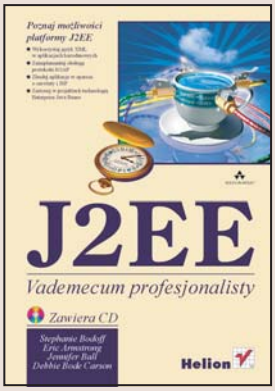
J2EE. Vademecum profesjonalisty. Wydanie II



Autorzy: Stephanie Bodoff, Eric Armstrong, Jennifer Ball,Debbie Bode Carson
Tłumaczenie: Adam Bochenek, Piotr Rajca, Jaromir Senczyk, Przemysław Szeremiota
ISBN: 83-7361-953-4
Tytuł oryginału: The J2EE Tutorial, Second Edition
Format: B5, stron: 1292

Poznaj możliwości platformy J2EE

- Wykorzystaj język XML w aplikacjach bazodanowych
- Zaimplementuj obsługę protokołu SOAP
- Zbuduj aplikacje w oparciu o serwlety i JSP
- Zastosuj w projektach technologię Enterprise Java Beans

Platforma Java 2 Enterprise Edition zdobyła już mocną pozycję na rynku serwerów aplikacji. Dzięki niej możliwe stało się tworzenie aplikacji korporacyjnych zgodnych z podstawowym założeniem przyświecającym twórcom Javy – „pisz raz, uruchamiaj wszędzie". Najnowsza wersja platformy J2EE została znacznie rozbudowana i zmodernizowana w porównaniu z poprzednimi. Dodano do niej możliwość obsługi usług WWW, rozszerzono i unowocześniono implementacje technologii serwletów i JSP oraz poprawiono wiele komponentów przydatnych przy tworzeniu aplikacji korporacyjnych.

„J2EE. Vademecum profesjonalisty. Wydanie II" to kompletny przewodnik po najnowszej wersji Java 2 Enterprise Edition, napisany przez członków zespołu zajmującego się rozwojem platformy J2EE, zatrudnionych w firmie Sun. Opisuje kluczowe komponenty Java 2 Platform, Enterprise Edition (J2EE) w wersji 1.4. Prezentuje rozwiązania konkretnych problemów napotykanych przez programistów, zilustrowane licznymi przykładami. W książce przedstawiono nie tylko komponenty J2EE, ale również współpracujące z nią technologie: JavaServer Pages Standard Tag Library (JSTL) oraz JavaServer Faces.

- Tworzenie aplikacji internetowej wykorzystującej język XML
- Stosowanie parsera SAX
- Analiza i przetwarzanie hierarchii DOM w dokumentach
- Korzystanie z XPath
- Implementacja usług WWW oraz obsługi protokołu SOAP
- Tworzenie aplikacji z wykorzystaniem serwletów i JSP
- Stosowanie technologii JavaServer Faces
- Projektowanie z wykorzystaniem Enterprise Java Beans

Jeśli chcesz zaprojektować i stworzyć aplikację z wykorzystaniem J2EE, w tej książce znajdziesz wszystkie informacje, jakie mogą Ci być do tego potrzebne.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 21.

Konfigurowanie aplikacji JavaServer Faces

Do obowiązków architekta aplikacji należą:

- Rejestrowanie obiektów pomocniczych dla danej aplikacji w celu udostępnienia ich wszystkim częściom aplikacji.
- Konfigurowanie komponentów pomocniczych i komponentów modelu, aby odwołania do nich na stronie powodowały tworzenie instancji o odpowiednich wartościach.
- Definiowanie reguł nawigacji dla każdej ze stron aplikacji gwarantujące poprawną sekwencję kolejnych stron aplikacji.
- Pakowanie wszystkich stron, obiektów i innych plików aplikacji, aby aplikacja mogła zostać zainstalowana dla dowolnego kontenera.

Zadaniem niniejszego rozdziału jest wyjaśnienie sposobu realizacji wymienionych zadań.

Plik konfiguracyjny zasobów aplikacji

Technologia JavaServer Faces stosuje przenośny format plików konfiguracyjnych (w postaci dokumentu XML) umożliwiających konfigurowanie zasobów. Architekt aplikacji tworzy jeden lub więcej plików zwanych plikami konfiguracyjnymi zasobów aplikacji, które używają tego formatu do rejestrowania i konfigurowania obiektów, a także definiowania reguł nawigacji. Plik konfiguracyjny zasobów aplikacji nosi zwykle nazwę *faces-config.xml*.

Plik konfiguracyjny zasobów aplikacji musi być zgodny z plikiem *http://java.sun.com/dtd/web-facesconfig_1_0.dtd*. Dodatkowo każdy taki plik musi zawierać następujące elementy w poniższym porządku:

- Numer wersji XML:

```
<?xml version="1.0"?>
```

- Deklarację DOCTYPE:

```
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD Javaserver Faces
Config 1.0//EN" "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
```

- Znacznik `faces-config` obejmujący wszystkie inne deklaracje:

```
<faces-config>
...
</faces-config>
```

Może istnieć więcej niż jeden plik konfiguracyjny zasobów aplikacji. Implementacja JavaServer Faces znajduje ten plik lub pliki, poszukując:

- Zasobu o nazwie */META-INF/faces-config.xml* w dowolnym pliku JAR w katalogu aplikacji */WEB-INF/lib* oraz za pomocą nadrzędnych classloaderów. Jeśli istnieje zasób o takiej nazwie, to zostaje załadowany jako zasób konfiguracji. Metoda ta jest użyteczna w przypadku spakowanej biblioteki zawierającej komponenty i obiekty wyświetlania.
- Parametru inicjalizacji kontekstu, `javax.faces.application.CONFIG_FILES`, który specyfikuje jedną lub więcej ścieżek (oddzielonych przecinkami) prowadzących do wielu plików konfiguracyjnych dla danej aplikacji. Metoda ta powinna być stosowana w przypadku rozbudowanych aplikacji, które delegują odpowiedzialność za konfigurację poszczególnych swoich części do różnych grup architektów aplikacji.
- Zasobu o nazwie *faces-config.xml* w katalogu */WEB-INF/* aplikacji. W ten sposób używają plików konfiguracyjnych najprostsze aplikacje.

Dostęp do zasobów zarejestrowanych dla aplikacji wymaga od programisty aplikacji użycia instancji klasy `Application`, która jest automatycznie tworzona dla każdej aplikacji. Instancja `Application` działa jako scentralizowana fabryka zasobów zdefiniowanych w pliku XML.

Podczas uruchamiania aplikacji implementacja JavaServer Faces tworzy pojedynczą instancję klasy `Application` i konfiguruje ją, wykorzystując informacje zawarte w pliku konfiguracyjnym zasobów aplikacji.

Konfigurowanie komponentów

Do tworzenia komponentów pomocniczych używanych przez aplikacje JavaServer Faces i umieszczenia ich w wybranym zakresie używamy mechanizmu tworzenia komponentów zarządzanych. Mechanizm ten konfigurujemy w pliku konfiguracyjnym zasobów aplikacji, używając elementu `managed-bean` do zdefiniowania każdego komponentu. Plik ten zostaje przetworzony podczas uruchamiania aplikacji. Gdy strona odwołuje się do komponentu, to implementacja JavaServer Faces inicjalizuje go zgodnie z konfiguracją zapisaną w pliku konfiguracyjnym zasobów aplikacji.

