Interactive Web Services with Java

JSP, Servlets, JWIG, SOAP, WSDL, UDDI

Anders Møller & **Michael I. Schwartzbach**

BRICS, University of Aarhus

http://www.brics.dk/~amoeller/WWW/

First published: April 2002
Latest revision: October 2003

About this tutorial...

This slide collection about Java Web service programming, JSP, Servlets, JWIG, SOAP, WSDL, and UDDI 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.

A PDF version suitable for printing and off-line browsing is available upon request.

See also our tutorial **The XML Revolution - Technologies for the future Web** covering XML and the essential related technologies.

Contents

(See also the tutorial The XML Revolution covering XML and the most essential related technologies, such as Namespaces, XInclude, XML Base, XLink, XPointer, XPath, DTD, XML Schema, DSD, XSLT, XQuery, DOM, SAX, and JDOM.)

Java and WWW

- using Java for Web service development

JSP, Servlets, and JWIG

Three Java-based technologies for making interactive Web services:

- JSP: resembles ASP/PHP
- Servlets: resembles Perl/C/VB CGI scripts
- JWIG: novel high-level language developed at BRICS/DAIMI

Before describing these, we will look into some more general aspects of Java and WWW.

(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, JWIG, 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 2006? Denmark!

Please enter your email address: john.doe@notmail.com

Go!

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="Go!">
</form>
```

The browser collects the reply in a **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

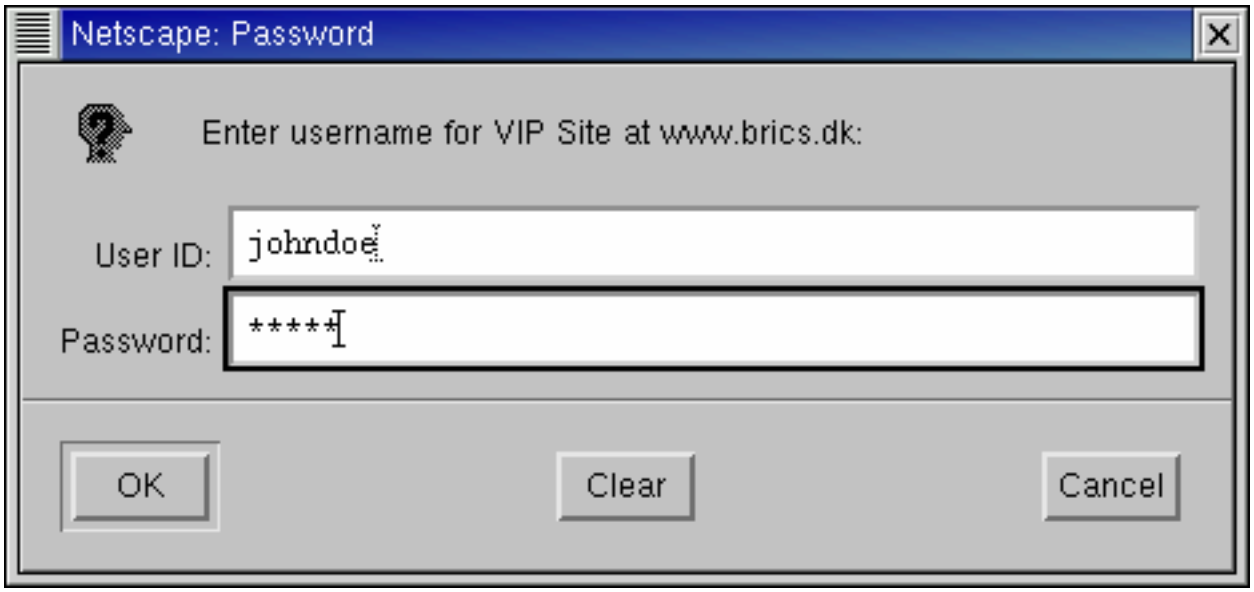
Authentication

Security aspects:

- **authentication** (access restriction)
  - username/password forms (`<input type="password">`)
  - HTTP Basic Autentication
  - 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 QWxhZGRpbjpvcGVuIHNlc2FtZQ==
  ```

  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)

