

---

Basic Research in Computer Science

BRICS NS-02-1 Møller & Schwartzbach: Interactive Web Services with Java: JSP, Servlets, and JWIG

## Interactive Web Services with Java

JSP, Servlets, and JWIG

Anders Møller  
Michael I. Schwartzbach

BRICS Notes Series

ISSN 0909-3206

NS-02-1

April 2002

**Copyright © 2002, Anders Møller & Michael I. Schwartzbach.  
BRICS, Department of Computer Science  
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work  
is permitted for educational or research use  
on condition that this copyright notice is  
included in any copy.**

**See back inner page for a list of recent BRICS Notes Series publications.  
Copies may be obtained by contacting:**

**BRICS  
Department of Computer Science  
University of Aarhus  
Ny Munkegade, building 540  
DK-8000 Aarhus C  
Denmark  
Telephone: +45 8942 3360  
Telefax: +45 8942 3255  
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide  
Web and anonymous FTP through these URLs:**

`http://www.brics.dk`  
`ftp://ftp.brics.dk`  
**This document in subdirectory NS/02/1/**

# **Interactive Web Services with Java**

***JSP, Servlets, and JMWIG***

**Anders Møller** & **Michael I. Schwartzbach**

**BRICS, University of Aarhus**

**<http://www.brics.dk/~amoeller/WWW/>**

# About this tutorial...

This slide collection about Java Web service programming, JSP, Servlets, and JMWIG is created by

**Anders Møller**

**<http://www.brics.dk/~amoeller>**

and

**Michael I. Schwartzbach**

**<http://www.brics.dk/~mis>**

at the BRICS research center at University of Aarhus, Denmark.

**Copyright © 2002 Anders Møller & Michael I. Schwartzbach**

**Reproduction of this slide collection is permitted on condition that it is distributed in whole, unmodified, and for free, and that the authors are notified.**

A PDF version suitable for printing and off-line browsing is available upon [request](#).

# Contents

1. [Java and WWW](#) - using Java for Web service development 4
2. [Servlets](#) - Java-based CGI scripts 28
3. [JSP](#) - a Java version of ASP/PHP 43
4. [JWIG](#) - a high-level language for developing Web services 53
5. [PowerForms](#) - declarative form-field validation 86

# Java and WWW

- using Java for Web service development

1. [Introduction](#)
2. [Interactive Web Services](#)
3. [Benefits from using Java](#)
4. [HelloWorld in JSP, Servlets, and JMWIG](#)
5. [Internet Architecture](#)
6. [HTTP - HyperText Transfer Protocol](#)
7. [HTML Forms](#)
8. [Authentication](#)
9. [SSL - Secure Sockets Layer](#)
10. [Session Tracking](#)
11. [A Web Server in 150 Lines](#)
12. [A Test Client](#)
13. [The `java.net` Package](#)
14. [Extra Things Worth Knowing](#)

# JSP, Servlets, and JMWIG

Three Java-based technologies for making interactive Web services:

- [JSP](#): resembles ASP/PHP
- [Servlets](#): resembles Perl/C/VB CGI scripts
- [JMWIG](#): novel high-level language developed at BRICS/DAIMI

Plan:

- intro: general stuff about Java and WWW
- Servlets
- JSP (are compiled to Servlets)
- JMWIG

(Requirements: we assume a basic knowledge of Java and HTML!)

# Interactive Web Services

Originally, the Web consisted of **static HTML pages**.

The **client-server** model:

1. a client initiates communication with a server (e.g. requesting a page)
2. the server responds (e.g. returns the page)

In an **interactive Web service**, the pages contain forms with information to the server, and the reply is generated dynamically.

Compared to static HTML pages, interactive Web services can provide:

- up-to-date information (replies generated at time of request)
- tailor-made information (reply generated dynamically by program based on user input and current server state)
- two-way communication (client can also send data to server)

Common service code layers:

- presentation (receive client requests, produce HTML replies)
- functionality (extract information, track user sessions)
- data (database / containers)

In JSP/Servlets/JWIG, the data layer consists of an SQL/JDBC database or `java.util` containers.

# Java and WWW

Java is an ideal framework for server-side Web programming:

- portability (well-defined semantics of language and standard libraries)
- platform independence (bytecode interpretation)
- secure runtime model (array bound checks, automatic garbage collection, bytecode verification, ...)
- sandboxing security (`SecurityManager`)
- dynamic loading (`ClassLoader`)
- data migration (serialization)
- Unicode (as HTML and XML)
- threads, concurrency control
- network access (`java.net.*`)
- cryptographic security (RSA, ...)
- applets (on client-side) are also Java
- ...

(compare with Perl/C/VB CGI scripts!)

# HelloWorld

Variants of a well-known program:

## ["Hello World" in JSP:](#)

```
<html><head><title>JSP</title></head>
<body><h1>Hello World!</h1>
This page was last updated: <%= new java.util.Date() %>
</body></html>
```

## ["Hello World" in Servlets:](#)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet</title></head>");
        out.println("<body><h1>Hello World!</h1>");
        out.println("This page was last updated: " + new java.util.Date());
        out.println("</body></html>");
    }
}
```

## ["Hello World" in Jwig:](#)

```
import dk.brics.jwig.runtime.*;

public class Hello extends Service
{
    public class Example extends Session
    {
        public void main() {
            XML x = [[ <html><head><title>JWIG</title></head><body>
                <h1><[what]></h1></body></html> ]];
            x = x <[ what = [[ Hello World! ]] ];
            show x;
        }
    }
}
```

Note that:

- **JSP** pages are HTML with embedded code (like ASP or PHP pages)
- **Servlets** are code with embedded HTML strings (like CGI scripts)

- in **JWIG**, XML (e.g. XHTML) is a built-in data-type

# Internet Architecture

The Web has many layers:

- our applications (JSP, Servlets, JMWIG, Explorer/Netscape)
  - *application layer* (HTTP) - GET/POST requests, URLs, MIME types
  - *transport layer* (TCP) - reliable communication, client/server sockets
  - *internet layer* (IP) - datagrams, IP numbers
  
  - *physical layer* (Ethernet) - bits
- we will mainly look at the upper two.

# HTTP - HyperText Transfer Protocol

The communication protocol of the WWW:

- 1991 - [the original HTTP protocol \(v0.9\)](#) - read [Tim Berners-Lee's design issues](#)
- 1996 - [HTTP/1.0](#)
- 1999 - [HTTP/1.1](#)

HTTP is **stateless** - interactions follow a request-response pattern with no protocol support for sessions consisting of multiple interactions between the same client and server.

A HTTP **URL** (Uniform Resource Locator) identifies a Web resource:

```
protocol://host:port/path?query
```

(Example: <http://www.google.com/search?q=interactive+web+services>)

- *protocol* - http, https, ...
- *host* - server name or IP number (e.g. freewig.brics.dk or 130.225.2.179)
- *port* - server local port (default: 80 for http, 443 for https)
- *path* - path on server to file or program (server decides interpretation)
- *query* - arguments to program (program decides interpretation, usually encoded name-value pairs)

A **request** from a client to a server is a TCP packet. Example:

```
GET /index.html HTTP/1.0
Accept: text/html, text/plain
User-Agent: Mozilla/4.76
Host: www.brics.dk
If-Modified-Since: Friday, 01-Mar-02 12:09:31 GMT
```

The **response** from the server to the client has the form:

```
HTTP/1.1 OK 200
Date: Mon, 08 Apr 2002 13:19:36 GMT
Server: Apache/1.3.23 (Unix) mod_bigwig/2.0 mod_perl/1.26 mod_ssl/2.8.7
OpenSSL/0.9.6c
Last-Modified: Tue, 05 Mar 2002 09:33:33 GMT
Expires: Fri, 05 Apr 2002 09:33:33 GMT
Content-Length: 3682
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title>BRICS - Basic Research in Computer Science</title></head>
<body bgcolor=white>
...
</body></html>
```

Request methods:

- **GET**: simple request, arguments (if any) are in *query* - response may be cached
- **POST**: bigger request, arguments follow request header - response should not be cached
- **HEAD**: as GET, but client only wants response header
- ...

Most common response codes:

- Successful 2xx:
  - 200 OK - requested resource follows
- Redirection 3xx:
  - 301 Moved Permanently - resource has moved, update bookmarks accordingly
  - 302 Moved Temporarily - resource has moved, but don't update bookmarks
  - 304 Not Modified - resource has not been modified since date given by conditional-GET
- Client Error 4xx:
  - 400 Bad Request - malformed request
  - 401 Unauthorized - proper user authentication not provided
  - 403 Forbidden - no permission to access resource
  - 404 Not Found - the requested resource does not exist
- Server Error 5xx:
  - 500 Internal Server Error - bug in server
  - 501 Not Implemented - required functionality not supported
  - 503 Service Unavailable - due to overloading or maintenance

**CGI** (Common Gateway Interface) is a standard for making HTTP servers start programs to handle requests.

**MIME** (Multipurpose Internet Mail Extensions) is used to describe message encodings (e.g. Content-Type).

# HTML Forms

Users send information to servers via [forms](#):

## The Poll Service

Who wins the World Cup 2002?  
Please enter your email address:

HTML source:

```
<h3>The Poll Service</h3>
<form action="http://freewig.brics.dk/users/laudrup/soccer.jsp" method="post">
Who wins the World Cup 2002?
<select name="bet">
<option value="fr">France!</option>
<option selected value="dk">Denmark!</option>
<option value="other country">someone else?</option>
</select><br>
Please enter your email address: <input type="text" name="email"><br>
<input type="submit" name="submit" value="Submit">
</form>
```

Browser collects reply in **query string**:

```
bet=other+country&email=john.doe%40notmail.com&submit=go
```

Values are **URL-encoded**: *space* becomes +, non-alphanumeric chars become *%hexcode* - assuming `enctype="application/x-www-form-urlencoded"` (the default)

## GET vs. POST?

- GET with hardwired querystring can be used in links (`<a href="...">...</a>`) :-)
- GET has server-specific limits on input lengths :-)
- GET querystrings usually end in the server logs :-)
- GET is (in principle) idempotent - results are cached unless explicitly "expired"
- GET is the default for `form` :-)

**Uploading files** requires POST and a different encoding of form data:

```
<form method="post" enctype="multipart/form-data" ...>
<input type="file" ...>
```

The response then contains:

- file name (or full path, depending on browser)
- data MIME type
- encoding type

# Authentication

Security aspects:

- **authentication** (access restriction)
  - username/password forms (`<input type="password" ...>`)
  - HTTP Basic Authentication
  - X.509 certificates
- **encryption** (confidentiality, integrity)
  - SSL - next page...

## HTTP Basic Authentication:

- authentication of client (not of server)
- familiar input dialogs
- handled at protocol level, not explicitly by application
- browsers can (optionally) remember name/password (without using cookies)



How it works:

- in first interaction, the server sends an HTML error message with a HTTP header (a "*challenge*"):

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="VIP Site"
```

where the *realm* is a sub-domain of the server

- the client responds by repeating the request, but adding another HTTP header (a "response"):  
`Authorization: Basic QWxhZGRpbjpvYGVuIHNLc2FtZQ==`  
containing the base64 encoding of "*name:password*"
- the server decodes the name and password, and checks with its access restrictions
- in subsequent interactions with that server, the browser can send the `Authorization` immediately without user involvement (convenient, but potentially dangerous)

Note: there is **no encryption** here!