Korzystając z mechanizmu tworzenia komponentów zarządzanych, możemy:

- Tworzyć komponenty za pomocą jednego, scentralizowanego pliku dostępnego dla całej aplikacji, zamiast warunkowo tworzyć instancje komponentów w różnych częściach aplikacji.
- Indywidualizować właściwości komponentów bez tworzenia dodatkowego kodu.

- Indywidualizować wartości właściwości komponentów podczas tworzenia komponentu bezpośrednio za pomocą pliku konfiguracyjnego.
- Za pomocą elementów `value` nadawać właściwości jednego komponentu zarządzanego wartość będącą wynikiem opracowania wyrażenia wiążącego wartość dla innego komponentu.

W niniejszym podrozdziale przedstawimy sposób inicjalizacji komponentów pomocniczych za pomocą mechanizmu tworzenia komponentów zarządzanych. Sposób implementacji właściwości komponentów pomocniczych wyjaśniliśmy w podrozdziale „Implementacja właściwości komponentu" w rozdziale 19. W podrozdziale „Implementacja metod komponentu pomocniczego", zamieszczonym w rozdziale 19., omówiliśmy sposób implementacji metod komponentu pomocniczego. Sposób odwołań znaczników komponentów do komponentów zarządzanych przedstawiliśmy w podrozdziale „Wiązanie wartości i instancji komponentów z zewnętrznymi źródłami danych" (rozdział 18.).

Stosowanie elementu managed-bean

Komponenty pomocnicze tworzymy za pomocą elementu `managed-bean` reprezentującego instancję klasy komponentu, która musi istnieć dla danej aplikacji. Element `managed-bean` jest przetwarzany przez implementację JavaServer Faces podczas działania aplikacji. Gdy strona aplikacji odwołuje się do komponentu, to implementacja JavaServer Faces tworzy jego instancję (jeśli nie istnieje żadna instancja komponentu) zgodnie z konfiguracją odpowiedniego elementu.

Poniżej przedstawiamy przykład konfiguracji komponentu zarządzanego dla aplikacji Księgarnia Duke'a:

```
<managed-bean>
  <managed-bean-name> NA </managed-bean-name>
    <managed-bean-class>
      model.ImageArea
    </managed-bean-class>
    <managed-bean-scope> application </managed-bean-scope>
    <managed-property>
      <property-name>shape</property-name>
      <value>poly</value>
    </managed-property>
    ...
  </managed-bean-name>
</managed-bean>
```

Element `managed-bean-name` definiuje klucz komponentu używany w danym zakresie. Jeśli tworzymy odwzorowanie komponentu interfejsu użytkownika na komponent pomocniczy, to atrybut `value` znacznika komponentu musi odpowiadać elementowi `managed-bean-name` aż do pierwszego znaku kropki. Na przykład poniższe wyrażenie odpowiada właściwości `shape` instancji `NA` klasy `ImageArea`:

```
value="#{NA.shape}"
```

Część wyrażenia przed znakiem kropki odpowiada elementowi `managed-bean-name` dla komponentu pomocniczego `ImageArea`. Więcej przykładów użycia atrybutu `value` do wiązania komponentów z właściwościami można znaleźć w podrozdziale „Stosowanie znaczników komponentów HTML" zamieszczonym w rozdziale 18.

Element `managed-bean-class` definiuje pełną nazwę klasy komponentu JavaBean używaną do tworzenia instancji komponentu. Obowiązkiem programisty aplikacji jest zapewnienie, żeby klasa ta odpowiadała konfiguracji komponentu w pliku konfiguracyjnym zasobów aplikacji. Przykładowo, definicje właściwości muszą zgadzać się ze skonfigurowanymi dla danego komponentu.

Element `managed-bean-scope` definiuje zakres, w którym przechowywany będzie komponent. Możliwe są cztery zakresy: `none`, `request`, `session` lub `application`. Jeśli zdefiniujemy komponent o zakresie `none`, to za każdym odwołaniem do tego komponentu będzie tworzona jego nowa instancja, która nie jest przechowywana w żadnym zakresie. Jedną z sytuacji, w której używamy zakresu `none`, jest odwołanie jednego komponentu zarządzanego do innego komponentu zarządzanego. Jeśli komponent ten powinien być tworzony tylko wtedy, gdy występuje odwołanie do niego, to powinien właśnie posiadać zakres `none`. Przykład inicjalizacji właściwości komponentu zarządzanego można znaleźć w podpunkcie „Inicjalizacja właściwości komponentów zarządzanych".

Element `managed-bean` może zawierać zero lub więcej elementów `managed-property`, z których każdy odpowiada jednej właściwości zdefiniowanej w klasie komponentu. Elementy są używane do zainicjalizowania wartości właściwości komponentu pomocniczego. Jeśli nie chcemy, aby określona właściwość została zainicjalizowana podczas tworzenia instancji komponentu pomocniczego, to nie umieszczamy dla niej definicji `managed-property` w pliku konfiguracyjnym zasobów aplikacji.

Jeśli element `managed-bean` nie zawiera innych elementów `managed-bean`, to może zawierać jeden element `map-entries` lub `list-entries`. Element `map-entries` konfiguruje zbiór komponentów będących instancjami typu `Map`. Element `list-entries` konfiguruje zbiór komponentów będących instancjami typu `List`.

Aby stworzyć odwzorowanie na właściwość zdefiniowaną za pomocą elementu `managed-property`, musimy zagwarantować, że część wyrażenia umieszczonego w atrybucie `value` poprzedzająca znak kropki zgadza się z elementem `property-name` elementu `managed-property`. We wcześniejszym przykładzie właściwość `shape` została zainicjalizowana wartością `poly`. W następnym podrozdziale omówimy bardziej szczegółowo posługiwanie się elementem `managed-property`.

Inicjalizacja właściwości za pomocą elementu managed-property

Element `managed-property` musi zawierać element `property-name`, który musi zgadzać się z nazwą danej właściwości w klasie komponentu. Element `managed-property` musi zawierać również jeden ze zbioru elementów (przedstawionych w tabeli 21.1) definiujący wartość właściwości. Wartość ta musi być takiego samego typu, jaki został zdefiniowany dla tej właściwości w klasie komponentu. Wybór elementu definiującego wartość zależy od typu właściwości zdefiniowanego w klasie komponentu. Wszystkie elementy używane do inicjalizacji wartości przedstawione zostały w tabeli 21.1.

Przykład inicjalizacji właściwości typu `String` za pomocą elementu `value` przedstawiliśmy w punkcie „Stosowanie elementu managed-bean". Elementu podrzędnego `value` możemy również używać do inicjalizowania wartości typów prostych i innych typów referencyjnych.

Tabela 21.1. *Elementy podrzędne elementu managed-property definiujące wartość właściwości*

Element	Wartość, którą definiuje
`list-entries`	Definiuje wartości listy
`map-entries`	Definiuje wartości mapy
`null-value`	Jawnie nadaje właściwości wartość `null`
`value`	Definiuje pojedynczą wartość, na przykład typu `String` lub `int`, bądź wyrażenie JavaServer Faces

Pozostała część tego podrozdziału opisuje sposób użycia elementu podrzędnego `value` i innych elementów podrzędnych do inicjalizacji właściwości typu `java.util.Map` tablicy oraz `Collection`, a także parametrów inicjalizacji.

