



Technologia i rozwiązania

JavaServer Faces 2.2

Mistrzowskie programowanie

JavaServer Faces 2.2

— to framework dla mistrzów programowania w Javie!



Anghel Leonard



Tytuł oryginału: Mastering JavaServer Faces 2.2

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-2419-0

Copyright © 2014 Packt Publishing

First published in the English language under the title ‘Mastering JavaServer Faces 2.2 – (9781782176466)’.

Polish edition copyright © 2016 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/jsf22m.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/jsf22m>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorze	11
O recenzentach	13
Wstęp	15
Rozdział 1. Dynamiczny dostęp do danych aplikacji JSF przy użyciu Expression Language (EL 3.0)	19
Składnia EL	20
Operatory EL	20
Hierarchia operatorów EL	21
Zastrzeżone słowa EL	21
Przetwarzanie natychmiastowe oraz opóźnione	22
Wyrażenia wartościowe EL	22
Odwołania do komponentów zarządzanych	23
Odwołania do właściwości komponentów zarządzanych	24
Odwołania do zagnieżdżonych właściwości komponentów zarządzanych	25
Odwołania do typów wyliczeniowych Java SE	27
Odwołania do kolekcji	27
Niejawne obiekty EL	29
Wyrażenia odwołujące się do metod	31
Teksty warunkowe w JSF	33
Pisanie własnego mechanizmu przetwarzającego	37
Przegląd EL 3.0	45
Stosowanie operatora przypisania	45
Stosowanie operatora konkatencji	45
Stosowanie operatora średnika	46
Poznawanie wyrażeń lambda	46
Stosowanie obiektów kolekcji	47
Podsumowanie	49

Rozdział 2. Komunikacja w JSF	51
Przekazywanie i pobieranie parametrów	52
Stosowanie parametrów kontekstu	52
Przekazywanie parametrów żądania przy użyciu znacznika <f:param>	52
Stosowanie parametrów widoku	55
Wywoływanie akcji z wykorzystaniem żądań GET	62
Przekazywanie atrybutów przy użyciu znacznika <f:attribute>	66
Ustawianie wartości właściwości przy użyciu obiektów nasłuchujących akcji	69
Przekazywanie parametrów przy użyciu zasięgu Flash	71
Zastępowanie znacznika <f:param> znacznikiem JSTL <c:set>	75
Przesyłanie danych w ciasteczkach	76
Stosowanie pól ukrytych	78
Przesyłanie haseł	79
Programowy dostęp do atrybutów komponentów interfejsu użytkownika	79
Przekazywanie parametrów przy użyciu wyrażeń odwołujących się do metod	80
Komunikacja przy użyciu atrybutu binding	81
Komunikacja pomiędzy komponentami zarządzanymi	83
Wstrzykiwanie jednego komponentu zarządzanego do drugiego	83
Komunikacja pomiędzy komponentami zarządzanymi przy użyciu mapy aplikacji lub sesji	85
Programowy dostęp do innych komponentów zarządzanych	86
Podsumowanie	87
Rozdział 3. Zasięgi JSF — długość życia i zastosowanie w komunikacji komponentów zarządzanych	89
Zasięgi JSF a zasięgi CDI	90
Zasięg żądania	92
Zasięg sesji	95
Zasięg widoku	98
Zasięg aplikacji	100
Zasięg konwersacji	102
Zasięg przepływu	105
Prosty przepływ	108
Przepływy z komponentami	112
Przepływy zagnieżdżone	114
Programowe konfigurowanie przepływów	118
Przepływy a przypadki nawigacji	120
Badanie przypadków nawigacji w przepływach	123
Stosowanie metod initializer i finalizer	124
Przełączanie przepływu	126
Pakowanie przepływów	129
Programowy zasięg przepływu	130
Zależny pseudozasięg	133
Zasięg none	134
Zasięg niestandardowy	134
Implementacja klasy zasięgu niestandardowego	135
Wyznaczanie wyrażeń EL zasięgów niestandardowych	136

Kontrola czasu istnienia zasięgu przy użyciu obiektu nasłuchującego akcji	139
Kontrola czasu istnienia zasięgu niestandardowego z użyciem obiektów NavigationHandler	141
Tworzenie instancji komponentów zarządzanych	144
Wstrzykiwanie komponentów	144
Podsumowanie	147
Rozdział 4. Konfigurowanie JSF przy użyciu plików XML i adnotacji — część 1.	149
Nowe przestrzenie nazw JSF 2.2	150
Programowa konfiguracja w JSF 2.2	151
Konfigurowanie komponentów zarządzanych w XML-u	152
Stosowanie wielu plików konfiguracyjnych	157
Konfiguracja ustawień lokalnych i wiązek zasobów	159
Konfiguracja walidatorów i konwerterów	161
Konfigurowanie nawigacji	169
Nawigacja niejawna	169
Nawigacja warunkowa	172
Nawigacja z wyłączeniem	175
Nawigacja programowa	177
Konfigurowanie obiektów nasłuchujących akcji	178
Obiekty nasłuchujące akcji aplikacji	180
Konfigurowanie metod nasłuchujących zdarzeń systemowych	183
Stosowanie znacznika <f:event>	183
Implementacja interfejsu SystemEventListener	185
Konfigurowanie metod nasłuchujących faz	191
Stosowanie adnotacji @ListenerFor oraz @ListenersFor	195
Podsumowanie	196
Rozdział 5. Konfigurowanie JSF przy użyciu plików XML i adnotacji — część 2.	197
Konfiguracja obiektów obsługi zasobów	198
Programowe dodawanie zasobów CSS i JS	205
Konfiguracja obiektu obsługi widoków	205
Przesłanie mechanizmów wizualizacji JSF	209
Stosowanie operacji wykonywanych po stronie klienta	215
Klasy wytwórcze JSF	219
Konfiguracja globalnego obiektu obsługi wyjątków	220
Konfiguracja klasy wytwórczej RenderKit	223
Konfiguracja PartialViewContext	224
Konfiguracja obiektu VisitContext	227
Konfiguracja obiektów ExternalContext	230
Konfiguracja Flash	233
Window ID API w JSF 2.2	235
Konfigurowanie cyklu życia	241
Konfigurowanie aplikacji	244
Konfigurowanie VDL	246
Połączone możliwości wielu klas wytwórczych	248
Podsumowanie	249

Rozdział 6. Korzystanie z danych tabelarycznych	251
Tworzenie prostej tabeli JSF	252
Klasa CollectionDataModel JSF 2.2	254
Sortowanie tabel	259
Sortowanie i DataModel — klasa CollectionDataModel	265
Usuwanie wiersza tabeli	267
Edycja i aktualizacja wierszy tabeli	269
Dodawanie nowych wierszy	272
Wyświetlanie numerów wierszy	274
Wybieranie pojedynczego wiersza	275
Wybieranie wielu wierszy	277
Zagnieżdżanie tabel	279
Podział tabel na strony	280
Generowanie tabel przy użyciu API JSF	286
Filtrowanie tabel	291
Określanie wyglądu tabel przy użyciu stylów	296
Zmiana koloru tła wierszy z użyciem atrybutu rowClasses	296
Podświetlanie wiersza wskazanego myszą	297
Podświetlanie wierszy po kliknięciu myszą	298
Podsumowanie	299
Rozdział 7. JSF i AJAX	301
Krótki przegląd cyklu życia JSF-AJAX	302
Prosty przykład JSF-AJAX na dobry początek	302
Atrybuty JSF-AJAX	303
Atrybuty execute oraz render	304
Atrybut listener	306
Atrybut event	307
Atrybut onevent — monitorowanie stanu AJAX-a po stronie klienta	308
Atrybut onerror — monitorowanie błędów AJAX-a po stronie klienta	309
Grupowanie komponentów w znaczniku <f:ajax>	311
Zastosowanie AJAX-a do aktualizacji pól formularzy po wystąpieniu błędów walidacji	312
Przyciski Anuluj i Wyczyść	314
Łączenie AJAX-a i zasięgu przepływu	318
Żądania zwrotne i AJAX	322
Warunkowe wyświetlanie i przetwarzanie żądań zwrotnych	324
Czy to nie jest żądanie AJAX?	327
AJAX i znacznik <f:param>	328
Kontrola kolejki żądań AJAX	329
Jawne wczytywanie pliku jsf.js	330
Prezentacja wartości parametrów	331
Metoda jsf.ajax.request i komponenty inne niż UICommand	332
Dostosowywanie zawartości pliku jsf.js	335
Implementacja AJAX-owego paska postępów (sygnalizator działania)	338
Podsumowanie	340

Rozdział 8. JSF 2.2 — HTML5 i przesyłanie plików na serwer	341
Korzystanie z HTML5 i JSF 2.2	341
Atrybuty przekazywane	342
Elementy przekazywane	344
JSF 2.2 — HTML5 i model Bean Validation 1.1 (Java EE 7)	346
Mechanizm przesyłania plików w JSF 2.2	347
Prosty przykład przesyłania plików z wykorzystaniem możliwości JSF 2.2	348
Stosowanie wielu elementów <h:inputFile>	350
Pobieranie informacji o przesyłanym pliku	351
Zapis przesłanych danych na dysku	353
Walidator przesyłanych plików	355
Przesyłanie plików z użyciem AJAX-a	356
Przesyłanie plików z podglądem	357
Przesyłanie większej liczby plików	364
Przesyłanie plików i nieokreślony pasek postępów	366
Przesyłanie plików i określony pasek postępów	368
Podsumowanie	371
Rozdział 9. Zarządzanie stanem w JSF	373
Zapisywanie stanu widoku w JSF	373
Częściowe zapisywanie stanu widoku	374
Częściowe zapisywanie stanu i przeglądanie drzewa	374
Zapisywanie stanu widoku na serwerze lub kliencie	375
Logiczne i fizyczne widoki JSF	378
Zapisywanie stanu w bazie danych — aplikacja eksperymentalna	379
Obsługa wyjątków ViewExpiredException	386
Serializacja stanu w sesji na serwerze	389
JSF 2.2 jest technologią bezstanową	391
Widoki bezstanowe oraz komponenty umieszczone w zasięgu widoku	392
Programowe wykrywanie widoków bezstanowych	394
Uwagi dotyczące bezpieczeństwa JSF	395
Cross-site request forgery (CSRF)	395
Cross-site scripting (XSS)	395
Wstrzykiwanie SQL	396
Podsumowanie	396
Rozdział 10. Niestandardowe komponenty JSF	397
Tworzenie komponentów niestandardowych, które nie są komponentami złożonymi	398
Tworzenie własnego obiektu obsługi znacznika	401
Tajniki konstrukcji komponentów niestandardowych	402
Tworzenie komponentów złożonych	413
Implementacja komponentu złożonego Temperature	416
Przekształcanie komponentu jQuery w komponent złożony	420
Pisanie pola do wyboru dat HTML5 jako komponentu złożonego	425
Wzbogacanie obrazka o akcje	429
Stosowanie facet złożonych	431
Walidacja lub konwersja danych wejściowych w komponentach złożonych	433

Sprawdzanie obecności atrybutu	435
Niebezpieczeństwa stosowania komponentów złożonych	435
Ukrywanie atrybutów przekazywanych w komponentach złożonych	436
Rozpowszechnianie komponentów złożonych w postaci plików JAR w JSF 2.2	439
Dodawanie komponentów złożonych w sposób programowy	441
Podsumowanie	443
Rozdział 11. Kontrakty biblioteki zasobów JSF 2.2 — motywy	445
Stosowanie kontraktów	446
Określanie wyglądu tabel przy użyciu kontraktów	448
Stosowanie kontraktów do określania wyglądu komponentów interfejsu użytkownika	451
Kontrakty stylów stosowane na urządzeniach różnych typów	453
Tworzenie kontraktów dla komponentów złożonych	458
Implementacja przełącznika motywów	460
Konfiguracja kontraktów w kodzie XML	467
Pakowanie kontraktów w plikach JAR	468
Podsumowanie	468
Rozdział 12. Szablony technologii Facelets	471
Krótka prezentacja znaczników technologii Facelets	471
Tworzenie prostego szablonu — PageLayout	474
Przekazywanie parametrów przy użyciu znacznika <ui:param>	477
Przekazywanie właściwości komponentów i metod akcji przy użyciu znacznika <ui:param>	479
Stosowanie znaczników <ui:decorate> oraz <ui:fragment>	481
Iteracja przy użyciu znacznika <ui:repeat>	484
Stosowanie znaczników <ui:include> oraz <f:viewParam>	487
Stosowanie znaczników <ui:include> oraz <ui:param>	489
Debugowanie z użyciem znacznika <ui:debug>	491
Usuwanie zawartości przy użyciu znacznika <ui:remove>	492
Stosowanie atrybutu jsfc	493
Rozszerzanie szablonu PageLayout	494
Programowe aspekty faceletów	499
Zagadnienia związane z klasą FaceletFactory	499
Stosowanie klasy FaceletCache	499
Klasa ResourceResolver zastąpiona klasą ResourceHandler	502
Programowe dołączanie faceletów	506
Tworzenie klasy TagHandler	507
Pisanie niestandardowych funkcji bibliotek znaczników faceletów	508
Pułapki stosowania faceletów	510
AJAX i znacznik <ui:repeat>	510
Przykład użycia znaczników <c:if> oraz <ui:fragment>	511
Przykład użycia znaczników <c:forEach> oraz <ui:repeat>	512
Podsumowanie	512
Dodatek A. Cykl życia JSF	515
Skorowidz	517

JSF 2.2 — HTML5 i przesyłanie plików na serwer

Niniejszy rozdział można podzielić na dwie główne części. Pierwsza z nich jest poświęcona wsparciu dla języka HTML5 dostępnemu w JSF 2.2, druga natomiast opisuje nowe mechanizmy **przesyłania plików na serwer** dodane w JSF 2.2. Oczywiście obie te części nie są ze sobą bezpośrednio powiązane, jednak, jak się przekonasz, komponenty do przesyłania plików dodane w JSF 2.2 można uatrakcyjnić, korzystając z nowych narzędzi wsparcia dla HTML5, a nowe atrybuty przekazywane (ang. *pass-through attributes*) mogą być bardzo przydatne w przypadku rozszerzania komponentu do przesyłania plików dodanego w JSF 2.2 przy użyciu nowych możliwości analogicznego komponentu języka HTML5.

