления товара в корзину

```
http://myserver.com/addToCart?cart=314159&item=1729
```

был бы неподходящим. Запросы GET должны быть идемпотентными. Под этим подразумевается то, что, допустим, двухкратная выдача запроса не должна приводить к иным последствиям, чем однократная. Кэшируемыми могут быть только такие запросы, которые подчиняются этому требованию. Запрос на "добавление товара к корзине" не является идемпотентным, поскольку после его двухкратного выполнения в корзине появятся два экземпляра одного и того же товара. Но в этом контексте запросы POST определенно являются приемлемыми. Таким образом, даже в веб-приложении, которое поддерживает метод REST, должны в определенной степени использоваться запросы POST.

В настоящее время в технологии JSF не предусмотрен стандартный механизм создания или применения "привлекательных URL", но начиная с версии JSF 2.0 имеется поддержка для запросов GET. Такая поддержка будет описана в следующих разделах настоящей главы.

Параметры просмотра

Рассмотрим запрос GET на отображение сведений об определенном товаре:

```
http://myserver.com/catalog?item=1729
```

Идентификатор товара задан как параметр запроса. После получения запроса значение этого параметра должно быть передано соответствующему бину. С этой целью могут использоваться параметры представления.

В верхней части страницы добавьте теги наподобие следующих:

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

При обработке запроса значение параметра запроса сведений о товаре передается методу setCurrentItem бина каталога.

Страница JSF может иметь любое количество параметров представления. Параметры представления могут проверяться и преобразовываться, как и любые другие параметры запроса. (В главе 7 изложены подробные сведения о преобразовании и проверке правильности.)

Часто возникает необходимость в получении дополнительных данных после задания параметров представления. Например, после того как будет задан параметр представления определенного товара, может потребоваться выбрать данные о свойствах товара из базы данных для дальнейшей подготовки к отображению страницы с описанием товара. В главе 8 будет показано, как возложить обязанности по выполнению этой работы на обработчик события preRenderView.

Ссылки запросов GET

В предыдущем разделе было показано, как в веб-технологии JSF обрабатываются запросы GET. В приложении, поддерживающем метод REST, может потребоваться предоставить пользователям возможность перемещаться по страницам с помощью запросов GET. В связи с этим возникает необходимость в добавлении на страницы кнопок и ссылок, после щелчков на которых вырабатываются запросы GET. С этой целью используются теги h:button и h:link. (С другой стороны, для выработки запросов POST служат теги h:commandButton и h:commandLink.)

В связи с этим при работе с такими запросами может потребоваться управлять идентификаторами целевого представления и параметрами запроса. Идентификатор целевого представления задается с помощью атрибута outcome. Он может представлять собой фиксированную строку:

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

Еще один вариант состоит в том, что может быть указано выражение значения:

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

Вызывается метод getSkipOutcome. Он должен выдать строку результата. Затем строка результата обычным образом передается в обработчик навигации, выдавая идентификатор целевого представления.

В этом состоит существенное различие между атрибутом `outcome` тега `h:button` и атрибутом `action` тега `h:commandButton`. Значение атрибута `outcome` вычисляется до подготовки страницы к отображению, что позволяет встроить требуемую ссылку в содержимое страницы. Однако вычисление значения атрибута `action` происходит только после фактически выполненного пользователем щелчка на кнопке. По этой причине в спецификации JSF для описания процесса вычисления идентификаторов целевых представлений для запросов `GET` используется термин "навигация с вытеснением".

Внимание! Выражение языка выражений в атрибуте `outcome` является выражением значения, а не выражением метода. В принципе действие кнопки можно применять для изменения состояния приложения определенным образом. Однако вычисление результата обработки ссылки с запросом `GET` не должно приводить к каким-либо изменениям, ведь в конечном итоге ссылка вычисляется только для потенциального использования в будущем.

Определение параметров запроса

Часто возникает необходимость включать параметры в связи с обработкой ссылки с запросом `GET`. Эти параметры могут быть получены из трех источников.