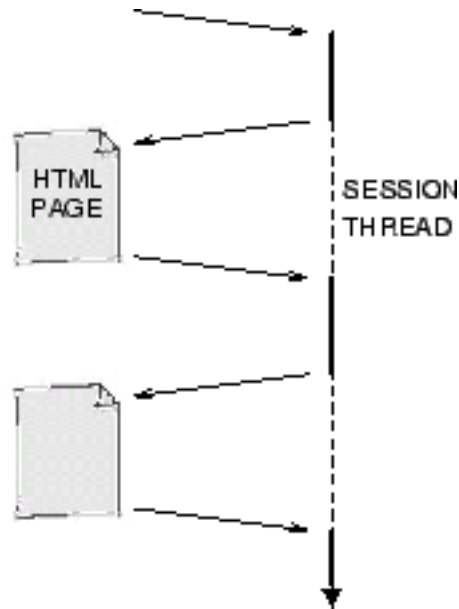
# SSL - Secure Sockets Layer

- [SSL](#) - the Secure Sockets Layer can be inserted between the *application layer* (HTTP) and the *transport layer* (TCP).
- using cryptography, it provides **privacy and reliability** of client-server communication and **authentication of the server**
- first, a secure channel is set up using (slow) public-key encryption (e.g. RSA) to generate a shared secret
- subsequently, communication is performed using (fast) symmetric encryption (e.g. DES)
- for Web services, just use the `https` protocol in URLs (assuming that a [trusted certificate](#) is generated for the server)
- J2SE 1.4 contains [Java Secure Socket Extension \(JSSE\)](#) providing full Java support for SSL ([javax.net.ssl](http://javax.net.ssl))

# Session Tracking

HTTP is stateless, but interactive Web services require user sessions.

A **session** is a sequence of related interactions between a client and a server:



In general, Web services have three kinds of data:

- **shared** data - global data, shared between all sessions
- **per-session** data - local data, private to each session
- **temporary** data - only used for a single interaction

Techniques for implementing sessions on top of HTTP:

- **URL rewriting**

Add user/session data to all URLs referring to the session:

```
http://mysite.com/buy;customer=wile_e_coyote
```

or

```
http://mysite.com/buy;session=117
```

- **hidden form fields**

Include

```
<input type="hidden" name="customer"
```

value="wile\_e\_coyote">  
in the response page.

- **Cookies** - allowing servers to store and access data at clients

A cookie contains:

- name
  - value
  - expiration time
  - domain (default: server name)
  - path (sub-domain)
  - secure flag (should only be transmitted via SSL)
- max 4KB, 20 per server, 300 total (for each browser)

How it works:

- servers create cookies by response headers:  
`Set-Cookie: customer=wile_e_coyote; path=/; expires=Wednesday, 09-Nov-99 23:12:40 GMT`
- clients include the relevant cookies in subsequent requests:  
`Cookie: customer=wile_e_coyote`  
based on the request URL and the cookie domain and path

Problems:

- not a security thread, but perhaps a *privacy* thread (the user is typically not aware of the cookies)
- users may disable cookies
- not easy for users to move a cookie to another machine

Benefit:

- for some services, cookies can store *all* session data (e.g. "shopping basket" applications)

- **session URL** (unique to JWIG!)

- every session is associated a **unique** and **persistent** URL
- explained [later](#)...

# A Web Server in 150 Lines

A simple but functioning HTTP [file server](#) in Java.

- listens for HTTP GET requests and sends back files
- sets MIME type based on file extensions
- simple access restrictions
- redirects directory requests

Read command-line arguments and open server socket:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class FileServer
{
    public static void main(String[] args)
    {
        // read arguments
        if (args.length!=2) {
            System.out.println("Usage: java FileServer <port> <wwwhome>");
            System.exit(-1);
        }
        int port = Integer.parseInt(args[0]);
        String wwwhome = args[1];

        // open server socket
        ServerSocket socket = null;
        try {
            socket = new ServerSocket(port);
        } catch (IOException e) {
            System.err.println("Could not start server: " + e);
            System.exit(-1);
        }
        System.out.println("FileServer accepting connections on port " + port);
```

Begin request-response loop:

```
        // request handler loop
        while (true) {
            Socket connection = null;
            try {
                // wait for request
                connection = socket.accept();
                BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                OutputStream out = new
BufferedOutputStream(connection.getOutputStream());
                PrintStream pout = new PrintStream(out);
```

Read first line of request to get file name:

```

// read first line of request (ignore the rest)
String request = in.readLine();
if (request==null)
    continue;
log(connection, request);
while (true) {
    String misc = in.readLine();
    if (misc==null || misc.length()==0)
        break;
}

```

Process request by checking that the request is well-formed and permitted. For directory requests that do not end in '/', redirect browser. For files, send back the contents:

```

// parse the line
if (!request.startsWith("GET") || request.length()<14 ||
    !(request.endsWith("HTTP/1.0") || request.endsWith("HTTP/1.1")))
{
    // bad request
    errorReport(pout, connection, "400", "Bad Request",
        "Your browser sent a request that " +
        "this server could not understand.");
} else {
    String req = request.substring(4, request.length()-9).trim();
    if (req.indexOf("..")!=-1 ||
        req.indexOf("/.ht")!=-1 || req.endsWith("~")) {
        // evil hacker trying to read non-wwwhome or secret file
        errorReport(pout, connection, "403", "Forbidden",
            "You don't have permission to access the
requested URL.");
    } else {
        String path = wwwhome + "/" + req;
        File f = new File(path);
        if (f.isDirectory() && !path.endsWith("/")) {
            // redirect browser if referring to directory without
final '/'

            pout.print("HTTP/1.0 301 Moved Permanently\r\n" +
                "Location: http://" +
                connection.getLocalAddress().getHostAddress()
+ ":" +
                connection.getLocalPort() + "/" + req +
"\r\n\r\n");

            log(connection, "301 Moved Permanently");
        } else {
            if (f.isDirectory()) {
                // if directory, implicitly add 'index.html'
                path = path + "index.html";
                f = new File(path);
            }
            try {
                // send file
                InputStream file = new FileInputStream(f);
                pout.print("HTTP/1.0 200 OK\r\n" +
                    "Content-Type: " + guessContentType(path)
+ "\r\n" +
                    "Date: " + new Date() + "\r\n" +
                    "Server: FileServer 1.0\r\n\r\n");
                sendFile(file, out); // send raw file

```

```

        log(connection, "200 OK");
    } catch (FileNotFoundException e) {
        // file not found
        errorReport(pout, connection, "404", "Not Found",
            "The requested URL was not found on this
server.");
    }
}
}
}
out.flush();

```

Catch exceptions and close connection:

```

    } catch (IOException e) { System.err.println(e); }
    try {
        if (connection != null) connection.close();
    } catch (IOException e) { System.err.println(e); }
}
}

```

Auxiliary methods for logging and error reporting:

```

private static void log(Socket connection, String msg)
{
    System.err.println(new Date() + " [" +
connection.getInetAddress().getHostAddress() +
        ":" + connection.getPort() + "] " + msg);
}

private static void errorReport(PrintStream pout, Socket connection,
        String code, String title, String msg)
{
    pout.print("HTTP/1.0 " + code + " " + title + "\r\n" +
        "\r\n" +
        "<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">\r\n" +
        "<TITLE>" + code + " " + title + "</TITLE>\r\n" +
        "</HEAD><BODY>\r\n" +
        "<H1>" + title + "</H1>\r\n" + msg + "<P>\r\n" +
        "<HR><ADDRESS>FileServer 1.0 at " +
        connection.getLocalAddress().getHostName() +
        " Port " + connection.getLocalPort() + "</ADDRESS>\r\n" +
        "</BODY></HTML>\r\n");
    log(connection, code + " " + title);
}

```

Auxiliary methods for guessing MIME type and sending file contents:

```
private static String guessContentType(String path)
{
    if (path.endsWith(".html") || path.endsWith(".htm"))
        return "text/html";
    else if (path.endsWith(".txt") || path.endsWith(".java"))
        return "text/plain";
    else if (path.endsWith(".gif"))
        return "image/gif";
    else if (path.endsWith(".class"))
        return "application/octet-stream";
    else if (path.endsWith(".jpg") || path.endsWith(".jpeg"))
        return "image/jpeg";
    else
        return "text/plain";
}

private static void sendFile(InputStream file, OutputStream out)
{
    try {
        byte[] buffer = new byte[1000];
        while (file.available()>0)
            out.write(buffer, 0, file.read(buffer));
    } catch (IOException e) { System.err.println(e); }
}
}
```

Obvious extensions:

- read all request header (If-Modified-Since, ...)
- HTTP authentication
- support POST and HEAD requests
- multithreading (maintain thread pool, pass each request to an idle thread)
- **dynamic reply generation** - plug in class files, e.g. Servlets or JMWIG programs!

# A Test Client

A simple HTTP [client](#) in Java.

- reads user's request, sends it to the server, and prints the reply
- useful for testing server implementations

Read command-line arguments:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class MultiClient
{
    public static void main(String[] args)
    {
        // read arguments
        if (args.length!=2) {
            System.out.println("Usage: java MultiClient <host> <port>");
            System.exit(-1);
        }
        String host = args[0];
        int port = Integer.parseInt(args[1]);

        System.out.println("MultiClient 1.0");
        System.out.println("Enter request followed by one empty line or 'quit' to
quit.");
        BufferedReader user = new BufferedReader(new InputStreamReader(System.in));

        try {
```

Read user's request and send it to the server:

```
mainloop:
    while (true) {
        // read user request
        StringBuffer req = new StringBuffer();
        boolean done = false, first = true;
        while (!done) {
            // get a line
            System.out.print(host + ":" + port + "> ");
            String line = user.readLine();
            if (line.equals("quit"))
                break mainloop;
            req.append(line).append("\r\n");
            if (line.length()==0 && !first)
                done = true; // done when reading blank line
            first = false;
        }
    }
```

Make connection to server and send the request:

```

Socket socket = null;
try {
    // create socket and connect (don't occupy port too long)
    socket = new Socket(host, port);
    socket.setSoTimeout(60000); // set timeout to 1 minute
    PrintStream out = new PrintStream(socket.getOutputStream());
    out.print(req); // send bytes in default encoding
    out.flush();
}

```

Get server reply and print it to standard out:

```

        // show reply
        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        while (true) {
            String line = in.readLine();
            if (line==null)
                break;
            System.out.println(line);
        }
}

```

Close connection and catch exceptions:

```

        } catch (Exception e) { System.err.println(e); }
        if (socket!=null)
            socket.close(); // close connection
    }
}
catch (Exception e) { System.err.println(e); }
}
}

```

- a good starting point for making a simple Web browser - "just" add an HTML parser and a GUI (as [Notscape](#) using `javax.swing.text.html`)...

# The java.net Package

- provides convenient access to application layer, transport layer, and internet layer.