Korzystanie z HTML5 i JSF 2.2

Każdy, kto zajmuje się tworzeniem aplikacji internetowych, bardzo entuzjastycznie podchodzi do poznawania i stosowania języka HTML5 udostępniającego nowy zestaw znaczników i możliwości, takich jak: `<audio>`, `<video>`, `<keygen>` itd. Począwszy od wersji 2.2, programiści JSF mogą korzystać z możliwości języka HTML5, używając:

- atrybutów przekazywanych;
- elementów przekazywanych (ang. *pass-through elements*; czyli kodu znacznikowego przyjaznego dla języka HTML).

Choć elementy i znaczniki przekazywane są inspirowane możliwościami języka HTML5, to jednak są one elementami JSF, których można używać także z innymi wersjami standardu HTML.

Mechanizmy te stanowią alternatywę dla pisania własnych zestawów mechanizmów wizualizacji. To doskonale rozwiązanie, gdyż język HTML5 wciąż jest w trakcie rozwoju, a to oznacza, że pisanie i dostosowywanie zestawów mechanizmów wizualizacji do bezustannych zmian języka może stanowić trudne wyzwanie.

Aby móc korzystać z języka HTML5 w JSF 2.0, konieczne będzie napisanie własnych zestawów mechanizmów wizualizacji, które będą obsługiwać nowe elementy i atrybuty tego języka.

Atrybuty przekazywane

Począwszy do JSF 2.2, dostępne są atrybuty przetwarzane przez komponenty JSF na serwerze oraz **atrybuty przekazywane** (ang. *pass-through attributes*), które są przetwarzane w trakcie działania aplikacji po stronie klienta.

Wygodnym elementem HTML5, którego można użyć do przeprowadzenia testów atrybutów przekazywanych, jest `<input>`. Do jego nowych możliwości można zaliczyć między innymi: nowe wartości atrybutu `type` (na przykład: `email`, `tel`, `color` oraz `reset`) oraz nowy atrybut `placeholder` (zawiera on tekst podpowiedzi wyświetlanej w pustym polu).

W kodzie HTML5 element ten można zapisać w następujący sposób:

```
<input placeholder="Wpisz adres e-mail zawodnika" type="email">
```

Korzystając z atrybutów przekazywanych, ten sam kod można wygenerować na pięć różnych sposobów:

- Umieścić atrybuty przekazywane w nowej przestrzeni nazw `http://xmlns.jcp.org/jsf/passthrough` (każdy programista JSF wie, czym są przestrzenie nazw oraz elementy z prefiksami. Stosowanie zarówno przestrzeni nazw, jak i elementów z prefiksami nie jest niczym szczególnym ani tajemniczym). Przekonajmy się zatem, w jaki sposób można wygenerować powyższy element HTML5, używając atrybutów przekazywanych JSF:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f5="http://xmlns.jcp.org/jsf/passthrough"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
...
<h:body>
  <h:inputText value="#{playersBean.email}" f5:type="email"
               f5:placeholder="Wpisz adres e-mail zawodnika"/>
...

```

Podczas prac nad tą książką wciąż trwała debata, jakiego prefiksu należy używać dla tej przestrzeni nazw. Początkowo wybrano prefiks p, jednak jest on powszechnie uznawany za prefiks biblioteki PrimeFaces, dlatego konieczne jest użycie innego. W tej książce zastosowałem prefiks f5, ale nic nie stoi na przeszkodzie, by zastąpić go dowolnym innym, który zwycięży w debatach i zyska największą popularność.

- Skorzystać ze znacznika `<f:passThroughAttribute>` w sposób pokazany w poniższym przykładzie:

```
<h:inputText value="#{playersBean.email}">
  <f:passThroughAttribute name="placeholder" value="Wpisz adres e-mail
zawodnika" />
  <f:passThroughAttribute name="type" value="email" />
</h:inputText>
```

- Atrybuty przekazywane mogą także pochodzić z komponentów zarządzanych. W takim przypadku należy je umieścić w kolekcji `Map<String, String>`, przy czym kluczami jej elementów są nazwy atrybutów, a wartościami — wartości tych atrybutów. Oto przykład takiego rozwiązania:

```
private Map<String, String> attrs = new HashMap<>();
...
attrs.put("type", "email");
attrs.put("placeholder", "Wpisz adres e-mail zawodnika");
```

W kodzie strony należy natomiast użyć znacznika `<f:passThroughAttributes>`, jak pokazano poniżej:

```
<h:inputText value="#{playersBean.email}">
  <f:passThroughAttributes value="#{playersBean.attrs}" />
</h:inputText>
```

- W przypadku stosowania języka Expression Language 3 (wchodzącego w skład środowiska Java EE 7) atrybuty można także definiować bezpośrednio, w sposób przedstawiony w poniższym przykładzie (w praktyce sprowadza się to do bezpośredniego zdefiniowania kolekcji `Map<String, String>` przy użyciu wyrażenia EL 3):

```
<h:inputText value="#{playersBean.email}">
  <f:passThroughAttributes value='#{{"placeholder":"Wpisz adres e-mail zawodnika",
  "type":"email"}}' />
</h:inputText>
```

Kompletna implementacja takiego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_8_01_1*.

- Atrybuty przekazywane można także dodawać programowo. Poniższy przykład pokazuje, w jaki sposób można wygenerować pole tekstowe HTML5 i dodać je do formularza:

```
<h:body>
  <h:form id="playerForm">
    ...
```

```

    </h:form>
</h:body>

...
FacesContext facesContext = FacesContext.getCurrentInstance();
UIComponent formComponent =
facesContext.getViewRoot().findComponent("playerForm");

HtmlInputText playerInputText = new HtmlInputText();
Map passThroughAttrs = playerInputText.getPassThroughAttributes();
passThroughAttrs.put("placeholder", "Wpisz adres e-mail zawodnika");
passThroughAttrs.put("type", "email");

formComponent.getChildren().add(playerInputText);
...

```

Kompletna implementacja takiego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_8_01_02*.

Elementy przekazywane

Programiści JSF ukrywają kod HTML pod postacią komponentów JSF. Dla twórców stron WWW kod JSF może wyglądać dosyć dziwnie, jednak generowany na jego podstawie kod HTML jest doskonale znany. W celu zmodyfikowania generowanego kodu HTML twórcy stron WWW muszą modyfikować kod JSF, co może być dla nich stosunkowo trudnym zadaniem. Na szczęście JSF 2.2 udostępnia wygodne rozwiązanie stworzone z myślą o języku HTML5. Są to tak zwane **elementy przekazywane** (ang. *pass-through elements*). Korzystając z tego mechanizmu, projektanci i twórcy stron mogą pisać czysty kod HTML, programiści JSF natomiast mogą przejmować go i kojarzyć elementy HTML z kodem wykonywanym na serwerze, dodając do nich lub zastępując odpowiednie atrybuty. JSF rozpoznaje takie atrybuty, jeśli należą one do przestrzeni nazw `http://xmlns.jcp.org/jsf`. Poniższy fragment kodu przedstawia przykład strony JSF, która nie zawiera nawet jednego znacznika JSF:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:jsf="http://xmlns.jcp.org/jsf">
  <head jsf:id="head">
    <title></title>
  </head>
  <body jsf:id="body">
    <form jsf:id="form">
      Imię:<input type="text" jsf:value="#{playersBean.playerName}"/>
      Nazwisko:<input type="text" jsf:value="#{playersBean.playerSurname}"/>
      <button jsf:action="#{playersBean.playerAction()}">Pokaż</button>
    </form>
  </body>
</html>

```

JSF przegląda elementy HTML w poszukiwaniu atrybutów należących do przestrzeni nazw `http://xmlns.jcp.org/jsf`. Dla każdego elementu zawierającego takie atrybuty JSF określi jego typ i zastąpi go odpowiednim komponentem JSF (na przykład znacznik `<head>` zostanie zastąpiony znacznikiem `<h:head>`, a `<input>` znacznikiem `<h:inputText>`). JSF doda komponenty do drzewa komponentów, które zostanie wyświetlone w przeglądarce jako kod HTML. Taki komponent JSF zostanie powiązany z konkretnym elementem, a jego atrybuty, w zależności od pochodzenia, zostaną określone na podstawie „normalnych” atrybutów lub atrybutów przekazywanych. Informacje na temat odwzorowywania komponentów JSF na elementy HTML można znaleźć na stronie <http://docs.oracle.com/javaee/7/api/javax/faces/view/facelets/TagDecorator.html>. W przypadku elementów HTML, które nie mają bezpośrednich odpowiedników (takich jak: `<div>` lub ``), JSF utworzy specjalny komponent, rodzinę komponentów, taką jak: `javax.faces.Panel`, oraz typ mechanizmu wizualizacji `javax.faces.passthrough.Element`; informacje na ten temat można znaleźć na stronie <http://docs.oracle.com/javaee/7/javaserver-faces-2-2/vlddocs-facelets/jsf/element.html>.

Kompletna implementacja takiego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_01_03*.

Ponieważ JSF zastępuje elementy HTML komponentami JSF, elementów tych można używać bez żadnych ograniczeń, co oznacza, że możemy korzystać z nich w kodzie JSF. Na przykład można używać walidatorów, konwerterów oraz znaczników `<f:param>`, co pokazano w poniższym przykładzie:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:jsf="http://xmlns.jcp.org/jsf"
  xmlns:f="http://xmlns.jcp.org/jsf/core">
  <head jsf:id="head">
    <title></title>
  </head>
  <body jsf:id="body">
    <form jsf:id="form">
      Imię:
      <input type="text" jsf:value="#{playersBean.playerName}"
        <f:validator validatorId="playerValidator"/>
      </input>
      <!-- albo w ten sposób -->
      <input type="text" jsf:value="#{playersBean.playerName}"
        jsf:validator="playerValidator"/>
      Nazwisko:
      <input type="text" jsf:value="#{playersBean.playerSurname}"
        <f:validator validatorId="playerValidator"/>
      </input>
      <!-- albo w ten sposób -->
      <input type="text" jsf:value="#{playersBean.playerSurname}"
        jsf:validator="playerValidator"/>
      <button jsf:action="#{playersBean.playerAction()}">Pokaż
        <f:param id="playerNumber" name="playerNumberParam" value="2014"/>
      </button>
```

```

    </form>
  </body>
</html>

```

Kompletna implementacja takiego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_8_01_04*.

JSF 2.2 — HTML5 i model Bean Validation 1.1 (Java EE 7)

Model Bean Validation 1.1 (patrz <https://docs.oracle.com/javasee/7/tutorial/bean-validation001.htm>) może być idealnym sposobem sprawdzania poprawności informacji wpisywanych przez użytkownika w aplikacjach tworzonych z użyciem JSF 2.2 i HTML5. Poniższy przykład pokazuje, jak można sprawdzić poprawność imienia i nazwiska w komponencie PlayersBean — nie są akceptowane wartości null, łańcuchy puste oraz łańcuchy krótsze od 3 znaków:

```

@Named
@RequestScoped
public class PlayersBean {

    private static final Logger logger = Logger.getLogger(PlayersBean.
        class.getName());

    @NotNull(message = "Imię nie może mieć wartości null ani być puste")
    @Size(min = 3,message = "Imię musi się składać z co najmniej 3 liter")
    private String playerName;
    @NotNull(message = "Nazwisko nie może mieć wartości null ani być puste")
    @Size(min = 3,message = "Nazwisko musi się składać z co najmniej 3 liter")
    private String playerSurname;
    ...
}

```

JSF może interpretować przesyłane puste łańcuchy znaków jako wartości null, jeśli w pliku *web.xml* zostanie użyty poniższy parametr kontekstu:

```

<context-param>
  <param-name>
    javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
  </param-name>
  <param-value>true</param-value>
</context-param>

```

W tym przypadku nie ma zatem potrzeby, by stosować znacznik `<f:validator>` lub atrybut `validator`. Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_8_02*.

Biblioteka OmniFaces udostępnia zestaw mechanizmów wizualizacji dla języka HTML5, wyposażony w możliwości stosowania jego nowych atrybutów. Więcej informacji na jego temat można znaleźć na stronie <http://showcase.omnifaces.org/>.

Mechanizm przesyłania plików w JSF 2.2

Programiści JSF długo czekali na wbudowany komponent służący do przesyłania plików. Az do momentu pojawienia się wersji 2.2 JSF problem ten obchodzono, stosując rozszerzenia JSF, takie jak PrimeFaces, RichFaces bądź też biblioteki, takie jak FileUpload należąca do projektu Apache Commons.

W JSF 2.2 dodano jednak komponent przeznaczony do przesyłania plików na serwer (w kodzie HTML jest on reprezentowany jako element `input` typu `file`). Komponent ten jest reprezentowany przez znacznik `<h:inputFile>` i można go stosować dokładnie tak samo jak wszystkie inne komponenty JSF. Wyczerpujące informacje o wszystkich dostępnych atrybutach znacznika `<h:inputFile>` można znaleźć w dokumentacji JSF, na stronie: <http://docs.oracle.com/javasee/7/javaserver-faces-2-2/vlddocs-facelets/h/inputFile.html>; poniżej przedstawionych zostało jedynie kilka najważniejszych spośród nich:

- `value`: określa plik reprezentowany jako obiekt `javax.servlet.http.Part`, który należy przesłać na serwer.
- `required`: zawiera wartość logiczną. Przyjmuje wartość `true`, jeśli użytkownik przed przesłaniem formularza musi określić wartość tego pola.
- `validator`: określa walidator używany przez ten komponent.
- `converter`: określa konwerter używany przez ten komponent.
- `valueChangeListener`: ten atrybut określa metodę, którą należy wywołać w przypadku zmiany wartości komponentu.

Komponent `<h:inputFile>` bazuje na specyfikacji Servlet 3.0, która wchodzi w skład platformy Java EE od jej wersji 6. Specyfikacja Servlet 3.0 udostępnia mechanizm do przesyłania plików na serwer bazujący na interfejsie `javax.servlet.http.Part` oraz adnotacji `@MultipartConfig`. Poniżej przedstawiony został prosty kod zgodny ze specyfikacją Servlet 3.0, pokazujący sposób implementacji przesyłu plików; trzeba przy tym pamiętać, że jest to servlet; zastosujemy go jeszcze pod koniec tego rozdziału:

```
@WebServlet(name = "UploadServlet", urlPatterns = {"/UploadServlet"})
@MultipartConfig(location="/folder", fileSizeThreshold=1024*1024,
    maxFileSize=1024*1024*3, maxRequestSize=1024*1024*3*3)

public class UploadServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        for (Part part : request.getParts()) {
            String filename = "";
            for (String s: part.getHeader("content-disposition").split(";")) {
                if (s.trim().startsWith("filename")) {
                    filename = s.split("=")[1].replaceAll("\\\"", "");
                }
            }
        }
    }
}
```



```

    }
  }
  part.write(filename);
}
}
}

```

Pobieżna analiza kodu źródłowego klasy `FacesServlet` implementacji JSF 2.2 pozwala zorientować się, że została ona opatrzona adnotacją `@MultipartConfig`; zrobiono to właśnie po to, by mogła ona obsługiwać typ danych używany w przypadku przesyłania plików na serwer.

Jeśli nie znasz sposobu przesyłania plików na serwer opisanego w specyfikacji Servlet 3.0, to więcej informacji na jego temat możesz znaleźć w poradniku na stronie <http://docs.oracle.com/javase/6/tutorial/doc/glrb.html>.

Po stronie klienta do przesyłania plików na serwer można użyć znacznika `<form>` oraz elementu HTML5 input typu `file`:

```

<form action="UploadServlet" enctype="multipart/form-data" method="POST">
  <input type="file" name="file">
  <input type="Submit" value="Prześlij plik">
</form>

```

W rzeczywistości komponent JSF 2.2 do przesyłu plików jest jedynie pewnego rodzaju opakowaniem dla powyższego kodu.

Prosty przykład przesyłania plików z wykorzystaniem możliwości JSF 2.2

W tym punkcie rozdziału przedstawione zostaną podstawowe czynności umożliwiające napisanie w JSF 2.2 aplikacji pozwalającej na przesyłanie plików na serwer. Choć aplikacja ta jest bardzo prosta, stanowi ona podstawę dla kolejnych przykładów przedstawionych w tym rozdziale.

W celu zaimplementowania mechanizmu przesyłania plików na serwer po stronie klienta należy wykonać następujące czynności:

1. W znaczniku `<h:form>` trzeba określić typ kodowania `multipart/form-data`, który pozwoli przeglądarce na wygenerowanie odpowiedniego żądania HTTP POST. Oto właściwa postać tego znacznika:

```

<h:form id="uploadForm" enctype="multipart/form-data">

```

2. Należy skonfigurować znacznik `<h:inputFile>` zgodnie z wymaganiami aplikacji. W tym przykładzie zostanie użyty następujący kod:

```

<h:inputFile id="fileToUpload" required="true"
  requiredMessage="Nie wybrano żadnego pliku..."
  value="#{uploadBean.file}"/>

```

3. Trzeba dodać przycisk (lub odnośnik), którego kliknięcie rozpocznie proces przesyłu pliku, na przykład taki:

```
<h:commandButton value="Prześlij plik" action="#{uploadBean.upload()}" />
```

Ewentualnie do formularza można także dodać jakieś znaczniki służące do obsługi komunikatów. Przykład formularza z takimi znacznikami został przedstawiony poniżej:

```
<h:messages globalOnly="true" showDetail="false"
  showSummary="true" style="color:red"/>
<h:form id="uploadFormId" enctype="multipart/form-data">
  <h:inputFile id="fileToUpload" required="true"
    requiredMessage="Nie wybrano żadnego pliku..."
    value="#{uploadBean.file}" />
  <h:commandButton value="Prześlij plik" action="#{uploadBean.upload()}" />
  <h:message showDetail="false" showSummary="true"
    for="fileToUpload" style="color:red" />
</h:form>
```

Poniższa lista opisuje natomiast czynności do zrealizowania w kodzie wykonywanym na serwerze:

1. Zazwyczaj wartością atrybutu `value` znacznika `<h:inputFile>` jest wyrażenie EL o postaci `#{komponent_przesylu.obiekt_part}`. Jeśli zmienimy *komponent_przesylu* na `uploadBean`, a *obiekt_part* na `file`, to uzyskamy wyrażenie o postaci `#{uploadBean.file}`. Obiekt `file` służy do przechowywania danych przesyłanego pliku w komponencie `UploadBean` jako instancji typu `javax.servlet.http.Part`. Jedyne, co musimy w tym celu zrobić, to zdefiniować właściwość `file` w komponencie, w dokładnie taki sam sposób, w jaki definiowane są wszystkie inne właściwości:

```
import javax.servlet.http.Part;
...
private Part file;
...
public Part getFile() {
    return file;
}

public void setFile(Part file) {
    this.file = file;
}
...
```

Przesłane dane można odczytać, używając metody `getInputStream` interfejsu `Part`.

2. Po kliknięciu przycisku *Prześlij plik* zostanie wywołana metoda `upload()`. W momencie wywoływania tej metody cała zawartość przesłanego pliku będzie już zapisana w obiekcie `file`, dzięki czemu będzie ją można pobrać w formie

strumienia (używając w tym celu metody `getInputStream`) i odpowiednio przetworzyć. Na przykład można użyć obiektu klasy `Scanner`, by przetworzyć przesłane dane na łańcuchy znaków, jak pokazano w poniższym przykładzie:

```
public void upload() {
    try {
        if (file != null) {
            Scanner scanner = new Scanner(file.getInputStream(), "UTF-8").
                useDelimiter("\\A");
            fileInString = scanner.hasNext() ? scanner.next() : "";
            logger.info(fileInString);
            FacesContext.getCurrentInstance().addMessage(null,
                new FacesMessage("Udało się pomyślnie przesłać plik na serwer!"));
        }
    } catch (IOException | NoSuchElementException e) {
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Nie udało się przesłać pliku!"));
    }
}
```

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_03*. W jej przypadku przesłane dane są przekształcane na łańcuchy znaków i wyświetlane w dzienniku, testując ją, warto zatem przesyłać zwyczajne pliki tekstowe zawierające czytelne informacje.

Stosowanie wielu elementów <h:inputFile>

Jeśli zastanawiasz się, czy w tym samym znaczniku <h:form> można umieścić więcej niż jeden znacznik <h:inputFile>, to powinieneś wiedzieć, że odpowiedź na to pytanie jest twierdząca. Należy przy tym nadać każdemu znacznikowi <h:inputFile> unikalny identyfikator i skojarzyć go z unikalną instancją typu `Part`. Aby zastosować dwa znaczniki <h:inputFile>, zmienimy formularz <h:form> w sposób przedstawiony poniżej (analogicznie można rozszerzyć formularz, tak by zawierał trzy, cztery znaczniki <h:inputFile> czy też jeszcze więcej):

```
<h:form id="uploadFormId" enctype="multipart/form-data">
    <h:inputFile id="fileToUpload_1" required="true"
        requiredMessage="Nie wybrano żadnego pliku..."
        value="#{uploadBean.file1}"/>
    <h:inputFile id="fileToUpload_2" required="true"
        requiredMessage="Nie wybrano żadnego pliku..."
        value="#{uploadBean.file2}"/>
    ...
    <h:message showDetail="false" showSummary="true"
        for="fileToUpload_1" style="color:red"/>
    <h:message showDetail="false" showSummary="true"
        for="fileToUpload_2" style="color:red"/>
</h:form>
```

```

...
<h:commandButton value="Prześlij pliki" action="#{uploadBean.upload()}" />
</h:form>

```

Teraz w kodzie wykonywanym po stronie serwera należy zdefiniować dwie instancje typu `Part`:

```

...
private Part file1;
private Part file2;
...
// akcesory i mutatory dla pól file1 i file2
...

```

Obie instancje `Part` należy obsłużyć w metodzie `upload`:

```

...
if (file1 != null) {
    Scanner scanner1 = new Scanner(file1.getInputStream(), "UTF-8");
    useDelimiter("\\A");
    fileInString1 = scanner1.hasNext() ? scanner1.next() : "";
    logger.info(fileInString1);
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Udało się pomyślnie przesłać plik 1 na serwer!"));
}
if (file2 != null) {
    Scanner scanner2 = new Scanner(file2.getInputStream(), "UTF-8");
    useDelimiter("\\A");
    fileInString2 = scanner2.hasNext() ? scanner2.next() : "";
    logger.info(fileInString2);
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Udało się pomyślnie przesłać plik 2 na serwer!"));
}
...

```

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_04*.

Pobieranie informacji o przesyłanym pliku

W przypadku przesyłania pliku na serwer do najpotrzebniejszych informacji należą: jego nazwa, rozmiar, typ zawartości oraz wielkość. W JSF informacje te są dostępne zarówno po stronie klienta, jak i serwera. Przyjrzyjmy się następującemu znacznikowi `<h:inputFile>`:

```

<h:form id="formUploadId" enctype="multipart/form-data">
  <h:inputFile id="fileToUpload" value="#{uploadBean.file}"
    required="true" requiredMessage="Nie wybrano żadnego pliku...">
    ...
  </h:inputFile>
</h:form>

```

A teraz przekonasz się, w jaki sposób można pobrać informacje o pliku wybranym do przesłania na serwer.

Po stronie klienta w tym celu należy wykonać następujące czynności:

- Nazwę pliku, jego wielkość (wyrażoną w bajtach) oraz typ zawartości można pobrać przy użyciu poniższego kodu JavaScript:

```
var file = document.getElementById('formUploadId:fileToUpload').files[0];
...
alert(file.name);
alert(file.size);
alert(file.type);
```

- Innym rozwiązaniem jest zastosowanie odpowiedniego wyrażenia w kodzie strony JSF (oczywiście w tym przypadku informacje o pliku będą dostępne dopiero po jego przesłaniu na serwer):

```
// identyfikator komponentu, formUploadId:fileToUpload
#{uploadBean.file.name}
```

```
// nazwa przesłanego pliku
#{uploadBean.file.submittedFileName}
```

```
// rozmiar przesłanego pliku
#{uploadBean.file.size}
```

```
// typ zawartości przesłanego pliku
#{uploadBean.file.contentType}
```

Po stronie serwera informacje o przesłanym pliku można pobrać w poniższy sposób:

- Do pobrania nazwy pliku, jego wielkości (wyrażonej w bajtach) oraz typu zawartości służy kilka metod interfejsu `Part`; oto przykład ich zastosowania:

```
...
private Part file;
...
System.out.println("Identyfikator komponentu file: " + file.getName());
System.out.println("Typ zawartości: " + file.getContentType());
System.out.println("Nazwa przesłanego pliku:" + file.getSubmittedFileName());
System.out.println("Wielkość pliku: " + file.getSize());
...

```

Jeśli łańcuch znaków zwracany przez tę metodę reprezentuje całą ścieżkę dostępu do pliku, a nie tylko jego nazwę, to konieczne będzie jej wyodrębnienie.

- Nazwę pliku można także pobrać z nagłówka `content-disposition`, jak pokazano w poniższym przykładzie:

```
private String getFileNameFromContentDisposition(Part file) {
    for (String content:file.getHeader("content-disposition").split(";")) {
        if (content.trim().startsWith("filename")) {
            return content.substring(content.indexOf('=') +1).trim().replace("\"", "");
        }
    }
    return null;
}
```

Przykładową postać nagłówka content-disposition przedstawia rysunek 8.1.

