Java EE – dzień 3

v3.1

Plan

1. Strony JSP
2. Model View Controller
3. Filtry

Strony JSP

JSP – ogólna charakterystyka

JSP czyli **Java Server Pages** – jest to technologia, która pozwala osadzać kod javy w plikach html.

Strona **JSP** – jest to dokument tekstowy, który może zawierać dwa typy danych:

- statyczne (np. html, xml),
- dynamiczne.

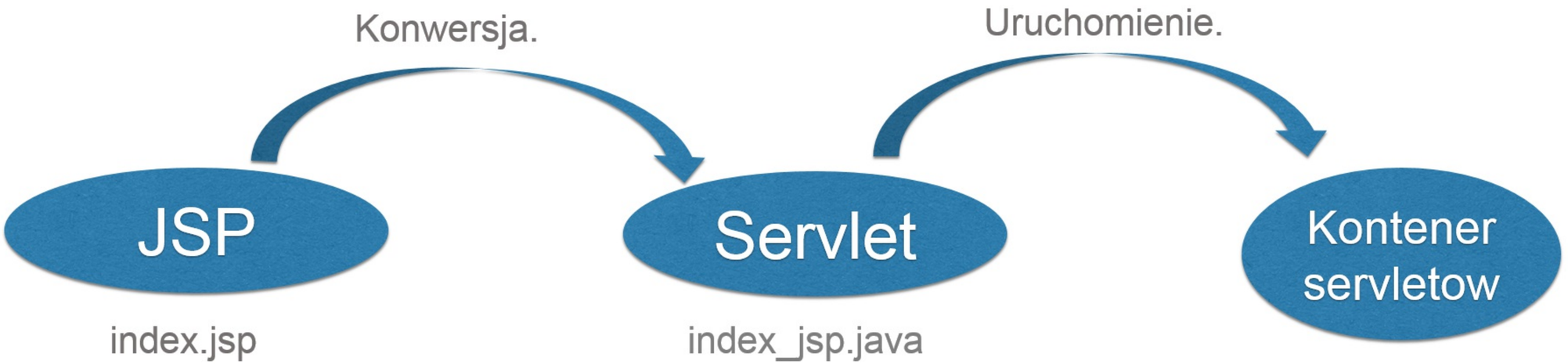
Strony JSP są automatycznie tłumaczone przez serwer aplikacji (np. Tomcat) na postać servletu.

W celu obsługi żądania, wygenerowany i skompilowany **`servlet`**, jest wywoływany przez serwer aplikacji.

Oznacza to, że mamy dostęp do pełnej technologii servletów z poziomu pliku JSP.

JSP

Proces uruchamiania strony JSP moglibyśmy zaprezentować w poniższy sposób:



JSP – zalety

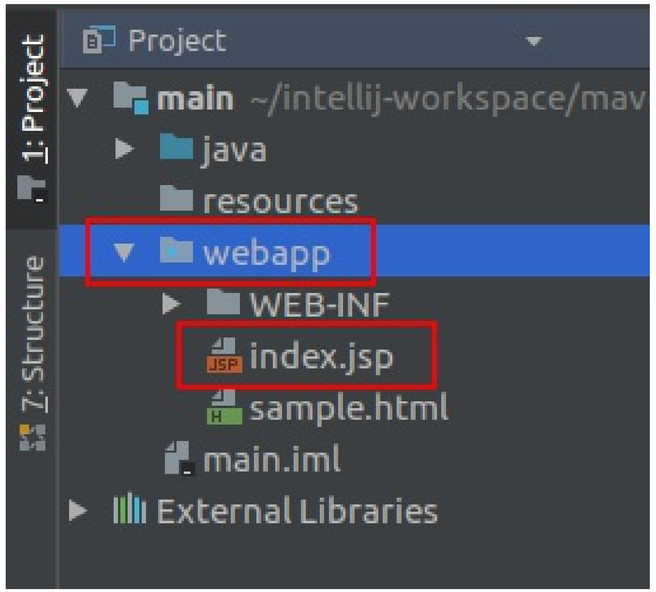
- Łatwiejsze tworzenie i wyświetlanie kodu html względem metod używanych w servletach (np.: **`println`**).
- Kod JSP jest tłumaczony na **`servlet`**, zawiera więc tożsame możliwości – co powoduje, że znając technologię servletów jesteśmy w stanie w szybkim czasie przyswoić JSP.
- Stosowanie technologii JSP jest wygodne, jeżeli prezentujemy dużą ilość kodu html.

Jest jednak także minus korzystania z JSP – umieszczając logikę aplikacji w ramach html – bardzo zmniejszamy czytelność.

JSP – lokalizacja plików

Pliki **`jsp`** umieszczamy w katalogu **`webapp`** naszego projektu.

Dzięki temu będziemy mieli do nich dostęp bezpośrednio z przeglądarki internetowej.



JSP – przykładowy plik

Przykład pliku **JSP** – zawiera zarówno elementy języka HTML i EL:

```
<html>
<head>
<title>Fahrenheit to Celsius</title>
</head>
<body>
<jsp:include page="header.jspf"/>
    <!-- kod html -->
<%-- ${param.foo} --%>
</body>
</html>
```

JSP – przykładowy plik

Przykład pliku **JSP** – zawiera zarówno elementy języka HTML i EL:

```
<html>
<head>
<title>Fahrenheit to Celsius</title>
</head>
<body>
<jsp:include page="header.jspf"/>
    <!-- kod html -->
<%-- ${param.foo} --%>
</body>
</html>
```

- Akcja

JSP – przykładowy plik

Przykład pliku **JSP** – zawiera zarówno elementy języka HTML i EL:

```
<html>
<head>
<title>Fahrenheit to Celsius</title>
</head>
<body>
<jsp:include page="header.jspf"/>
    <!-- kod html -->
<%-- ${param.foo} --%>
</body>
</html>
```

- Akcja
- Komentarz

JSP – uruchomienie

Plik **jsp** możemy wywołać bezpośrednio z poziomu przeglądarki.