The most relevant classes and methods:

- **URL**
  - `URL(String spec)` - constructor
  - `URLConnection.openConnection()` - creates TCP connection of type given by the protocol
- **URLConnection (interface) / HttpURLConnection (subclass)** - for making HTTP requests
  - `void setRequestMethod(String method)` - set method (GET/POST/... - default is GET)
  - `void setDoOutput(boolean dooutput)` - intend to use connection for output
  - `void setDoInput(boolean doinput)` - intend to use connection for input
  - `OutputStream getOutputStream()` - output stream
  - `InputStream getInputStream()` - input stream
  - `void setRequestProperty(String key, String value)` - set request header
  - `Hashtable getHeaderFields()` - read response header
  - `Object getContent()` - read input and convert to an object
  - ...
- **URLEncoder / URLDecoder** - for encoding/decoding special chars in URLs (application/x-www-form-urlencoded)
  - `String encode(String s)`
  - `String decode(String s)`

[Example:](#)

```
import java.net.*;
import java.io.*;

public class AltaVista {
    public static void main (String args[])
    {
        try {
            // make connection
            URL url = new URL("http://www.altavista.com/cgi-bin/query?q=" +
                URLEncoder.encode(args[0]));
            URLConnection connection = url.openConnection();
            connection.setDoInput(true);
            InputStream in = connection.getInputStream();

            // read reply
            StringBuffer b = new StringBuffer();
            BufferedReader r = new BufferedReader(new InputStreamReader(in));
            String line;
            while ((line = r.readLine()) != null)
                b.append(line);
            String s = b.toString();

            // look for first search result, if any
```

```

        if (s.indexOf(">We found 0 results") != -1)
            System.out.println("No results found.");
        else {
            int i = s.indexOf("\"status='")+9;
            int j = s.indexOf("'", i);
            System.out.println("First result: " + s.substring(i, j));
        }
    }
    catch (Exception e) { e.printStackTrace(); }
}
}

```

(sends a query to the AltaVista search engine and extracts the first result)

Other useful classes in `java.net`:

- `InetAddress` - IP addresses (DNS lookup, etc.)
- `Socket, ServerSocket` - TCP sockets (as in example [client](#) and [server](#))
- `ProtocolHandler` - defines mapping for `URL.openConnection()` to concrete `URLConnection`
- `ContentHandler` - defines mapping for `getObject()` to `Object` (based on MIME type)

Other relevant packages and technologies:

- Applets: `java.applet`
- Remote Method Invocation: `java.rmi`
- [XML](#) ([JDOM](#), [JAXP](#))

# Extra Things Worth Knowing

Other useful Java features for Web service development:

- **serialization** - "implements `Serializable`"
  - allows objects and object structures to be stored, transferred, and reconstructed
- **security manager** - "`SecurityManager`" and policy files
  - can restrict all interactions with the program environment and resources (file system, network, ...)
- **thread synchronization** - "`synchronized(x) {...}`"
  - concurrency control is essential in any multi-threaded system

# Servlets

- Java-based CGI scripts

1. [Introduction](#)
2. [Requests](#)
3. [Responses](#)
4. [Servers](#)
5. [Deployment](#)
6. [Servlet Contexts](#)
7. [Sessions](#)
8. [Security](#)
9. [Listeners](#)
10. [Filters](#)
11. [JDBC and JDOM](#)

# Introduction

Servlets are written in pure Java using the [Servlet API](#):

- like CGI scripts, Servlets follow the **request-response** pattern from HTTP
- the API provides full access to the HTTP protocol layer

["Hello World" in Servlets](#):

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet</title></head>");
        out.println("<body><h1>Hello World!</h1>");
        out.println("This page was last updated: " + new java.util.Date());
        out.println("</body></html>");
    }
}
```

- a servlet is a sub-class of `HttpServlet`
- the `doGet` method defines the request handler for GET requests
- the `HttpServletRequest` object contains all information about the request
- the `HttpServletResponse` object is used for generating the response
- a server makes only **one instance** of each Servlet class

Useful methods in `HttpServlet` to be implemented in sub-classes:

- `void init()`
- `void doGet(HttpServletRequest, HttpServletResponse)`
- `void doPost(HttpServletRequest, HttpServletResponse)`
- `void doHead(HttpServletRequest, HttpServletResponse)`
- `void log(String)`, `void log(String, Throwable)`
- `String getServletName()`, `String getServletInfo()`
- `void destroy()`

Useful predefined methods:

- `ServletConfig getServletConfig()` - reads server configuration for this Servlet
- `ServletContext getServletContext()` - explained later...

Exceptions: `ServletException`

# Requests

- the full request is available in the given [HttpServletRequest](#):

- `String getHeader(String)`, `Enumeration getHeaders(String)` - reads [HTTP request headers](#)
- `String getParameter(String)` - parses and decodes querystring (for GET) or body (for POST)
- `InputStream getInputStream()` - for reading raw POST request body
- `String getRemoteHost()` - returns client IP address
- ...

Example:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Requests extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet Request GET</title></head><body>");
        out.println("The value of <tt>username</tt> is: <tt>" +
            htmlEscape(request.getParameter("username")) + "</tt>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        doGet(request, response);
    }

    private String htmlEscape(String s)
    {
        StringBuffer b = new StringBuffer();
        for (int i = 0; i<s.length(); i++) {
            char c = s.charAt(i);
            switch (c) {
                case '<': b.append("&lt;"); break;
                case '>': b.append("&gt;"); break;
                case '"': b.append("&quot;"); break;
                case '\\': b.append("&apos;"); break;
                case '&': b.append("&amp;"); break;
                default: b.append(c);
            }
        }
        return b.toString();
    }
}
```

# Responses

- the response is generated using the given [HttpServletResponse](#):

- void addHeader(String name, String value) - add name/value pair to header
- void setStatus(int sc) - set [status code](#) (default: SC\_OK =200)
- void sendError(int sc, String msg) - send error reply
- void sendRedirect(String url) - redirect browser to given page
- ServletOutputStream getOutputStream() - output stream for response body
- ...

- always make the header *before* sending the response body

[Example:](#)

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Responses extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        long expires = new Date().getTime() + 1000*60*60*24;
        response.setContentType("text/html");
        response.addDateHeader("Expires", expires);
        ServletOutputStream out = response.getOutputStream();
        out.println("<html><head><title>Servlet Response</title></head><body>");
        out.println("<h1>Todays News</h1>");
        out.println(getNews());
        out.println("<p><hr>");
        out.println("<i>This news item expires " + new Date(expires));
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.sendError(response.SC_METHOD_NOT_ALLOWED, "Que? - No habla POST!");
    }

    private String getNews()
    {
        return "Nothing has happened since yesterday...";
    }
}
```

(setContentType and addDateHeader are convenience methods on top of setHeader and addHeader...)

**Warning:** browsers usually **cache** responses to GET request - use response.addDateHeader("Expires", 0) to disable caching.

# Servers

- **Apache [Tomcat 4.0](#)** - the official reference implementation for Servlets 2.3 and JSP 1.2 (and Open Source!)
- Trifork's [Enterprise Application Server](#)
- Macromedia's [JRun](#)
- New Atlanta's [ServletExec](#)
- Caucho's [Resin](#)
- Gefion Software's [LiteWebServer](#)
- ...

- installation and server configuration are of course implementation dependent, but service deployment is essentially the same

Micro-Installation-HOWTO™ for Tomcat 4.0 on RedHat Linux using RPM:

1. download and install [tomcat4-4.0.1-1.noarch.rpm](#) and [tomcat4-webapps-4.0.1-1.noarch.rpm](#)
2. edit `/etc/tomcat4/conf/tomcat4.conf` (set dir to JDK)
3. edit `/etc/init.d/tomcat4` (may want to change runlevels for `chkconfig`)
4. edit `/etc/passwd` (correct dir to `/var/tomcat4`)
5. run `/etc/init.d/tomcat start` (as root)
6. test the installation by viewing <http://HOST:8180/examples/servlet/HelloWorldExample> in a browser

# Deployment

A Servlet **Web application** is structured in a directory:

<code>myapplication/</code>	contains auxiliary files (e.g. HTML, GIF, CSS, JSP files), can be in sub-directories
<code>myapplication/WEB-INF/</code>	contains deployment descriptor, <code>web.xml</code>
<code>myapplication/WEB-INF/classes/</code>	contains Servlet class files (in appropriate sub-directories, if non-default package names)
<code>myapplication/WEB-INF/lib/</code>	contains extra jar files

Using the normal `jar` tool, a complete Web application can be wrapped into a **portable** Web Archive (`.war`).

## The deployment descriptor: `web.xml`

provides control of:

- URL mapping (from URLs to files)
- initialization parameters
- timeout settings
- directory listings
- error pages
- security constraints, client authentication
- filters and listeners (explained later...)

Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <!-- assign Name and Initialization Parameters to Manager Servlet -->
  <servlet>
    <servlet-name>Manager</servlet-name>
    <servlet-class>org.apache.catalina.servlets.ManagerServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
  </servlet>

  <!-- define the Manager Servlet mapping -->
  <servlet-mapping>
    <servlet-name>Manager</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
```

```
<!-- define a Security Constraint on this application -->
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Entire Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>

<!-- define the Login Configuration for this application -->
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Tomcat Manager Application</realm-name>
</login-config>
</web-app>
```

- for simple applications, the default deployment descriptor is sufficient.

Default mapping from URLs to files:

- **Servlets:** `http://HOST:PORT/myapplication/servlet/package.servletclass` (omit "`package.`" if default package)
- **auxiliary files:** `http://HOST:PORT/myapplication/file`

Warning: if not using the default deployment descriptor, [make sure](#) that the default URL mapping (the "invoker servlet") is deactivated (using `servlet-mapping`)!

# Servlet Contexts

Each *Web application* has a [ServletContext](#) object (returned by `getServletContext()`):

- allows data to be **shared** between interactions and different servlets
- contains a server **log**
- access to global server configuration
- other Web application's servlet contexts are accessible (can be restricted)

Shared state:

- `void setAttribute(String name, Object object)`
- `Object getAttribute(String name)`

Any object can be stored - and freely modified.

(This is a good alternative to servlet class instance variables which are private to the servlet.)

# Sessions

Session state is maintained in a [HttpSession](#) object:

- `request.getSession(true)` returns the current `HttpSession` object or generates a new
- `boolean isNew()` - returns true if new session
  
- `Object getAttribute(String name)` - reads per-session state
- `void setAttribute(String name, Object value)` - writes per-session state
  
- `String getId()` - session info
- `long getCreationTime()`
- `long getLastAccessedTime()`
- `void setMaxInactiveInterval(int seconds)`

- this resembles the management of [shared state](#).

A typical example:

```
...
HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart) session.getAttribute("shoppingcart");
if (cart==null) {
    cart = new ShoppingCart();
    session.setAttribute("shoppingcart", cart);
}
addItemToCart(cart);
...
```

Under the hood:

- uses **cookies** or **URL rewriting**
- since URL rewriting may be used, URLs to our own Servlets should always be passed through `response.encodeURL(String)`

[Cookies](#) can also be controlled manually...

Note: this is a rather low-level approach - the session flow is not explicit in the code!