```
formUploadId
-----22975721424845
Content-Disposition: form-data; name="formUploadId:fileToUpload"; filename="RafaelNadal.jpg"
Content-Type: image/jpeg
```

Rysunek 8.1. Przykładowa postać nagłówka content-disposition

Powyższe rozwiązanie można bardzo łatwo zrozumieć, jeśli przeanalizujemy żądanie POST (używając w tym celu dodatku Firebug lub innego wyspecjalizowanego narzędzia dla programistów). Na powyższym rysunku został przedstawiony wybrany fragment informacji prezentowanych przez metodę `getFileNameFromContentDisposition`.

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_8_05*.

Zapis przesłanych danych na dysku

W poprzednich przykładach plik przesłany na serwer był konwertowany na łańcuch znaków i wyświetlany w konsoli. Jednak zazwyczaj po odebraniu pliku będziemy chcieli zapisać jego zawartość na serwerze, w określonym katalogu (załóżmy, że będzie to katalog *D:\files*). Do tego celu można skorzystać z klasy `FileOutputStream` w sposób przedstawiony w poniższym przykładzie:

```
try (InputStream inputStream = file.getInputStream();
    FileOutputStream outputStream = new FileOutputStream("D:" +
        File.separator + "files" + File.separator +
        file.getSubmittedFileName())) {

    int bytesRead = 0;
    final byte[] chunk = new byte[1024];
    while ((bytesRead = inputStream.read(chunk)) != -1) {
        outputStream.write(chunk, 0, bytesRead);
    }

    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Udało się pomyślnie przesłać plik na serwer!"));
}
```

```

    } catch (IOException e) {
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Nie udało się przesłać pliku!"));
    }

```

Jeśli chcemy wykonywać operacje wejścia-wyjścia z użyciem bufora, to możemy w tym celu skorzystać z klas `BufferedInputStream` oraz `BufferedOutputStream`.

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_06*. Jeśli wolisz pobierać nazwę pliku z nagłówka `content-disposition`, to takie rozwiązanie zostało zaimplementowane jako aplikacja *przykład_8_07*.

Innym rozwiązaniem pozwalającym na zapisywanie zawartości przesyłanych plików na serwerze jest zastosowanie metody `Part.write`. W takim przypadku położenie zapisywanego pliku należy określić przy użyciu znacznika `<multipart-config>` (<https://docs.oracle.com/javasee/7/tutorial/servlets011.htm>). Oprócz położenia można także ustawić maksymalną wielkość pliku, wielkość żądania oraz próg wielkości pliku. Wszystkie te informacje są zapisywane w pliku `web.xml`. Poniżej przedstawiono przykładowy fragment tego pliku:

```

<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
  <multipart-config>
    <location>D:\files</location>
    <max-file-size>1310720</max-file-size>
    <max-request-size>20971520</max-request-size>
    <file-size-threshold>50000</file-size-threshold>
  </multipart-config>
</servlet>

```

Jeśli plik konfiguracyjny nie będzie zawierał określenia miejsca docelowego plików, to zostanie użyta lokalizacja domyślna, którą jest "".

W tym przypadku przesłany plik zostanie zapisany we wskazanym miejscu, pod nazwą przekazaną jako argument wywołania metody `Part.write`:

```

try {
    file.write(file.getSubmittedFileName());
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Udało się pomyślnie przesłać plik na serwer!"));
} catch (IOException e) {
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage("Nie udało się przesłać pliku!"));
}

```


Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_08*.

Walidator przesyłanych plików

W większości przypadków trzeba będzie w jakiś sposób ograniczać to, jakie pliki użytkownicy mogą przesyłać na serwer. Zazwyczaj ograniczenia te dotyczą długości nazwy pliku, jego wielkości lub typu zawartości. Na przykład można zdecydować się na odrzucanie plików, które:

- mają nazwy o długości przekraczającej 25 znaków;
- nie są obrazami w formatach PNG lub JPG;
- przekraczają 1 MB wielkości.

Takie warunki można zaimplementować w formie następującej klasy:

```
@FacesValidator
public class UploadValidator implements Validator {

    private static final Logger logger =
        Logger.getLogger(UploadValidator.class.getName());

    @Override
    public void validate(FacesContext context, UIComponent component, Object value)
        throws ValidatorException {

        Part file = (Part) value;

        // WERYFIKACJA DŁUGOŚCI NAZWY PLIKU
        String name = file.getSubmittedFileName();
        logger.log(Level.INFO, "WERYFIKACJA NAZWY PLIKU: {0}", name);
        if (name.length() == 0) {
            FacesMessage message = new FacesMessage("Błąd przesyłania pliku: Nie można
                określić nazwy pliku!");
            throw new ValidatorException(message);
        } else if (name.length() > 25) {
            FacesMessage message = new FacesMessage("Błąd przesyłania pliku: Nazwa pliku
                jest zbyt długa!");
            throw new ValidatorException(message);
        }

        // WERYFIKACJA TYPU ZAWARTOŚCI PLIKU
        if (!(("image/png".equals(file.getContentType()) &&
            ("image/jpeg".equals(file.getContentType())))) {
            FacesMessage message = new FacesMessage("Błąd przesyłania pliku:
                Można przesyłać tylko obrazy PNG i JPG!");
            throw new ValidatorException(message);
        }
    }
}
```

```
// WERYFIKACJA WIELKOŚCI PLIKU (rozmiar pliku nie może być większy niż 1 MB)
if (file.getSize() > 1048576) {
    FacesMessage message = new FacesMessage("Błąd przesyłania pliku: Nie można
        przesyłać plików większych niż 1 MB!");
    throw new ValidatorException(message);
}
}
```

Taki walidator należy jeszcze dodać do znacznika `<h:inputFile>`, jak pokazano w poniższym przykładzie:

```
<h:inputFile id="fileToUpload" required="true"
    requiredMessage="Nie wybrano żadnego pliku..."
    value="#{uploadBean.file}">
    <f:validator validatorId="uploadValidator" />
</h:inputFile>
```

Teraz na serwerze będzie można zapisywać wyłącznie pliki spełniające podane kryteria. W przypadku odrzucenia pliku na stronie zostanie wyświetlony komunikat informujący o zbyt długiej nazwie, przekroczeniu limitu wielkości lub nieodpowiednim typie pliku.

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_09*.

Przesyłanie plików z użyciem AJAX-a

Mechanizm przesyłania plików zaimplementowany w JSF pozwala na wykorzystanie możliwości technologii AJAX poprzez połączenie znaczników `<h:inputFile>` i `<f:ajax>` bądź też znaczników `<h:commandButton>` (do inicjalizacji przesyłu) i `<f:ajax>`. W pierwszym z tych dwóch przypadków formularz do przesyłu plików zazwyczaj przyjmuje następującą postać:

```
<h:form enctype="multipart/form-data">
    <h:inputFile id="fileToUpload" value="#{uploadBean.file}" required="true"
        requiredMessage="Nie wybrano żadnego pliku..."
        <!-- <f:ajax listener="#{uploadBean.upload()}" render="@all"/>
            w JSF 2.2.0 należy użyć @all -->
        <f:ajax listener="#{uploadBean.upload()}" render="fileToUpload"/>
        <!-- działa w JSF 2.2.5 -->
    </h:inputFile>
    <h:message showDetail="false" showSummary="true" for="fileToUpload"
        style="color:red"/>
</h:form>
```

Atrybut `render` powinien zawierać identyfikatory komponentów, które należy odświeżyć po zakończeniu przesyłania pliku. W JSF 2.2.0, w związku z błędem implementacji, konieczne jest użycie słowa kluczowego `@all` zamiast identyfikatorów. W nowszych wersjach JSF błąd ten został już naprawiony i w JSF 2.2.5 wszystko działa zgodnie z oczekiwaniami.

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_10*.

W drugim z opisanych wcześniej przypadków kombinację znaczników `<f:ajax>` oraz `<h:commandButton>` można zastosować tak, jak pokazano w poniższym przykładzie:

```
<h:form enctype="multipart/form-data">
  <h:inputFile id="fileToUpload" value="#{uploadBean.file}" required="true"
    requiredMessage="Nie wybrano żadnego pliku..."/>
  <h:commandButton value="Upload" action="#{uploadBean.upload()}">
    <!-- <f:ajax execute="fileToUpload" render="@all"/> use @all in JSF 2.2.0 -->
    <f:ajax execute="fileToUpload" render="fileToUpload"/> <!-- działa w JSF 2.2.5 -->
  </h:commandButton>
  <h:message showDetail="false" showSummary="true"
    for="fileToUpload" style="color:red"/>
</h:form>
```

Kompletna wersja tego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przykład_8_11*.

Przesyłanie plików z podglądem

Bardzo cenną i wygodną cechą komponentów do przesyłu plików jest możliwość wyświetlania podglądu obrazka przed jego przesłaniem na serwer. Poniższy rysunek 8.2 pokazuje rozwiązanie, które zaimplementujemy w tym punkcie rozdziału.



Rysunek 8.2. Przesyłanie plików na serwer z podglądem

Innymi słowy, kiedy użytkownik wybierze obrazek, należy go automatycznie i bezzwłocznie przesłać na serwer przy użyciu technologii AJAX, zapewniając w ten sposób możliwość wyświetlenia go na stronie natychmiast po wybraniu z lokalnego komputera. Żądanie POST wygenerowane przez AJAX spowoduje zapisanie pliku w obiekcie Part (nadamy mu nazwę `file`) na serwerze. Po zakończeniu obsługi żądania AJAX konieczne jest odświeżenie jakiegoś komponentu pozwalającego na wyświetlanie obrazków, takiego jak `<h:graphicImage>`. Komponent ten odwoła się do serwera, używając żądania GET. Komponent zarządzany odpowiedzialny za obsługę przesyłanych plików zostanie umieszczony w zasięgu sesji, dlatego też serwlet będzie

mógł pobrać instancję tego komponentu z sesji i użyć obiektu `file` reprezentującego przesłany obrazek. Dzięki temu serwlet będzie mógł przesłać zawartość obrazka (jego bajty), używając w tym celu strumienia wyjściowego odpowiedzi; będzie też mógł wygenerować jego miniaturę i przesłać ją, aby ograniczyć ilość przesyłanych danych. Następnie, kiedy użytkownik kliknie przycisk inicjujący przesyłanie wybranego pliku, należy go jedynie zapisać na dysku.

Tak wygląda idea działania tego rozwiązania. Teraz zajmiemy się jego implementacją i rozbudowaniem o sprawdzanie poprawności, przycisk pozwalający na anulowanie przesyłania oraz wyświetlaniem informacji o wybranym pliku obok jego podglądu.

W celu zaimplementowania przedstawionego rozwiązania należy wykonać następujące czynności:

1. Przygotować formularz, który będzie automatycznie przysyłał obrazek na serwer z użyciem AJAX-a:

```
<h:form enctype="multipart/form-data">
  <h:inputFile id="uploadFileId" value="#{uploadBean.file}" required="true"
    requiredMessage="Nie wybrano żadnego pliku...">
    <f:ajax render=":previewImgId :imgNameId :uploadMessagesId"
      listener="#{uploadBean.validateFile()}" />
  </h:inputFile>
</h:form>
```

2. W ramach obsługi żądania AJAX zostanie wywołana metoda `validateFile`. Ta metoda, wykonywana na serwerze, potrafi sprawdzić nazwę pliku, jej długość, wielkość pliku oraz typ jego zawartości. Poniżej przedstawiony został jej kod:

```
...
private Part file;
...

public void validateFile() {
    // WERYFIKACJA DŁUGOŚCI NAZWY PLIKU
    String name = file.getSubmittedFileName();
    if (name.length() == 0) {
        resetFile();
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
            "Błąd przesyłania pliku: Nie można określić nazwy pliku!"));
    } else if (name.length() > 25) {
        resetFile();
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
            "Błąd przesyłania pliku: Nazwa pliku jest zbyt długa!"));
    } else // WERYFIKACJA TYPU ZAWARTOŚCI PLIKU
        if (!(!"image/png".equals(file.getContentType()) &&
            !"image/jpeg".equals(file.getContentType())) {
            resetFile();
            FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
                "Błąd przesyłania pliku: Można przysyłać tylko obrazy PNG i JPG!"));
        } else // WERYFIKACJA WIELKOŚCI PLIKU (rozmiar pliku nie może być większy niż 1 MB)
            if (file.getSize() > 1048576) {
```

```

        resetFile();
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
            "Błąd przesyłania pliku: Nie można przesyłać plików większych niż 1 MB!"));
    }
}

```

3. Jeśli warunki określone w metodzie `validateFile` nie będą spełnione, to zostanie wywołana metoda `resetFile`. To bardzo prosta metoda, która odtwarza początkowy stan obiektu pliku. Co więcej, wywołuje ona także metodę `delete`, która usuwa tymczasowe zasoby przydzielone dla przesłanego pliku (w tym także pliki tymczasowo zapisane na dysku). Metoda `resetFile` została zdefiniowana w następujący sposób:

```

public void resetFile() {
    try {
        if (file != null) {
            file.delete();
        }
    } catch (IOException ex) {
        Logger.getLogger(UploadBean.class.getName()).log(Level.SEVERE, null, ex);
    }
    file = null;
}

```

4. Po zakończeniu obsługi żądania AJAX na stronie zostaną odświeżone komponenty o identyfikatorach: `previewImgId`, `imgNameId` oraz `uploadMessagesId`. Poniższy kod przedstawia komponenty o identyfikatorach `previewImgId` oraz `imgNameId`. W naszej aplikacji przykładowej identyfikator `uploadMessagesId` odpowiada komponentowi `<h:messages>`:

```

...
<h:panelGrid columns="2">
    <h:graphicImage id="previewImgId"
        value="/PreviewServlet/#{header['Content-Length']}"
        width="#{uploadBean.file.size gt 0 ? 100 : 0}"
        height="#{uploadBean.file.size gt 0 ? 100 : 0}"/>
    <h:outputText id="imgNameId"
        value="#{uploadBean.file.submittedFileName}
        #{empty uploadBean.file.submittedFileName ? ' ' : ','}
        #{uploadBean.file.size} #{uploadBean.file.size gt 0 ? 'bajtów' : ''}"/>
</h:panelGrid>
...

```

5. Atrybut `value` znacznika `<h:graphicImage>` odwołuje się do serwletu `PreviewServlet`. Serwlet ten udostępnia obrazek, generując jego zawartość przy użyciu strumienia wyjściowego odpowiedzi, dzięki czemu zapewnia możliwość wyświetlania podglądu obrazka. Aby uniknąć problemów z mechanizmem pamięci podręcznej, konieczne jest dodanie do adresu obrazka losowego ciągu znaków (doskonale do tego celu nada się liczba określająca wielkość zawartości żądania). Dzięki zastosowaniu

takiego rozwiązania dla każdego żądania będzie wyświetlany odpowiedni, a nie ten sam, obrazek. Poniżej przedstawiony został fragment implementacji serwletu:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {

    // w razie potrzeby można dodać bufor, stosując odpowiedni strumień wyjściowy
    OutputStream out = response.getOutputStream();

    response.setHeader("Expires", "Sat, 6 May 1995 12:00:00 GMT");
    response.setHeader("Cache-Control", "no-store, no-cache, must-revalidate");
    response.addHeader("Cache-Control", "post-check=0, pre-check=0");
    response.setHeader("Pragma", "no-cache");

    int nRead;
    try {
        HttpSession session = request.getSession(false);
        if (session.getAttribute("uploadBean") != null) {
            UploadBean uploadBean = (UploadBean) session.getAttribute("uploadBean");
            if (uploadBean.getFile() != null) {
                try (InputStream inStream = uploadBean.getFile().getInputStream()) {
                    byte[] data = new byte[1024];
                    while ((nRead = inStream.read(data, 0, data.length)) != -1) {
                        out.write(data, 0, nRead);
                    }
                }
            }
        }
    } finally {
        out.close();
    }
}
```

6. Powyższy kod zapisze wszystkie bajty przesłanego obrazka w strumieniu wyjściowym odpowiedzi. W przypadku wyświetlania podglądu przesyłanych obrazków często jest stosowana technika polegająca na zmniejszaniu obrazka i generowaniu jego miniaturki, by ograniczyć liczbę przesyłanych bajtów. W Javie obrazek można przeskalować na wiele różnych sposobów, a jeden z prostszych i szybszych został przedstawiony poniżej:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {

    OutputStream out = response.getOutputStream();

    response.setHeader("Expires", "Sat, 6 May 1995 12:00:00 GMT");
    response.setHeader("Cache-Control", "no-store, no-cache, must-revalidate");
    response.addHeader("Cache-Control", "post-check=0, pre-check=0");
```

```

response.setHeader("Pragma", "no-cache");

try {
    HttpSession session = request.getSession(false);
    if (session.getAttribute("uploadBean") != null) {
        UploadBean uploadBean = (UploadBean) session.getAttribute("uploadBean");
        if (uploadBean.getFile() != null) {
            BufferedImage image = ImageIO.read(uploadBean.getFile().getInputStream());
            BufferedImage resizedImage =
                new BufferedImage(100, 100, BufferedImage.TYPE_INT_ARGB);
            Graphics2D g = resizedImage.createGraphics();
            g.drawImage(image, 0, 0, 100, 100, null);
            g.dispose();
            ImageIO.write(resizedImage, "png", out);
        }
    }
} finally {
    out.close();
}
}

```

7. Następnie na stronie należy dodać dwa przyciski: *Prześlij obraz* oraz *Anuluj*. Kliknięcie pierwszego z nich rozpoczyna przesyłanie pliku, a kliknięcie drugiego przerywa operację. Poniższy kod przedstawia oba te przyciski:

```

<h:form>
  <h:commandButton value="Prześlij plik"
    action="#{uploadBean.saveFileToDisk()}" />
  <h:commandButton value="Anuluj" action="#{uploadBean.resetFile()}" />
</h:form>

```

8. Po kliknięciu przycisku *Prześlij obraz* metoda `saveFileToDisk` zapisze przesłany plik na dysku; oto jej kod:

```

public void saveFileToDisk() {

    if (file != null) {
        // w razie potrzeby można dodać bufor, stosując odpowiedni strumień wyjściowy
        try (InputStream inputStream = file.getInputStream();
            FileOutputStream outputStream = new FileOutputStream("D:" +
                File.separator + "files" + File.separator + file.getSubmittedFileName())) {

            int bytesRead;
            final byte[] chunk = new byte[1024];
            while ((bytesRead = inputStream.read(chunk)) != -1) {
                outputStream.write(chunk, 0, bytesRead);
            }

            resetFile();
        }
    }
}

```



```

        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Udało się pomyślnie przesłać plik na serwer!"));
    } catch (IOException e) {
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Nie udało się przesłać pliku!"));
    }
}
}
}

```

I gotowe! Kompletna wersja tej aplikacji bez generowania podglądu wybranego obrazka jest dostępna w kodach dołączonych do książki jako *przykład_8_13*. Z kolei aplikacja prezentująca podgląd obrazka jest dostępna jako *przykład_8_12*.

Proces weryfikacji wybranego pliku można usunąć z serwera i przenieść do przeglądarki. Ta wersja aplikacji także jest dostępna w kodach dołączonych do książki, jako *przykład_8_14*. Kod JavaScript używany do weryfikacji pliku jest całkiem prosty, co pokazano w poniższym przykładzie:

```

<script type="text/javascript">
    function validateFile() {
        // <![CDATA[
        document.getElementById('formSaveId:uploadHiddenId').value = false;
        document.getElementById('validationId').innerHTML = "";

        var file = document.getElementById('formUploadId:fileToUpload').files[0];

        document.getElementById('fileNameId').innerHTML = "<b>Nazwa pliku:</b> " +
            file.name;

        if (file.size > 1048576)
            fileSize = (Math.round(file.size * 100 / (1048576)) / 100).toString() + 'MB';
        else
            fileSize = (Math.round(file.size * 100 / 1024) / 100).toString() + 'KB';
        document.getElementById('fileSizeId').innerHTML = "<b>Wielkość pliku:</b> " +
            fileSize;
        document.getElementById('fileContentTypeId').innerHTML = "<b>Typ pliku:</b> " +
            file.type;

        // WERYFIKACJA DŁUGOŚCI NAZWY PLIKU
        if (file.name.length === 0) {
            clearUploadField();
            document.getElementById('validationId').innerHTML =
                "<ul><li>Błąd przesyłania pliku: Nie można określić nazwy pliku!</li></ul>";
            return false;
        }

        if (file.name.length > 25) {

```

```

clearUploadField();
document.getElementById('validationId').innerHTML =
    "<ul><li>Błąd przesyłania pliku: Nazwa pliku jest zbyt długa!</li></ul>";
return false;
}

// WERYFIKACJA TYPU ZAWARTOŚCI PLIKU
if (file.type !== "image/png" && file.type !== "image/jpeg") {
clearUploadField();
document.getElementById('validationId').innerHTML = "<ul><li>Błąd przesyłania
    pliku: Można przesyłać tylko obrazy PNG i JPG!</li></ul>";
return false;
}

// WERYFIKACJA WIELKOŚCI PLIKU (rozmiar pliku nie może być większy niż 1 MB)
if (file.size > 1048576) {
clearUploadField();
document.getElementById('validationId').innerHTML = "<ul><li>Błąd przesyłania
    pliku: Nie można przysyłać plików większych niż 1 MB!</li></ul>";
return false;
}

document.getElementById('formSaveId:uploadHiddenId').value = true;
return true;
//]]>
}

function clearUploadField()
{
document.getElementById('previewImgId').removeAttribute("src");
document.getElementById('imgNameId').innerHTML = "";
document.getElementById('uploadMessagesId').innerHTML = "";
var original = document.getElementById("formUploadId:fileToUpload");
var replacement = document.createElement("input");

replacement.type = "file";
replacement.id = original.id;
replacement.name = original.name;
replacement.className = original.className;
replacement.style.cssText = original.style.cssText;
replacement.onchange = original.onchange;
// ...dalsze atrybuty

original.parentNode.replaceChild(replacement, original);
}
</script>

```

Przesyłanie większej liczby plików

Domyślnie JSF 2.2 nie zapewnia możliwości przesyłania większej liczby plików, jednak taki rezultat można łatwo uzyskać, wprowadzając kilka modyfikacji. W celu zapewnienia możliwości przesyłania kilku plików należy się skoncentrować na dwóch wymienionych poniżej zagadnieniach:

- Zapewnieniu możliwości wyboru większej liczby plików.
- Przesłaniu wszystkich wybranych plików na serwer.

Jeśli chodzi o pierwsze z tych zagadnień, to możliwość wyboru większej liczby plików zapewnia element `input` języka HTML5 z atrybutem `multiple` oraz mechanizm atrybutów przekazywanych JSF 2.2. W przypadku zastosowania tego atrybutu i przypisania mu wartości `true` okno dialogowe przeglądarki pozwala na zaznaczenie więcej niż jednego pliku. A zatem pierwsze zagadnienie sprowadza się do wprowadzania minimalnych modyfikacji:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f5="http://xmlns.jcp.org/jsf/passthrough">
...
<h:form id="uploadFormId" enctype="multipart/form-data">
  <h:inputFile id="fileToUpload" required="true" f5:multiple="multiple"
              requiredMessage="Nie wybrano żadnego pliku..."
              value="#{uploadBean.file}"/>
  <h:commandButton value="Prześlij" action="#{uploadBean.upload()}" />
</h:form>
```

Drugi problem jest nieco bardziej złożony, gdyż w przypadku wybrania większej liczby plików przesłanie każdego kolejnego pliku sprawi, że JSF nadpisze wcześniejszą instancję `Part`. To zrozumiałe zachowanie, ponieważ standardowo używana jest tylko jedna instancja typu `Part`, a w tym przypadku potrzebna jest ich kolekcja. Rozwiązanie tego problemu wymaga skoncentrowania się na mechanizmie wizualizacji komponentu do przesyłania plików. Nosi on nazwę `FileRenderer` (i jest rozszerzeniem klasy `TextRenderer`), a kluczowym elementem naszego problemu jest przedstawiona poniżej implementacja jego metody `decode` (w szczególności fragment kodu wyróżniony pogrubieniem):

```
@Override
public void decode(FacesContext context, UIComponent component) {

    rendererParamsNotNull(context, component);

    if (!shouldDecode(component)) {
        return;
    }

    String clientId = decodeBehaviors(context, component);

    if (clientId == null) {
        clientId = component.getClientId(context);
    }
}
```

```

    }

    assert (clientId != null);
    ExternalContext externalContext = context.getExternalContext();
    Map<String, String> requestMap = externalContext.getRequestParameterMap();

    if (requestMap.containsKey(clientId)) {
        setSubmittedValue(component, requestMap.get(clientId));
    }

    HttpServletRequest request = (HttpServletRequest) externalContext.getRequest();
    try {
        Collection<Part> parts = request.getParts();
        for (Part cur : parts) {
            if (clientId.equals(cur.getName())) {
                component.setTransient(true);
                setSubmittedValue(component, multiple);
            }
        }
    } catch (IOException | ServletException ioe) {
        throw new FacesException(ioe);
    }
}

```

Wyróżniony fragment kodu powoduje nadpisywanie instancji Part, jednak bez trudu można go zmodyfikować tak, by operował nie na jednej instancji Part, lecz na ich liście:

```

    try {
        Collection<Part> parts = request.getParts();
        List<Part> multiple = new ArrayList<>();
        for (Part cur : parts) {
            if (clientId.equals(cur.getName())) {
                component.setTransient(true);
                multiple.add(cur);
            }
        }
        this.setSubmittedValue(component, multiple);
    } catch (IOException | ServletException ioe) {
        throw new FacesException(ioe);
    }
}

```

Oczywiście aby zastosować tak zmodyfikowany kod, konieczne jest utworzenie własnego mechanizmu wizualizacji i prawidłowe skonfigurowanie go w pliku *faces-config.xml*.

Następnie, w poniższy sposób, można zdefiniować w komponencie listę instancji Part:

```

...
private List<Part> files;

public List<Part> getFile() {

```

```

    return files;
}

public void setFile(List<Part> files) {
    this.files = files;
}
...

```

Każdy element tej listy jest plikiem, a to oznacza, że można je zapisać na dysku, przeglądając tę listę przy użyciu poniższego kodu:

```

...
for (Part file : files) {
    ...

    try (InputStream inputStream = file.getInputStream();
        FileOutputStream outputStream =
            new FileOutputStream("D:" + File.separator + "files" + File.separator +
                file.getSubmittedFileName())) {

        int bytesRead = 0;
        final byte[] chunk = new byte[1024];
        while ((bytesRead = inputStream.read(chunk)) != -1) {
            outputStream.write(chunk, 0, bytesRead);
        }

        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Udało się pomyślnie przesłać plik " +
                file.getSubmittedFileName() + " na serwer!"));
    } catch (IOException e) {
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Nie udało się przesłać pliku!"));
    }
}
}

```

Kompletna wersja tej aplikacji bez generowania podglądu wybranego obrazka jest dostępna w kodach dołączonych do książki jako *przykład_8_15*.