Prześledźmy zatem co oznaczają kolejne elementy poniższego adresu, który przykładowo możemy wpisać w przeglądarkę:

http://localhost:8080/jspFiles/file.jsp

- **http://localhost:8080** – adres naszego serwera,
- **jspFiles** – folder podrzędny w katalogu **webapp** – nie jest wymagany, pliki można umieszczać bezpośrednio w katalogu **webapp**,
- **file.jsp** – nazwa pliku jsp.

W kolejnym rozdziale dowiemy się jak połączyć pliki **jsp** oraz **servlety**.

Język wyrażeń – Expression Language

Expression Language (**EL**) – powstał, aby wprowadzić uproszczenie kodu tradycyjnych znaczników JSP.

Wyrażenia mają bardzo prostą składnię, wystarczy użyć notacji:

`${zmienna.pole}`

Przykład:

Pobranie i wyświetlenie parametru o nazwie **`myname`** będzie miało postać:

`${param.myname}`

W pliku JSP wyrażenia **EL** mogą występować:

- w statycznym html:

```
<h1>${param.myname}</h1>
```

- jako atrybut innych znaczników:

```
<c:out value="${login}"
       default="brak" />
```

Język wyrażeń – Expression Language

Expression Language (**EL**) – powstał, aby wprowadzić uproszczenie kodu tradycyjnych znaczników JSP.

Wyrażenia mają bardzo prostą składnię, wystarczy użyć notacji:

`${zmienna.pole}`

Przykład:

Pobranie i wyświetlenie parametru o nazwie **`myname`** będzie miało postać:

`${param.myname}`

W pliku JSP wyrażenia **EL** mogą występować:

- w statycznym html:

```
<h1>${param.myname}</h1>
```

- jako atrybut innych znaczników:

```
<c:out value="${login}"
       default="brak" />
```

  - **`<c:out>`** – znacznik z biblioteki JSTL (elementy tej biblioteki będziemy omawiać podczas kursu).

EL – obiekty predefiniowane

W wyrażeniach mamy dostęp do predefiniowanych zmiennych, np:

- `pageContext` – kontekst strony JSP (zawiera m.in. obiekty request, response),
- `param` – pojedynczy parametr żądania,
- `paramValues` – tablica parametru żądania,
- `header` – wartość nagłówka HTTP,
- `headerValues` – tablica wartości nagłówka HTTP,
- `cookie` – wartość ciasteczka,
- `initParam` – wartość parametru inicjalizacyjnego.

EL – pobieranie wartości

Przykłady:

- Pobieranie parametru żądania o nazwie **bar**:

```
${param.bar}
```

- Pobieranie nazwy aplikacji (kontekstu):

```
${pageContext.request.contextPath}
```

- Pobieranie wartości ciasteczka o nazwie **foo**:

```
${cookie.foo.value}
```

EL – operatory

W EL występują poniższe operatory:

- **arytmetyczne:** `+ - * / % mod div`
- **logiczne:** `and or not && || !`
- **porównania:** `== != => > <= < lt gt le ge eq ne`
- **empty**
- **warunkowy:** `A ? B : C`

Przykłady:

```
${2*8}
${8/2}
${8 div 2}
${8 lt 2}
${8 <= 2}
${false || false}
${true and true}
${empty param.name ? "empty" : "not"}
${empty param.name}
```

EL – komentarze

Aby dodać komentarze w JSP, używamy poniższej konstrukcji:

```
<%-- Tu umieszczamy komentarz --%>
```

Include

Realizując projekty oparte o technologię JSP zauważymy z czasem, że duża ilość kodu jest powtarzalna – przykładem będzie **`header`** i **`footer`** strony.

Prostym sposobem na rozwiązanie problemu będzie podział strony na tzw. fragmenty, wg wzoru przedstawionego obok, a następnie załączanie ich w każdym pliku, który powinien posiadać te elementy.

Include

Za pomocą dyrektywy **include** mamy możliwość fizycznego włączenia zawartości jednego pliku w drugi.

Daje to duże możliwości wyłączania powtarzanego kodu do oddzielnych plików.

Ułatwia to znacznie ewentualne modyfikacje elementów wspólnych w naszej aplikacji.

Rozszerzenie załączanego pliku może być dowolne (np. jsp, html).

Aby odróżnić pliki aplikacji od plików używanych jako fragmenty można użyć do ich oznaczenia rozszerzenia **jspf** – nie jest to jednak konieczne – poprawne będzie także użycie rozszerzenia **jsp**.

```
<html>
  <body>
    <%@ include file="header.jspf" %>
    <%@ include file="footer.jspf" %>
  </body>
</html>
```

JSP

Warto zapoznać się z Code Conventions udostępnionym przez firmę Oracle dotyczącymi Technologii JSP.

http://www.oracle.com/technetwork/articles/javase/code-convention-138726.html

Zadania

Model View Controller

Co warto wiedzieć na początku?

Wzorzec projektowy (design pattern)

W inżynierii oprogramowania uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych.

Pokazuje powiązania i zależności między klasami oraz obiektami. Ułatwia tworzenie, modyfikację, oraz pielęgnację kodu źródłowego.

- Jest opisem rozwiązania, a nie jego implementacją.
- Wzorce projektowe stosowane są w projektach wykorzystujących programowanie obiektowe.
- Źródło: http://pl.wikipedia.org/wiki/Wzorzec_projektowy_(informatyka)

Model-View-Controller (MVC)

Czym jest MVC?