# Security

Authentication and encryption can be controlled

- **declaratively** - using the deployment descriptor (`web.xml`)
- **operationally** - by explicit service code
- or a combination

**Authentication** using `web.xml`:

- **form-based** authentication
  1. make file with usernames, passwords, and roles
  2. design login and login-failure pages
  3. specify URLs that require authentication (and optionally, also SSL)
- **HTTP Basic** authentication
  1. make file with usernames, passwords, and roles
  2. specify URLs that require authentication (and optionally, also SSL)

**SSL:**

- requires the JSSE package
- requires a public-key server certificate

- details are server specific

`HttpServletRequest` contains useful security methods:

- `String getRemoteUser()` - who has logged in?
- `boolean isUserInRole(String role)` - what is the user's abstract role?
- `boolean isSecure()` - does the connection use SSL?

# Listeners

- event handlers

Events:

- servlet context events ([ServletContextListener](#)):
  - initialize
  - destroy
- servlet context attribute events ([ServletContextAttributeListener](#)):
  - set
  - add
  - remove
- session events ([HttpSessionListener](#)):
  - create
  - invalidate
  - time out
- session attribute events ([HttpSessionAttributeListener](#)):
  - set
  - add
  - remove

Implement the appropriate interface, register your listener (in `web.xml`).

Example listener and deployment declaration:

```
public class DataListener implements ServletContextAttributeListener
{
    public void attributeReplaced(ServletContextAttributeEvent event)
    {
        if (event.getName().equals("some_data"))
            updateSomeOtherData();
    }
    ...
}
```

```
<listener>
  <listener-class>DataListener</listener-class>
</listener>
```

- useful for implementing dependencies between data
- but only if data is modified explicitly via `setAttribute` - and not via some methods/fields in the data

- also useful for monitoring the running service

# Filters

- inserting **hooks** before and after requests are processed
- **wrappers** modify the request and the response

A [Filter](#) can modify

- the incoming request, e.g.
  - redirect
  - check security requirements
  - perform logging
- the outgoing response, e.g.
  - data compression or encryption
  - [XSLT transformation](#)
  - perform logging

`FilterChain`: multiple filters are processed in deployment order in a stack discipline with the Servlet in the bottom.

Supplementary, [HttpServletRequestWrapper](#) and [HttpServletResponseWrapper](#) provide **wrappers** to modify the request/response.

Example filter and deployment declaration:

```
public class TraceFilter implements Filter
{
    private ServletContext context;

    public void init(FilterConfig config)
        throws ServletException
    {
        context = config.getServletContext();
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException
    {
        context.log("[ "+request.getRemoteHost()+" ] request: "+
                    ((HttpServletRequest) request).getRequestURL());
        chain.doFilter(request, response);
        context.log("[ "+request.getRemoteHost()+" ] done");
    }
}
```

```
<filter>
  <filter-name>myfilter</filter-name>
  <filter-class>TraceFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>myfilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

# JDBC and JDOM

- alternatives to [ServletContext](#):

- [JDBC](#) - API for connecting to SQL databases
- [JDOM](#) / [JAXP](#) - APIs for manipulating XML documents (see the [XML Tutorial!](#))

# JSP - JavaServer Pages

- a Java-version of ASP/PHP

1. [JSP Pages](#)
2. [Deployment](#)
3. [Translation into Servlets](#)
4. [Combining JSP and Servlets](#)
5. [Custom Tag Libraries](#)
6. [The Standard Tag Library \(JSTL\)](#)

# JSP Pages

## JSP:

- HTML (or XML) **templates**
- **embedded Java code** generates HTML dynamically
- **user-defined tags** referring to Java code that generates HTML dynamically
  
- = Servlets inside-out (JSP pages are translated into Servlets)
- all Servlet features are directly available

Another [Hello-World](#) example:

```
<% response.addDateHeader("Expires", 0); %>
<html><head><title>JSP</title></head>
<body><h1>Hello World!</h1>
<%! private int hits = 0; %>
You are visitor number <% synchronized(this) { out.println(++hits); } %>
since the last time the service was restarted.
<p>
This page was last updated: <%= new java.util.Date().toLocaleString() %>
</body></html>
```

- Expressions: `<%= expression %>`
- Statements: `<% statement %>`
- Declarations: `<%! declaration %>`
- JSP directives: `<%@ directive %>`

Alternative XML syntax:

- `<jsp:expression>...</jsp:expression>`
- `<jsp:scriptlet>...</jsp:scriptlet>`
- `<jsp:declaration>...</jsp:declaration>`
- `<jsp:directive.../>`
- ([<jsp:include.../>](#))

Pre-declared variables:

- `HttpServletRequest request`
- `HttpServletResponse response`
- `HttpSession session`
- `JspWriter out`
- `ServletContext application`
- `ServletConfig config`
- [PageContext pageContext](#)

Directives:

- [include](#)
- [page](#)
- taglib

# Deployment

- just put the `.jsp` files in the [Web application directory](#)
- the server will take care of translation and compilation

# Translation into Servlets

Translation is extremely simple - doesn't even need to parse the HTML or Java code!

```
HTML(/XML)          -> out.write("...");
<%= expression %>  -> out.print(expression);
<% statement %>     -> statement
<%! declaration %> -> declaration (in Servlet class)
<%@ directive %>   -> instruction to translator, e.g. include file
```

Example:

```
<% response.addDateHeader("Expires", 0); %>
<html><head><title>JSP</title></head>
<body><h1>Hello World!</h1>
<%! private int hits = 0; %>
You are visitor number <% synchronized(this) { out.println(++hits); } %>
since the last time the service was restarted.
<p>
This page was last updated: <%= new java.util.Date().toLocaleString() %>
</body></html>
```

is by Tomcat translated [into](#) the following Servlet code (slightly abbreviated):

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class HelloWorld2$jsp extends HttpJspBase {
    private int hits = 0;

    private static boolean _jspx_initiated = false;

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException
    {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {
            if (_jspx_initiated == false)
                synchronized (this) {
                    if (_jspx_initiated == false) {
                        _jspx_init();
                        _jspx_initiated = true;
                    }
                }
        }
    }
}
```

```

    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=ISO-8859-1");
    pageContext = _jspxFactory.getPageContext(this, request, response, "",
true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    response.addDateHeader("Expires", 0);
    out.write("\r\n<html><head><title>JSP</title></head>\r\n<body><h1>Hello
World!</h1>\r\n");
    out.write("\r\nYou are visitor number ");
    synchronized(this) { out.println(++hits); }
    out.write("\r\nsince the last time the server was
restarted.\r\n<p>\r\nThis page was last updated: ");
    out.print( new java.util.Date().toLocaleString() );
    out.write("\r\n</body></html>");
} catch (Throwable t) {
    if (out != null && out.getBufferSize() != 0) out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
} finally {
    if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
}
}
}

```

Note: since translation is on the lexical level, the following is perfectly acceptable in a JSP page:

```

<% if (Math.random() < 0.5) { %>
    Have a <b>nice</b> day!
<% } else { %>
    Have a <b>lousy</b> day!
<% } %>

```

# Combining JSP and Servlets

Common approach:

- **Servlets** handle the **contents** (using lots of Java code)
- **JSP pages** handle the **presentation** (using lots of HTML)

- communicate using `HttpSession` attributes, forward requests using `RequestDispatcher`

Example Servlet receiving the original request:

```
public class Register extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        String email = request.getParameter("email");
        HttpSession session = request.getSession(true);
        session.setAttribute("email", email);
        RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("/present.jsp");
        dispatcher.forward(request, response);
    }
}
```

JSP page producing the final response:

```
<html><head>mailing list</head><body>
<h1>Welcome!</h1>
You have registered the following address:
<tt><%= session.getAttribute("email") %></tt>
<p><a href="continue">Continue</a>
</body></html>
```

- this quickly becomes a mess...

Often, applications are composed of [JavaBean Components](#).

# Custom Tag Libraries

- making abbreviations for commonly used JSP fragments

Example [JSP page](#) with custom tag:

```
<%@ taglib uri="/WEB-INF/tlds/mytags.tld" prefix="my" %>
<my:wrapper style="k001">
  <b>hello!</b>
</my:wrapper>
```

Tag Handler code (compile and put in WEB-INF/classes/mytaglib/WrapperTag.class):

```
package mytaglib;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class WrapperTag extends TagSupport {
    private String style;

    public void setStyle(String style) {
        this.style = style;
    }

    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            if (style.equals("k001")) {
                out.print("<html><head><title>MyCoolService</title></head><body
bgcolor=\"red\">");
            } else {
                // ...
            }
        } catch (IOException e) { System.out.println("Error in WrapperTag: "+e); }
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() {
        try {
            JspWriter out = pageContext.getOut();
            out.print("</body></html>");
        } catch (IOException e) { System.out.println("Error in WrapperTag: "+e); }
        return EVAL_PAGE;
    }
}
```

(See the [API](#).)

Tag Library Descriptor file (put in WEB-INF/tlds/mytags.tld):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>MyTags</short-name>

  <tag>
    <name>wrapper</name>
    <tag-class>mytaglib.WrapperTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>style</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

There are [lots of free tag libraries!](#)

# The Standard Tag Library (JSTL)

- a [large library of custom tags](#)

Topics:

- control structures (conditionals, iteration, ...)
- text formatting
- XML
- database

- JSTL 1.0 was released 11 April 2002

View Sun's [JSTL Tutorial](#).

# JWIG - High-Level Web Services in Java

- a novel approach

1. [JWIG](#)
2. [Sessions in JWIG](#)
3. [XML Templates](#)
4. [Static Guarantees](#)
5. [A Tiny Example](#)
6. [The Service Class](#)
7. [The Session Class](#)
8. [The XML Class](#)
9. [Code Gaps](#)
10. [The JWIG API](#)
11. [The JWIG System](#)
12. [Runtime System](#)
13. [A Larger Example](#)
14. [The Data Layer](#)
15. [The Functionality Layer](#)
16. [Generating Dynamic XML](#)
17. [The Presentation Layer](#)
18. [The Template Manager](#)
19. [The Development Cycle](#)
20. [Risky Business](#)
21. [Static Analysis](#)
22. [The JWIG Analyzer](#)
23. [Checking Summary Graphs](#)
24. [Catching Errors](#)
25. [DSD2 Schemas](#)

# JWIG

The [JWIG](#) system is another Java-based framework for programming Web applications.