- Строка результата.
- Параметры просмотра.
- Вложенные теги `f:param`.

Если один и тот же параметр задан более одного раза, то приоритет получает последнее значение в этом списке присваивания.

Рассмотрим более подробно, как осуществляется выбор применяемых значений.

Параметры могут быть заданы в строке результатов примерно так:

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

Обработчик навигации удаляет параметры из результата, вычисляет идентификатор целевого представления и добавляет параметры. В этом примере идентификатором целевого представления является `/index.xhtml?q=1`.

При передаче многочисленных параметров следует обязательно предусмотреть экранирование с помощью разделителя `&`:

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

Безусловно, в строке результата можно использовать выражение значения, как в следующем примере:

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

Предусмотрен удобный сокращенный способ включения всех параметров представления в строку запроса. Он состоит в том, что добавляется один атрибут:

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

С помощью этого способа можно переносить все параметры представления с одной страницы на другую, что представляет собой обычное требование к приложению, поддерживающему метод `REST`.

Для определения параметров представления может использоваться тег `f:param`. Например:

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

Аналогичные преимущества, связанные с включением параметров представления, можно получить при работе со ссылками перенаправления, которые также являются запросами GET. Но вместо задания значения атрибута в теге следует добавлять параметр в результат:

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

К сожалению, вложенные теги f:param не включаются в запрос.

Если для задания правил навигации служат файлы конфигурации в коде XML, используйте атрибут include-viewparams и вложенные теги view-param:

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

В применяемом синтаксисе имеются некоторые несогласованности, но они не должны вызывать обеспокоенность. Программист просто должен работать более внимательно, а это способствует повышению уровня его квалификации.

Добавление ссылок, обеспечивающих формирование закладок, в приложение для викторины

Рассмотрим приложение для викторины, которое использовалось ранее в этой главе для демонстрации способов навигации. Можно ли добиться того, чтобы это приложение в большей степени поддерживало метод REST?

Запрос GET не подходит для передачи ответа, поскольку такие запросы не являются идемпотентными. Передача ответа приводит к изменению счета. Но ссылки, поддерживающие метод REST, можно применить для перехода от одного задания викторины к другому.

Чтобы чрезмерно не усложнять это приложение, мы предусмотрим единственную ссылку, применимую для формирования закладки, в целях перехода к следующему заданию. Для этого используется параметр представления:

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

Ссылка определяется следующим образом:

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

Метод getSkipOutcome бина QuizBean возвращает индекс или значение "done", в зависимости от того, доступны ли дополнительные задания:

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

Ссылка, полученная в итоге, выглядит примерно так (рис. 3.8):

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

Теперь имеется возможность создать закладку на эту ссылку, что позволяет вернуться к любому заданию викторины.

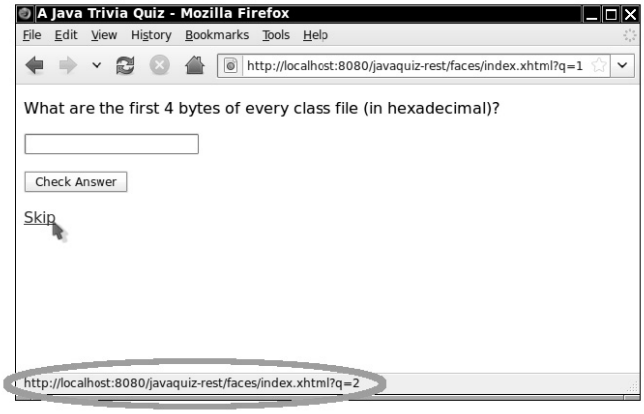

Рис. 3.8. Ссылка, поддерживающая метод REST

В листинге 3.7 показана страница `index.xhtml` с параметром представления и тегом `h:link`. В листинге 3.8 приведен модифицированный код бина `QuizBean`. Мы добавили метод `setCurrentProblem` и изменили механизм вычисления количества очков. Поскольку теперь появилась возможность снова и снова возвращаться к одному и тому же заданию, мы должны исключить возможность того, чтобы пользователь зарабатывал дополнительные очки, отвечая на один и тот же вопрос несколько раз.