Jest to wzorzec architektoniczny służący do organizowania struktury aplikacji posiadających graficzne interfejsy użytkownika. Zakłada podział aplikacji na trzy połączone z sobą warstwy: **model**, **widok** i **kontroler**.

Model

Jest reprezentacją pewnego problemu, jej struktury danych. Model jest samodzielny.

Model zajmuje się dostarczaniem danych do wykorzystania w aplikacji. Do poprawnej pracy nie wymaga obecności dwóch pozostałych części MVC. Komunikacja z nimi zachodzi w sposób niejawny.

Model-View-Controller (MVC)

Widok

Jest odpowiedzialny za prezentację danych w obrębie graficznego interfejsu użytkownika.

Może składać się z podwidoków zarządzających mniejszymi elementami składowymi.

Widoki mają bezpośrednie referencje do modeli, z których pobierają dane, gdy otrzymują od kontrolera żądanie ich wyświetlenia.

Możliwe są różne widoki tych samych danych.

Kontroler

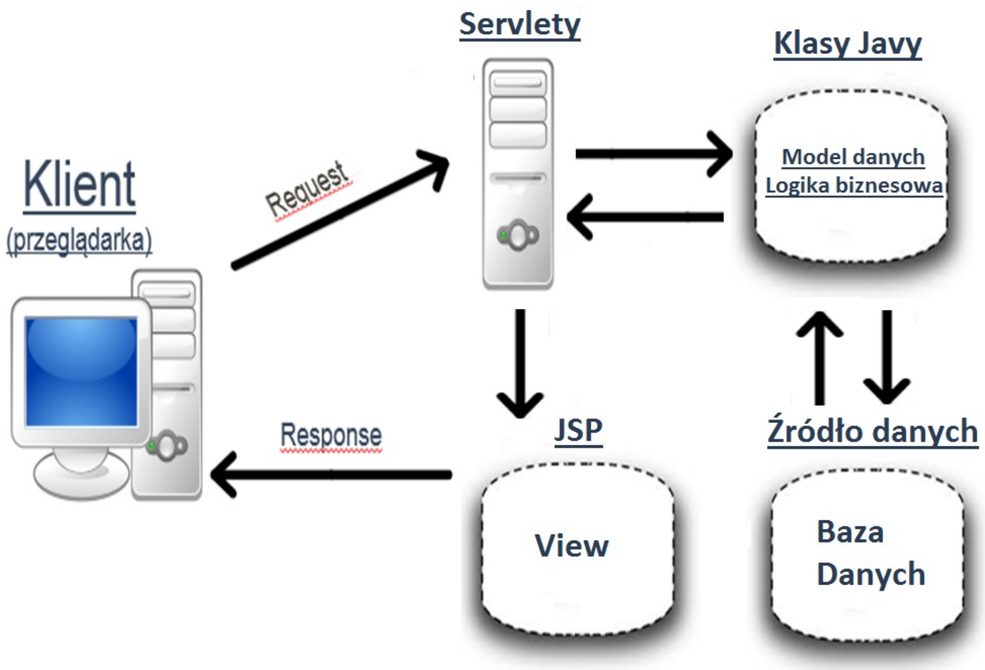
Jego zadaniem jest odbiór, przetworzenie oraz analiza danych wejściowych od użytkownika.

Po przetworzeniu odebranych danych kontroler może wykonać następujące czynności:

- zmienić stan modelu.
- odświeżyć widok.
- przełączyć sterowanie na inny kontroler.

Model-View-Controller (MVC)

Schemat MVC



Jak to działa?

Po wpisaniu w przeglądarkę adresu internetowego odnajdywany jest odpowiedni servlet (kontroler) który ma obsłużyć żądanie.

Servlet wykonuje odpowiednie operacje i przekazuje atrybuty do widoku (plik JSP).

JSP wyświetla otrzymane dane, generując wynikowy html przesłany do przeglądarki.

Warstwa modelu – to np. klasy implementujące znany już nam wzorzec Active Record.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.
  "myAttribute1" - nazwa atrybutu – będziemy z niej korzystać w pliku JSP.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.

  1 - wartość atrybutu – możemy przekazać dowolny obiekt.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.
- Możemy przekazywać atrybuty dowolnych typów.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.
- Możemy przekazywać atrybuty dowolnych typów.
- **"/first.jsp"** - wskazujemy plik JSP do którego zostanie przekazane żądanie.

MVC – przykład - servlet

Poniżej przedstawiona jest dobrze już nam znana metoda **doGet**, przekazująca atrybut do odpowiedniego pliku JSP.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        request.setAttribute("myAttribute1", 1);
        request.setAttribute("myAttribute2", "two");
        getServletContext().getRequestDispatcher("/first.jsp")
            .forward(request, response);
}
```

- Za pomocą metody **setAttribute** ustawiamy atrybut, który następnie odczytamy w pliku **jsp**.
- Możemy przekazywać atrybuty dowolnych typów.
- **"/first.jsp"** - wskazujemy plik JSP do którego zostanie przekazane żądanie.
- Za pomocą metody **forward** przekazujemy żądanie do pliku JSP.

MVC – przykład - JSP

Przekazane w servlecie parametry odbieramy w pliku JSP w zależności od potrzeb w jeden z poniższych sposobów.

```
${myAttribute}

<c:out value="${myAttribute}" default="empty attribute"/>
```

MVC – przykład - JSP

Przekazane w servlecie parametry odbieramy w pliku JSP w zależności od potrzeb w jeden z poniższych sposobów.