It inherits all the usual benefits from Java, but includes four unique features:

- a stronger **session** concept
- XML **templates** as first-class values
- static **guarantees** about the behavior of running services
- **shared state** is accessed through usual scope mechanisms

JWIG is implemented by:

- providing a set of Java packages `dk.brics.jwig`
- extending the Java syntax using a desugarer as preprocessor
- supplying a special module for the Apache server

# Sessions in Jwig

In CGI-scripts, Servlets, and JSP the session concept is **implicit**:

- the identity of a session must be stored in cookies, hidden fields, or the URL
- local state must be explicitly saved and restored across interactions with the client

In Jwig, the session concept is **explicit**:

- a `Session` is written as a sequential program (just like an ordinary Java method)
- interactions with the client are similar to remote method invocations
- the local state is automatically the full state of the thread running the `Session`:
  - all local variables, regardless of their type or complexity
  - the full stack of method invocations

The Hello Word example:

```
import dk.brics.jwig.runtime.*;

public class Hello extends Service
{
    public class Example extends Session
    {
        public void main() {
            XML x = [[ <html><head><title>JWIG</title></head><body>
                       <h1><[what]></h1></body></html> ]];
            x = x <[ what = [[ Hello World! ]] ];
            show x;
            XML y = [[ <html><head><title>JWIG</title></head><body>
                       Goodbye!</body></html> ]];
            show y;
        }
    }
}
```

# XML Templates

In CGI-scripts and Servlets, XML values are **implicit**:

- all values appear as the output from `print` statements
- the functionality and presentation of a service are completely intertwined

In JSP, XML values are **partly** explicit:

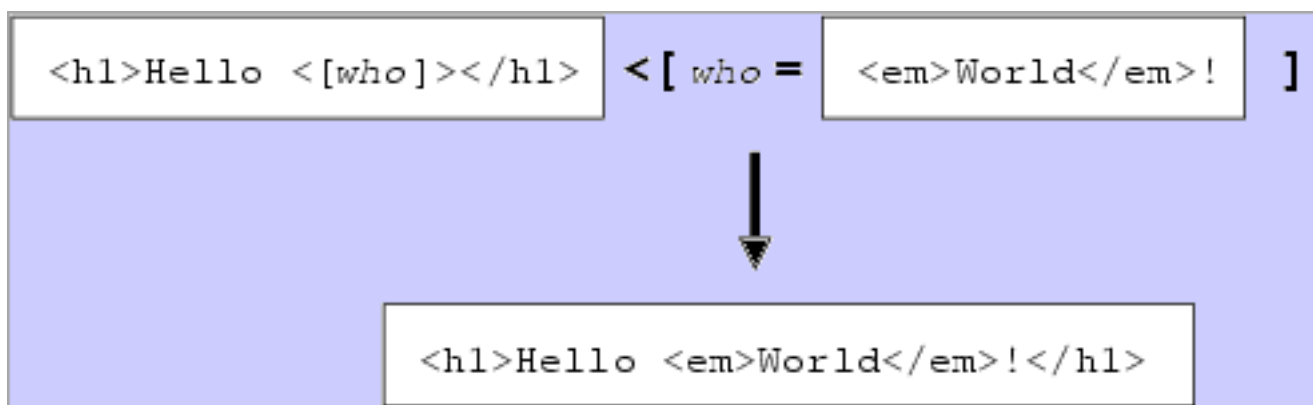
- part of some values are written as constants
- for simple applications, this provides some separation of functionality and presentation

In JMWIG, XML values are **explicit**:

- they are all instances of an `XML` class
- they are *first-class* values, just like `String` values
- they are subject to computations

Instances of `XML` are **templates**, that is XML fragments containing named **gaps**.

Gaps may at runtime be **plugged** with other templates or strings:



# Static Guarantees

In JSP and Servlets, interaction with the client is a **risky** business:

- the client may receive **invalid** XML documents (e.g. XHTML)
- the server may receive **unexpected** form fields

The Web service may **fail** in either of these cases.

In JWIG, client interaction is viewed abstractly as a **remote method invocation**:

- the compiler usually checks that argument and result types of methods are valid

JWIG can provide similar guarantees for interactions by **analyzing**:

- whether the (dynamically generated) XML document is **valid** according to the appropriate XML schema
- whether the forms **fields** being received are as **expected**

# A Tiny Example

The following is a tiny, self-contained Jtwig [service](#):

```
import dk.brics.jtwig.runtime.*;

public class MyService extends Service {

    int counter = 0;

    synchronized int next() { return ++counter; }

    public class ExampleSession extends Session {

        XML wrapper =
            [[ <html>
                <head><title>Jtwig</title></head>
                <body>
                    <[body]>
                </body>
            </html> ]];

        XML hello =
            [[ <form>
                Enter your name: <input name="handle"/>
                <input type="submit" name="continue" value="continue" />
            </form> ]];

        XML greeting =
            [[ <form>
                Hello <[who]>, you are user number <[count]>
                <input type="submit" name="continue" value="continue" />
            </form> ]];

        XML goodbye =
            [[ Goodbye <[who]> ]];

        public void main() {
            show wrapper<[body=hello]>;
            String name = receive handle;
            show wrapper<[body=greeting<[who=name,count=next()]]>;
            exit wrapper<[body=goodbye<[who=name]]>;
        }
    }
}
```

Notable features:

- the entire service is a subclass of a `Service` class
- sessions are inner subclasses of a `Session` class
- `wrapper` is a variable of type `XML`
- XML constants are written in special syntax `[ [ . . . ] ]`
- gaps are enclosed in `<[ . . . ]>`
- `counter` is a shared variable
- `name` is a local variable
- interactions are performed by the `show` statement
- form fields are received by the `receive` expression
- XML values can be plugged together `<[ . . . ]`

Look at other [small examples](#).

# The Service Class

A service is specified as a subclass of the `Service` class.

A `Service` object corresponds to an instance of an installed service and contains:

- **shared data**, which are simply the fields in the object
- an inner class for each kind of **session** offered to the clients

The `Service` class offers the following features:

- `checkpoint()` and `rollback()` of the **shared state** (using serialization)
- handling of **cookies** (for use by the programmer, not for encoding session ID!)
- **logging** of events
- setting of **timeouts**
- **SSL** support
- **blocking** of incoming requests

# The Session Class

A session is specified as a subclass of the `Session` class.

A `Session` object corresponds to an single thread communicating with a particular client, and contains:

- **local data**, which are simply the fields in the object

The `Session` class offers the following features:

- the `show`, `receive`, and `exit` methods for **interactions**
- **temporary reply** documents for impatient clients (see [example](#), [source](#))
- **access control**, using HTTP Authentication (see [example](#), [source](#))

There are two **variations** of sessions:

- the `Page` class which disallows `show` statements (only allows `exit`)
- the `Seslet` class which allows communication with programs rather than human clients

# The XML Class

Templates are implemented by the `XML` class.

An `XML` object corresponds to an XML fragment possibly containing named gaps.

The `XML` class offers the following features:

- template **constants**
- the `plug` method for **composing** templates
- **printing** the XML document represented by an object
- the `get` method for dynamically **loading** a template from a URL

The JWIG system allows an elaborate **syntactic sugar** for these constructions.

A template constant is written as:

```
[[ <table border="0" cellspacing="0">
  <tr><td align="left">
    <a href=[js]>
      
    </a>
  </td></tr>
  <tr><td align="left">
    <[name]>
  </td></tr>
</table>
<[rest]>
]]
```

where `js`, `name`, and `rest` are gaps.

Templates are **plugged** by writing:

```
options = options<[rest=templateOption<[inx=versionInx(contents[i]),
                                         date=versionDate(contents[i])]]];
```

This is different from [JDOM](#) documents in the following ways:

- templates need not be constructed bottom-up
- large chunks are written in normal syntax
- the underlying data structure exploits sharing of common fragments
- however, templates cannot be deconstructed (future work)

- and different from [Servlets](#) in the following ways:

- documents need not be constructed one line at a time
- well-formedness is easily guaranteed

- escaping of special characters is automatic
- specialized [analyses](#) is possible!

# Code Gaps

JWIG can emulate the JSP style with embedded code using **code gaps**:

```
import dk.brics.jwig.runtime.*;

public class Today extends Service {
    int counter = 0;

    synchronized int next() { return ++counter; }

    public class View extends Page {
        public void main() {
            exit [[ <html><head><title>JWIG</title></head><body>
                Today is <{ return new Date(); }>
                <p/>
                You are visitor number <{ return next(); }>
                </body></html> ]];
        }
    }
}
```

- the code gaps are evaluated when the document is shown
- only service fields and methods are visible in code gaps

# The JWIG API

The JWIG classes have a **fully documented** [API](#).

The available packages are:

- `dk.brics.jwig.runtime`: services, sessions, and templates
- `dk.brics.jwig.desugar`: JWIG to Java converter
- `dk.brics.jwig.runwig`: extension to Apache
- `dk.brics.jwig.analysis`: static guarantees

# The Jwig System

Consider again the [MyService](#) example.

The following commands are used (on UNIX/Linux) to create and maintain a working service:

**Compile** the source code:

```
jwig compile MyService.jwig
```

This creates the following files:

```
MyService$ExampleSession.class  MyService.class
```

Obtain **static guarantees**:

```
jwig analyze *.class
```

**Install** the service:

```
jwig install /home/mis/jwig-mis/MyService *.class
```

**Run** the service with the URL:

```
http://freewig.brics.dk/jwig-mis/MyService/MyService*ExampleSession
```

The service can be **updated**:

```
jwig update /home/mis/jwig-mis/MyService *.class
```

**Uninstall** the service:

```
jwig uninstall /home/mis/jwig-mis/MyService
```

# Runtime System

At runtime, each session is allocated:

- one JVM thread (persistent through the session lifetime)
- one sub-directory (containing the thread's files)
- one URL (referring to `index.html` in the session sub-directory)

The `index.html` page always contains the newest response shown to the client.

This is different from the JSP/Servlet solutions:

- we are not using cookies, URL rewriting, or hidden fields
- the URL functions as an **identity** of the session
- sessions can be **bookmarked** (suspended and later resumed)
- the **history buffer** of the browser is not filled with references to obsolete requests

# A Larger Example

The **challenge** is to make a dynamic version of [www.musikbestilling.dk](http://www.musikbestilling.dk).

Large parts of this service are **reconstructed** in a [JWIG version](#).

It has been made **dynamic** in the following ways:

- artists are described in a database updated by the administrator
- artists may edit all personal information
- customers may view and book available dates
- bookings are available to artists who may confirm og cancel
- customers are automatically notified by e-mail
- artists may add and remove available dates

Another dynamic aspect allows editing of the XHTML in a running service...

# The Data Layer

The data is described in four classes:

- [Kunstner.jwig](#)
- [Kunde.jwig](#)
- [Booking.jwig](#)
- [Anmeldelse.jwig](#)

The data classes implement Comparable and Serializable to allow sorting and checkpointing:

```
import java.util.*;
import java.io.*;

class Booking implements Comparable,Serializable {
    static int ids = 0;
    String id;
    static final int FRI = 0;
    static final int RESERVERET = 1;
    static final int BEKRÆFTET = 2;
    String kunde;
    String kunstner;
    int dag,måned,år;
    int status;

    public Booking(String kunstner,
                   int dag,
                   int måned,
                   int år) {
        this.id=String.valueOf(ids++);
        this.kunstner=kunstner;
        this.dag=dag;
        this.måned=måned;
        this.år=år;
        this.status=Booking.FRI;
    }

    public int compareTo(Object o) {
        Calendar c = Calendar.getInstance();
        c.set(år,måned-1,dag);
        Date d = c.getTime();
```

```
Booking b = (Booking)o;  
c.set(b.år,b.måned-1,b.dag);  
Date e = c.getTime();  
return -d.compareTo(e);  
}  
}
```

Objects of these classes are stored in collection objects:

```
HashMap kunstnere;  
HashMap kunder;  
HashMap bookings;  
HashMap anmeldelser;
```

The [HashMap](#) class is the lazy programmer's friend!

# The Functionality Layer

Look at the part of the [Musik](#) service that permits booking of artists.