Листинг 3.7. Файл `javaquiz-rest/web/index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
   <f:metadata>
      <f:viewParam name="q" value="#{quizBean.currentProblem}"/>
   </f:metadata>
   <h:head>
      <title>#{msgs.title}</title>
   </h:head>
   <h:body>
      <h:form>
         <p>#{quizBean.question}</p>
         <p><h:inputText value="#{quizBean.response}"/></p>
         <p><h:commandButton value="#{msgs.checkAnswer}"
         action="#{quizBean.answerAction}"/></p>
         <p><h:link outcome="#{quizBean.skipOutcome}" value="Skip">
            <f:param name="q" value="#{quizBean.currentProblem + 1}"/>
         </h:link>
         </p>
      </h:form>
   </h:body>
</html>
```

Листинг 3.8. Файл `javaquiz-rest/src/java/com/corejsf/QuizBean.java`

```
package com.corejsf;

import java.io.Serializable;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import javax.inject.Named;
   // или import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // или import javax.faces.bean.SessionScoped;

@Named // или @ManagedBean
@SessionScoped
public class QuizBean implements Serializable {
   private int currentProblem;
   private int tries;
   private String response = "";
   private String correctAnswer;

   // Здесь формулировки задач приведены в коде. В реальном приложении
   // они должны считываться из базы данных
   private ArrayList<Problem> problems = new ArrayList<Problem>(Arrays.asList(
      new Problem(
         "What trademarked slogan describes Java development? Write once, ...",
         "run anywhere"),
      new Problem(
         "What are the first 4 bytes of every class file (in hexadecimal)?",
         "CAFEBABE"),
      new Problem(
         "What does this statement print? System.out.println(1+\"2\");",
         "12"),
        new Problem(
           "Which Java keyword is used to define a subclass?",
           "extends"),
        new Problem(
           "What was the original name of the Java programming language?",
           "Oak"),
        new Problem(
           "Which java.util class describes a point in time?",
           "Date")));

    private int[] scores = new int[problems.size()];

    public String getQuestion() {
       return problems.get(currentProblem).getQuestion();
    }

    public String getAnswer() { return correctAnswer; }

    public int getScore() {
       int score = 0;
       for (int s : scores) score += s;
       return score;
    }

    public String getResponse() { return response; }
    public void setResponse(String newValue) { response = newValue; }

```