```
${myAttribute}

<c:out value="${myAttribute}" default="empty attribute"/>
```

- Zawartość atrybutu wyświetlamy za pomocą EL, podając nazwę parametru umieszczoną pomiędzy znakami **${** oraz **}**

MVC – przykład - JSP

Przekazane w servlecie parametry odbieramy w pliku JSP w zależności od potrzeb w jeden z poniższych sposobów.

```
${myAttribute}

<c:out value="${myAttribute}" default="empty attribute"/>
```

- Zawartość atrybutu wyświetlamy za pomocą EL, podając nazwę parametru umieszczoną pomiędzy znakami **${** oraz **}**
- Wyświetlenie przy pomocy znacznika JSTL - znaczniki zostaną omówione w kolejnym rozdziale.

Szablony

Istnieje wiele alternatywnych rozwiązań szablonowych, stanowiących alternatywę dla JSP.

Część z nich powstała z powodu początkowych ograniczeń technologii JSP – we wczesnych jej wersjach.

Ich możliwości oraz mnogość można zobaczyć:

https://en.wikipedia.org/wiki/Comparison_of_web_template_engines

Szablony

Zalety:

- Ułatwiają podział zadań i obowiązków w zespole.
- Separacja kodu HTML i Java.
- Czytelność kodu.

Dokonując wyboru konkretnego rozwiązania szablonów warto zwrócić uwagę na jego popularność oraz to czy jest nadal rozwijany.

Pamiętajmy że Java jest już dawno pełnoletnia więc cześć rozwiązań może nie być już wspierana.

JSTL

Technologia JSP umożliwia korzystanie z dodatkowych bibliotek znaczników.

W ramach zajęć omówimy bibliotekę JSTL a dokładniej elementy podstawowe tej biblioteki.

Aby zacząć korzystać z biblioteki musimy w naszym pliku JSP dodać jej definicję za pomocą dyrektywy **`taglib`**, w następujący sposób:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

oraz dodać w pliku pom.xml poniższy fragment:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

JSTL

Technologia JSP umożliwia korzystanie z dodatkowych bibliotek znaczników.

W ramach zajęć omówimy bibliotekę JSTL a dokładniej elementy podstawowe tej biblioteki.

Aby zacząć korzystać z biblioteki musimy w naszym pliku JSP dodać jej definicję za pomocą dyrektywy **`taglib`**, w następujący sposób:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

- **`prefix="c"`** - prefiks oznacza, że od takiej wartości będą rozpoczynać się nasze znaczniki np. **`<c:forEach`**.

oraz dodać w pliku pom.xml poniższy fragment:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

JSTL - out

Znacznik JSTL **<c:out>** - służy do wyświetlania wartości zmiennych. Mogą to być zarówno przekazane jako parametry jak również utworzone np. w skryptlecie.

```
<c:out value="<b> bold text </b>" escapeXml="false"/>

<c:out value="${str}" default="default value"/>
```

JSTL - out

Znacznik JSTL **<c:out>** - służy do wyświetlania wartości zmiennych. Mogą to być zarówno przekazane jako parametry jak również utworzone np. w skryptlecie.

```
<c:out value="<b> bold text </b>" escapeXml="false"/>

<c:out value="${str}" default="default value"/>
```

- **escapeXml="false"** - takie ustawienie parametru powoduje, że nie zostaną dodane znaki ucieczki

JSTL - out

Znacznik JSTL **<c:out>** - służy do wyświetlania wartości zmiennych. Mogą to być zarówno przekazane jako parametry jak również utworzone np. w skryptlecie.

```
<c:out value="<b> bold text </b>" escapeXml="false"/>

<c:out value="${str}" default="default value"/>
```

- **escapeXml="false"** - takie ustawienie parametru powoduje, że nie zostaną dodane znaki ucieczki
- **default="default value"** - domyślna wartość.

JSTL - forEach

Znacznik JSTL **<c:forEach>** - oferuje sposób przejścia po elementach tablicy lub kolekcji, w sposób bardzo zbliżony do znanej nam już pętli **for**, schematycznie przedstawionej poniżej:

```
for(Typ nazwaObiektu : Tablica){
}
```

Schematycznie składnia wygląda następująco:

```
<c:forEach var="varName"
           items="myItems">
</c:forEach>
```

JSTL - forEach

Znacznik JSTL **<c:forEach>** - oferuje sposób przejścia po elementach tablicy lub kolekcji, w sposób bardzo zbliżony do znanej nam już pętli **for**, schematycznie przedstawionej poniżej:

```
for(Typ nazwaObiektu : Tablica){
}
```

Schematycznie składnia wygląda następująco:

```
<c:forEach var="varName"
           items="myItems">
</c:forEach>
```

- **var="varName"** - dowolna nazwa zmiennej którą będziemy się posługiwać wewnątrz pętli.

JSTL - forEach

Znacznik JSTL **<c:forEach>** - oferuje sposób przejścia po elementach tablicy lub kolekcji, w sposób bardzo zbliżony do znanej nam już pętli **for**, schematycznie przedstawionej poniżej:

```
for(Typ nazwaObiektu : Tablica){
}
```

Schematycznie składnia wygląda następująco:

```
<c:forEach var="varName"
           items="myItems">
</c:forEach>
```

- **var="varName"** - dowolna nazwa zmiennej którą będziemy się posługiwać wewnątrz pętli.
- **items="myItems"** - tablica, lub kolekcja której elementy będziemy iterować.

JSTL – forEach - przykład

Kod metody **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String[] animals = {"Pies", "Kot", "Nietoperz", "Ważka"};
        request.setAttribute("animals", animals);
        getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
                    .forward(request, response);
    }
```

JSTL – forEach - przykład

Kod metody **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String[] animals = {"Pies", "Kot", "Nietoperz", "Ważka"};
        request.setAttribute("animals", animals);
        getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
                    .forward(request, response);
    }
```

- Tworzymy tablicę napisów.

JSTL – forEach - przykład

Kod metody **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String[] animals = {"Pies", "Kot", "Nietoperz", "Ważka"};
        request.setAttribute("animals", animals);
        getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
                .forward(request, response);
    }
```

- Tworzymy tablicę napisów.
- Ustawiamy atrybut o nazwie **animals**.