Its functionality is described by the following main method:

```
public void main() {
    while (true) {
        show(genIndhold(titleMusikere,genMusikere()));
        Kunstner k = (Kunstner)kunstnere.get(receive who);
        show(genIndhold("..... "+k.navn+" !",genPræsentation(k)));
        if ((receive submit).equals("book")) {
            show(genIndhold("..... bestil "+k.navn+" !",genBooking(k)));
            String[] booking = receive[] booking;
            if (booking.length>0) {
                Kunde u = new Kunde(receive navn,
                                    receive adresse,
                                    receive postnummer,
                                    receive by,
                                    receive telefon,
                                    receive email);

                kunder.put(u.id,u);
                for (int i=0; i<booking.length; i++) {
                    Booking b = (Booking)bookings.get(booking[i]);
                    b.status = Booking.RESERVERET;
                    b.kunde = u.id;
                }
                save();
                exit genIndhold("..... mange tak !",templateKvittering<[navn=k.navn]);
            }
        }
    }
}
```

Notable points:

- it *is* just a simple sequential program
- the *business logic* is easy to recognize
- no XHTML is immediately visible

# Generating Dynamic XML

Dynamic XML is best generated using `genXYZ()` methods.

Look at the clickable table of artists:

```
<form action="..." >
  <input type="hidden" value="" name="who" />
  <script type="text/javascript">
    function handle(id) {
      document.forms[0]['who'].valueid;
      document.forms[0].submit();
    }
  </script>
  <table border="2">
    <tr>
      <td class="menu" align="center" valign="middle" width="135" height="40">
        Diskoteker
      </td>
      <td class="menu" align="center" valign="middle" width="135" height="40">
        Solister
      </td>
      <td class="menu" align="center" valign="middle" width="135" height="40">
        Strippere
      </td>
    </tr>
    <tr>
      <td class="item" align="center" valign="middle" width="135" height="40">
        <a class="item" href="javascript:handle(3)">Disko Keld</a>
      </td>
      <td class="item" align="center" valign="middle" width="135" height="40">
        <a class="item" href="javascript:handle(0)">Karl Børge</a>
      </td>
      <td class="item" align="center" valign="middle" width="135" height="40">
        <a class="item" href="javascript:handle(2)">Lola Pagola</a>
      </td>
    </tr>
    <tr>
      <td class="item" align="center" valign="middle" width="135" height="40"></td>
      <td class="item" align="center" valign="middle" width="135" height="40">
        <a class="item" href="javascript:handle(1)">Karl Ejnar</a>
      </td>
      <td class="item" align="center" valign="middle" width="135" height="40"></td>
    </tr>
    <tr>
      <td class="item" align="center" valign="middle" width="135" height="40"></td>
      <td class="item" align="center" valign="middle" width="135" height="40">
        <a class="item" href="javascript:handle(4)">Party-Anders</a>
      </td>
      <td class="item" align="center" valign="middle" width="135" height="40"></td>
    </tr>
  </table>
</form>
```

This is a very dynamic document:

- the number of columns depends in the contents of the database
- the rows must be filled out with "trailing" blank data

This is very simple using XML templates. First find the data:

```
XML genMusikere() {
    Vector kats = new Vector();
    Vector cols = new Vector();
    Iterator it = new TreeSet(kunstnere.values()).iterator();
    while (it.hasNext()) {
        Kunstner k = (Kunstner)it.next();
        boolean found = false;
        for (int i=0; i<kats.size() && !found; i++) {
            if (kats.elementAt(i).equals(k.kategori)) {
                found = true;
                ((Vector)cols.elementAt(i)).add(k);
            }
        }
        if (!found) {
            kats.add(k.kategori);
            Vector v = new Vector();
            v.add(k);
            cols.add(v);
        }
    }
    return templateMusikere<[kategorier=genKategorier(kats),
                               cols=genCols(cols)]>;
}
```

Then generate the table header:

```
XML genKategorier(Vector kats) {
    XML x = [[<[rest]>]];
    for (int i=0; i<kats.size(); i++) {
        x = x<[rest=templateKategori<[kategori=(String)kats.elementAt(i)]]>;
    }
    return x<[rest=[[ ]]]>;
}
```

Finally, generate the table entries:

```
XML genCols(Vector cols) {
    XML x = [[<[row]>]];
    boolean done = false;
    int i = 0;
    while (!done) {
        int empty = 0;
        XML r = [[<[rest]>]];
        for (int j=0; j<cols.size(); j++) {
            Vector v = (Vector)cols.elementAt(j);
            if (v.size()>i) {
                Kunstner k = (Kunstner)v.elementAt(i);
                r = r<[rest=templateMusiker<[navn=k.navn,code=genCode(k.id)]]>;
            }
        }
        if (empty>0) {
            r = r<[rest=templateMusiker<[navn="",code=""]>]> empty times;
        }
        x = x<[rest=r]>;
        done = true;
    }
}
```

```
    } else {  
        r = r<[rest=templateTom];  
        empty++;  
    }  
}  
done = empty==cols.size();  
if (!done) x = x<[row = templateRow<[data=r<[rest=[[[]]]]]];  
i++;  
}  
return x<[row=[[[]]]];  
}
```

We still haven't seen any XHTML markup...

# The Presentation Layer

So far, we have assumed very little about the concrete XHTML markup:

- the `genXYZ()` methods **assume** some gaps are present
- the receive operations **expects** an input field named `who`

Other than that, we are free to design the look of our service through a collection of XML template constants:

```
private static XML templateMusikere = [[
  <form>
    <input type="hidden" name="who" value=""/>
    <script type="text/javascript">
      function handle(id) {
        document.forms[0]['who'].value = id;
        document.forms[0].submit();
      }
    </script>
    <table border="2">
      <tr><[kategorier]></tr>
      <[cols]>
    </table>
  </form>
]];

private static XML templateKategori = [[
  <td class="menu" align="center" valign="middle" width="135" height="40">
    <[kategori]>
  </td>
  <[rest]>
]];

private static XML templateRow = [[
  <tr><[data]></tr>
  <[row]>
]];

private static XML templateTom = [[
  <td class="item" align="center" valign="middle" width="135" height="40">
  </td>
  <[rest]>
]];

private static XML templateMusiker = [[
  <td class="item" align="center" valign="middle" width="135" height="40">
    <a class="item" href=[code]><[navn]></a>
  </td>
  <[rest]>
]];
```

# The Template Manager

In Jtwig, we can **browse** and **edit** all the XML templates in a [template manager](#).

This is in fact just another Jtwig service, [TempMan.jtwig](#)

In particular, we can **interactively** modify the graphical design during the lifetime of a [running](#) session thread!

This requires that we use the `get` operation:

```
private static XML templateMusikere;
private static XML templateKategori;
private static XML templateMusiker;
private static XML templateTom;
private static XML templateRow;

public class Refresh extends Page {
    public void main() {
        templateMusikere =
            get "file:/home/mis/Musik/templates/Musikere/Musikere";
        templateKategori =
            get "file:/home/mis/Musik/templates/Musikere/Kategori";
        templateMusiker =
            get "file:/home/mis/Musik/templates/Musikere/Musiker";
        templateTom =
            get "file:/home/mis/Musik/templates/Musikere/Tom";
        templateRow =
            get "file:/home/mis/Musik/templates/Musikere/Row";
    }
}
```

# The Development Cycle

In many Web application projects, programmers and graphical designers **fight for control**.

Different scenarios:

- the programmer designs the service and asks the designer for advice (which is perhaps ignored)
- the designer makes a bunch of static pages and ask the programmer to make them come alive
- the programmer and designer sits down and tries to work together

In all cases, there are **bottlenecks** and **problems** of communication.

Using the JWIG approach, there is a **contract** between the two:

*there are these 33 templates that must contain specific gaps and input fields*

Within these constraints, programmers and designers can work **independently**.

Future work in JWIG:

- automatic support for negotiating and checking contracts
- FrontPage-style tool for editing templates

# Risky Business

Interaction with clients is normally completely unchecked:

- the client may receive **invalid** XML documents (e.g. XHTML, WML)
- the server may receive **unexpected** form fields

These correspond to various failure modes for the service:

- the Web browser shows ugly XHTML pages
- the WAP phone goes dead, because it cannot handle invalid WML code
- the service is missing data, which the client should have provided
- the client purchased several items, but only the first is shipped
- ...

JWIG provides means for making this interface **safer**!

# Static Analysis

All **interesting** properties of programs are (sadly) **undecidable** (this is [Rice's Theorem](#)):

- *does my program terminate (the [Halting Problem](#))?*
- *how much heap space does my program need?*
- *can my program ever dereference a `null` pointer?*
- *will my program correctly sort this list?*
- *can my program only generate valid XHTML documents?*
- *which input fields are defined in this dynamically generated XHTML document?*

Instead of giving up, compiler writers resort to **static analysis**:

- don't try to decide the question exactly
- settle for an approximative answer
- only use safe answers

For the Halting Problem, the answers would be:

- *Yes, your program definitely terminates*
- *I don't think your program terminates, but I'm not really sure*

The **engineering challenge** is to give useful answers as often as possible.

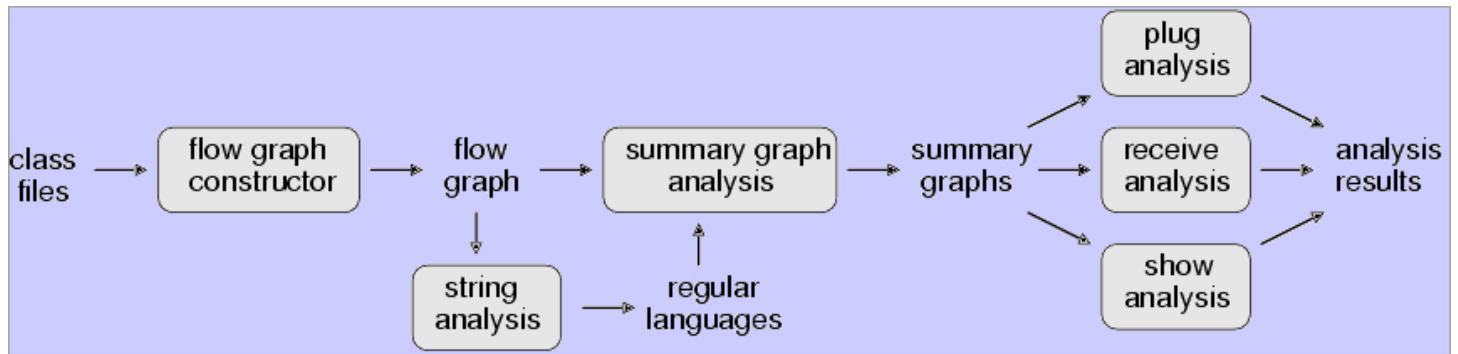
Static analysis is a **standard technique** involving:

- defining an abstraction of the properties we are interested in (a [lattice](#))
- extracting a [control flow graph](#) for the program
- defining dataflow equations for all program constructions
- obtaining a minimal solution using fixed-point iteration

This is enough to provide reasonable accuracy for analyzing JWIG programs.

