Architektura zorientowana na usługi										
2	Temat:	***REST***	Zadania:							Data:
	Autor:	**Sylwia Kaleta**	1	2	3	4	5	6	7	8	**17 X 2018**
	Autor:	**Kamil Wanat**	M	M	M	E	E	H	-	-	**18:00-19:30**

Wstęp:

Celem laboratoriów było zapoznanie się z technologią REST. Zadania laboratoryjne pozwalały opanować implementację interfejsu oraz klienta REST-owego. Ponadto przećwiczyliśmy przesyłanie parametrów w zapytaniu oraz obsługę cookies.

Zadanie 1. Podstawowa usługa REST

Zadanie polegało na utworzeniu Webservice'u REST obsługującego czasowniki PUT,POST,GET oraz DELETE, mającego na celu obsługę listy użytkowników pewnego systemu. Lista osób została zaimplementowana przy użyciu sesyjnego EJB, które przechowuje wymagane dane oraz udostępnia prosty interfejs do manipulowania tymi danymi, w szczególności: dodawanie użytkownika, wyszukiwanie po loginie, usuwanie użytkownika. Jest to podstawy zbiór metod pozwalający na wykonanie zadania. Webservice implementujący REST zapisany został w pliku GenericResource.java. Przykładowa implementacja funkcji obsługującej żądanie typu GET widoczna jest poniżej:

```
@GET
  @Consumes({"text/plain"})
  @Path("/users")
  public Response getUsersList(
       @QueryParam("sortBy") String sortBy,
       @QueryParam("sortDir") String sortDir,
       @QueryParam("page") String page,
       @QueryParam("count") String count)
  {
    return performGetReq(sortBy, sortDir, page, count, 0);
  }
```

Zadanie 2. Negocjacja treści

Zadanie polegało na zrefaktoryzowaniu kodu z zadania numer jeden w taki sposób, aby możliwe było zwracanie różnych typów danych, w zależności jaki typ danych został podany jako dane żądania. Wyróżnione zostały następujące typy: plain-text, XML, JSON oraz HTML. Aby możliwe było obsłużenie wymienionych typów, metody zaimplementowane w zadaniu pierwszym musiały zostać powielone. Dzięki temu każda z metod może obsługiwać wybrany typ danych. Dodatkowo przesyłanie odpowiedzi zostało zmienione. Obecnie każda metoda przyjmująca żądanie przesyła do metod wewnętrznych informację o tym jaki typ danych został dostarczony, co przekłada się bezpośrednio na zwracany typ danych. Informacja przesyłana jest w postaci liczby całkowitej która następnie interpretowana jest w metodzie private StringBuilder getStringBuilder(List<User> userList, int type). Kolejne liczby odpowiadają następującym typom danych zwracanych:

1 – HTML
2 – XML
3 – JSON
4 – plain text

Przefomatowane odpowiednio dane zapisywane są do StringBuildera a następnie zwracane do klienta w odpowiedzi Response. Poniżej widoczne jest przygotowanie danych w formacie JSON:

```
else if(type==2)
    {
```

```
        Integer counter = 0;
        for(User usr: userList)
        {
           counter++;
           sb.append("<user ID="+counter.toString()+">\n");
           sb.append("\t<login>\n");
           sb.append("\t\t");
           sb.append(usr.toString());
           sb.append("\n\t</login>\n");
           sb.append("</user>\n");
        }
     }
```

Zadanie 3. Parametry w zapytaniu

Celem zadania było zaimplementowanie obsługi dodatkowych parametrów w zapytaniu typu GET. Osiągnięto to poprzez dodanie @QueryParam w metodach obsługujących zapytanie typu GET. Na poniższym listingu widoczna jest implementacja takiego zapytania zwracającego listę użytkowników w formacie XML.

```
@GET
   @Consumes({"text/xml"})
   @Path("/users")
   public Response getUsersListXml(
         @QueryParam("sortBy") String sortBy,
         @QueryParam("sortDir") String sortDir,
         @QueryParam("page") String page,
         @QueryParam("count") String count)
   {
      return performGetReq(sortBy, sortDir, page, count, 2);
   }
```

Zadanie 4. Obsługa błędów

Zadanie polegało na dodaniu podstawowej obsługi błędów. Wymagało to dodania odpowiednich statusów błędu w odpowiedzi na żądanie oraz stosownych komentarzy określających miejsce oraz rodzaj błędu. Modyfikacji dokonano bezpośrednio w ciele metod odpowiedzialnych za odebranie oraz przetworzenie danego żądania. Na poniższym listingu widoczny jest fragment ustawiający kod błędu o numerze 409 wraz z komentarzem o próbie dodania użytkownika o istniejącym loginie. W tym przypadku próba dodania nowego użytkownika nie powiedzie się a klient otrzyma stosowny komunikat.

```
if(!dodano)
     {
        status = 409;
        msg = "Uzytkownik o takim loginie istnieje";
     }
     return Response.status(status).entity(msg).build();
```

Zadanie 5. Cookie

Celem zadania było umożliwienie logowania użytkowników, sprawdzania który użytkownik jest obecnie zalogowany oraz wylosowywania. Dodatkowym wymaganiem było ustawienie pliku cookies, oraz zapisanie id sesji po stronie serwera. Aby to osiągnąć w klasie reprezentującej użytkownika dodane zostało pole session reprezentujące numer sesji użytkownika. Jeśli użytkownik nie jest zalogowany pole to jest równe zero. W przypadku poprawnego zalogowania pole przyjmuje wartość ID sesji. Zalogowanie użytkownika polega na utworzeniu nowego ciasteczka oraz modyfikacji pola session w liście użytkowników. Widoczne jest to na poniższym listingu:

```
userListContainer.setSession(parts[0].trim(), Long.toString(milis));
```

cookie = new NewCookie("name", Long.toString(milis));
Wylogowanie polega na usunięciu pliku Cookiem (ustawienie pustego) oraz zmianie id sesji użytkownika na wartość 0.

Zadanie 6 Klient REST

Wymaganiem zadania była implementacja dwóch klientów REST w wybranych technologiach. Jeden z klientów zaimplementowany został z wykorzystaniem języka pyton. Jest to prosty klient umożliwiający na zalogowanie użytkownika, wyświetlenie aktywnych użytkowników, wylogowanie użytkownika oraz zakończenie pracy klienta. Menu wyświetlane jest przy pomocy poniższego kodu:

```
while(True):

print("Wybierz opcje:")

print("1. loguj")

print("2. pokaz aktywnych uzytkownikow")

print("3. wyloguj")

print("4. wyjdz")

option = input()
```

Drugi z klientów zaimplementowany został z wykorzystaniem technologii JavaScript. Dzięki temu mogliśmy porównać dwa podejścia do programowania klienta: przy pomocy rozwiązań przeglądarkowych oraz konsolowych. Pozwoliło nam to na lepsze zrozumienie działania klientów REST-owych.

Podsumowanie:

Zadania wymagane w laboratorium okazały się czasochłonne oraz problematyczne. Dodatkowo nieściśle zdefiniowana treść zadań prowadziła do konfliktów interpretacyjnych. Z tego względu czas wykonania zadań znacznie się wydłużał, gdyż konieczne było ustalenie właściwej interpretacji postawionego problemu.