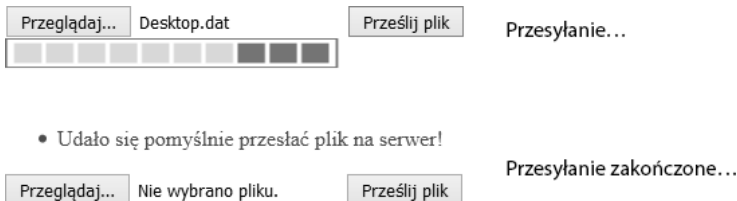
Przesyłanie plików i nieokreślony pasek postępów

W przypadku przesyłania niewielkich plików cały proces trwa bardzo krótko, kiedy jednak w grę wchodzi duże pliki, to zakończenie ich przesyłania może zająć kilka sekund, a nawet kilka minut. W takich przypadkach bardzo dobrym zwyczajem jest implementacja paska postępów informującego użytkownika o stanie procesu przesyłania. Najprostszy pasek stanu to pasek nieokreślony, nazwany tak, gdyż pokazuje jedynie fakt trwania procesu, lecz nie udostępnia żadnych informacji o szacunkowym czasie jego zakończenia ani o ilości przetworzonych bajtów.

W celu zaimplementowania takiego paska postępów konieczne jest stworzenie mechanizmu przesyłu korzystającego z technologii AJAX. Mechanizm AJAX zaimplementowany w JSF pozwala określać, kiedy zostaje rozpoczęte oraz zakończone żądanie. Informacje te są dostępne po stronie klienta, dzięki czemu nieokreślony pasek postępów można w prosty sposób zaimplementować, używając poniższego kodu:

```
<script type="text/javascript">
  function progressBar(data) {
    if (data.status === "begin") {
      document.getElementById("uploadMsgId").innerHTML="";
      document.getElementById("progressBarId").setAttribute("src",
        "./resources/progress_bar.gif");
    }
    if (data.status === "complete") {
      document.getElementById("progressBarId").removeAttribute("src");
    }
  }
</script>
...
<h:body>
  <h:messages id="uploadMsgId" globalOnly="true" showDetail="false"
  showSummary="true" style="color:red"/>
  <h:form id="uploadFormId" enctype="multipart/form-data">
    <h:inputFile id="fileToUpload" required="true"
      requiredMessage="Nie wybrano żadnego pliku..."
      value="#{uploadBean.file}"/>
    <h:message showDetail="false" showSummary="true" for="fileToUpload"
      style="color:red"/>
    <h:commandButton value="Prześlij plik" action="#{uploadBean.upload()}">
      <f:ajax execute="fileToUpload" onevent="progressBar" render=":uploadMsgId
      @form"/>
    </h:commandButton>
  </h:form>
  <div>
    <img id="progressBarId" width="250px;" height="23"/>
  </div>
</h:body>
```

Przykładowy wygląd takiego paska postępów został przedstawiony na rysunku 8.3.



Rysunek 8.3. Nieokreślony pasek postępów

Kompletna wersja tej aplikacji bez generowania podglądu wybranego obrazka jest dostępna w kodach dołączonych do książki jako *przykład_8_16*.

Przesyłanie plików i określony pasek postępów

Określony pasek postępów jest znacznie bardziej złożony. Zazwyczaj bazuje on na obiekcie nasłuchującym, który jest w stanie monitorować liczbę przesłanych bajtów danych (jeśli korzystałeś z komponentu `FileUpload` wchodzącego w skład projektu Apache Commons, to na pewno miałeś okazję zaimplementować taki obiekt nasłuchujący). W JSF 2.2 klasa `FacesServlet` została opatrzona adnotacją `@MultipartConfig`, dzięki czemu może obsługiwać przesyłanie plików na serwer, jednak nie udostępnia żadnego interfejsu obiektu nasłuchującego, który można by wykorzystać do śledzenia przebiegu tego procesu. Co więcej, klasa ta jest sfinalizowana, co oznacza, że nie można jej rozszerzać.

Cóż, powyższe czynniki nieco ograniczają potencjalne rozwiązania tego problemu. Aby zaimplementować pasek postępów obsługiwany na serwerze, komponent obsługujący przesyłanie plików musi zostać zaimplementowany jako odrębna klasa (serwlet) i musi udostępniać odpowiedni interfejs obiektu nasłuchującego. Alternatywne rozwiązanie działające po stronie klienta mogłoby polegać na zastosowaniu żądań POST, które oszukiwałyby implementację klasy `FacesServlet`, udając, że zostały wygenerowane przez plik *jsf.js*.

W tym punkcie rozdziału przedstawione zostanie rozwiązanie bazujące na obiekcie `XMLHttpRequest Level 2`, dostępnym w języku HTML5 (który pozwala na przesyłanie i odbieranie strumieni typów `Blob`, `File` oraz `FormData`), zdarzeniach postępu dostępnych w HTML5 (w przypadku przesyłania plików na serwer zdarzenia te udostępniają informacje o całkowitej liczbie przekazanych oraz przesłanych bajtów), elemencie paska postępów HTML5 oraz niestandardowym serwlecie korzystającym z możliwości wprowadzonych w specyfikacji Java Servlet 3.0. Jeśli nie znasz wspomnianych możliwości języka HTML5, to warto przeczytać odpowiednią dokumentację, aby się z nimi zapoznać.

Jeśli znasz wymienione możliwości języka HTML5, bez trudu będziesz mógł zrozumieć przedstawione poniżej fragmenty kodów obsługujące pasek postępów po stronie klienta. Jako pierwszy przedstawiony został kod JavaScript:

```
<script type="text/javascript">
function fileSelected() {
    hideProgressBar();
    updateProgress(0);
    document.getElementById("uploadStatus").innerHTML = "";
    var file = document.getElementById('fileToUploadForm:fileToUpload').files[0];
    if (file) {
        var fileSize = 0;
        if (file.size > 1048576)
            fileSize = (Math.round(file.size * 100 / (1048576)) / 100).toString() + 'MB';
        else
            fileSize = (Math.round(file.size * 100 / 1024) / 100).toString() + 'KB';
    }
}
```



```

        document.getElementById('fileName').innerHTML = 'Nazwa: ' + file.name;
        document.getElementById('fileSize').innerHTML = 'Wielkość: ' + fileSize;
        document.getElementById('fileType').innerHTML = 'Typ: ' + file.type;
    }
}

function uploadFile() {
    showProgressBar();
    var fd = new FormData();
    fd.append("fileToUpload",
        document.getElementById('fileToUploadForm:fileToUpload').files[0]);

    var xhr = new XMLHttpRequest();
    xhr.upload.addEventListener("progress", uploadProgress, false);
    xhr.addEventListener("load", uploadComplete, false);
    xhr.addEventListener("error", uploadFailed, false);
    xhr.addEventListener("abort", uploadCanceled, false);
    xhr.open("POST", "UploadServlet");
    xhr.send(fd);
}

function uploadProgress(evt) {
    if (evt.lengthComputable) {
        var percentComplete = Math.round(evt.loaded * 100 / evt.total);
        updateProgress(percentComplete);
    }
}

function uploadComplete(evt) {
    document.getElementById("uploadStatus").innerHTML =
        "Przesyłanie pliku zostało pomyślnie zakończone!";
}

function uploadFailed(evt) {
    hideProgressBar();
    document.getElementById("uploadStatus").innerHTML = "Nie można przesłać pliku!";
}

function uploadCanceled(evt) {
    hideProgressBar();
    document.getElementById("uploadStatus").innerHTML =
        "Przesyłanie zostało przerwane!";
}

var updateProgress = function(value) {
    var pBar = document.getElementById("progressBar");
    document.getElementById("progressNumber").innerHTML = value + "%";
    pBar.value = value;
}

```

```

function hideProgressBar() {
    document.getElementById("progressBar").style.visibility = "hidden";
    document.getElementById("progressNumber").style.visibility = "hidden";
}

function showProgressBar() {
    document.getElementById("progressBar").style.visibility = "visible";
    document.getElementById("progressNumber").style.visibility = "visible";
}
</script>

```

A poniżej został przedstawiony element HTML do przesyłania plików korzystający z powyższego kodu JavaScript:

```

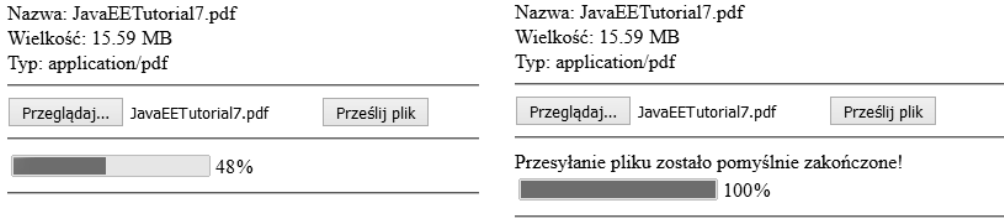
<h:body>
  <hr/>
  <div id="fileName"></div>
  <div id="fileSize"></div>
  <div id="fileType"></div>
  <hr/>
  <h:form id="fileToUploadForm" enctype="multipart/form-data">
    <h:inputFile id="fileToUpload" onchange="fileSelected();" />
    <h:commandButton type="button" onclick="uploadFile()" value="Prześlij plik" />
  </h:form>
  <hr/>
  <div id="uploadStatus"></div>
  <table>
    <tr>
      <td>
        <progress id="progressBar" style="visibility: hidden;"
          value="0" max="100"></progress>
      </td>
      <td>
        <div id="progressNumber" style="visibility: hidden;">0 %</div>
      </td>
    </tr>
  </table>
  <hr/>
</h:body>

```

Przykładowa postać takiej strony została przedstawiona na rysunku 8.4.

Serwlet zastosowany w tym rozwiązaniu, UploadServlet, został przedstawiony we wcześniejszej części rozdziału. Kompletna wersja tej aplikacji bez generowania podglądu wybranego obrazka jest dostępna w kodach dołączonych do książki jako *przykład_8_17*.

W razie konieczności użycia takiego paska postępów podczas przesyłania większej liczby plików można rozszerzyć powyższą aplikację lub zastosować jedno z gotowych rozwiązań, takich jak PrimeFaces Upload, RichFaces Upload czy też jQuery Upload Plugin.



Rysunek 8.4. Pasek postępów pokazujący bieżący stan procesu przesyłania pliku

Podsumowanie

W tym rozdziale dowiedziałeś się, jak korzystać z możliwości języka HTML5 przy użyciu atrybutów i elementów przekazywanych JSF 2.2. W drugiej części przedstawione zostały sposoby korzystania z nowego komponentu do przesyłania plików dostępnego w JSF 2.2 (proste przesyłanie, przesyłanie większej liczby plików, przesyłanie obrazków z podglądem, przesyłanie z wykorzystaniem nieokreślonego oraz określonego paska postępów).

Do zobaczenia w następnym rozdziale, w którym poznasz kolejną wspaniałą możliwość JSF 2.2 — widoki bezstanowe.

Skorowidz

A

adnotacja, 149, 197

 @Advanced, 166

 @ApplicationScoped, 100

 @Column, 290

 @Component, 163, 167

 @ConversationScoped, 102

 @CustomScoped, 134

 @Dependent, 133

 @EJB, 165

 @FacesComponent, 399, 414

 @FacesConverter, 167

 @FacesValidator, 163

 @FlowDefinition, 119

 @Inject, 165, 166, 217

 @ListenerFor, 195

 @ListenersFor, 195

 @ManagedBean, 91

 @ManagedProperty, 146

 @MultipartConfig, 348

 @Named, 91, 163, 167

 @PostConstruct, 100, 120, 133

 @RequestScoped, 163, 167

 @ResourceDependencies, 214

 @ResourceDependency, 214, 219

 @SessionScoped, 96

 @ViewScoped, 98

adres URL, 176

AJAX, 225, 301

akcesor, 68

akcje widoku, 62, 64

aktualizacja

 pól formularzy, 312

 wierszy tabeli, 269

API DOM, 151

API JSF, 79, 286

aplikacja, 100

arkusz stylów, 450

atak

 typu CSRF, 395

 typu wstrzykiwanie SQL, 396

 typu XSS, 395

atrybut

 action, 70

 actionListener, 178, 180

 binding, 68, 81

 componentType, 420

 event, 217, 307

 execute, 304

 jsfc, 493

 leftside, 422

 listener, 306

 multiple, 364

 onclick, 298

 onerror, 309

 onevent, 308

 onmouseout, 297

 onmouseover, 297

 render, 304

 rendered, 270

 resetValues, 314

 rightside, 422

 rowClasses, 296

 style, 296

 styleClasses, 296

atrybuty

 JSF-AJAX, 303

 przekazywane, 342, 436

 znacznika <uirepeat>, 484

B

baza, base, 29
 baza danych MongoDB, 383
 bezpieczeństwo, 395
 bezstanowość, 391
 biblioteka
 <h:inputText>, 68
 ICEfaces, 67
 MyFaces, 67
 OmniFaces, 166, 222
 PrimeFaces, 67, 450
 RichFaces, 67
 błędy walidacji, 312

C

ciasteczka, cookies, 76
 metody, 78
 CSRF, Cross-site request forgery, 395
 CSS, 452, 454
 cykl życia, 241
 JSF, 515
 JSF-AJAX, 302
 czas istnienia zasięgu, 139, 141
 częściowa prezentacja, 302
 częściowe
 przetwarzanie, 302
 zapisywanie stanu, 374