JSTL – forEach - przykład

Kod metody **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String[] animals = {"Pies", "Kot", "Nietoperz", "Ważka"};
        request.setAttribute("animals", animals);
        getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
                .forward(request, response);
    }
```

- Tworzymy tablicę napisów.
- Ustawiamy atrybut o nazwie **animals**.
- Wskazujemy stronę jsp, która wyświetli przekazane parametry.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
<c:forEach items="${animals}" var="animal">
    <tr>
        <td>${animal}</td>
    </tr>
</c:forEach>
</table>
```

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
<c:forEach items="${animals}" var="animal">
    <tr>
        <td>${animal}</td>
    </tr>
</c:forEach>
</table>
```

- **var="animal"** - określamy nazwę, którą będziemy się posługiwać w pętli.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
<c:forEach items="${animals}" var="animal">
    <tr>
        <td>${animal}</td>
    </tr>
</c:forEach>
</table>
```

- **var="animal"** - określamy nazwę, którą będziemy się posługiwać w pętli.
- **items="${animals}"** - wskazujemy zmienną do iteracji.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
<c:forEach items="${animals}" var="animal">
    <tr>
        <td>${animal}</td>
    </tr>
</c:forEach>
</table>
```

- **var="animal"** - określamy nazwę, którą będziemy się posługiwać w pętli.
- **items="${animals}"** - wskazujemy zmienną do iteracji.
- Wyświetlanie elementu.

JSTL – forEach - przykład

Ten sam przykład z wykorzystaniem listy stringów metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    List<String> animals = Arrays.asList("Pies", "Kot", "Nietoperz", "Ważka");
    request.setAttribute("animals", animals);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

JSTL – forEach - przykład

Ten sam przykład z wykorzystaniem listy stringów metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    List<String> animals = Arrays.asList("Pies", "Kot", "Nietoperz", "Ważka");
    request.setAttribute("animals", animals);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

- Tworzymy listę napisów. Reszta jest taka sama jak w przykładzie wyżej. Odbiór po stronie JSP nie zmienia się.

JSTL – forEach - przykład

Przykład z wykorzystaniem listy obiektów klasy Book metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    List<Book> books =
    Arrays.asList(
        new Book("Thinking in Java", "Bruce Eckel"),
        new Book("Harry Potter i Komnata Tajemnic", "J.K. Rowling"));
    request.setAttribute("books", books);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

JSTL – forEach - przykład

Przykład z wykorzystaniem listy obiektów klasy Book metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    List<Book> books =
    Arrays.asList(
        new Book("Thinking in Java", "Bruce Eckel"),
        new Book("Harry Potter i Komnata Tajemnic", "J.K. Rowling"));
    request.setAttribute("books", books);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

- Tworzymy listę książek - obiekty klasy Book.

JSTL – forEach - przykład

Przykład z wykorzystaniem listy obiektów klasy Book metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    List<Book> books =
    Arrays.asList(
        new Book("Thinking in Java", "Bruce Eckel"),
        new Book("Harry Potter i Komnata Tajemnic", "J.K. Rowling"));
    request.setAttribute("books", books);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

- Tworzymy listę książek - obiekty klasy Book.
- Używamy konstruktora (title, author).

JSTL – forEach - przykład

Przykład z wykorzystaniem listy obiektów klasy Book metoda **doGet** servletu:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    List<Book> books =
    Arrays.asList(
        new Book("Thinking in Java", "Bruce Eckel"),
        new Book("Harry Potter i Komnata Tajemnic", "J.K. Rowling"));
    request.setAttribute("books", books);
    getServletContext().getRequestDispatcher("/mvc/forEach.jsp")
            .forward(request, response);
}
```

- Tworzymy listę książek - obiekty klasy Book.
- Używamy konstruktora (title, author).
- Ustawiamy atrybut o nazwie **books**.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
    <c:forEach items="${books}" var="book">
        <tr>
            <td>${book.title}, ${book.author}</td>
        </tr>
    </c:forEach>
</table>
```

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
    <c:forEach items="${books}" var="book">
        <tr>
            <td>${book.title}, ${book.author}</td>
        </tr>
    </c:forEach>
</table>
```

- **var="book"** - określamy nazwę, którą będziemy się posługiwać w pętli.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
    <c:forEach items="${books}" var="book">
        <tr>
            <td>${book.title}, ${book.author}</td>
        </tr>
    </c:forEach>
</table>
```

- **var="book"** - określamy nazwę, którą będziemy się posługiwać w pętli.
- **items="${books}"** - wskazujemy zmienną do iteracji.

JSTL – forEach - przykład

Kod pliku jsp - **forEach.jsp**

```
<table>
    <c:forEach items="${books}" var="book">
        <tr>
            <td>${book.title}, ${book.author}</td>
        </tr>
    </c:forEach>
</table>
```

- **var="book"** - określamy nazwę, którą będziemy się posługiwać w pętli.
- **items="${books}"** - wskazujemy zmienną do iteracji.
- Wyświetlanie elementu. Klasa Book powinna posiadać metody **getTitle()** i **getAuthor()**.

JSTL – forEach - przykład

Przykładowa klasa Book z pliku **Book.java**