Odwołania do parametru inicjalizacji

Kolejną ważną zaletą mechanizmu tworzenia komponentów zarządzanych jest możliwość odwołań do obiektów niejawnych przez właściwości komponentu.

Załóżmy, że pewna strona akceptuje dane wprowadzane przez użytkownika, w tym jego adres. Załóżmy również, że większość użytkowników mieszka w rejonie o takim samym kodzie pocztowym. Komponent kodu pocztowego może wyświetlić ten kod, przechowując go w obiekcie niejawnym i odwołując się do niego podczas wyświetlania strony.

Kod możemy zapamiętać jako początkową, domyślną wartość niejawnego obiektu kontekstu `initParam`, dodając parametr kontekstu do aplikacji i nadając mu wartość za pomocą programu *deploytool*. Na przykład, aby nadać parametrowi kontekstu o nazwie `defaultAreaCode` wartość `650`, uruchomimy *deploytool*, wybierzemy naszą aplikację z drzewa, wybierzemy zakładkę `Context`, dodamy nowy parametr kontekstu i wprowadzimy `defaultAreaCode` w polu *Coded Parameter* i `650` w polu *Value*.

Następnie musimy stworzyć deklarację `managed-bean`, która skonfiguruje właściwość odwołującą się do tego parametru:

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>CustomerBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>areaCode</property-name>
    <value-ref>initParam.defaultAreaCode</value-ref>
  </managed-property>
  ...
</managed-bean>
```

Dostęp do kodu podczas wyświetlania strony wymaga odwołania się do właściwości przez atrybut `value` znacznika komponentu `area`:

```
<h:inputText id=area value="#{customer.areaCode}"
```

Pobieranie wartości z innych obiektów niejawnych odbywa się w podobny sposób. Listę obiektów niejawnych zawiera tabela 18.9.

Inicjalizacja właściwości będących mapą

Element `map-entries` użyty wewnątrz elementu `managed-property` służy inicjalizacji wartości właściwości komponentu pomocniczego, która posiada typ `java.util.Map`. Poniżej przedstawiamy definicję elementu `map-entries` pochodzącą z pliku *web-facesconfig_1_0.dtd* umieszczonego pod adresem *http://java.sun.com/dtd/ web-facesconfig_1_0.dtd* i definiującego plik konfiguracyjny zasobów aplikacji:

```
<!ELEMENT map-entries (key-class?, value-class?, map-entry*) >
```

Definicja ta pokazuje, że element `map-entries` może zawierać opcjonalny element `key-class`, opcjonalny element `value-class` oraz zero lub więcej elementów `map-entry`.

Element `map-entry` zdefiniowany został następująco:

```
<!ELEMENT map-entry (key, (null-value|value )) >
```

Poniżej przedstawiamy przykład użycia elementu `map-entries`:

```
<managed-bean>
  ...
  <managed-property>
    <property-name>prices</property-name>
    <map-entries>
      <map-entry>
        <key>My Early Years: Growing Up on *7</key>
        <value>30.75</value>
      </map-entry>
      <map-entry>
        <key>Web Servers for Fun and Profit</key>
        <value>40.75</value>
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

Mapa utworzona przez powyższy element `map-entries` zawiera dwie pozycje. Domyślnie wszystkie klucze i wartości mapy przekształcane są na typ `java.lang.String`. Chcąc używać kluczy innego typu, musimy zagnieździć element `key-class` wewnątrz elementu `map-entries`:

```
<map-entries>
  <key-class>java.math.BigDecimal</key-class>
  ...
</map-entries>
```

Powyższa deklaracja spowoduje przekształcenie wszystkich kluczy na typ `java.math.BigDecimal`. Oczywiście sami musimy zadbać o to, by przekształcenie takie było możliwe. Klucz użyty w poprzednim przykładzie jest typu `String` i nie może zostać przekształcony na typ `java.math.BigDecimal`.

Jeśli chcemy, aby również wszystkie wartości na mapie były innego typu, to po elemencie `key-class` umieszczamy element `value-class`:

```
<map-entries>
  <key-class>int</key-class>
  <value-class>java.math.BigDecimal</value-class>
  ...
</map-entries>
```

Zwróćmy uwagę, że element ten konfiguruje typ tylko wszystkich elementów podrzędnych `value`.

Pierwszy element `map-entry` w ostatnim przykładzie zawiera element podrzędny `value`. Element ten definiuje pojedynczą wartość, która zostanie przekształcona na typ określony przez komponent.

Drugi element `map-entry` definiuje element `value`, który odwołuje się do właściwości innego komponentu. Odwołanie do innego komponentu przez właściwość komponentu pozwala tworzyć system złożony z precyzyjnie zdefiniowanych obiektów. Na przykład obiekt obsługi formularza należący do zakresu żądania może posiadać wskaźnik na obiekt obsługi bazy danych umieszczony w zakresie aplikacji. Oba te obiekty razem mogą obsługiwać formularz. Zauważmy przy tym, że umieszczenie odwołania do innego komponentu spowoduje utworzenie jego instancji, jeśli ona jeszcze nie istnieje.

Zamiast używać elementu `map-entries`, można również przypisać całą mapę, wykorzystując element `value` zawierający wyrażenie, którego typem jest mapa.

Inicjalizacja właściwości będących tablicami lub listami

Element `values` używany jest do zainicjalizowania wartości właściwości typu tablicowego lub typu `List`. Każda pojedyncza wartość tablicy lub listy zostaje zainicjalizowana za pomocą elementu `value` lub `null-value`. Oto przykład:

```
<managed-bean>
  ...
  <managed-property>
    <property-name>books</property-name>
    <values>
      <value-type>java.lang.String</value-type>
      <value>Web Servers for Fun and Profit</value>
      <value>#{myBooks.bookId[3]}</value>
      <null-value/>
    </values>
  </managed-property>
</managed-bean>
```

Powyższy przykład inicjalizuje właściwość typu tablicowego lub `List`. Typ utworzonej struktury danych jest określony przez typ odpowiadającej jej właściwości komponentu. Element `values` definiuje listę wartości należących do tablicy lub listy. Element `value` określa pojedyncza wartość tablicy bądź listy i może odwoływać się do właściwości innego komponentu. Element `null-value` spowoduje wywołanie metody `setBooks` z argumentem równym `null`. Wartość `null` nie może zostać podana dla właściwości będącej typu prostego języka Java, na przykład `int` lub `boolean`.

Inicjalizacja właściwości komponentów zarządzanych

Czasami może zdarzyć się sytuacja, w której będziemy chcieli stworzyć komponent odwołujący się do innych komponentów zarządzanych, tworząc w ten sposób graf lub drzewo komponentów. Załóżmy na przykład, że chcemy stworzyć komponent reprezentujący informacje o kliencie, w tym adres do korespondencji i adres zameldowania, które także są komponentami. Przedstawiona poniżej deklaracja `managed-bean` tworzy instancję klasy `CustomerBean`, która posiada dwie właściwości typu `AddressBean`: jedną reprezentującą adres do korespondencji i drugą reprezentującą adres zameldowania. W ten sposób powstaje proste drzewo, którego korzeniem jest komponent `CustomerBean`, a liśćmi dwa komponenty `AddressBean`.

