Technologia i rozwiązania

JavaServer Faces 2.2 Mistrzowskie programowanie

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

Anghel Leonard

Helion

[PACKT] PUBLISHING

Tytuł oryginału: Mastering JavaServer Faces 2.2

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-2419-0

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

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.

Spis treści

8

ROZDZIAŁ

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 doskonałe 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 `<span>`), 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/vdldocs-facelets/jsf/element.html*.

Kompletna implementacja takiego rozwiązania jest dostępna w kodach dołączonych do książki jako aplikacja *przyklad_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/javaee/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. Aż 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/javaee/7/javaserver-faces-2-2/vdldocs-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 serwlet; 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/javaee/6/tutorial/doc/glrbb.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 *przyklad_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: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 *przyklad_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[] chunck = new byte[1024];
  while ((bytesRead = inputStream.read(chunck)) != -1) {
    outputStream.write(chunck, 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 *przyklad_8_06*. Jeśli wolisz pobierać nazwę pliku z nagłówka `content-disposition`, to takie rozwiązanie zostało zaimplementowane jako aplikacja *przyklad_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/javaee/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 *przyklad_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 *przyklad_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 *przyklad_8_10*.

W drugim z opisanych wcześniej przypadków kombinację znaczników `<f:ajax>` oraz `<h:command ↪Button>` 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 *przyklad_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 przesyłał 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 przesyłać 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[] chunck = new byte[1024];
      while ((bytesRead = inputStream.read(chunck)) != -1) {
        outputStream.write(chunck, 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 *przyklad_8_13*. Z kolei aplikacja prezentująca podgląd obrazka jest dostępna jako *przyklad_8_12*.

Proces weryfikacji wybranego pliku można usunąć z serwera i przenieść do przeglądarki. Taka wersja aplikacji także jest dostępna w kodach dołączonych do książki, jako *przyklad_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 przesyłać 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[] chunck = new byte[1024];
      while ((bytesRead = inputStream.read(chunck)) != -1) {
        outputStream.write(chunck, 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 *przyklad_8_15*.

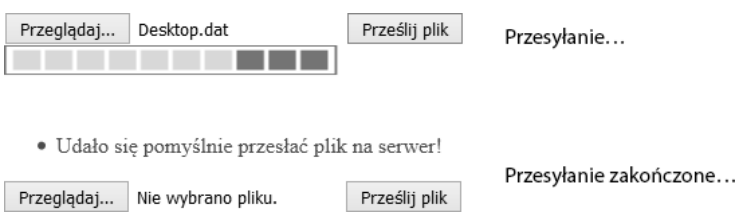
Przesyłanie plików i nieokreślony pasek postępów

W przypadku przesyłania niewielkich plików cały proces trwa bardzo krótko, kiedy jednak w grę wchodzą 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 *przyklad_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 `XMLHttp`
`↪Request` 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>
```

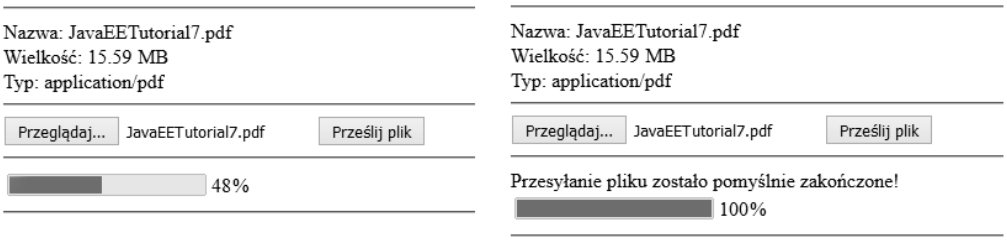
A poniżej został przedstawiony element HTML do przesyłania plików korzystający z powyższego kodu JavaScript:

```
  <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 *przyklad_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

B

C

D

E

F

G

H

I

J

K

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ź

Ż

JavaServer Faces 2.2 Mistrzowskie programowanie

JavaServer Faces (JSF) 2.2 jest dziś najważniejszym fra- meworkiem służącym do budowy internetowych interfejsów użytkownika aplikacji sieciowych i stanowi podstawowy komponent platformy Java Enterprise Edition. W porównaniu z poprzednimi wydaniami 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
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-2419-0

cena: 89,00 zł