```
public class Book {
        private String title;
        private String author;
        public Book(String title, String author) {
                this.title = title;
                this.author = author;
        }
        public String getTitle() {
                return title;
        }
        public String getAuthor() {
                return author;
        }
}
```

JSTL – forEach

Oprócz elementów **`items`** oraz **`var`** w ramach pętli możemy określić dodatkowe atrybuty np.:

Atrybut	Opis	Wymagalność	Wartość domyślna
`begin`	Element od którego zaczynamy iterować.	Nie	0
`end`	Element do którego iterujemy.	Nie	Ostatni element
`step`	Skok o jaki przesuwa się nasza pętla.	Nie	1
`varStatus`	Nazwa zmiennej dla statusu pętli.	Nie	–

JSTL – forEach - varStatus

Omawiany atrybut **`varStatus`** – to zmienna typu: **`LoopTagStatus`** posiadająca kilka przydatnych metod.

- **`index`** / **`getIndex()`** – liczony od 0 aktualnie iterowany element.
- **`count`** / **`getCount()`** – liczony od 1 obrót pętli.
- **`first`** / **`isFirst()`** - wartość typu logicznego określająca czy jest to element pierwszy.
- **`last`** / **`isLast()`** – wartość typu logicznego określająca czy jest to element ostatni.

Zakładając że określiliśmy atrybut **`varStatus`** w następujący sposób:

```
<c:forEach varStatus="loopStatus">
```

Mamy możliwość wywołać np.:

```
${loopStatus.isFirst()}
```

```
${loopStatus.first}
```

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

<c:out value="${animal}"/> - Wyświetlanie elementu.

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

Ustawiony atrybut **begin="2"** – więc pierwszym elementem jest **Nietoperz** . Elementy liczymy od 0.

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

count można porównać do liczby aktualnych obrotów pętli.

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

Następny wynik to **Jeż** z powodu ustawionego atrybutu **step="2"**.

JSTL – forEach - przykład

Dla poniższej tablicy napisów przekazanej z servletu:

```
String[] animals = { "Pies", "Kot", "Nietoperz", "Ważka", "Jeż",
                     "Surykatka", "Dziobak" };
```

w wyniku wykonania pętli **forEach** w jsp:

```
<c:forEach var="animal" begin="2" end="5" step="2" varStatus="theCount"
           items="${animals}">
  index = ${theCount.index}, count = ${theCount.count}, <c:out value="${animal}"/>
</c:forEach>
```

otrzymamy poniższy wynik:

```
index = 2, count = 1, Nietoperz
index = 4, count = 2, Jeż
```

Kolejnego elementu nie ma ponieważ atrybut **end="5"**.

JSTL - forTokens

Znacznik JSTL **<c:forTokens>** - oferuje funkcjonalność zbliżoną do znanej nam klasy **StringTokenizer**.

```
<c:forTokens items="www.coderslab.pl" delims="." var="site">
  <c:out value="${site}"/> <br/>
</c:forTokens>
```

JSTL - forTokens

Znacznik JSTL **<c:forTokens>** - oferuje funkcjonalność zbliżoną do znanej nam klasy **StringTokenizer**.

```
<c:forTokens items="www.coderslab.pl" delims="." var="site">
  <c:out value="${site}"/> <br/>
</c:forTokens>
```

- **items="www.coderslab.pl"** - wartość do podziału.

JSTL - forTokens

Znacznik JSTL **<c:forTokens>** - oferuje funkcjonalność zbliżoną do znanej nam klasy **StringTokenizer**.

```
<c:forTokens items="www.coderslab.pl" delims="." var="site">
  <c:out value="${site}"/> <br/>
</c:forTokens>
```

- **items="www.coderslab.pl"** - wartość do podziału.
- **delims="."** - znak podziału.

JSTL - forTokens

Znacznik JSTL **<c:forTokens>** - oferuje funkcjonalność zbliżoną do znanej nam klasy **StringTokenizer**.

```
<c:forTokens items="www.coderslab.pl" delims="." var="site">
  <c:out value="${site}"/> <br/>
</c:forTokens>
```

- **items="www.coderslab.pl"** - wartość do podziału.
- **delims="."** - znak podziału.
- **var="site"** - zmienna, którą będziemy się posługiwać w pętli.

JSTL - forTokens

Znacznik JSTL **<c:forTokens>** - oferuje funkcjonalność zbliżoną do znanej nam klasy **StringTokenizer**.

```
<c:forTokens items="www.coderslab.pl" delims="." var="site">
  <c:out value="${site}"/> <br/>
</c:forTokens>
```

- **items="www.coderslab.pl"** - wartość do podziału.
- **delims="."** - znak podziału.
- **var="site"** - zmienna, którą będziemy się posługiwać w pętli.
- Wyświetlanie wartości.

Instrukcje warunkowe - if

W JSTL można korzystać z instrukcji warunkowej **`if`**.

Służy do tego tag:

`<c:if test="warunek"> </c:if>`

Należy zwrócić uwagę, że nie występuje on w wersji z **`else`** – jak zapewne byśmy się tego spodziewali.

```
<c:if test="${not empty param.myName}">
  Jest ok - jest param
</c:if>
```

Instrukcje warunkowe - choose

Do dyspozycji mamy również tag **choose** – jego działanie można porównać do znanej nam konstrukcji **switch** ... **case**.