# The Jwig Analyzer

The Jwig analyzer works as follows:



The three checks **guarantee** that:

- only gaps that are present will be plugged
- all input fields are present when received
- all XML shown is valid

On the example program:

```
import dk.brics.jwig.runtime.*;

public class Greetings extends Service {
    String greeting = null;

    public class Welcome extends Session {
        XML cover = [[ <html>
            <head><title>Welcome</title></head>
            <body bgcolor=[color]>
                <{ if (greeting==null)
                    return [[ <em>Hello World!</em> ]];
                else
                    return [[ <b><g></b> ]] <[g=greeting];
            }>
            <[contents]>
        </body>
        </html> ]];

        XML getinput = [[ <form>Enter today's greeting:
            <input type="text" name="salutation"/>
            <input type="submit"/>
        </form> ]];

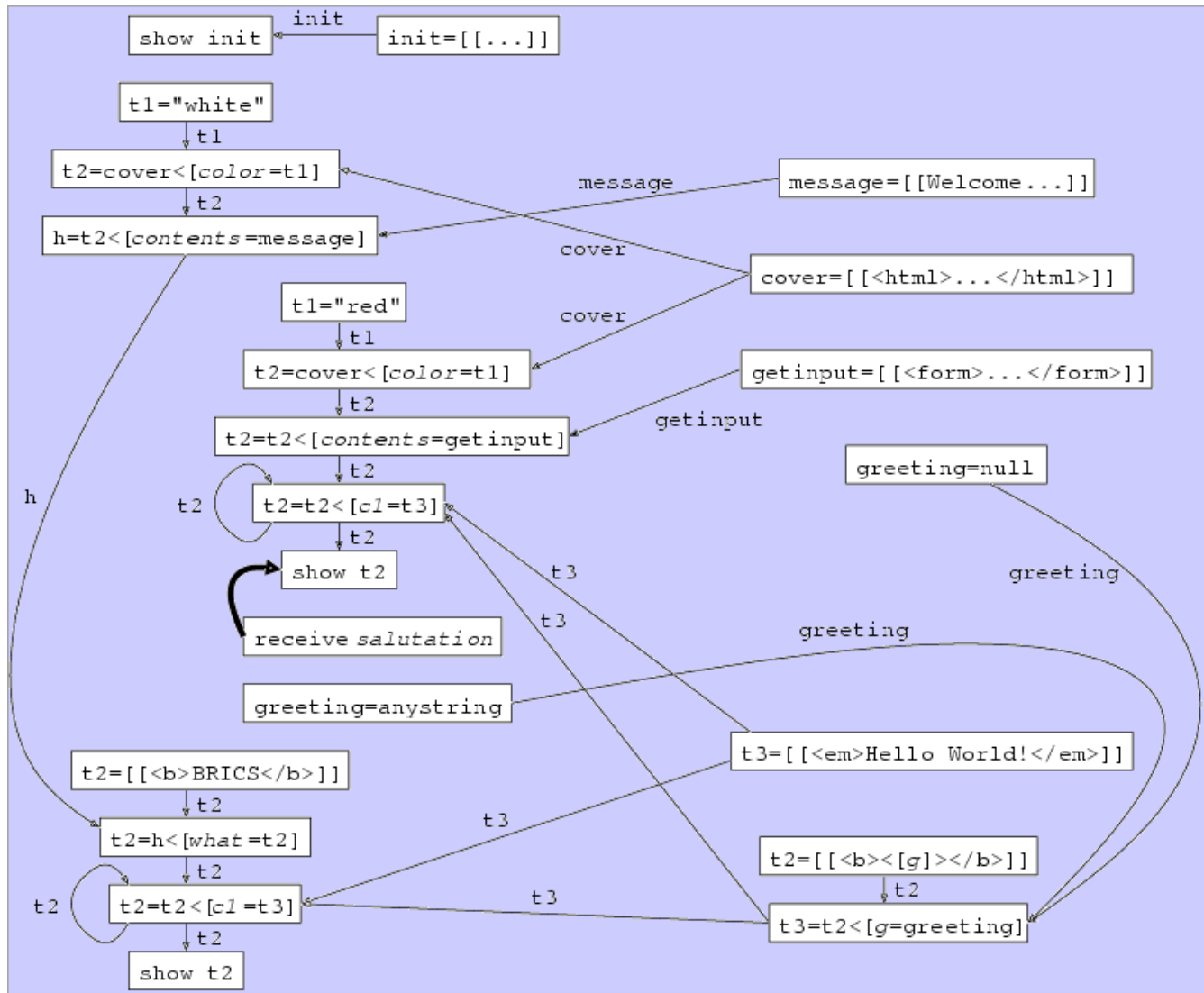
        XML message = [[ Welcome to <[what]>. ]];

        public void main() {
            XML h = cover<[color="white",contents=message];
            if (greeting==null) {
                show cover<[color="red",contents=getinput];
                greeting = receive salutation;
            }
            exit h<[what=[[<b>BRICS</b>]]];
        }
    }
}
```

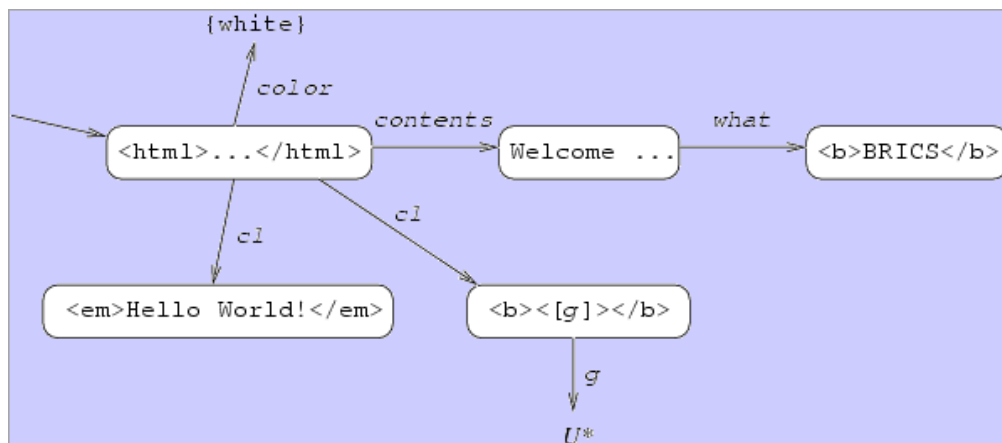
the analyzer goes through eight phases:

1. single methods
2. code gaps
3. method invocations
4. exceptions
5. show and receive operations
6. arrays
7. field variables
8. graph simplification

to construct a flow graph:



The possible documents being shown at the exit statement are then approximated by a summary graph:



No errors are found, in this case.

The key idea is the notion of **summary graphs**: a summary graph approximates the set of XML templates that may appear at a given program point for a given variable or expression.

- a **node** represents a constant template from the program source
- an **edge** represents a possible plug operation

# Checking Summary Graphs

Summary graphs are the basis for checking our three desired properties:

- only gaps that are present will be plugged
- all input fields are present when received
- all XML shown is valid

For every expression of the form:

`X<[g=Y]`

we must check that all documents described by the summary graph obtained for  $\mathcal{Y}$  contains a gap named  $g$ .

For every expression of the form:

`receive f`

we must first find all statements of the form:

`show X`

that are relevant to this point in the execution, and for each of those determine if all documents described by the summary graph obtained for  $\mathcal{X}$  contains a field named  $f$ .

For every statement of the form:

`show X`

it is checked that all documents described by the summary graph obtained for  $\mathcal{X}$  are valid XHTML documents.

# Catching Errors

If we introduce an error by forgetting the name attribute:

```
import dk.brics.jwig.runtime.*;

public class Greetings extends Service {
    String greeting = null;

    public class Welcome extends Session {
        XML cover = [[ <html>
            <head><title>Welcome</title></head>
            <body bgcolor=[color]>
                <{ if (greeting==null)
                    return [[ <em>Hello World!</em> ]];
                else
                    return [[ <b><[g]></b> ]] <[g=greeting];
                }>
                <[contents]>
            </body>
        </html> ]];

        XML getinput = [[ <form>Enter today's greeting:
            <input type="text" name="salutation"/>
            <input type="submit"/>
        </form> ]];

        XML message = [[ Welcome to <[what]>. ]];

        public void main() {
            XML h = cover<[color="white",contents=message];
            if (greeting==null) {
                show cover<[color="red",contents=getinput];
                greeting = receive salutation;
            }
            exit h<[what=[[<b>BRICS</b>]]];
        }
    }
}
```

then the following error message is produced:

```

*** Field `salutation' is never available on line 30
*** Invalid XHTML at line 29
--- element 'input': requirement not satisfied:
<or>
  <attribute name="type">
    <union>
      <string value="submit" />
      <string value="reset" />
    </union>
  </attribute>
  <attribute name="name" />
</or>
2 problems encountered.

```

This is fast enough to run in practice (time in seconds):

Name	Lines	Templates	Shows	Time
Chat	80	4	3	5.370
Guess	94	8	7	7.147
Calendar	133	6	2	7.029
Memory	167	9	6	9.718
TempMan	238	13	3	7.719
WebBoard	766	32	24	9.769
Bachelor	1,078	88	14	115.641
Jaoo	3,923	198	9	35.997

# DSD2 Schemas

Validity of XML documents can only be checked if **valid documents** are **specified formally**.

This can be done using various formalism:

- [DTD](#)
- [XML Schema](#)
- [Schematron](#)
- [Trex](#)
- [Examplotron](#)
- [RELAX NG](#)

We have developed yet another formalism, DSD2, because:

- it is more expressive and yet simpler
- it is well-suited for analyzing summary graphs

DSD2 can be used to specify XML languages such as:

- [XHTML](#)
- [PowerForms](#)
- [DSD2](#)

# PowerForms

- declarative input field validation

1. [Input Field Validation](#)
2. [The PowerForms Language](#)
3. [Cool Examples](#)
4. [PowerForms Constraints](#)
5. [PowerForms Expressions](#)
6. [Regular Expressions](#)
7. [The PowerForms Engine](#)
8. [The PowerForms Translator](#)
9. [PowerForms in Jwig](#)

# Input Field Validation

More detailed **requirements** may be imposed on input fields:

- *the phone number must be 8 digits*
- *only legal e-mail addresses may be written*
- *the password must be at least 7 characters long and not just contain letters*

Sometimes, various **dependencies** between fields are required:

- *the shipping fee varies with the customers country of residence*
- *only married persons must specify their spouse*
- *at most 3 pizza toppings may be chosen*

Often, Web programmers are faced with an apparent choice:

- **client-side** validation checks requirements in the browser (using JavaScript)
- **server-side** validation checks requirements on the server (in Java)

However, **both** checks must really be performed:

- client-side validation gives quick responses, saves bandwidth, and lightens the burden of the server
- server-side validation is required since the client may cheat or just not permit JavaScript

This gives the programmer an unpleasant task:

- validation is tricky to get right
- it must be implemented in two different programming languages

# The PowerForms Language

[PowerForms](#) is an [XML language](#) for **specifying constraints** on input fields in XHTML forms:

- advanced formats for individual fields
- complex interdependencies among several fields

A PowerForms **translator** automatically generates the corresponding code for both the client and the server.

# Cool Examples

Hello World	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
Age	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
Date	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
<a href="#">E-mail</a>	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
<a href="#">URI</a>	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
Letter	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
License	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
Password	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
NYC	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>
Customize	<a href="#">XHTML</a>	<a href="#">PowerForms</a>	<a href="#">Demo</a>

# PowerForms Constraints

PowerForms allows [constraints](#) to be defined for named fields (of every kind).