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>
    com.mycompany.mybeans.CustomerBean
  </managed-bean-class>
  <managed-bean-scope> request </managed-bean-scope>
  <managed-property>
    <property-name>mailingAddress</property-name>
    <value>addressBean</value>
  </managed-property>
  <managed-property>
    <property-name>streetAddress</property-name>
    <value>addressBean</value>
  </managed-property>
  <managed-property>
    <property-name>customerType</property-name>
    <value>New</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>addressBean</managed-bean-name>
  <managed-bean-class>
    com.mycompany.mybeans.AddressBean
  </managed-bean-class>
  <managed-bean-scope> none </managed-bean-scope>
  <managed-property>
    <property-name>street</property-name>
    <null-value/>
  <managed-property>
  ...
</managed-bean>
```

Pierwsza deklaracja komponentu `CustomerBean` (czyli zawierająca element `managed-bean-name customer`) tworzy komponent `CustomerBean` dostępny w zakresie żądania. Komponent ten posiada dwie właściwości: `mailingAddress` i `streetAddress`. Właściwości te używają elementu `value`, odwołując się do komponentu o nazwie `addressBean`.

Druga deklaracja komponentu zarządzanego definiuje komponent klasy `AddressBean`, ale nie tworzy go, ponieważ element `managed-bean-scope` definiuje zakres `none`. Przypomnijmy, że zakres `none` oznacza, że komponent tworzony jest tylko wtedy, gdy wystąpi odwołanie do niego. Ponieważ zarówno właściwość `mailingAddress`, jak i `streetAddress` odwołują się do `adressBean` za pomocą elementu `value`, to podczas tworzenia komponentu klasy `CustomerBean` zostają również utworzone dwie instancje klasy `AddressBean`.

Tworząc obiekt odwołujący się do innego obiektu, należy unikać odwoływania się do obiektów o krótszym okresie życia, ponieważ uniemożliwia to odzyskanie zasobów zakresu, który przestaje istnieć. Na przykład obiekt należący do zakresu sesji nie może odwoływać się do obiektu istniejącego w zakresie żądania. Obiekty należące do zakresu `none` nie mają określonego czasu istnienia i dlatego mogą odwoływać się jedynie do obiektów tego samego zakresu. Dozwolone odwołania przedstawione zostały w tabeli 21.2.

Tabela 21.2. *Dozwolone odwołania pomiędzy obiektami różnych zakresów*

Obiekt należący do zakresu	Może wskazywać obiekt należący do zakresu
`none`	`none`
`application`	`none, application`
`session`	`none, application, session`
`request`	`none, application, session, request`

Inicjalizacja map i list

Oprócz konfigurowania właściwości typu `Map` i `List` można również konfigurować bezpośrednio obiekty typu `Map` oraz `List` i odwoływać się do nich bezpośrednio ze znacznika zamiast do właściwości obudowujących obiekty typu `Map` i `List`.

Aplikacja Księgarnia Duke'a konfiguruje obiekt typu `List`, aby zainicjalizować listę darmowych biuletynów, spośród których użytkownik może subskrybować wybrane na stronie *bookcashier.jsp*:

```
<managed-bean>
  ...
<managed-bean-name>newsletters</managed-bean-name>
  <managed-bean-class>
    java.util.ArrayList
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value-class>javax.faces.model.SelectItem</value-class>
    <value>#{newsletter0}</value>
    <value>#{newsletter1}</value>
    <value>#{newsletter2}</value>
    <value>#{newsletter3}</value>
  </list-entries>
</managed-bean>
<managed-bean>
  <managed-bean-name>newsletter0</managed-bean-name>
  <managed-bean-class>
    javax.faces.model.SelectItem
  </managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <managed-property>
    <property-name>label</property-name>
    <value>Duke's Quarterly</value>
  </managed-property>
  <managed-property>
```

```
    <property-name>value</property-name>
    <value>200</value>
  </managed-property>
</managed-bean>
...
```

Powyższa konfiguracja inicjalizuje obiekt typu `List` o nazwie `newsletters`. Lista ta składa się z instancji typu `SelectItem` będących również komponentami zarządzanymi. Więcej informacji na temat klasy `SelectItem` można znaleźć w punkcie „Komponenty UISelectItem, UISelectItems i UISelectItemGroup" zamieszczonym w rozdziale 18. Zwróćmy uwagę, że, w przeciwieństwie do przykładu zamieszczonego w podpunkcie „Inicjalizacja właściwości będących mapą", tym razem lista biuletynów nie jest właściwością komponentu zarządzanego (nie jest obudowana elementem `managed-property`). Lista ta jest teraz sama komponentem zarządzanym.

Rejestracja komunikatów

Jeśli chcemy używać komunikatów niestandardowych, to musimy udostępnić je aplikacji podczas jej uruchamiania. Możemy to osiągnąć na dwa sposoby: umieszczając komunikat w kolejce instancji `FacesContext` (patrz podrozdział „Lokalizacja" w rozdziale 19.) lub rejestrując komunikat w pliku konfiguracyjnym zasobów aplikacji.

Poniżej przedstawiamy fragment pliku rejestrującego komunikaty dla aplikacji Księgarnia Duke'a:

```
<application>
  <message-bundle>
    resource.ApplicationMessages
  </message-bundle>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>fr</supported-locale>
    <supported-locale>es</supported-locale>
  </locale-config>
</application>
```

Powyższy zbiór elementów spowoduje umieszczenie w instancji `Application` komunikatów znajdujących się w podanym zestawie `ResourceBundle` o nazwie `resources.ApplicationMessages`.

Element `message-bundle` reprezentuje zbiór zlokalizowanych komunikatów i musi zawierać pełną ścieżkę dostępu do zestawu `ResourceBundle` zawierającego zlokalizowane komunikaty, `resource.ApplicationMessages` w tym przypadku.

Element `locale-config` zawiera listę obsługiwanych lokalizacji, w tym lokalizacji domyślnej. Element ten umożliwia systemowi ustalenie właściwej lokalizacji na podstawie konfiguracji języka dla przeglądarki. Aplikacja Księgarnia Duke'a umożliwia ręczny wybór lokalizacji i w związku z tym zastosowanie elementu `locale-config` nie jest w jej przypadku konieczne.

Elementy `supported-locale` i `default-locale` akceptują dwuliterowe kody zdefiniowane przez standard ISO-639 (patrz *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*). Warto zawsze sprawdzać, czy rzeczywiście umieściliśmy w zestawie `ResourceBundle` komunikaty dla lokalizacji, które określiliśmy za pomocą tych elementów.

Dostęp do zlokalizowanego komunikatu umieszczonego w zestawie zasobów wymaga od programisty aplikacji użycia odpowiedniego klucza (patrz podrozdział „Lokalizacja" w rozdziale 19.).

Rejestracja niestandardowego walidatora

Jeśli programista aplikacji dostarcza własnej implementacji interfejsu `Validator`, to musi zarejestrować niestandardowy walidator w pliku konfiguracyjnym zasobów aplikacji za pomocą elementu `validator`:

```
<validator>
  ...
  <validator-id>FormatValidator</validator-id>
  <validator-class>validators.FormatValidator</validator-
class>
  <attribute>
    ...
    <attribute-name>formatPatterns</attribute-name>
    <attribute-class>java.lang.String</attribute-class>
  </attribute>
</validator>
```

Elementy podrzędne `validator-id` i `validator-class` są obowiązkowe. Element `validator-id` reprezentuje identyfikator, pod którym zostanie zarejestrowana klasa walidatora. Identyfikator ten jest używany przez klasę znacznika odpowiadającą niestandardowemu znacznikowi `validator`.

Element `validator-class` reprezentuje pełną nazwę klasy implementującej interfejs `Validator`.

Element `attribute` identyfikuje atrybut związany z implementacją interfejsu `Validator`. Posiada obowiązkowe elementy podrzędne `attribute-name` i `attribute-class`. Element `attribute-name` odnosi się do nazwy atrybutu określonej przez znacznik `validator`, natomiast `attribute-class` identyfikuje typ wartości związanej z atrybutem.

Sposób implementacji interfejsu `Validator` został omówiony w podrozdziale „Implementacja niestandardowego walidatora" (rozdział 19.).

Sposób odwołań do walidatora na stronach aplikacji wyjaśniliśmy w punkcie „Stosowanie niestandardowego walidatora" (rozdział 18.).

Rejestracja niestandardowego konwertera

Podobnie jak w przypadku niestandardowego walidatora również niestandardowy konwerter wymaga rejestracji. Poniżej przedstawiamy konfigurację elementu `converter` dla konwertera `CreditCardConverter` używanego przez aplikację Księgarnia Duke'a:

```
<converter>
  <description>
    Rejestruje implementację konwertera,
    converters.CreditCardConverter używając
    identyfikatora creditcard.

  </description>
  <converter-id>creditcard</converter-id>
  <converter-class>
    converters.CreditCardConverter
  </converter-class>
</converter>
```

Element `converter` reprezentuje implementację interfejsu `Converter` i zawiera obowiązkowe elementy podrzędne `converter-id` i `converter-class`.

Element `converter-id` określa identyfikator używany przez atrybut `converter` znacznika komponentu interfejsu użytkownika w celu zastosowania konwertera do przekształcenia danych komponentu. Przykład odwołania do niestandardowego konwertera przez znacznik komponentu zamieściliśmy w punkcie „Stosowanie niestandardowego konwertera" w rozdziale 18.

Element `converter-class` identyfikuje klasę implementującą interfejs `Converter`.

Sposób implementacji niestandardowego konwertera omówiliśmy w podrozdziale „Implementacja niestandardowego konwertera" w rozdziale 19.

Konfigurowanie reguł nawigacji

W podrozdziale „Model nawigacji" zamieszczonym w rozdziale 17. wyjaśniliśmy, że konfigurowanie nawigacji polega na utworzeniu zbioru reguł umożliwiających wybór kolejnej strony na skutek kliknięcia przycisku lub łącza. Reguły nawigacji definiuje się w pliku konfiguracyjnym zasobów aplikacji.

Każda reguła nawigacji określa sposób przejścia od określonej strony do zbioru innych stron. Implementacja JavaServer Faces wybiera odpowiednią regułę nawigacji dla aktualnie wyświetlanej strony.

Po wybraniu właściwej reguły nawigacji wybór kolejnej strony zależy od metody akcji wywołanej na skutek kliknięcia komponentu oraz wyniku logicznego zwróconego przez te metodę lub podanego przez znacznik komponentu.

Wynik logiczny może być dowolnie zdefiniowany przez programistę aplikacji. W tabeli 21.3 przedstawione zostały najczęściej używane wyniki.

Tabela 21.3. *Typowe łańcuchy wyników logicznych*

Wynik	Znaczenie
`success`	Wszystko przebiegło poprawnie. Przejdź do kolejnej strony.
`failure`	Wystąpił błąd. Przejdź do strony błędu.
`logon`	Użytkownik powinien się najpierw zalogować. Przejdź do strony logowania.
`no results`	Brak wyników wyszukiwania. Przejdź ponownie do strony wyszukiwania.

Zwykle metoda akcji przetwarza dane formularza bieżącej strony. Może na przykład sprawdzać, czy nazwa użytkownika i hasło wprowadzone w formularzu zgadzają się z zapisanymi w pliku. Jeśli tak, to metoda zwraca wynik `success`. W przeciwnym razie zwraca `failure`. Przykład ten pokazuje, że na ustalenie następnej strony ma wpływ metoda przetwarzająca akcję oraz zwracany przez nią wynik logiczny.

Poniżej przedstawiamy regułę nawigacji, która mogłaby zostać użyta w omówionym właśnie przykładzie:

```
<navigation-rule>
  <from-view-id>/logon.jsp</from-view-id>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/storefront.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action>
    <from-outcome>failure</from-outcome>
    <to-view-id>/logon.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Reguła ta definiuje możliwe sposoby nawigacji dla strony *logon.jsp*. Każdy element `navigation-case` definiuje osobny przypadek ścieżki nawigacji dla tej strony. Pierwszy element `navigation-case` oznacza, że jeśli metoda `LogonForm.logon` zwróci wynik `success`, to należy przejść do strony *storefront.jsp*. Drugi element `navigation-case` sugeruje, że gdy metoda `LogonForm.logon` zwróci wynik `failure`, to wyświetlona zostanie strona *logon.jsp*.

Konfiguracja nawigacji dla danej aplikacji jest zbiorem reguł nawigacji. Każda taka reguła jest zdefiniowana przez element `navigation-rule` w pliku *faces-config.xml*.