```
<c:choose>
  <c:when test="${condition1}">
    // Jeżeli warunek jest prawdziwy
  </c:when>
  <c:when test="${condition2}">
    // Jeżeli warunek jest prawdziwy
  </c:when>
  <c:otherwise>
    // Jeżeli żaden powyższy nie był prawdziwy
  </c:otherwise>
</c:choose>
```

JSTL – url

Do budowy poprawnych odnośników w naszej aplikacji z użyciem **JSTL** wykorzystujemy tag:

```
<c:url value = "/urlAddress"/>
```

Przykład odnośnika:

```
<a href='<c:url value="/Servlet02" />'>Link</a>
```

Przykład załączania css:

```
<link href='<c:url value="/dist/css/sb-admin-2.css" />' rel="stylesheet">
```

JSTL

Oprócz poznanej już standardowej biblioteki znaczników występują jeszcze inne bardziej wyspecjalizowane:

- **Formatujące**

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

- **XML**

```
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
```

- **SQL**

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

- **Funkcje**

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

JSTL – SQL

Mimo udostępnienia przez bibliotekę JSTL możliwości nawiązania połączenia z bazą danych z poziomu pliku SQL – nie jest to zalecana praktyka i nie będziemy omawiali takiego sposobu.

Rozwiązaniem preferowanym w nowoczesnych aplikacjach jest wyszczególnienie odpowiednich warstw, z których każda ma odrębną rolę.

Omówieniem takiego podziału na przykładzie servletów i JSP zajmiemy się w następnym module.

JSTL - SQL

Omówienie wszystkich możliwych znaczników z przykładami:

http://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm

http://o7planning.org/en/10429/java-jsp-standard-tag-library-jstl-tutorial

Zadania

Servlety i JDBC

Import sterownika

Aby dodać sterownik MySQL wystarczy dodać następującą zależność w pliku **pom.xml**:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency>
```

Zadania

Filtry

Filtry

Filtry – to mechanizm, który daje możliwość przechwycenia żądania, przychodzącego do servletu, oraz generowanej przez niego odpowiedzi.

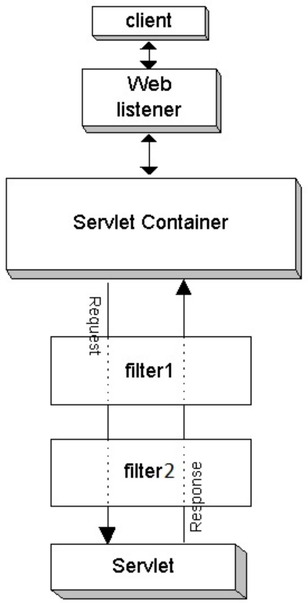
Może modyfikować nagłówki żądania i odpowiedzi.

Filtry mogą służą do:

- mechanizmu uwierzytelniania i autoryzacji (logowanie i uprawnienia)
- logowania
- konwersji danych

Filtry

Schematyczną zasadę aplikacji filtrów przedstawia rysunek:



Zalety filtrów:

- Jest niezależny od servletu, do którego jest przypinany
- Możemy przypinać wiele filtrów do jednego adresu

Filtry

Filtrem jest klasa implementująca interfejs **javax.servlet.Filter**.

Filtr posiada metody:

`void init(FilterConfig filterConfig)` – jest wywoływana przy utworzeniu filtra.

`void destroy()` – wywoływana przy zakończeniu działania filtru

`void doFilter (ServletRequest, ServletResponse, FilterChain)` – główna metoda realizująca zadania filtra.

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Metoda realizująca logikę filtra.

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Właściwa logika filtra.

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Za pomocą tej metody przekazujemy obsługę do metody **doFilter** kolejnego filtra – jeżeli istnieje lub przechodzimy do servletu.

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Wymagana przez interfejs metoda – zazwyczaj pozostaje pusta.

Filtr – przykład

W **IntelliJ** filtr możemy stworzyć wybierając **New –> Filter**.

```
public class LogInfo implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException, ServletException {
        System.out.println("Zapis do loga");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
}
```

Metoda inicjalizacji – możemy ją wykorzystać do pobrania i ustawienia parametrów inicjalizacji filtra – analogicznie jak ma to miejsce w przypadku servletów.

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Określamy nazwę filtra.

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Podajemy pełną pakietową nazwę klasy.

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Opcjonalnie możemy zdefiniować parametry – tak jak w przypadku servletów.

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Określamy mapowanie dla filtra.

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Podajemy nazwę określoną w tagu **<filter-name>** .

Filtry – konfiguracja

Następnym krokiem jest określenie adresów dla jakich filtr ma zostać uruchomiony.

Definicje określamy w pliku **web.xml** za pomocą tagu **<filter>**

lub za pomocą adnotacji, np:

```
@WebFilter("/*")
public class LogInfo implements Filter{
    // kod filtra
}
```

```
<filter>
  <filter-name>LogInfo</filter-name>
  <filter-class>
    pl.coderslab.LogInfo
  </filter-class>
  <init-param>
    <param-name>foo</param-name>
    <param-value>bar</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogInfo</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Definiujemy dla jakich adresów url filtr ma zostać wywołany /* – oznacza, że zostanie wywołany dla wszystkich.

Filtr – kodowanie

Kodowanie możemy ustawić bezpośrednio w każdej metodzie **`doGet`** servletu .

W celu uniknięcia powielania kodu możemy w tym celu wykorzystać filtr.

```
public class EncodingFilter implements Filter {
    private String charsetEncoding = "utf-8";
    private String contentType = "text/html";
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filterChain) throws IOException, ServletException {
        request.setCharacterEncoding(charsetEncoding);
        response.setContentType(contentType);
        response.setCharacterEncoding(charsetEncoding);
        filterChain.doFilter(request, response);
    }
    public void destroy() {}
}
```

Filtr – kodowanie

Kodowanie możemy ustawić bezpośrednio w każdej metodzie **`doGet`** servletu .

W celu uniknięcia powielania kodu możemy w tym celu wykorzystać filtr.