Note that an field with a given name may occur multiple times.

All constraints must be **satisfied** before the form can be **submitted**.

A constraint is a **decision tree**:

- the nodes are **expressions**
- the leaves are also **expressions**

# PowerForms Expressions

An [expression](#) evaluates to either **true** or **false**.

It decides a **property** of the values of a given input field:

- the number of **occurrences** of the field (upper and lower bounds)
- **equality** to another field
- **inequality** to another field (compared as numbers or strings)
- **matching** a regular expression
- boolean connectives (**and**, **or**, **not**)

# Regular Expressions

PowerForms use [regular expressions](#) to describe allowed formats of input values.

There are several operators:

- the **empty** language
- any **character**
- any **string**
- a **constant** string
- a **set** of characters
- a **range** of characters
- an **interval** of integers
- **repetitions** (possibly with upper and lower bounds)
- the **complement** operator
- the **optional** operator
- the **union** operator
- the **intersection** operator
- the **concatenation** operator
- a **named** regular expression
- a previously compiled **DFA** located by a URL

All relevant formats can be specified by this formalism, such as:

- dates
- phone numbers
- URLs
- e-mail addresses
- ISBN numbers (with checksums even)

# The PowerForms Engine

The PowerForms engine **monitors** an XHTML page while it is being viewed by the client.

It cannot be **submitted**, until all constraints are **satisfied**.

While the form is being filled in, the engine generates dynamic **feedback**:

- invalid `options` in a `select` menu are **filtered** away
- invalid `checkbox` and `radio` buttons are **popped up**
- values for `text` fields are given a **status**:
  - *green light* means that the current value is valid
  - *yellow light* means that the current value is a prefix of a valid value
  - *red light* means that the current value and all extensions are invalid

The corresponding icons may be **customized**.

# The PowerForms Translator

A PowerForms specification and an XHTML document are **translated** into a new XHTML document containing JavaScript code.

The regular expressions are translated into DFAs.

To evaluate all interdependencies correctly, the generated code must perform a fixed-point computation of the constraints whenever the value of some field changes.

The tiny XHTML document:

```
<html>
  <head>
    <title>PowerForms Example - hello</title>
  </head>
  <body>
    <form>
      Please enter "Hello World!": <input type="text" name="t" size="12"/>
    </form>
  </body>
</html>
```

and the PowerForms specification:

```
<powerforms>
  <constraint field="t">
    <const value="Hello World!"/>
  </constraint>
</powerforms>
```

are translated into the following result:

```
<html>
  <head>
    <title>PowerForms Example - hello</title>
  </head>
  <body>
    <script type='text/javascript'
src='http://www.brics.dk/~ricky/powerforms/misc/PowerForms.js'></script>
    <script type='text/javascript'
src='http://www.brics.dk/~ricky/powerforms/misc/Select.js'></script>
    <script type='text/javascript'
src='http://www.brics.dk/~ricky/powerforms/misc/Expression.js'></script>
    <script type='text/javascript'
src='http://www.brics.dk/~ricky/powerforms/misc/Dfa.js'></script>
    <script type='text/javascript'
src='http://www.brics.dk/~ricky/powerforms/misc/Status.js'></script>
    <link href='http://www.brics.dk/~ricky/powerforms/misc/PowerForms.css'
rel='stylesheet' type='text/css' title='PowerForms status stylesheet' />
    <script type='text/javascript'>
      var pwf_event = null;
      var dfa_0 = new Dfa(new Array(new Array(false, 'H', 'H', 11), new Array(false,
'!', '!', 2), new Array(true), new Array(false, 'o', 'o', 8), new Array(false, 'd',
'd', 1), new Array(false, 'l', 'l', 4), new Array(false, 'o', 'o', 9), new
Array(false, 'l', 'l', 12), new Array(false, 'r', 'r', 5), new Array(false, ' ', ' ',
10), new Array(false, 'W', 'W', 3), new Array(false, 'e', 'e', 7), new Array(false,
```

```

'l', 'l', 6)));
dfa_0.filter = function(s) { return dfa_0.run(s) == PowerForms.ACCEPT; }

function pwf_constraint_0() {
  var accept = true;
  var change = false;
  var alert = null;
  var error = null;
  var messages = new Array();
  var state, element;
  element = document.forms[0].elements[0];
  state = dfa_0.run(element.value);
  switch (state) {
    case PowerForms.CRASH:
      messages[messages.length] = 'Invalid input in field t';
      accept = false;
      break;
    case PowerForms.REJECT:
      messages[messages.length] = 'Incomplete input in field t';
      accept = false;
      break;
  }
  pwf_status_0.update(state);
  return Array(accept, change, messages);
}

function pwf_update(functions) {
  var accept = true;
  var change = false;
  var msgs = new Array(functions.length);
  do {
    accept = true;
    change = false;
    for (var i = 0; i != functions.length; i++) {
      var info = functions[i]();
      if (info[1]) {
        change = true;
      }
      if (info[0]) {
        msgs[i] = null;
      } else {
        accept = false;
        msgs[i] = info[2];
      }
    }
  } while (change);
  var messages = new Array();
  for (var i = 0; i != msgs.length; i++) {
    if (msgs[i] != null) {
      for (var j = 0; j != msgs[i].length; j++) {
        messages[messages.length] = msgs[i][j];
      }
    }
  }
  return Array(messages.length == 0, change, messages);
}

function pwf_update_0(e) {
  pwf_event = e;
  return pwf_update(new Array(pwf_constraint_0));
}

```

```

function pwf_update_form_0() {
    return pwf_update(new Array(pwf_constraint_0));
}

function pwf_submit_form_0(name, value, ignoreconstraints) {
    if (document.forms[0].submitted) {
        return false;
    }
    pwf_event = null;
    if (!ignoreconstraints) {
        var info = pwf_update_form_0();
        if (!info[0]) {
            var messages = info[2];
            confirm(messages.length+' error'+(messages.length == 1 ? '' : 's')+ ' on
form:'+'\n'+messages.join('\n'));
            return false;
        }
    }
    if (name != null) {
        document.forms[0].elements[1].name = name;
        document.forms[0].elements[1].value = (value == null ? '' : value);
    }
    document.forms[0].submitted = true;
    document.forms[0].onsubmit = null;
    document.forms[0].submit();
    return true;
}

var pwf_status_0 = null;

function pwf_init() {
    var element;
    pwf_status_0 = new Status(PowerForms.IMAGE, 'pwf-id-1',
'http://www.brics.dk/~ricky/powerforms/misc/na.gif',
'http://www.brics.dk/~ricky/powerforms/misc/red.gif',
'http://www.brics.dk/~ricky/powerforms/misc/yellow.gif',
'http://www.brics.dk/~ricky/powerforms/misc/green.gif');
    element = document.forms[0].elements[0];
    PowerForms.addElementById('pwf-id-0', element);
    if (element.addEventListener) {
        element.addEventListener('onkeyup', pwf_update_0, true);
        element.addEventListener('onchange', pwf_update_0, true);
    } else {
        element.onkeyup = isAcademicBrowser ? function(e) { this.blur();
this.focus(); pwf_update_0; } : pwf_update_0;
        element.onchange = pwf_update_0;
    }
    pwf_update_form_0();
    if (!document.getElementById) {
        document.getElementById = PowerForms.getElementById;
    }
}
</script>

<form onsubmit="return pwf_submit_form_0(null, null, false);">
    Please enter "Hello World!":
    <input id="pwf-id-0" name="t" type="text" size="12" onkeyup='pwf_update_0();'
onchange='pwf_update_0();' /><img id='pwf-id-1' name='pwf-id-1'
src='http://www.brics.dk/~ricky/powerforms/misc/blank.gif' />
    <input type='hidden' name='' />
</form>

```

```
<script type='text/javascript'>
  pwf_init();
</script>
</body>
</html>
```

# PowerForms in Jwig

PowerForms can be run as a [stand-alone](#) tool, but is also integrated into Jwig, as shown in this [example](#):

```
import dk.brics.jwig.runtime.*;
import java.util.*;

public class PowerFreebie extends Service {

    public class HowMany extends Session {

        static final int MAX = 5;

        XML templateAsk = [[ <html>
                            <body>
                                <form>
                                    <[msg]>
                                    <p/>
                                    How many free T-shirts do you want?
                                    <input name="amount" type="text"/>
                                    <input name="continue" type="submit"/>
                                </form>
                            </body>
                        </html>
                    ]];

        XML templateReply = [[ <html>
                                <body>
                                    You will receive <[amount]> k001 T-shirts any day
                                </body>
                            </html>
                    ]];

        XML format = [[ <powerforms>
                        <constraint field="amount">
                            <match>
                                <interval low="1" high=[high]/>
                            </match>
                        </constraint>
                    </powerforms>
                ]];

        public void main() {
            String msg = "";
            int amount;
            boolean ok = false;
            while (!ok) {
                ok = true;
                try {
                    show templateAsk<[msg=msg] powerforms format<[high=MAX];
                } catch (PowerFormsValidateException e) {
                    ok = false;
                    msg = e.getMessage();
                }
            }
        }
    }
}
```

```
    amount = Integer.parseInt(receive amount);  
    exit templateReply<[amount=amount];  
  }  
}  
}
```

Notable points:

- the PowerForms specification is constructed with XML templates
- JWIG can statically check validity with respect to the schema for PowerForms
- this example is immune to dishonest clients

## Recent BRICS Notes Series Publications

- NS-02-1 Anders Møller and Michael I. Schwartzbach. *Interactive Web Services with Java*. April 2002. 99 pp.
- NS-01-8 Anders Møller and Michael I. Schwartzbach. *The XML Revolution (Revised)*. December 2001. 186 pp. This revised and extended report supersedes the earlier BRICS Report NS-00-8.
- NS-01-7 Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '01*, (Aalborg, Denmark, August 25, 2001), August 2001. vi+97 pp.
- NS-01-6 Luca Aceto and Prakash Panangaden, editors. *Preliminary Proceedings of the 8th International Workshop on Expressiveness in Concurrency, EXPRESS '01*, (Aalborg, Denmark, August 20, 2001), August 2001. vi+139 pp.
- NS-01-5 Flavio Corradini and Walter Vogler, editors. *Preliminary Proceedings of the 2nd International Workshop on Models for Time-Critical Systems, MTCS '01*, (Aalborg, Denmark, August 25, 2001), August 2001. vi+ 127pp.
- NS-01-4 Ed Brinksma and Jan Tretmans, editors. *Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '01*, (Aalborg, Denmark, August 25, 2001), August 2001. viii+156 pp.
- NS-01-3 Martin Hofmann, editor. *Proceedings of the 3rd International Workshop on Implicit Computational Complexity, ICC '01*, (Aarhus, Denmark, May 20–21, 2001), May 2001. vi+144 pp.
- NS-01-2 Stephen Brookes and Michael Mislove, editors. *Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01*, (Aarhus, Denmark, May 24–27, 2001), May 2001. viii+279 pp.
- NS-01-1 Nils Klarlund and Anders Møller. *MONA Version 1.4 — User Manual*. January 2001. 83 pp.
- NS-00-8 Anders Møller and Michael I. Schwartzbach. *The XML Revolution*. December 2000. 149 pp.