Interactive Web Applications and Services with Java and XML
Course, Fall 2005

Henning Niss and Troels C. Damgaard

IT University of Copenhagen

Lecture 1
Interactive web apps and services with Java and XML

Teachers:

- Henning Niss
- Troels C. Damgaard

ITU

hniss@itu.dk  tcd@itu.dk

Time:

- Tuesdays 17–19 (lectures) and 19–21 (exercises)
  (auditorium 4)  (rooms 4A56 & 4A58)

This week:

- Lectures but no exercises this week!
(More) practical information

Main material:
  course notes ("Kompendium") in bookshop

Web page:
- http://www.itu.dk/courses/IWSJ/E2005/

Mailing lists and newsgroup:
- IWSJE2005_stud@itu.dk and it-c.courses.IWSJ
What are web applications and services?

Examples!
- ITU “Find Person”;
- Amazon “Search”;
- WebserviceX “Currency Converter”.
Definitions of web services

“Official” World Wide Consortium definition:
A Web service is a **software system** designed to support interoperable **machine-to-machine interaction** over a network. It has an **interface described** in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using **SOAP-messages**, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

[“Web Services Glossary”, 2004]

**webservices.xml.com:**
At a minimum, however, a Web service is any piece of software that **makes itself available over the Internet** and uses a standardized **XML messaging** system. [“Top Ten FAQs for Web services”, 2001]

Definition used in this course:
A **web service (web application)** is **dynamic functionality available over the HTTP protocol**.
Course contents

How to best construct such web applications / services?

- Core Java-based technologies for web application development:
  - servlets,
  - Java Server Pages (JSP).

- Novel techniques for easier and safer web application development:
  - Xact (type-safe XML generation),
  - JWIG (a Java-based framework for safe web applications).

- Application-to-application XML-based technologies:
  - XML over HTTP,
  - SOAP, WSDL, and UDDI.
Course goals

Provide a basis for understanding and developing web services.

After the course, the student should be able to

- describe fundamental principles and concepts;
- describe and compare Java-based technologies;
- implement web applications using these;
- describe and compare XML-based technologies;
- implement web services using these.
Prerequisites

Implementation experience with Java and XML.

- you should be able to
  - design, implement, and test moderate size Java programs
  - (inheritance, packages, and external libraries).
  - design, . . ., programs for XML manipulation in Java
  - (JDOM).

- for example by having taken
  - “Introductory programming” (“Grundlæggende programmering”),
  - a programming intensive project and/or follow-up course, and
  - “XML Processing”.

hniss (ITU)
Course format

Format:

- Weekly lectures and exercises
  - lectures describes technologies,
  - exercises gives practical experience.
- Three mandatory “mini projects”
  - opportunity to gain in-depth practical experience,
  - required in order to be eligible for exam,
  - not supposed to require more time than usual.

Exam:

- Written exam with written material allowed — no PC (form: A2).

Material:

- *An Introduction to XML and Web Technologies* (Møller & Schwartzbach),
- supplemented by research papers and software manuals.
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
Web applications and services

A web service (web application) is dynamic functionality available through the HTTP protocol.


Client-server model:

1. *client* sends a request to *server*;
2. server constructs reply, and sends to client.
Web services vs static pages

Web used to consist mostly of static pages.

In contrast, web services can provide

- dynamics (replies generated at time of request);
- user-specific information (based on user input and server state);
- two-way communication (client can send data to server).
Two types of web services

An *interactive* web service is

- used by a human,
- constructed using forms, and
- dynamic (server generates reply based on input in the form);
- example: “Find Person”.

An *Application-to-Application* web service is

- “used” by programs,
- specified using some standard format (in XML),
- dynamic;
- example: “Currency Converter”.

Writing a web service

- Web services are everywhere!
- Looks simple, but much complexity underneath.
  - presentation (HTML and JavaScript)
  - authentication (HTTP, e.g.)
  - dynamics (PHP/ASP/JSP/Servlets)
  - data (XML and databases)
  - interoperability (XML technologies)
- Current technologies don’t provide much support.
  - understand the technologies
  - use them in practice
  - look for better alternatives
Web interaction

1. Client requests web page
2. Server returns an HTML document
3. Client renders HTML
1. Client requests web page
2. Server returns an HTML form
3. Client sends form data
4. Server invokes program
5. Program talks to external data source
6. Program / server returns XML document
7. Client requests stylesheet
Issues when writing web services

Multiple clients accessing the web service simultaneously leads to issues.