```
public class EncodingFilter implements Filter {
    private String charsetEncoding = "utf-8";
    private String contentType = "text/html";
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filterChain) throws IOException, ServletException {
        request.setCharacterEncoding(charsetEncoding);
        response.setContentType(contentType);
        response.setCharacterEncoding(charsetEncoding);
        filterChain.doFilter(request, response);
    }
    public void destroy() {}
}
```

Ustawiamy kodowanie dla żądania.

Filtr – kodowanie

Kodowanie możemy ustawić bezpośrednio w każdej metodzie **`doGet`** servletu .

W celu uniknięcia powielania kodu możemy w tym celu wykorzystać filtr.

```
public class EncodingFilter implements Filter {
    private String charsetEncoding = "utf-8";
    private String contentType = "text/html";
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filterChain) throws IOException, ServletException {
        request.setCharacterEncoding(charsetEncoding);
        response.setContentType(contentType);
        response.setCharacterEncoding(charsetEncoding);
        filterChain.doFilter(request, response);
    }
    public void destroy() {}
}
```

Ustawiamy typ zwracanej wartości.

Filtr – kodowanie

Kodowanie możemy ustawić bezpośrednio w każdej metodzie **`doGet`** servletu .

W celu uniknięcia powielania kodu możemy w tym celu wykorzystać filtr.

```
public class EncodingFilter implements Filter {
    private String charsetEncoding = "utf-8";
    private String contentType = "text/html";
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain filterChain) throws IOException, ServletException {
        request.setCharacterEncoding(charsetEncoding);
        response.setContentType(contentType);
        response.setCharacterEncoding(charsetEncoding);
        filterChain.doFilter(request, response);
    }
    public void destroy() {}
}
```

Ustawiamy kodowanie dla odpowiedzi.

Filtr – kodowanie – definicja

```
<filter>
  <filter-name>EncodingFilter</filter-name>
  <filter-class>pl.coderslab.EncodingFilter </filter-class>
  <init-param>
      <param-name>charsetEncoding</param-name>
      <param-value>UTF-8</param-value></init-param>
  <init-param>
    <param-name>contentType</param-name>
    <param-value>text/html</param-value></init-param>
</filter>
  <filter-mapping>
    <filter-name>EncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Filtr – kodowanie – definicja

```
<filter>
  <filter-name>EncodingFilter</filter-name>
  <filter-class>pl.coderslab.EncodingFilter </filter-class>
  <init-param>
      <param-name>charsetEncoding</param-name>
      <param-value>UTF-8</param-value></init-param>
  <init-param>
    <param-name>contentType</param-name>
    <param-value>text/html</param-value></init-param>
</filter>
  <filter-mapping>
    <filter-name>EncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Określamy nazwę filtra oraz klasę.

Filtr – kodowanie – definicja

```
<filter>
  <filter-name>EncodingFilter</filter-name>
  <filter-class>pl.coderslab.EncodingFilter </filter-class>
  <init-param>
      <param-name>charsetEncoding</param-name>
      <param-value>UTF-8</param-value></init-param>
  <init-param>
    <param-name>contentType</param-name>
    <param-value>text/html</param-value></init-param>
</filter>
  <filter-mapping>
    <filter-name>EncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Określamy parametry dla filtra.

Filtr – kodowanie – definicja

```
<filter>
  <filter-name>EncodingFilter</filter-name>
  <filter-class>pl.coderslab.EncodingFilter </filter-class>
  <init-param>
      <param-name>charsetEncoding</param-name>
      <param-value>UTF-8</param-value></init-param>
  <init-param>
    <param-name>contentType</param-name>
    <param-value>text/html</param-value></init-param>
</filter>
  <filter-mapping>
    <filter-name>EncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

Określamy mapowanie.

Filtr – kodowanie – c.d.

Modyfikujemy klasę filtra tak by korzystał ze zdefiniowanych parametrów.

```
public void init(FilterConfig filterConfig) throws ServletException {
    String encodingParam = filterConfig.getInitParameter("charsetEncoding");
    String charsetParam = filterConfig.getInitParameter("contentType");
    if (encodingParam != null && charsetParam!= null) {
        contentType = encodingParam;
        charsetEncoding = charsetParam;
    }
}
```

Filtr – kodowanie – c.d.

Modyfikujemy klasę filtra tak by korzystał ze zdefiniowanych parametrów.

```
public void init(FilterConfig filterConfig) throws ServletException {
    String encodingParam = filterConfig.getInitParameter("charsetEncoding");
    String charsetParam = filterConfig.getInitParameter("contentType");
    if (encodingParam != null && charsetParam!= null) {
        contentType = encodingParam;
        charsetEncoding = charsetParam;
    }
}
```

Pobieramy parametry z konfiguracji.

Filtr – kodowanie – c.d.

Modyfikujemy klasę filtra tak by korzystał ze zdefiniowanych parametrów.

```
public void init(FilterConfig filterConfig) throws ServletException {
    String encodingParam = filterConfig.getInitParameter("charsetEncoding");
    String charsetParam = filterConfig.getInitParameter("contentType");
    if (encodingParam != null && charsetParam!= null) {
        contentType = encodingParam;
        charsetEncoding = charsetParam;
    }
}
```

Pobieramy parametry z konfiguracji.

Jeżeli parametry istnieją przypisujemy ich wartości do prywatnych atrybutów klasy filtra.

Filtry – argumenty

Zwróćmy uwagę że atrybuty metody **doFilter** są typu **ServletRequest** oraz **ServletResponse**

```
public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain) throws IOException, ServletException {
    // kod filtra
    chain.doFilter(request, response);
}
```

W celu pobrania parametrów sesji, np. podczas implementacji logowania należy wykonać rzutowania na typ **HttpServletRequest**, np:

```
HttpServletRequest httpRequest = (HttpServletRequest) request;
System.out.println(httpRequest.getSession().getAttribute("userName"));
```

Analogicznie możemy rzutować **ServletResponse** na **HttpServletResponse**

Zadania