Reguły nawigacji zastosowane dla aplikacji Księgarnia Duke'a są bardzo proste. Poniżej przedstawiamy dwie bardziej skomplikowane reguły, które mogłyby zostać użyte dla tej aplikacji:

```
<navigation-rule>
  <from-view-id>/catalog.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/bookcashier.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>out of stock</from-outcome>
    <from-action>
```

```
      #{catalog.buy}
    </from-action>
    <to-view-id>/outofstock.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>error</from-outcome>
    <to-view-id>/error.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Pierwsza reguła nawigacji w tym przykładzie oznacza, że aplikacja może przejść od strony *catalog.jsp* do strony:

- *bookcashier.jsp*, jeśli zamawiana pozycja jest na stanie,
- *outofstock.jsp*, gdy zamawianej pozycji nie ma na stanie.

Druga reguła nawigacji oznacza, że wystąpienie błędu spowoduje przejście od dowolnej strony do strony *error.jsp*.

Każdy element `navigation-rule` odpowiada jednemu identyfikatorowi drzewa komponentów zdefiniowanemu przez opcjonalny element `from-view-id`. Oznacza to, że każda reguła definiuje wszystkie możliwe sposoby nawigacji dla określonej strony aplikacji. Jeśli element `from-view-id` nie został zdefiniowany, to reguły nawigacji zdefiniowane przez element `navigation-rule` odnoszą się do wszystkich stron aplikacji. Element `from-view-id` umożliwia również stosowanie wzorców. Na przykład przedstawiony poniżej element `from-view-id` oznacza, że reguła nawigacji odnosi się do wszystkich stron umieszczonych w katalogu *books*:

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

Przedstawiony przykład reguł nawigacji pokazuje, że element `navigation-rule` może zawierać zero lub więcej elementów `navigation-case`. Element `navigation-case` definiuje zbiór kryteriów. Jeśli zostaną one spełnione, to aplikacja przejdzie do strony zdefiniowanej przez element `to-view-id` zagnieżdżony w tym samym elemencie `navigation-case`.

Kryteria nawigacji definiowane są za pomocą opcjonalnych elementów `from-outcome` i `from-action`. Element `from-outcome` definiuje wynik logiczny, na przykład `success`. Element `from-action` używa wyrażenia wiążącego metodę do odwołania się do metody akcji zwracającej łańcuch `String` będący wynikiem logicznym.

Elementy `navigation-case` porównywane są z wynikiem i wyrażeniem wiążącym metodę w następującej kolejności:

- Przypadki określające wartość `from-outcome` i `from-action`. Oba te elementy mogą zostać użyte jeśli metoda akcji zwraca różne wyniki w oparciu o wykonywane przez nią operacje.
- Przypadki określające jedynie wartość `from-outcome`. Element `from-outcome` musi zgadzać się albo z wynikiem zdefiniowanym przez atrybut `action` komponentu `UICommand` albo z wynikiem zwróconym przez metodę, do której odwołuje się ten komponent.
- Przypadki określające jedynie wartość `from-action`. Wartość ta musi zgadzać się z wyrażeniem `action` określonym przez znacznik komponentu.

Jeśli spełniony zostanie jeden z tych przypadków, to do wyświetlania zostanie wybrane drzewo komponentów zdefiniowane przez element `to-view-id`.

Sposób użycia atrybutu `action` znacznika komponentu odwołującego się do metody akcji wyjaśniliśmy w punkcie „Odwołanie do metody nawigacji" (rozdział 18.). Sposób implementacji metody akcji przedstawiliśmy w punkcie „Implementacja metody nawigacji" w rozdziale 19.

Rejestracja niestandardowej klasy wyświetlania w pakiecie wyświetlania

Pakiet wyświetlania definiuje dla każdego obsługiwanego komponentu interfejsu użytkownika zbiór obiektów `Renderer` wyświetlających komponent w różny sposób u klienta obsługiwanego przez ten pakiet. Na przykład standardowy komponent klasy `UISelectOne` definiuje komponent, który umożliwia użytkownikowi wybór jednego elementu ze zbioru elementów. Komponent ten może być wyświetlany przez klasę wyświetlania `Listbox`, `Menu` lub `Radio`. Każda z tych klas tworzy inny wygląd komponentu. Klasa `Listbox` tworzy menu, które może wyświetlać cały zbiór wartości. Klasa `Menu` wyświetla jednocześnie tylko podzbiór wszystkich elementów. Klasa `Radio` wyświetla zbiór przycisków wyboru.

Gdy programista aplikacji tworzy niestandardową klasę wyświetlania (patrz „Delegowanie wyświetlania do klasy wyświetlania" w rozdziale 20.), to musi zarejestrować ją we właściwym pakiecie wyświetlania. Ponieważ niestandardowy komponent mapy graficznej został zaimplementowany jako komponent HTML, to klasa wyświetlania `AreaRenderer` (a także `MapRenderer`) powinna być zarejestrowana w pakiecie wyświetlania HTML.

Klasę wyświetlania rejestrujemy w pliku konfiguracyjnym zasobów aplikacji za pomocą elementu `render-kit`. Poniżej przedstawiamy przykład konfiguracji takiego elementu dla klasy `AreaRenderer` należącej do aplikacji Księgarnia Duke'a:

```
<render-kit>
  <renderer>
    <renderer-type>DemoArea</renderer-type>
    <renderer-class>
      renderkit.AreaRenderer
    </renderer-class>
    <attribute>
      <attribute-name>onmouseout</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>onmouseover</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <attribute>
      <attribute-name>styleClass</attribute-name>
      <attribute-class>java.lang.String</attribute-class>
    </attribute>
    <supported-component-class>
      <component-class>
        components.AreaComponent
```

```
      </component-class>
    </supported-component-class>
  </renderer>
  ...
```

Element `render-kit` reprezentuje implementację `RenderKit`. Jeśli element ten nie zostanie wyspecyfikowany, to domyślnie zakładany jest pakiet wyświetlania HTML. Element `renderer` reprezentuje implementację `Renderer`. Zagnieżdżając element `renderer` wewnątrz elementu `render-kit`, rejestrujemy klasę wyświetlania w pakiecie wyświetlania `RenderKit` związanym z elementem `render-kit`.

Sposób użycia elementu `renderer-type` przez obiekt obsługi znacznika wyjaśniamy w następnym podrozdziale. Element `renderer-class` określa pełną nazwę klasy implementującej interfejs `Renderer`.

Elementy `component-family` i `render-type` są używane przez komponent do wyszukania odpowiedniego obiektu wyświetlania. Identyfikator `component-family` musi zgadzać się z wartością zwracaną przez metodę `getFamily` klasy komponentu. Identyfikator `render-type` musi zgadzać się z wartością zwracaną przez metodę `getRendererType` klasy obsługi znacznika. Element `attribute` nie jest używany podczas wykonania aplikacji, a jedynie dostarcza programom narzędziowym informacji o atrybutach obsługiwanych przez implementację `Renderer`.

Rejestracja niestandardowego komponentu

Niestandardowe komponenty również wymagają rejestracji w pliku konfiguracyjnym zasobów aplikacji. Poniżej przedstawiamy element `component` rejestrujący komponent klasy `AreaComponent`:

```
<component>
  <component-type>DemoArea</component-type>
  <component-class>
    components.AreaComponent
  </component-class>
  <property>
    <property-name>alt</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>coords</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <property>
    <property-name>shape</property-name>
    <property-class>java.lang.String</property-class>
  </property>
  <component-extension>
    <component-family>Area</component-family>
    <renderer-type>DemoArea</renderer-type>
  </component-extension>
</component>
```

Element `component-type` określa nazwę, pod którą zarejestrowany zostanie komponent. Inne obiekty, odwołując się do komponentu, będą używać właśnie tej nazwy. Element `component-class` określa pełną nazwę klasy komponentu. Elementy `property` specyfikują właściwości komponentu i ich typy.

Element `component-extension` identyfikuje zbiór komponentów i ich obiektów wyświetlania. Identyfikator `component-family` musi zgadzać się z wartością zwracaną przez metodę `getFamily` komponentu. Identyfikator `renderer-type` musi zgadzać się z wartością zwracaną przez metodę `getRendererType` obiektu obsługi znacznika. Takie rozwiązanie umożliwia wyświetlanie komponentu przez różne obiekty wyświetlania i wyświetlanie różnych komponentów przez ten sam obiekt wyświetlania.

Podstawowe wymagania aplikacji JavaServer Faces

Oprócz poprawnego skonfigurowania aplikacji musimy spełnić także inne wymagania związane z odpowiednim spakowaniem wymaganych plików i dostarczeniem pliku deskryptora instalacji. W niniejszym podrozdziale omówimy sposób wykonania tych zadań.

Aplikacja JavaServer Faces musi być zgodna ze specyfikacją Servlet 2.3 (lub nowszą) oraz specyfikacją JavaServer Pages 1.2 (lub nowszą). Wszystkie aplikacje zgodne z tymi specyfikacjami są pakowane w pliku WAR, który musi spełniać określone wymagania, aby możliwe było jego wykonanie przez różne kontenery. Plik WAR aplikacji JavaServer Faces musi zawierać co najmniej:

- deskryptor instalacji aplikacji internetowej o nazwie *web.xml* konfigurujący zasoby wymagane przez aplikację internetową,
- zbiór plików JAR zawierających podstawowe klasy,
- zbiór klas aplikacji, stron JavaServer Faces i innych wymaganych zasobów takich jak pliki graficzne,
- plik konfiguracyjny zasobów aplikacji.

Plik WAR ma zwykle następującą strukturę katalogów:

```
index.html
strony JSP
WEB-INF/
   web.xml
   faces-config.xml
   deskryptory bibliotek znaczników (opcjonalne)
   classes/
      pliki klas
      pliki Properties
   lib/
      pliki JAR
```

Plik *web.xml* (deskryptor instalacji), zbiór plików JAR i zbiór plików aplikacji muszą być umieszczone w katalogu *WEB-INF* pliku WAR. Zwykle do skompilowania klas aplikacji używa się programu *asant*. Pliki pakuje się do pliku WAR za pomocą programu *deploytool*, który umożliwia również instalację pliku WAR.

Programy *asant* i *deploytool* są dołączone do serwera Sun Java System Application Server Platform Edition 8. Sposób tworzenia pliku WAR przez program *asant* konfigurujemy za pomocą pliku *build.xml*. Każdy z przykładów omawianych w tej książce posiada własny plik *build.xml*, na którym można się wzorować, tworząc własny plik.

Konfigurowanie aplikacji za pomocą programu deploytool

Aplikacje internetowe są konfigurowane za pomocą elementów umieszczonych w deskryptorze instalacji aplikacji internetowej. Program *deploytool* generuje taki deskryptor, gdy tworzymy plik WAR i umieszcza w nim elementy, gdy tworzymy komponenty aplikacji i związane z nimi klasy. Elementy te możemy modyfikować za pomocą inspektorów związanych z plikami WAR.

Deskryptor instalacji dla aplikacji JavaServer Faces musi konfigurować:

- serwlet używany do przetwarzania żądań JavaServer Faces,
- odwzorowanie serwletu przetwarzającego żądania,
- ścieżkę dostępu do pliku konfiguracyjnego zasobów jeśli nie został on umieszczony w domyślnym katalogu.

Deskryptor instalacji może również:

- określać miejsce przechowywania stanu komponentów,
- ograniczać dostęp do stron zawierających znaczniki JavaServer Faces,
- włączać kontrolę poprawności dokumentów XML,
- weryfikować niestandardowe obiekty.

W tym podrozdziale przedstawimy więcej szczegółów związanych z realizacją wymienionych zadań za pomocą programu *deploytool*.

Identyfikacja serwletu przetwarzającego żądania

Jedno z wymagań odnośnie aplikacji JavaServer Faces polega na tym, aby wszystkie żądania dotyczące komponentów aplikacji przechodziły przez serwlet `FacesServlet`. Instancja klasy `FacesServlet` zarządza cyklem przetwarzania żądań przez aplikację internetową i inicjalizuje zasoby wymagane przez technologię JavaServer Faces. Aby spełnić te wymagania, należy:

1. W oknie dialogowym *Edit Contents* kreatora *Web Component* dodać do pliku WAR plik *jsf-api.jar* znajdujący się w katalogu *<J2EE_HOME>/lib/*. Plik ten jest konieczny dla zapewnienia dostępu do instancji `FacesServlet` podczas konfigurowania aplikacji za pomocą programu *deploytool*.
2. W oknie dialogowym *Choose Component Type* kreatora *Web Component* wybrać przycisk *Servlet* i kliknąć *Next*.
3. Z listy rozwijalnej *Servlet Class* wybrać klasę `FacesServlet`.
4. Na liście rozwijalnej *Startup Load Sequence Position* wprowadzić wartość `1` oznaczającą, że klasa `FacesServlet` powinna zostać załadowana podczas uruchamiania aplikacji. Klikamy *Finish*.

5. Z drzewa wybieramy komponent `FacesServlet`.
6. Wybieramy zakładkę *Aliases* i klikamy *Add*.
7. W polu *Aliases* wprowadzamy ścieżkę dostępu do `FacesServlet`. Użytkownicy aplikacji będą musieli wstawić tę ścieżkę do łańcucha URL. W przypadku aplikacji `guessNumber` ścieżką tą jest */guess/**.

Zanim aplikacja JavaServer Faces uruchomi pierwszą stronę JSP, kontener musi wywołać instancję `FacesServlet`, aby rozpocząć cykl życia stron omówiony w podrozdziale „Cykl życia strony JavaServer Faces", rozdział 17.

Aby zagwarantować wywołania instancji `FacesServlet`, tworzymy odwzorowanie w sposób opisany wyżej w punktach od 5. do 7.

Odwzorowanie to używa przedrostka pozwalającego zidentyfikować strony JSP posiadające zawartość JavaServer Faces. Dlatego też adres URL pierwszej strony musi zawierać to odwzorowanie. Istnieją dwa sposoby osiągnięcia tego celu:

- Autor strony może dołączyć do aplikacji stronę HTML, która posiada URL pierwszej strony JSP. Musi on zawierać ścieżkę do `FacesServlet` tak jak poniższy znacznik, który używa odwzorowania zdefiniowanego dla aplikacji `guessNumber`:

```
<a href="guess/greeting.jsp">
```

- Użytkownicy aplikacji mogą włączyć ścieżkę do `FacesServlet` od adresu URL pierwszej strony, wprowadzając go w przeglądarce, co ilustruje poniższy przykład dla aplikacji `guessNumber`:

```
http://localhost:8080/guessNumber/guess/greeting.jsp
```

Drugi sposób umożliwia użytkownikom rozpoczęcie pracy z aplikacją od pierwszej strony JSP zamiast od strony HTML. Wymaga jednak od użytkowników określenia pierwszej strony JSP. Jeśli wybrany zostanie pierwszy sposób, użytkownicy muszą jedynie wprowadzić:

```
http://localhost:8080/guessNumber
```

Zamiast odwzorowania za pomocą przedrostka */guess/** można również zdefiniować rozszerzenie za pomocą rozszerzenia, na przykład **.faces*. Jeśli serwer otrzyma żądanie skierowane do strony JSP o rozszerzeniu *.faces*, to kontener wyśle żądanie do instancji `FacesServlet`, który będzie oczekiwać istnienia odpowiedniej strony JSP o tej samej nazwie. Na przykład, jeśli łańcuch URL żądania będzie mieć postać *http://localhost/bookstore6/bookstore.faces*, to instancja `FacesServlet` odwzoruje go na stronę *bookstore.jsp*.

Określenie ścieżki dostępu do pliku konfiguracyjnego zasobów aplikacji

W podrozdziale „Plik konfiguracyjny zasobów aplikacji" wyjaśniliśmy, że aplikacja może posiadać wiele plików konfiguracyjnych zasobów aplikacji. Jeśli pliki te nie zostaną umieszczone w domyślnie przeszukiwanych katalogach lub nie będą posiadać nazwy *faces-config.xml*, należy określić ścieżki dostępu do tych plików. Ścieżki te określamy za pomocą programu *deploytool* w sposób następujący:

1. Wybieramy z drzewa plik WAR.
2. Wybieramy panel *Context* i klikamy *Add*.
3. W polu *Coded Parameter* wprowadzamy `javax.faces.application.CONFIG_FILES`.
4. W polu *Value* wprowadzamy ścieżkę dostępu do pliku konfiguracyjnego zasobów aplikacji. Na przykład ścieżka dostępu do pliku konfiguracyjnego zasobów aplikacji `guessNumber` ma postać */WEB-INF/faces-config.xml*.
5. Powtarzamy punkty od 2. do 4. dla każdego pliku konfiguracyjnego zasobów aplikacji.

Określenie miejsca przechowywania stanu

Implementując metody interfejsu `StateHolder` (patrz punkt „Przechowywanie i odtwarzanie stanu" w rozdziale 20.) określamy w deskryptorze instalacji, czy stan komponentu będzie przechowywany u klienta czy na serwerze. W tym celu konfigurujemy parametr kontekstu za pomocą programu *deploytool*:

1. Wybieramy aplikację z drzewa programu *deploytool*.
2. Wybieramy panel *Context* i klikamy *Add*.
3. W polu *Coded Parameter* wprowadzamy `javax.faces.STATE_SAVING_METHOD`.
4. W polu *Value* wprowadzamy `client` lub `server` w zależności od tego, czy chcemy, by stan był przechowywany u klienta czy na serwerze.

Jeśli stan jest przechowywany u klienta, to stan całego widoku zostaje umieszczony na stronie w ukrytym polu. Implementacja JavaServer Faces domyślnie przechowuje stan u klienta. Aplikacja Księgarnia Duke'a również przechowuje swój stan u klienta.

Ograniczanie dostępu do komponentów JavaServer Faces

Oprócz identyfikacji instancji `FacesServlet` i stworzenia jej odwzorowania musimy również zapewnić, że wszystkie aplikacje używają instancji `FacesServlet` do przetwarzania komponentów JavaServer Faces. W tym celu konfigurujemy ograniczenia.

1. Wybieramy plik WAR z drzewa programu *deploytool*.
2. Wybieramy panel *Security*.
3. Klikamy *Add Constraints* i w polu *Security Constraints* wprowadzamy `Restrict Access to JSP Pages`.
4. Klikamy *Add Collections* i w polu *Web Resource Collections* wprowadzamy `Restrict Access to JSP Pages`.
5. Klikamy *Edit Collections*.
6. W oknie dialogowym *Edit Collections of Web Resource Collections* klikamy *Add URL Pattern* i wprowadzamy ścieżkę dostępu do strony, do której chcemy ograniczyć dostęp, na przykład */response.jsp*.
7. W sposób opisany w punkcie 6. wprowadzamy ścieżki dostępu do wszystkich stron JSP aplikacji i klikamy *OK*.

Włączanie kontroli poprawności plików XML

Aplikacja internetowa zawiera jeden lub więcej plików konfiguracyjnych zasobów aplikacji napisanych w języku XML. Możemy zarządzać od implementacji JavaServer Faces, aby sprawdzała poprawność tych plików. W tym celu nadajemy fladze `validateXML` wartość `true`:

1. Wybieramy plik WAR z drzewa programu *deploytool*.
2. Wybieramy panel *Context* i klikamy *Add*.
3. W polu *Coded Parameter* wprowadzamy `com.sun.faces.validateXml`.
4. W polu *Value* wprowadzamy `true`. Wartością domyślną jest `false`.

Weryfikacja niestandardowych obiektów

Jeśli aplikacja zawiera niestandardowe obiekty takie jak komponenty, konwertery, walidatory i obiekty wyświetlania, to podczas uruchamiania aplikacji możemy zweryfikować możliwość utworzenia tych obiektów. W tym celu nadajemy fladze `verifyObjects` wartość `true`:

1. Wybieramy plik WAR z drzewa programu *deploytool*.
2. Wybieramy panel *Context* i klikamy *Add*.
3. W polu *Coded Parameter* wprowadzamy `com.sun.faces.verifyObjects`.
4. W polu *Value* wprowadzamy `true`. Wartością domyślną jest `false`.

Flaga ta powinna mieć w normalnych warunkach wartość `false`, ponieważ weryfikacja obiektów wiąże się z dodatkowym kosztem.

Dołączanie wymaganych plików JAR

Aplikacje JavaServer Faces wymagają do prawidłowego działania kilku plików JAR. Należą do nich:

- *jsf-api.jar* (zawiera klasy `javax.faces.*`),
- *jsf-impl.ja*r (zawiera klasy implementacji JavaServer Faces),
- *jstl.jar* (wymagany, gdy używamy znaczników JSTL oraz używany przez klasy implementacji JavaServer Faces),
- *standard.jar* (wymagany, gdy używamy znaczników JSTL oraz używany przez klasy referencyjnej implementacji JavaServer Faces),
- *commons-beanutils.jar* (definiowanie i dostęp do właściwości komponentów JavaBeans),
- *commons-digester.jar* (przetwarzanie dokumentów XML),
- *commons-collections.jar* (rozszerzenia szkieletu kolekcji Java 2),
- *commons-logging.jar* (ogólny mechanizm zapisu informacji do dziennika).

Pliki *jsf-api.jar* i *jsf-impl.jar* znajdują się w katalogu *<J2EE_HOME>/lib/*. Plik *jstl.jar* został włączony do pliku *appserv-jstl.jar*. Pozostałe z wymienionych plików JAR włączono do pliku *appserv-rt.jar* również umieszczonego w katalogu *<J2EE_HOME>/lib/*.

Pakując i instalując aplikację JavaServer Faces za pomocą programu *deploytool*, nie musimy pakować żadnego z plików JAR z wyjątkiem pliku *jsf-api.jar*. Plik *jsf-api.jar* musi być spakowany razem z aplikacją, aby umożliwić dostęp do instancji `FacesServlet` i konfigurację jej odwzorowania.

Dołączanie klas, stron i innych zasobów

Podczas pakowania aplikacji internetowej za pomocą programu *deploytool* zauważymy, że program ten automatycznie pakuje wiele plików aplikacji we właściwych katalogach pliku WAR. Wszystkie strony JSP zostają umieszczone na najwyższym poziomie pliku WAR. Pliki TLD i *web.xml* tworzone przez program *deploytool* zostają umieszczone w katalogu *WEB-INF*. Wszystkie pakiety trafiają do katalogu *WEB-INF/classes*. Jednak program *deploytool* nie kopiuje pliku *faces-config.xml* w katalogu *WEB-INF*. Pakując aplikację, powinniśmy sami skopiować plik *faces-config.xml* do katalogu *WEB-INF*.