Be aware of:
- concurrency;
- communication;
- security;
- synchronization.
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
Java and web services

Java and related technologies are great for web service programming!

Java is

- well-defined (semantics and libraries);
- portable (bytecode interpreter);
- secure (array bounds check, garbage collection, sandboxing security);
- equipped with rich set of libraries (Unicode, XML, network access, ...).
Java-based offerings

Java-based offerings for writing web services:

*Java Server Pages/JSP*
  somewhat like ASP and PHP.

*Servlets*
  somewhat like CGI scripts in Perl/C/VB.

*JWIG*
  high-level language targetted at web services.
Hello World with JSP

- JSP pages are HTML with embedded code (like ASP and PHP).
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest req,
                      HttpServletResponse resp)
               throws IOException, ServletException {
        resp.setContentType("text/html");
        PrintWriter o = resp.getWriter();
        o.println("<html>");
        o.println("<head><title>Servlet</title></head>");
        o.println("<body><h1>Hello World!</h1>");
        o.println("This page was last updated: 
                      + new java.util.Date());
        o.println("</body></html>"ADATA nuisis (ITU) Introduction Lecture 1 26 / 76
}

Servlets are code with embedded HTML strings.
Hello World in Xact / JWIG

import dk.brics.jwig.runtime.*;

public class Hello extends Service {
    public class Example extends Session {
        public void main() {
            XML x = <![CDATA[
                <html>
                    <head>
                        <title>JWIG</title>
                    </head>
                    <body>
                        <h1><what></h1>
                        This page was last updated: <![CDATA[when]]>
                    </body>
                </html>
            ]];
            x.plug("what", "Hello World!");
            x.plug("when", new java.util.Date());
            show x;
        }
    }
}

- XML (and here XHTML) is a built-in datatype.
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
Internet architecture

Many layers to a web service

1. *applications* (JSP, Servlets, JWIG)
2. *application layer* (HTTP — GET and POST)
3. *transport layer* (TCP — reliable communication, sockets, ports)
4. *internet layer* (IP — datagrams, IP addresses)
5. *physical layer* (Ethernet)

In this course we will look at the first two.
Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is used to communicate with web servers/services.

- Client/server model.
- Request/reply protocol. Both types of messages in ASCII.
- Stateless.
- Uniform Ressource Identifiers identifies resources.
- Example URIs:
  - http://www.itu.dk/
  - http://www.google.com/search?q=IWSJ

- HTTP URLs: \texttt{http://host [ : port ] [ path [ ? query ] ]}
HTTP Requests

- Request grammar:
  
  request ::= method SP request-URI SP HTTP/ (digit)+ . (digit)+ CRLF (message-header CRLF)* [CRLF message-body]

  method ::= GET | POST | ...

- Example request:

  GET /courses/IWSJ/E2005/index.html HTTP/1.1
  Host: www.itu.dk
  <blank line>
Example HTTP Request

GET /search?q=IWSJ HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.2) Gecko/20040803
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png;*/,*/;q=0.5
Accept-Language: da,en-us;q=0.8,en;q=0.5,sw;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.google.com/
HTTP Responses

- Response grammar:
  
  response ::= http-version SP status-code SP reason CRLF
  
  (message-header CRLF)* [CRLF message-body]

- Example response:

  HTTP/1.1 200 OK
  Date: Mon, 29 Aug 2005 15:04:54 GMT
  ...
  
  <html><body>... </body></html>

- Common status codes:

  1xx Informational
  2xx Successful
    200 OK
  3xx Redirection
    301 Moved Permanently
    302 Moved Temporarily
  4xx Client error
    400 Bad Request
    404 Not Found
  5xx Server error
    500 Internal Server Error
    501 Not Implemented
Example HTTP Response

HTTP/1.1 302 Found
Date: Mon, 29 Aug 2005 15:12:52 GMT
Server: Apache/2.0.53 (Unix)
Location: http://www1.itu.dk/
Content-Length: 312
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
  <head>...</head>
  <body>...</body>
</html>
Request methods

*GET*

simple request; arguments (if any) in URL—follows `?`; response may be cached.

*POST*

larger request; arguments follow request header; response should not be cached.

*HEAD*

as *GET*, but client only wants response header.

*PUT*

upload a resource.

*DELETE*

delete a resource.

...
POST or GET?

- **GET** mainly intended for retrieving data
  - should be safe for clients;
  - should be idempotent;
  - cacheable.