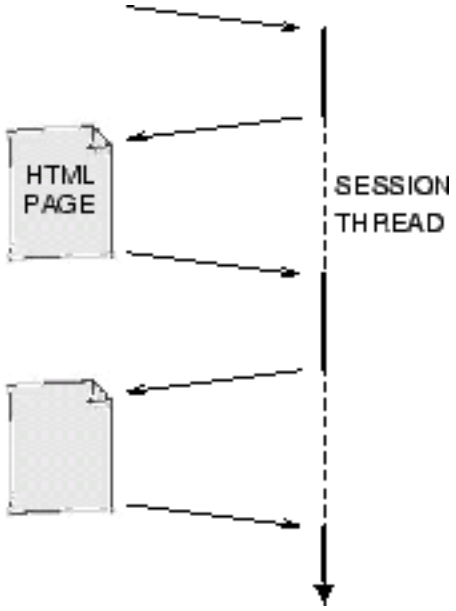
- for Web services, just use the **`https`** protocol in URLs (assuming that a trusted certificate is generated for the server)

- J2SE 1.4 contains **Java Secure Socket Extension (JSSE)** providing full Java support for SSL (`javax.net.ssl`)

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 threat, but perhaps a *privacy* threat (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 JWIG 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

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
- divided into two layers: generic and HTTP-specific

"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, but runs many threads

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

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

Useful predefined methods:

- `void log(String), void log(String,Throwable)`
- `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`...)

Try POST

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

Servers

- **Apache Tomcat 4.1** - 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

Deployment

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

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

(For some reason, servlet class fields are rarely used to share data!?)

Alternatives to `ServletContext`:

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

Sessions

Session state is maintained in an `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...

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

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)

Authentication:

- form-based
- HTTP Basic

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)

- 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>
```

Request Dispatcher

Services are often composed of many Servlets (and JSP pages) working together.

Forwarding a request to another Servlet using `RequestDispatcher`:

```
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/my_other_app/servlet/SomeServlet");
dispatcher.forward(request, response);
```

`request.setAttribute(String,String)` can be used to supply extra data to the new handler

- `forward` - complete transfer of control to another Servlet
- `include` - insert result from running another Servlet (another kind of "wrapping")

JSP - JavaServer Pages

- a Java-version of ASP/PHP

JSP Pages

JSP:

- developed as response to ASP and PHP
- 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 direcly 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>
```

Note that only an expert Java programmer could write this:

- `java.util.Date()` implements `toLocaleString()`
- `synchronized(this) { .. }` is necessary

Inserting code in XML templates:

- 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_inited = 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_inited == false)
```

```
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = 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="k00l">
  <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("k00l")) {
              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

View Sun's JSTL Tutorial.

No Need to Program?

- the tag library approach goes to great length to avoid (blatant) programming.

Part of the philosophy behind JSP(/ASP/PHP) is:

- anyone can write HTML documents
- Web services shouldn't be any harder
- programming is not necessary
- just choose the desired behavior from a tag library

This quickly breaks down:

- new behavior requires ever more tag libraries
- Java details creep in anyway

The tag code:

```
<c:forEach var="item" items="$session:cart.
items">
   ...
   <tr>
      <td align="right" bgcolor="#ffffff">
      <c:expr value="$item.quantity"/>
   </td>
   ...
</c:forEach>
```

abbreviates the JSP code:

```
<%
   Iterator i = cart.getItems().iterator
();
   while (i.hasNext()) {
      ShoppingCartItem item =
         (ShoppingCartItem)i.next();
      ...
%>
      <tr>
      <td align="right"
bgcolor="#ffffff">
      <%=item.getQuantity()%>
      </td>
      ...
<%
   }
%>
```

This only works under a lot of assumptions:

- the `cart` implements a `getItems()` method
- `getItems()` returns an object that implements an `iterator()` method
- a `ShoppingCartItem` implements a `getQuantity()`method

Debugging rapidly becomes a pain...

Model-View-Controller

- a software architecture to structure complex Web applications.

The application is structured in three layers:

- the **model** containing all data and operations
  - implemented as Java classes
- the **views** creating various presentations
  - implemented as JSP pages
- the **controller** receiving requests, updating the model, and delegating to views
  - implemented as a single Servlet

The MVC architecture dates back to Smalltalk-80, where it was used for implementing GUIs.