```
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.}
```

Расширенные правила навигации

Приемы, описанные в предыдущих разделах, должны удовлетворять большинству практических потребностей, связанных с навигацией. В этом разделе рассматриваются оставшиеся правила для элементов навигации, которые могут присутствовать в файле `faces-config.xml`. На рис. 3.9 показана синтаксическая диаграмма для допустимых элементов.

На заметку! Как уже было описано в разделе "Настройка конфигурации бинов" главы 2 на стр. 64, информация о навигации также может размещаться и в других файлах конфигурации, а не только в стандартном файле `faces-config.xml`.

Синтаксическая диаграмма, приведенная на рис. 3.9, показывает, что каждый элемент, `navigation-rule` и `navigation-case`, может иметь произвольное описание, а также элементы `display-name` и `icon`. Эти элементы предназначены для использования в инструментальных средствах разработки программ, поэтому более подробно здесь рассматриваться не будут.

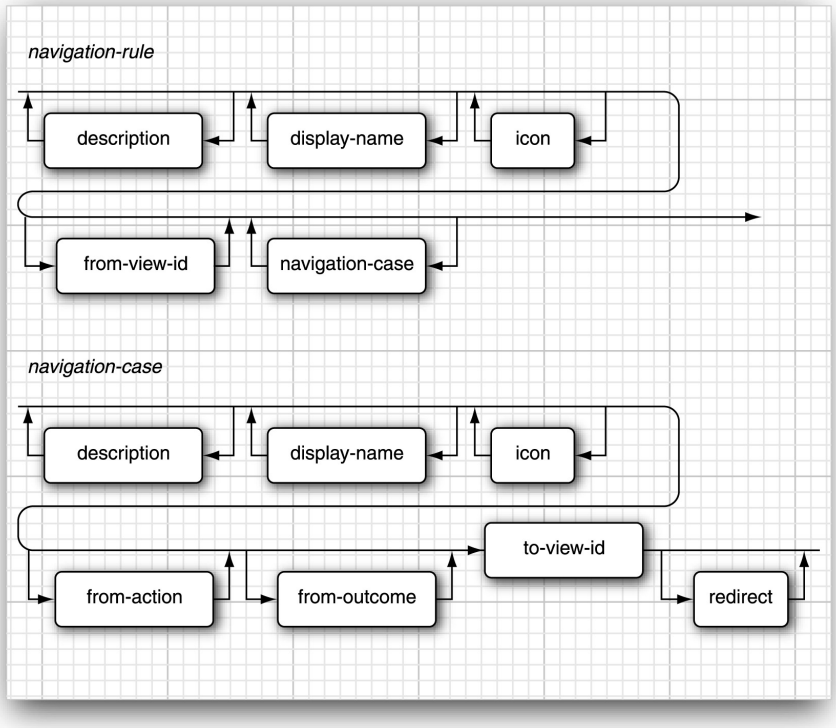

Рис. 3.9. Синтаксическая диаграмма для элементов навигации

Подстановочные знаки

В элементе `from-view-id` правила навигации можно использовать подстановочные знаки, например:

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

Это правило будет действовать для всех страниц, имена которых начинаются с префикса `/secure/`. Допускается применение только одного символа `*`, который должен находиться в конце строки идентификатора.

Если с шаблоном согласуется несколько правил с подстановочными знаками, то выбирается правило с наибольшей длиной.

На заметку! Вместо того чтобы не учитывать элемент `from-view-id`, можно также использовать один из следующих подходов для указания правила, распространяющегося на все страницы:

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

или

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

Использование элемента `from-action`

Элемент `navigation-case` имеет более сложную структуру по сравнению с теми, которые рассматривались ранее. Кроме элемента `from-outcome`, имеется также элемент `from-action`. Дополнительные возможности, связанные с применением двух элементов, могут оказаться удобными, если необходимо связать два отдельных действия с одной и той же строкой результата.

Предположим, например, что в приложении с викториной метод `startOverAction` возвращает не строку `"startOver"`, а строку `"again"`. Ту же строку может возвращать и метод `answerAction`. Для проведения различий между этими двумя вариантами навигации можно использовать элемент `from-action`. Содержимое этого элемента обязательно должно полностью соответствовать строке выражения метода в атрибуте `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>
```

На заметку! Обработчик навигации не вызывает метод, указанный в разграничителях `#{...}`. Этот метод вызывался до завершения функционирования обработчика навигации. Обработчик навигации просто использует строку `from-action` в качестве ключа для поиска соответствующего варианта навигации.

Возможности условной навигации JSF 2.0

Начиная с версии JSF 2.0 предусмотрена возможность задавать элемент `if`, который активизирует определенный вариант навигации только при выполнении заданного условия. Для этого условия должно быть предусмотрено выражение значения. Ниже приведен соответствующий пример.

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

Динамические идентификаторы целевого представления JSF 2.0

Элемент `to-view-id` может представлять собой выражение значения, и в этом случае происходит его вычисление. Полученный результат используется как идентификатор представления. Рассмотрим пример:

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

В этом примере вызывается метод `getNextViewID` бина викторины для получения идентификатора целевого представления.

Резюме

Выше были описаны все средства управления навигацией, предусмотренные в технологии JSF. Следует помнить, что в наиболее простом случае эти средства являются весьма несложными: действия кнопок и ссылок могут просто возвращать результат, который определяет следующую страницу. Но если необходимо обеспечить более развитое управление, платформа JSF предоставит все необходимые инструментальные средства.

В следующей главе будут приведены все необходимые сведения о стандартных компонентах JSF.