- **POST** mainly intended operations with side-effects
  - may have “uncontrolled” side-effects;
  - not cacheable;
  - no size limits.
Manual “browsing”

Often helpful to be able to send requests to web server *manually*.

- A *telnet* client is a good tool for this.
- *telnet* can connect to any port;
- *HTTP* is an ASCII protocol;
- **example**: `telnet www.itu.dk http`. 
Documents

- Originally mostly static content.
- Increasingly, more and more dynamic content.
  - generated using Servlets, JSP, etc;
  - database backed;
  - interaction through forms.
Tim Berners-Lee proposed the original HTTP protocol.

1991  The original HTTP protocol (a few informal pages)
1996  HTTP1.0 (informational RFC; 60 pages)
1999  HTTP1.1 (standard draft RFC; 176 pages)
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
Interactive web services receive input via (HTML) forms.

Example: the “Find Persons” form on http://www.itu.dk/.

```
<h2>Simple Search</h2>

<form action="find_person.sml" method="post">
  <input type="hidden" name="lang" value="en">
  ...
  <b>Find person:</b>
  <input type="text" name="pat" size="43">
  <input type="submit" value="Search">
</form>
```
When submitting (in this case clicking “Search”), the browser constructs a request (using the method specified) containing values for all the inputs.

- **Examples:**
  - **POST:**
    ```
    POST /ucs/people/find_person.sml HTTP/1.1
    Host: adm.itu.dk
    Content-Length: 23
    
    pat=Mads+Tofte&lang=eng
    ```
  - **GET:**
    ```
    GET /ucs/people/find_person.sml?pat=Mads+Tofte&lang=eng HTTP/1.1
    Host: adm.itu.dk
    ```

- **Values, whether using GET or POST, are encoded.**
Often a web service needs to authenticate a user. HTTP provides protocol-level authentication.

- client authentication;
- username/password forms;
- familiar dialogs.
Sequence of events:

1. server tells client to authorize itself
   HTTP/1.1 401 Authorization Required
   WWW-Authenticate: Basic realm="ITU’s intranet"

2. client sends credentials (name:password) in Base 64 encoded form
   GET /Intranet HTTP/1.1
   Authorization: Basic SGVubmluZ05pc3M6SVdT

3. server checks credentials;

4. browser may repeat the Authorization-header.
Secure sockets layer

Authentication does not provide confidentiality or integrity (or encryption).

- **Secure Socket Layer (SSL)** can be used between the application layer and the transport layer.
  - privacy and reliability of client/server communication using cryptography;
  - server authentication;
  - simple to use for users (swap http with https in URLs).

- **Sequence of events:**
  1. a secure channel using public-key crypto (e.g., RSA) set up;
  2. a shared secret generated;
  3. communication by symmetric crypto (e.g., DES).
Sessions (1)

The **HTTP** protocol is *stateless*!

...however, many web services are *stateful*.

- Example: user has selected a number of books & entered credit-card information, but not provided a delivery address.
- In general, a *session* is a sequence of related client-server interactions.

---

![Diagram](https://via.placeholder.com/150)

- HTML page
- HTML page
- session

---

hniss (ITU)  Introduction  Lecture 1  53 / 76
Sessions (2)

Three kinds of state:

- *shared state* (global, all sessions)
- *session state* (local, each session)
- *temporary/transient state* (single interaction)

Examples:

- shared: book catalog;
- session: cart contents, current “program counter”;  
- temporary: price of items in cart.
Implementing sessions in HTTP:

- URL rewriting: add session data to all URLs
  
  `http://www.itu.dk/Internet;hniss-session`

- hidden form fields: include hidden fields with session data
  
  `<input type="hidden" name="id" value="hniss-session">`

- cookies
  
  (server `Set-Cookie`; client includes `Cookie`)

- session URL (JWIG).
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
Web of Jokes

Running example throughout the course.

Infrastructure for a world-wide network of servers and clients for authoring, publishing, and reading jokes.

- jokes in a domain-specific XML language;
- a joke server is a web service storing jokes
  - submit (upload) a joke;
  - retrieve jokes (in a certain category);
- the joke metaserver knows about registered joke servers;
- a joke client is a web application for humans
  - author jokes an upload to “local” joke server;
  - read jokes from all joke servers known to the metaserver.
Overview
Joke language

Jokes consists of

- sequence of setups;
- one punchline.