Struts is a framework for implementing MVC architectures in Java, offering:

- XML configuration files for the controller
- poor man's PowerForms
- poor man's XML templates

JWIG - High-Level Web Services in Java

- a novel approach

JWIG

The JWIG system is another Java-based framework for programming Web applications.

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

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

The JWIG prototype 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**:

- individual interactions are tied together by the `action` attributes in the forms
- the client identity of a session must be stored in cookies, hidden fields, or the URL
- local state must be explicitly saved and restored across all 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 World" 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>
                       <form><input type="submit" /></form>
                       </body></html> ]];
            x = x <[ what = [[ Hello World! ]] ];
            show x;
            XML y = [[ <html><head><title>JWIG</title></head><body>
                       Goodbye!</body></html> ]];
            exit 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:

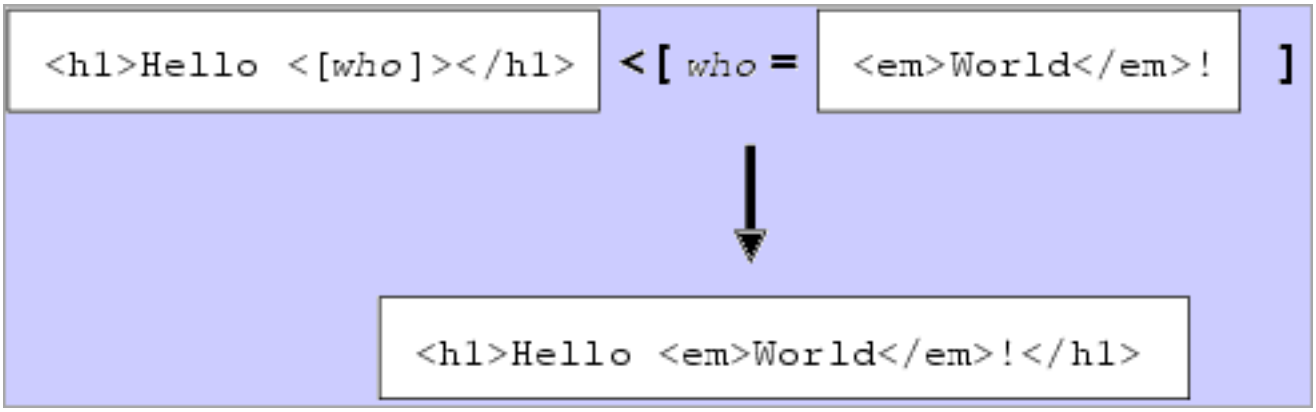
- part of some values are written as constants
- for simple applications, this provides some separation of functionality and presentation

In JWIG, 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 Web service written using Servlets:

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

public class Enter extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet Demo</title></head><body>" +
                    "<form action=\"Hello\">" +
                    "Enter your name: <input name=\"handle\" />" +
                    "<input type=\"submit\" value=\"Continue\" /></form>" +
                    "</body></html>");
    }
}
```

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

public class Hello extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        String name = (String) request.getParameter("handle");
        if (name==null) {
            response.sendError(response.SC_BAD_REQUEST, "Illegal request");
            return;
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet Demo</title></head><body>");
        ServletContext context = getServletContext();
        if (context.getAttribute("users")==null)
            context.setAttribute("users", new Integer(0));
        int users = ((Integer) context.getAttribute("users")).intValue() + 1;
        context.setAttribute("users", new Integer(users));
        HttpSession session = request.getSession(true);
        session.setAttribute("name", name);
        out.println("<form action=\"Goodbye\">" +
                    "Hello " + name + ", you are user number " + users +
                    "<input type=\"submit\" value=\"Continue\" /></form>" +
                    "</body></html>");
```

```
    }
}
```