D

dane tabelaryczne, 251
 debugowanie, 491
 deskryptor web.xml, 204
 diagram
 aplikacji, 113
 aplikacji z przepływem, 115
 witryny, 109
 dodawanie wierszy, 272
 dołączanie faceletów, 506
 dostęp
 do atrybutów komponentów, 79
 do ciasteczek, 77
 do komponentów zarządzanych, 86
 do mapy żądania, 95
 dynamiczny dostęp do danych aplikacji, 19
 działanie zasięgu konwersacji, 104

E

EAR, Enterprise ARchive, 164
 edycja wierszy tabeli, 269
 EJB, Enterprise JavaBean, 164
 EL 3.0, 45
 elementy
 adnotacji @FacesComponent, 399
 przekazywane, 344
 Expression Language, 19, 136

F

facelety, 471
 metadanych widoku, 499
 widoku, 499
 filtrowanie tabel, 291
 Flash, 233
 formularz dodawania wierszy, 272, 274
 funkcje
 bibliotek znaczników faceletów, 508
 wyrażeń, 508

G

generator stylów CSS3, 451
 generowanie tabel, 286
 grupowanie komponentów, 311

H

hasła, 79
 hierarchia
 klasy DataModel, 265
 operatorów, 21
 HTML5, 341

I

identyfikator
 dokumentu, 108
 konwersacji, 104
 okna, 236
 implementacja
 interfejsu SystemEventListener, 185
 klasy ResponseStateManager, 382
 komparatora, 266
 komponentu niestandardowego, 405

- komponentu złożonego, 416
- motywów, 445
- przełącznika motywów, 460
- informacje o przesyłanym pliku, 351
- instancje komponentów zarządzanych, 144
- interfejs
 - ActionListener, 179, 180
 - ClientBehaviorHolder, 216
 - ComponentSystemEventListener, 195
 - Converter, 166
 - EditableValueHolder, 434
 - java.io.Serializable, 97, 98
 - NamingContainer, 410
 - PhaseListener, 191
 - StateHelper, 403
 - SystemEventListener, 185
 - użytkownika, 79
 - Validator, 62
 - VisitCallback, 227
- iteracja, 484

J

- język deklarowania widoków, VDL, 246, 445, 471
- jQuery Upload Plugin, 370
- JSF-AJAX, 302

K

- catalog
 - resources, 198
 - WEB-INF, 202
- klasa
 - ActionListenerWrapper, 181
 - Application, 245
 - ApplicationFactory, 245
 - ATPSinglesRankings, 41
 - ClientBehaviorBase, 218
 - ClientWindow, 238
 - ClientWindowFactory, 237
 - CollectionDataModel, 254, 259, 265, 266, 267
 - ComponentHandler, 401
 - ConfigurableNavigationHandler, 178
 - CreateCustomScope, 139
 - CustomClientWindow, 238
 - CustomFaceletCache, 501
 - CustomFlash, 234
 - CustomFlowHandler, 131
 - CustomParameter, 132

- CustomRenderKit, 383
- CustomScopeNavigationHandler, 142
- DataModel, 265, 284
- DestroyCustomScope, 140
- ELResolver, 37
- ExceptionHandler, 220
- ExceptionHandlerWrapper, 220
- ExternalContext, 233
- ExternalContextFactory, 232
- ExternalContextWrapper, 232
- FaceletCache, 499
- FaceletFactory, 499
- FacesServlet, 348
- FlashFactory, 233
- FlashWrapper, 234
- FlowBuilder, 119, 124
- FlowHandler, 130
- FullAjaxExceptionHandlerFactory, 222
- FullVisitContext, 289
- HtmlDataTable, 284
- java.util.ArrayList, 256
- java.util.HashMap, 257
- java.util.HashSet, 257
- java.util.LinkedHashMap, 258
- java.util.LinkedHashSet, 257
- java.util.LinkedList, 256
- java.util.TreeMap, 258
- java.util.TreeSet, 257
- javax.faces.model.DataModel, 265
- LifecycleFactory, 241
- LifecycleWrapper, 241
- MessagesRenderer, 212
- NavigationCaseBuilder, 121
- NavigationHandler, 141
- NavigationHandlerWrapper, 122
- PartialViewContext, 224, 225
- PlayerBean, 63
- PlayerListener, 179
- PlayerResource, 200
- PlayersDataModel, 286
- RafaRenderer, 214
- Renderer, 213, 214
- RenderKit, 223
- ResourceHandler, 502
- ResourceHandlerWrapper, 200
- ResourceResolver, 205, 502
- ResponseStateManager, 380, 382
- ResponseWriterWrapper, 211
- TagHandler, 507

- klasa
 - UIInput, 59
 - UIViewParameter, 55, 59, 62
 - VariableResolver, 40
 - ViewDeclarationLanguageFactory, 246
 - ViewHandler, 380
 - VisitCallback, 229
 - VisitContext, 228, 229
 - VisitContextFactory, 228
 - WriteStateInDB, 383
 - zasięgu niestandardowego, 135
- klasy
 - CSS, 452
 - FlowHandlerFactory, 130
 - opakowujące, 178
 - wytwórcze, 130, 219, 223, 248
- klient szablonu, 476
- kolejka żądań AJAX, 329
- kolekcja, 27
 - typu HashSet, 284
 - typu Map, 257, 271
 - typu Set, 270
- komentarz, 492
- komparator, 266
- komponent, 392
 - jQuery, 420
 - listy rozwijanej, 410
 - PlayersBean, 72, 86
 - pomocniczy, 417
 - ProfileBean, 86
 - TempBean, 412
 - Temperature, 416
 - ThemeSwitcher, 464, 465, 466
 - ThemeSwitcherBean, 464
 - UICommand, 333, 334
 - zarządzany PlayersBean, 68, 101
- komponenty
 - CDI, 90
 - interfejsu użytkownika, 81
 - najwyższego poziomu, 438
 - niestandardowe, 402
 - niestandardowe, 397
 - zarządzane, 23, 83
 - złożone, 397, 413, 429
 - facety, 431
 - implementacja, 416
 - konwersja danych, 433
 - niebezpieczeństwa, 435
 - pole do wyboru dat, 425
 - programowe dodawanie, 441
 - przekształcanie komponentu jQuery, 420
 - rozpowszechnianie, 439
 - sprawdzanie obecności atrybutu, 435
 - tworzenie, 413
 - tworzenie kontraktów, 458
 - ukrywanie atrybutów, 436
 - walidacja danych, 433
 - wzbogacanie obrazka, 429
 - zliczanie elementów podrzędnych, 437
- komunikacja, 51
 - komponentów zarządzanych, 89
 - między komponentami zarządzanymi, 83, 85
- komunikat błędu, 315
- komunikaty, 160
- konfiguracja
 - Flash, 233
 - globalnego obiektu obsługi wyjątków, 220
 - klasy PartialViewContext, 224
 - klasy RenderKit, 223
 - kontraktów, 467
 - konwerterów, 161
 - obiektów ExternalContext, 230
 - obiektów obsługi zasobów, 198
 - obiektu obsługi widoków, 205
 - obiektu VisitContext, 227
 - ustawień lokalnych, 159
 - walidatorów, 161
 - wiązek zasobów, 159
- konfigurowanie
 - aplikacji, 244
 - cyklu życia, 241
 - JSF, 149, 197
 - komponentów zarządzanych, 152
 - metod nasłuchujących faz, 191
 - metod nasłuchujących zdarzeń systemowych, 183
 - nawigacji, 169
 - obiektów nasłuchujących akcji, 178
 - przepływów, 118
 - VDL, 246
- konstrukcja komponentów niestandardowych, 402
- kontrakt, 445, 446
 - allddevices, 457
 - browserpc, 453
 - Device320, 453
 - Device480, 453
 - Device640, 453
 - jsfui, 452

tableBlue, 449
 tableGreen, 449
 kontrakty stylów, 453
 kontrola kolejki żądań, 329
 konwersacja, 102
 konwersja danych, 433
 konwerter, 161

L

listy, 154
 rozwijane, 410

M

mapa
 aplikacji, 85
 inicjalizacyjna, 153
 sesji, 85
 żądania, 95
 mechanizm
 przesyłania plików, 347
 przetwarzający, 37, 43
 Window ID, 235
 wizualizacji, 210, 213
 metadane, 55
 metoda, 31
 addCompositeComponent, 442
 addValuesToFlashAction, 74
 afterPhase, 191
 beforePhase, 191
 begin, 102
 ClientWindowFactory.getClientWindow, 237
 compare, 259
 ConfigurableNavigationHandler.inspectFlow,
 123
 createResource, 200, 202, 505
 createViewResource, 505
 decode, 402, 412
 deletePlayerAction, 333
 encodeBegin, 214, 402, 414
 encodeChildren, 402
 encodeEnd, 402
 end, 102
 evaluateExpressionGet, 87
 ExternalContext.setClientWindow, 236
 FaceletContext.includeFacelet, 506
 FaceletContext.setAttribute, 506
 finalizer, 125

findComponent, 80, 288
 FlowBuilder.finalizer, 124
 flowBuilder.id, 119
 FlowBuilder.initializer, 124
 getAsObject, 166
 getAttributes, 80
 getFacelet, 500
 getPhaseId, 191, 192
 getPlayerName, 56
 getPlayerSurname, 56
 getRenderIds, 225
 getRequestPath, 201
 getReturnValue, 114
 getScript, 218
 getSinglesRankings, 40, 42
 getSinglesRankingsReversed, 42
 getSinglesRankingsUpperCase, 42
 getState, 380, 381
 getToFlowDocumentId, 121
 getValue, 40
 getWrapped, 200
 greetingsAction, 84
 handleFileUpload, 67
 init, 61, 63
 initializer, 124
 isTransient, 102
 jsf.ajax.request, 332, 333
 navigateHelper, 174
 parametersAction, 70, 71
 playerDone, 176
 playerLogin, 174
 populateApplicationConfiguration, 151
 processAction, 79, 181
 processEvent, 195
 pullValuesFromFlash, 74
 resetFile, 359
 ResponseStateManager.writeState, 380
 saveFileToDisk, 361
 setRowCount, 284
 setRowIndex, 266
 showSelectedPlayer, 275
 termsAcceptedAction, 75
 tournamentFinalize, 126
 tournamentInitialize, 126
 UIViewRoot.restoreViewScopeState(), 100
 validateFile, 359
 visitTree, 227, 228
 write, 211
 writeState, 380

metody
 ciasteczek, 78
 nasłuchujące faz, 191
 nasłuchujące zdarzeń systemowych, 183
 zwrotne, 151
 model Bean Validation, 346
 monitorowanie
 błędów, 309
 stanu, 308
 mutator, 68

N

nadużywanie komponentów, 146
 nawigacja, 120, 169
 niejawna, 169
 programowa, 169, 177
 warunkowa, 169, 172
 z wyłączeniem, 169, 175
 NetBeans IDE 8.0, 90
 nieokreślony pasek postępow, 367

O

obiekt
 ConfigurableNavigationHandler, 65
 Conversation, 102
 ExternalContext, 230
 Flash, 233
 flowScope, 110
 java.util.Map, 154
 nasłuchujący akcji, 139
 NavigationHandler, 141
 obsługi znacznika, 401
 PartialViewContext, 227
 RenderKitFactory, 224
 ResourceHandler, 204
 ResponseWriter, 402
 UIViewRoot, 80
 VisitContext, 227
 obiekty
 kolekcji, 47
 nasłuchujące akcji, 69, 178, 180, 182
 niejawne, 29, 30, 31
 obsługi widoków, 205
 obsługi wyjątków, 220
 obsługi zasobów, 198

obsługa
 widoków, 205
 wyjątków, 220
 wyjątków ViewExpiredException, 386
 zasobów, 198
 zdarzenia akcji, 178
 znacznika, 401
 odwołania
 do kolekcji, 27
 do komponentów zarządzanych, 23
 do metod, 80
 do typów wyliczeniowych, 27
 do właściwości komponentów zarządzanych, 24
 do zagnieżdżonych właściwości
 komponentów zarządzanych, 25
 operacja końcowa, 47
 operacje
 po stronie klienta, 215
 pośrednie, 47
 operator, 20
 konkatencji, 45
 przypisania, 45
 średnika, 46

P

pakiet
 JAR, 151
 javax.enterprise.context, 96, 100
 javax.faces.bean., 100
 pakowanie
 kontraktów, 468
 przepływów, 129
 parametr
 ActionListener, 178
 javax.faces.SERIALIZE_SERVER_STATE,
 390
 parametry
 inicjalizacyjne kontekstu, 153
 kontekstu, 52
 wejściowe, 107
 widoku, 55
 wyjściowe, 107
 żądania, 52
 pasek postępow, 338
 nieokreślony, 366
 określony, 368