Joke metadata:

- category;
- unique id;
- date;
- title;
- author (optional).

```xml
<schema ...>
  <element name="collection">
    <complexType> <sequence>
      <element ref="jml:joke" minOccurs="0" maxOccurs="unbounded"/>
    </sequence> </complexType>
  <unique name="joke-id-uniqueness">
    <selector xpath=".//jml:joke"/>
    <field xpath="@id"/>
  </unique>
  </element>
</schema>
```
Mad Cows

Two cows are talking.
The first cow: "Hey, did you hear about that mad cow disease?"
The second cow: "Yeah, but I’m not worried about it."
The first cow: "Why not?"
The second cow: "I’m a duck."
Showing jokes

Use XSLT and CSS (Sec. 12.2.3).
Operations

Joke server, metaserver, and client implemented as services.

Joke server operations:

- list (joke categories);
- retrieve(category) (jokes of a specified category);
- submit() (joke in body of POST request).

Joke metaserver operations:

- register(list-url, retrieve-url) (a joke server);
- discover (joke servers).

Joke client “operations”:

- read jokes (contacts metaserver, shows categories & jokes);
- write jokes (collecting joke data and metadata).
Overview

1. Web services;
2. Java and WWW;
3. HTTP;
4. Web service concepts;
5. Web of jokes;
6. Java examples.
TCP/IP and Java

Main abstraction: sockets.

- TCP is connection-oriented;
- two endpoints (both sockets);
- socket: both IP-address and port number;
- example: 130.226.142.6, port 80.
import java.net.*;
import java.io.*;
public class SimpleServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
            while (true) {
                Socket con = ss.accept();
                InputStreamReader in =
                    new InputStreamReader(con.getInputStream());
                StringBuffer msg = new StringBuffer();
                int c;
                while ((c = in.read())!=0) // we use 0 as end-of-message marker
                    msg.append((char)c);
                PrintWriter out = new PrintWriter(con.getOutputStream());
                out.print("Simon says: \"+msg);
                out.flush();
                con.close();
            }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
import java.net.*;
import java.io.*;
public class SimpleClient {
    public static void main(String[] args) {
        try {
            Socket con = new Socket(args[0], Integer.parseInt(args[1]));
            PrintStream out = new PrintStream(con.getOutputStream());
            out.print(args[2]);
            out.write(0); // mark end of message
            out.flush();
            InputStreamReader in =
                new InputStreamReader(con.getInputStream());
            int c;
            while ((c = in.read())!=-1)
                System.out.print((char)c);
            con.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
import java.net.*;
import java.io.*;
public class ImFeelingLucky {
    public static void main(String[] args) {
        try {
            Socket con = new Socket("www.google.com", 80);
            String req = "/search?"+
                "q="+URLEncoder.encode(args[0], "UTF8")+"&"+
                "btnI="+URLEncoder.encode("I'm Feeling Lucky", "UTF8");
            BufferedWriter out =
                new BufferedWriter(new OutputStreamWriter(con.
                    getOutputStream(), "UTF8"));
            out.write("GET "+req+" HTTP/1.1\r\n");
            out.write("Host: www.google.com\r\n");
            out.write("User-Agent: IXWT\r\n");
            out.flush();
            // ...
        }
    }
}
BufferedReader in =
    new BufferedReader
    (new InputStreamReader(con.getInputStream()));
String line;
System.out.print("The prophet spoke thus: ");
while ((line = in.readLine()) != null) {
    if (line.startsWith("Location:")) {
        System.out.println("Direct your browser to "+
            line.substring(9).trim()+
            " and you shall find great happiness in life.");
        break;
    } else if (line.trim().length()==0) {
        System.out.println("I am sorry - my crystal ball is blank.");
        break;
    }
}
con.close();
} catch (IOException e) {
    System.err.println(e);
} }
import java.net.*;
import java.io.*;
public class ImFeelingLucky2 {
    public static void main(String[] args) {
        try {
            String req = "http://www.google.com/search?q=" +
                URLEncoder.encode(args[0], "UTF8") + "&btnI=" +
                URLEncoder.encode("I’m Feeling Lucky", "UTF8");
            HttpURLConnection con =
                (HttpURLConnection) (new URL(req)).openConnection();
            con.setRequestProperty("User-Agent", "IXWT");
            con.setInstanceFollowRedirects(false);
            String loc = con.getHeaderField("Location");
            System.out.print("The prophet spoke thus: ");
            if (loc!=null)
                System.out.println("Direct your browser to "+loc+
                        " and you shall find great happiness in life.");
            else
                System.out.println("I am sorry - my crystal ball is blank.");
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}