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

public class Goodbye extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        HttpSession session = request.getSession(false);
        if (session==null) {
            response.sendError(response.SC_BAD_REQUEST, "Illegal request");
            return;
        }
        String name = (String) session.getAttribute("name");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet Demo</title></head><body>" +
                    "Goodbye " + name + "</body></html>");
        session.invalidate();
    }
}
```

- a session in this service consists of a sequence of interactions: `Enter`, `Hello`, `Goodbye`
- the Servlets are tied together by the `action` attributes in the forms
- lots of low-level details...
- no Web-specific static guarantees...

An equivalent program written in JWIG:

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

public class ExampleService extends Service
{
    int counter = 0;
    synchronized int next() { return ++counter; }

    public class ExampleSession extends Session
    {
        XML wrapper = [[ <html><head><title>JWIG Demo</title></head>
                         <body><[body]></body></html> ]];
        XML form =  [[ <form><[contents]>
                       <input type="submit" value="Continue" /></form> ]];
        XML hello = [[ Enter your name: <input name="handle" /> ]];
        XML greeting = [[ Hello <[who]>, you are user number <[count]> ]];
        XML goodbye = [[ Goodbye <[who]> ]];

        public void main() throws IOException
```

```
        {
            XML x = wrapper<[body=form];
            show x<[contents=hello];
            String name = receive handle;
            show x<[contents=greeting<[who=name,count=next()]];
            exit wrapper<[body=goodbye<[who=name]];
        }
    }
}
```

Notable features in the JWIG version:

- 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 recived 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)

The XML Class

Template 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 elaborate **syntactic sugar** for these constructions.

A template constant is written as:

```
[[ <table border="0" cellspacing="0">
     <tr><td align="left">
       <a href=[js]>
         <img border="0" src="http://www.brics.dk/~mis/file.gif"/>
       </a>
     </td></tr>
     <tr><td align="left">
       <[name]>
     </td></tr>
   </table>
  <[rest]> ]]
```

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

Templates are **plugged** by writing, for instance:

```
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 linearly
- well-formedness is easily guaranteed
- escaping of special characters is automatic
- specialized analysis 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 Page and Seslet Classes

- alternatives to the `Session` class.

`Page`:

- only **one client interaction** (no `show` statements, only `exit`)
- resembles Servlets or JSP (together with code gaps)
- allows more efficient implementation (since thread has short life time)

`Seslet`:

- intended for interaction with non-XHTML clients, e.g. **applets**
- input/output is supplied using an `InputStream` and `OutputStream`

Other variants can be imagined, e.g. `CachedPage`, an automatically cached version of `Page`.

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

As mentioned, **syntactic sugar** is provided for much of the functionality.

The JWIG System

Consider again the `ExampleService` example.

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

Compile the source code:

```
jwig compile ExampleService.jwig
```

This creates the following files:

```
ExampleService$ExampleSession.class  ExampleService.class
```

Obtain **static guarantees**:

```
jwig analyze *.class
```

Install the service:

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

Run the service with the URL:

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

The service can be **updated**:

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

Uninstall the service:

```
jwig uninstall /home/mis/jwig-mis/ExampleService
```

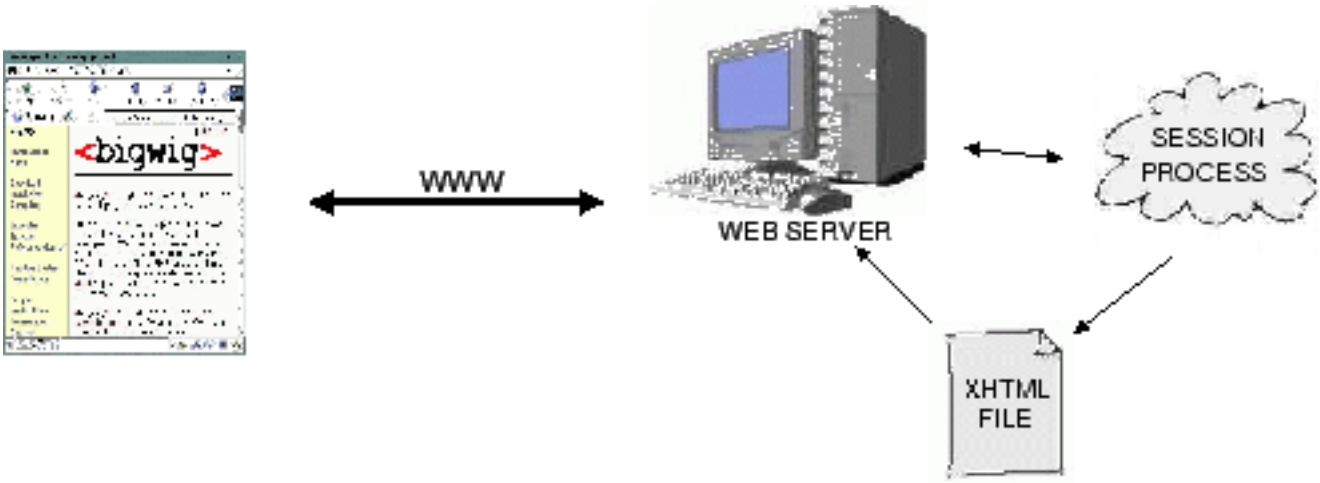
Runtime System

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.

A garbage collector takes care of removing dead session directories.



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

The JAOO Site

The `www.jaoo.dk` site is a conference administration system written in JWIG. It has:

- 4,000 lines of code
- 45 sessions
- almost 200 XHTML templates (259K total)
- complete separation between code and templates

Model-View-Controller

Most Web services are built according to the *Model-View-Controller* pattern:

- the **model** contains the underlying data
- the **view** generates different presentations of these data
- the **controller** handles interactions with clients

Models in JAOO

The model in the JAOO site is a collection of XML documents, each describing people, events, locations, etc.