- plik
 - cc.taglib.xml, 440
 - confirm.xhtml, 111
 - delete.taglib.xml, 216
 - faces-config.xml, 65, 122, 131, 153, 154, 157, 215, 440
 - index.xhtml, 57
 - jsf.js, 330, 335
 - jsfcc.jar, 440
 - rafa.css, 201
 - registration-flow.xml, 111, 119, 128
 - schedule-flow.xml, 120
 - schedule-flow.xml, 118
 - styles.css, 454
 - web.xml, 153, 216
 - pliki
 - CSS, 199
 - EAR, 164
 - JAR, 151, 439, 468
 - konfiguracyjne, 157
 - WAR, 164
 - XML, 149, 197
 - pobieranie
 - ciasteczka, 77
 - informacji, 351
 - parametrów, 52
 - podświetlanie wiersza, 297
 - podział tabel, 280
 - pola ukryte, 78
 - pole do wyboru dat, 425
 - prezentacja wartości parametrów, 331
 - PrimeFaces Upload, 370
 - programowa konfiguracja, 151
 - programowe
 - aspekty faceletów, 499
 - dodawanie komponentów złożonych, 441
 - dodawanie zasobów, 205
 - dołączanie faceletów, 506
 - konfigurowanie przepływów, 118
 - wykrywanie widoków bezstanowych, 394
 - programowy
 - dostęp do atrybutów komponentów, 79
 - dostęp do komponentów, 86
 - dostęp do mapy żądania, 95
 - zasięg przepływu, 130
 - przekazywanie, 171
 - atrybutów, 66
 - metod do szablonów, 481
 - parametrów, 52, 71, 80, 477
 - parametrów żądania, 52
 - właściwości, 479
 - przekierowanie, 97, 171, 176
 - przekształcanie komponentu jQuery, 420
 - przełączanie przepływu, 126
 - przełącznik motywów, 460, 461
 - przepływ, 105, 108
 - schedule, 131
 - sterowania, 71
 - przepływy
 - z komponentami, 112
 - zagnieżdżone, 114
 - przesłanie mechanizmów wizualizacji, 209
 - przestrzeń nazw, 150
 - przesyłanie
 - danych, 76
 - hasel, 79
 - plików, 347, 356, 364, 368
 - plików na serwer, 341
 - plików z podglądem, 357
 - przetwarzanie
 - natychmiastowe, 22
 - opóźnione, 22
 - żądań zwrotnych, 324
 - przycisk
 - Anuluj, 314
 - Wyczyść, 314
 - przypadki
 - nawigacji, 120
 - nawigacji w przepływach, 123
 - pseudozasięg, 133
- ## R
- reguła nawigacyjna, 65
 - rejestrowanie użytkownika, 72
 - relacyjna baza danych, 283
 - responsywne arkusze stylów, 457
 - RichFaces Upload, 370
 - rodzina komponentu, 213
 - rozszerzanie szablonu, 494, 498
- ## S
- schemat szablonu, 495
 - serializacja stanu, 389
 - sesja HTTP, 96
 - singleton, 244

składnia EL, 20
 słowa zastrzeżone, 21
 słowo kluczowe
 @all, 304
 @form, 304
 @none, 304
 @this, 304
 sortowanie, 265
 listy dat, 260
 listy liczb, 260
 tabel, 259
 sprawdzanie obecności atrybutu, 435
 stała CONTENT, 41
 stan widoku, 373, 374
 stosowanie
 atrybutu jsfc, 493
 faceletów, 510
 facet złożonych, 431
 klasy FaceletCache, 499
 komponentów złożonych, 435
 kontraktów, 446
 obiektów kolekcji, 47
 operatora konkatencji, 45
 operatora przypisania, 45
 operatora średnika, 46
 parametrów kontekstu, 52
 parametrów widoku, 55
 pól ukrytych, 78
 strona
 confirm.xhtml, 115, 121, 126
 done.xhtml, 171
 notournament.xhtml, 128
 registration.xhtml, 113
 schedule.xhtml, 116
 sponsored.xhtml, 141
 thanks.xhtml, 118
 struktura katalogów kontraktów, 447
 style, 296
 sygnalizator działania, 338
 szablon PageLayout, 474, 494
 szablony Facelet, 445, 471

Ś

śledzenie okna klienta, 236

T

tabela, 252
 aktualizacja wierszy, 269
 dodawanie wierszy, 272
 edycja wierszy, 269
 filtrowanie, 291
 generowanie, 286
 kolor tła wierszy, 296
 określanie wyglądu, 296
 podświetlanie wiersza, 297
 podział na strony, 280
 sortowanie, 259
 usuwanie wiersza, 267
 wybieranie pojedynczego wiersza, 275
 wybieranie wielu wierszy, 277
 wyświetlanie numerów wierszy, 274
 zagnieżdżanie, 279
 technologia bezstanowa, 391
 teksty warunkowe, 33
 tworzenie
 klasy TagHandler, 507
 komponentów niestandardowych, 398
 komponentów złożonych, 413
 kontraktów, 458
 leniwe, 144
 szablonu, 474
 tabeli, 252
 typ
 komponentu, 400
 mechanizmu wizualizacji, 400
 typy wyliczeniowe, 27

U

ukrywanie atrybutów, 436
 uporządkowanie
 bezwzględne, 157
 częściowe, 157
 ustawianie wartości właściwości, 69
 usuwanie
 wiersza tabeli, 267
 zawartości, 492
 użycie
 atrybutu binding, 81
 kontraktów, 448
 mapy aplikacji, 85

V

VDL, View Declaration Language, 246, 445, 471

W

walidacja, 312, 433

walidator, 56, 59, 161

przesyłanych plików, 355

WAR, Web application ARchive, 164

wartości

właściwości, 69

żądania, 65

wartość

null, 40, 155, 435

void, 174

warunkowe wyświetlanie żądań, 324

wczytywanie leniwe, 283

wersjonowanie

aplikacji, 199

zasobów, 203

węzeł

początkowy, 105

powrotu, 105

widoku, 106

wiązka zasobów, 159

widok, 98, 106, 373

widoki

bezstanowe, 392

fizyczne, 378

logiczne, 378

Window ID API, 235

wizualizacja JSF, 209

właściwości

komponentów zarządzanych, 24

zagnieżdżone komponentów zarządzanych, 25

właściwość, 29

first, 280

responseCode, 308

responseText, 308

responseXML, 308

rowCount, 280

rows, 280

source, 308

wstrzykiwanie

komponentów, 83, 144–147

sesyjnych komponentów, 164

SQL, 396

wtyczka ddSlick, 460

wybieranie

pojedynczego wiersza, 275

wielu wierszy, 277

wygląd

aplikacji, 465

komponentów interfejsu użytkownika, 451

komponentu złożonego, 420

tabeli, 296, 448

wyjątek, 220

AbortProcessingException, 182

java.io.NoSerializableException, 390

java.io.NotSerializableException, 390

ViewExpiredException, 376, 378, 386

wykrywanie widoków bezstanowych, 394

wyrażenia

EL zasięgów niestandardowych, 136

odwołujące się do metod, 31

wartościowe, 22

wyrażenie lambda

ciało funkcji, 46

operator lambdy, 46

parametry, 46

wyświetlanie

numerów wierszy, 274

żądań zwrotnych, 324

wywołanie

akcji, 62

metody, 106

przepływu, 106

wzbogacanie obrazka o akcje, 429

X

XML, 152

XSS, Cross-site scripting, 395

Z

zagnieżdżanie tabel, 279

zależny pseudozasięg, 133

zapis na dysku, 353

zapisywanie stanu widoku, 373, 385

częściowe, 374

na kliencie, 375

na serwerze, 375

w bazie danych, 379

zarządzanie stanem, 373

- zasięg
 - aplikacji, 100, 144, 145
 - bezkontekstowy, 133
 - CDI, 90
 - Flash, 71
 - JSF, 90
 - konwersacji, 102
 - niestandardowy, 134
 - none, 134, 144
 - normalny, 133
 - przepływu, 105, 130, 318
 - sesji, 95, 144, 145
 - widoku, 98, 315, 392
 - żądania, 92, 144, 316
- zasięgi
 - CDI, 92
 - JSF, 92
- zasoby, 198
 - CSS, 205
 - JS, 205
- zastosowanie wartości żądania, 65
- zdarzenia systemowe, 183, 189
- zdarzenie
 - onfocus, 217
 - PostKeepFlashValueEvent, 189
 - PostPutFlashValueEvent, 189
 - PostRestoreStateEvent, 190
 - PostValidateEvent, 163, 187
 - PreClearFlashEvent, 189
 - PreRemoveFlashValueEvent, 189
 - PreRenderView, 61, 63
- zintegrowane środowisko programistyczne, 90
- zmiana widoku, 322
- zmienna
 - CLASSPATH, 199
 - msg, 160
- znacznik
 - <absolute-ordering>, 158
 - <behavior-id>, 216
 - <c:forEach>, 512
 - <c:set>, 75
 - <cc:actionSource>, 429, 430
 - <cc:editableValueHolder>, 434
 - <cc:facet>, 432
 - <c:if>, 511
 - <cc:insertChildren>, 428
 - <cc:insertFacet>, 432, 433
 - <cc:interface>, 420
 - <cc:renderFacet>, 432
 - <confirmDelete>, 217
 - <datalist>, 428
 - <div>, 450
 - <el-resolver>, 40
 - <else>, 174
 - <f:actionListener>, 179, 182
 - <f:ajax>, 311, 332
 - <f:attribute>, 66, 163
 - <f:converter>, 59
 - <f:event>, 183
 - <f:metadata>, 55, 62
 - <f:param>, 52, 66, 75, 328
 - <f:resetValues>, 314
 - <f:setPropertyActionListener>, 69
 - <f:validator>, 59
 - <f:view>, 159, 453
 - <f:viewAction>, 61–64
 - <f:viewParam>, 55, 56, 61, 487
 - <flow-call>, 115
 - <flow-definition>, 107
 - <flow-reference>, 115
 - <flow-return>, 108
 - <from-outcome>, 108
 - <h:button>, 170
 - <h:commandButton>, 57, 217
 - <h:commandLink>, 431
 - <h:dataTable>, 41, 82, 251, 255, 283
 - <h:graphicImage>, 359, 431
 - <h:head>, 345
 - <h:inputFile>, 350
 - <h:inputSecret>, 79
 - <h:inputText>, 57, 110, 218, 345
 - <h:link>, 58, 170
 - <h:message>, 315
 - <h:messages>, 225
 - <h:outputFormat>, 160
 - <h:outputText>, 270
 - <h:panelGroup>, 442
 - <h:selectOneRadio>, 275
 - <if>, 174
 - <inbound-parameter>, 115, 117
 - <managed-bean>, 153
 - <managed-property>, 153
 - <message-bundle>, 161
 - <multipart-config>, 354
 - <null-value/>, 155
 - <o:viewParam>, 62
 - <option>, 463
 - <outbound-parameter>, 115, 116

<render-kit>, 224
 <resource-bundle>, 161
 <start-node>, 108
 <tag-name>, 216
 <to-view-id>, 174
 <ui:component>, 473
 <ui:composition>, 472
 <ui:debug>, 473, 491
 <ui:decorate>, 474, 481
 <ui:define>, 472
 <ui:fragment>, 473, 481, 486, 511
 <ui:include>, 472, 487, 489, 512
 <ui:insert>, 472
 <ui:param>, 473, 477, 479, 489
 <ui:remove>, 474, 492
 <ui:repeat>, 473, 484, 510, 512
 <value>, 153
 <view>, 107, 128

znak wieloznaczny, 170
 zwrócenie wyniku przepływu, 107

Ż

źródło, 47

ż

żądania

AJAX, 321, 327
 HTTP, 92, 96
 początkowe, 322
 zwrotne, 322

żądanie

GET, 57, 62, 169
 POST, 169

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

JavaServer Faces 2.2

Mistrzowskie programowanie

JavaServer Faces (JSF) 2.2 jest dziś najważniejszym frameworkiem służącym do budowy internetowych interfejsów użytkownika aplikacji sieciowych i stanowi podstawowy komponent platformy Java Enterprise Edition. W porównaniu z poprzednimi wydaniem JSF 2.2 został wzbogacony o wiele nowych funkcji. Z pewnością przydadzą się one programistom i znacznie podniosą efektywność pracy nad aplikacjami.

Niniejsza książka jest skierowana do programistów korzystających z JSF. Przedstawiono w niej wszystkie istotne zagadnienia związane z tworzeniem aplikacji za pomocą JSF 2.2. Czytelnik z pewnością doceni zarówno przejrzyste instrukcje, pozwalające na pełne wykorzystanie możliwości JSF 2.2, jak i liczne ćwiczenia, które stanowią doskonałą pomoc w tworzeniu imponujących aplikacji internetowych.

JavaServer Faces 2.2 — to framework dla mistrzów programowania w Javie!



W książce omówiono:

- język wyrażeń (EL) z uwzględnieniem najważniejszych aspektów EL 2.2 oraz EL 3.0
- zagadnienia związane z komunikacją w JSF oraz zasięgami JSF 2.2
- artefakty JSF i ich konfigurację
- język HTML5, technologię AJAX oraz pojęcie stanu widoku JSF
- tworzenie komponentów niestandardowych i komponentów złożonych

Anghel Leonard — jest niekwestionowanym autorytetem w dziedzinie programowania w Javie. Posiada kilkunastoletnie doświadczenie w pracy z Javą SE, Javą EE oraz z wieloma frameworkami Javy. Jest autorem kilkudziesięciu książek i artykułów poświęconych różnym technologiom Javy. Ostatnio tworzy świetne aplikacje internetowe na potrzeby systemów GIS.

[PACKT] open source
PUBLISHING community experience distilled

Helion

44765 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne

☎ 0 801 339900

📠 0 601 339900

Sprawdź najnowsze promocje:

● <http://helion.pl/promocje>

● http://helion.pl/najchetedniej_czytane

● <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

● <http://helion.pl/nawosci>

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

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2419-0



9 788328 324190

Informatyka w najlepszym wydaniu

cena: 89,00 zł