```
<talk>
  <speaker>AndrewHunt</speaker>
  <speaker>DaveThomas</speaker>
  <title>What is an Agile Method</title>
  <abstract>
    Giga Information Group predicts that 2/3 of all corporate IT will be
    using an Agile method within the next 18 months, and that roughly 25%
    are examining agile processes currently.  What is an Agile method,
    and why are all these companies interested in it?

    Andy Hunt and Dave Thomas are two of the 17 founders of the Agile
    Alliance, whose Agile Manifesto has been extensively written about
    throughout the industry and in mainstream press such as "The Economist".
    In this talk, they will discuss the Agile Principles and Practices,
    the problems that they solve, and give a brief overview of the leading
    Agile methodlogies.
  </abstract>
  <day>Wednesday</day>
  <start hour="11" minute="15"/>
  <stop hour="12" minute="00"/>
  <track>agile</track>
  <room>tools</room>
  <link>http://www.eos.dk/jaooslides/session_AgileMethods.zip</link>
  <linktext>Slides</linktext>
</talk>
```

```
<speaker>
  <name>Andy Hunt</name>
  <company>Independent Consultant</company>
  <photo>http://www.eos.dk/jaoo/2002/20020206/speakers/images/AndyHunt.gif</photo>
  <bio>
    Andy Hunt is an avid woodworker and musician, but curiously, he is more in
    demand as a consultant. He has worked in telecommunications, banking, financial
    services and utilities, as well as more exotic fields such as medical imaging,
    graphic arts, and Internet services. Andy specializes in blending tried-and-
true
    techniques with leading-edge technologies, creating novel -- but practical --
    solutions.
  </bio>
</speaker>
```

Views in JAOO

Views are generated by methods that read XML files using JDOM and stitch XML templates together. For example, the events schedule is generated by:

```
XML genEvents(TreeSet t) {
  XML x = templateScheduleDay;
  String hour1 = "";
  String minute1 = "";
  String hour2 = "";
  String minute2 = "";
  XML y = templateScheduleBlock;
  Iterator i = t.iterator();
  while (i.hasNext()) {
    Element e = (Element)i.next();
    String h1 = e.getChild("start").getAttributeValue("hour");
    String m1 = e.getChild("start").getAttributeValue("minute");
    String h2 = e.getChild("stop").getAttributeValue("hour");
    String m2 = e.getChild("stop").getAttributeValue("minute");
    if (!h1.equals(hour1) || !m1.equals(minute1) || !h2.equals(hour2) || !m2.equals
(minute2)) {
       if (!hour1.equals(""))
         x = x<[events = templateScheduleEvents<[start=genHM(hour1,minute1),
stop=genHM(hour2,minute2),block=y]];
       y = templateScheduleBlock;
       hour1 = h1; minute1 = m1; hour2 = h2; minute2 = m2;
    }
    if (e.getName().equals("talk"))
      y=y<[event=templateScheduleTalk<[
                    track=genColor(e.getChildText("track")),
                    speakers=genEventSpeakers(e),
                    title=e.getChildText("title"),
                    extra=genExtra(e.getChildText("link"),e.getChildText
("linktext")),
                    room=genRoom(e.getChildText("room"))]];
    else
      y=y<[event=templateScheduleBreak<[title=e.getChildText("title")]];
  }
  if (!hour1.equals(""))
    x = x<[events = templateScheduleEvents<[start=genHM(hour1,minute1),stop=genHM
(hour2,minute2),block=y]];
  return x;
}
```

Notice that the code does not contain **any** mark-up tags, but only refers to template constants such as:

```
const XML templateScheduleEvents = [[
<tr>
<td valign="middle" align="center" width="100"><[start]> - <[stop]>  </td>
<td valign="top"><[block]></td>
</tr>
<tr><td colspan="5"><hr></td></tr>
<[events]>
]]
```

Controllers in JAOO

Controllers are simply programmed as sessions. For example, the business cards interactions are implemented as:

```
public class Search extends Session {
  public void main() {

    read registrations.xml;
    show loginPage;
    String id = receive id;
    if (id is not in registrations.xml) exit loginErrorPage;

    read cards.xml;
    Element p;
    if (id is in cards.xml) p = information from cards.xml;
    else p = information from registrations.xml;

    String filter = "";
    while (true) {
      show searchPage<[filter=filter,cards = generate cards using filter];
      if ((receive action).equals("filter")) filter = filter;
      if ((receive action).equals("update")) {
         if ((receive pin) matches p) {
            show updatePage<[fields = values from p];
            update p with received values;
            update cards.xml with p;
         }
      }
    }
  }
}
```

The Template Manager

In JWIG, we can **browse** and **edit** all the XML templates in a template manager.

This is in fact just another JWIG service, `TempMan.jwig`

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:

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

public class TempEx extends Service {
  static XML templateWrapper;
  static XML templateHello;
  static XML templateGoodbye;

  public TempEx() {refresh();}

  static void refresh() {
    try {
      templateWrapper = get "file:/home/mis/TempEx/templates/Wrapper";
      templateHello = get "file:/home/mis/TempEx/templates/Hello";
      templateGoodbye = get "file:/home/mis/TempEx/templates/Goodbye";
    } catch (Exception e) { e.printStackTrace(); }
  }

  public class Refresh extends Page {
    public void main() {
        refresh();
    }
  }

  public class Example extends Session {
    public void main() {
      show templateWrapper<[what=templateHello];
      exit templateGoodbye;
    }
  }
}
```

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
- ...

Most sites in fact generate invalid (X)HTML:

Site	Errors on cover page
www.altavista.com	29
www.cnn.com	58
www.sun.com	19
www.ibm.com	30
www.microsoft.com	123
www.google.com	27
www.w3c.org	0

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

Static Analysis

All **interesting** properties of the behavior 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.

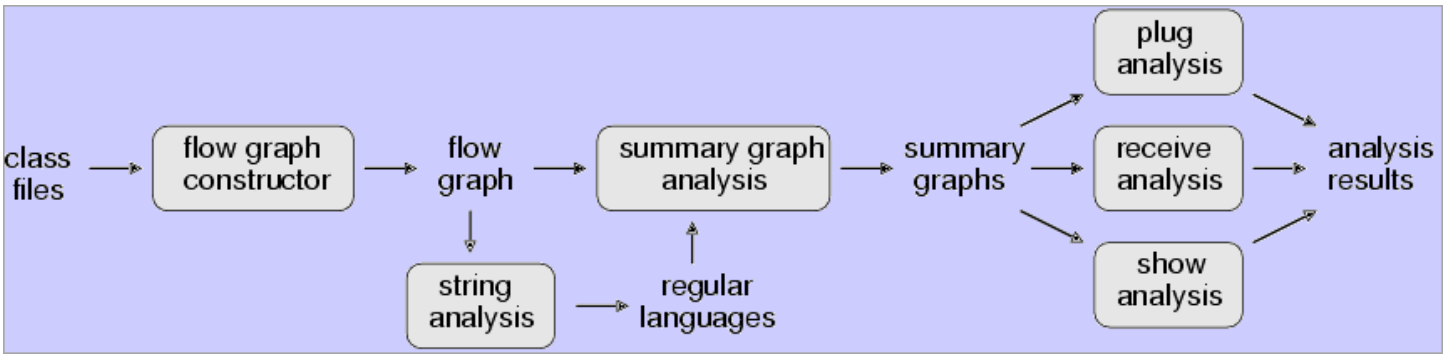
Static analysis is a **standard technique** involving:

- 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:

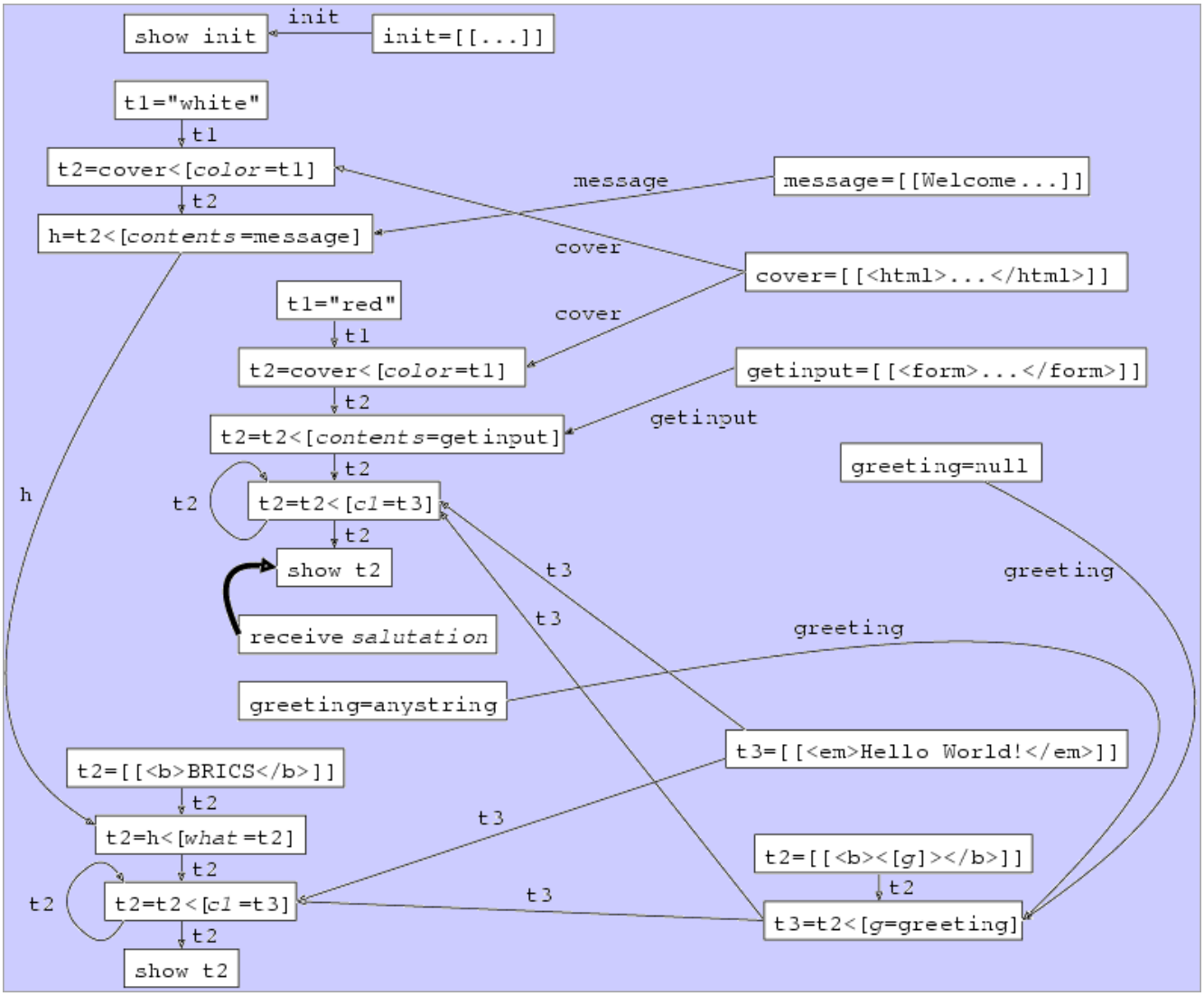
```
import dk.brics.jwig.runtime.*;

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:

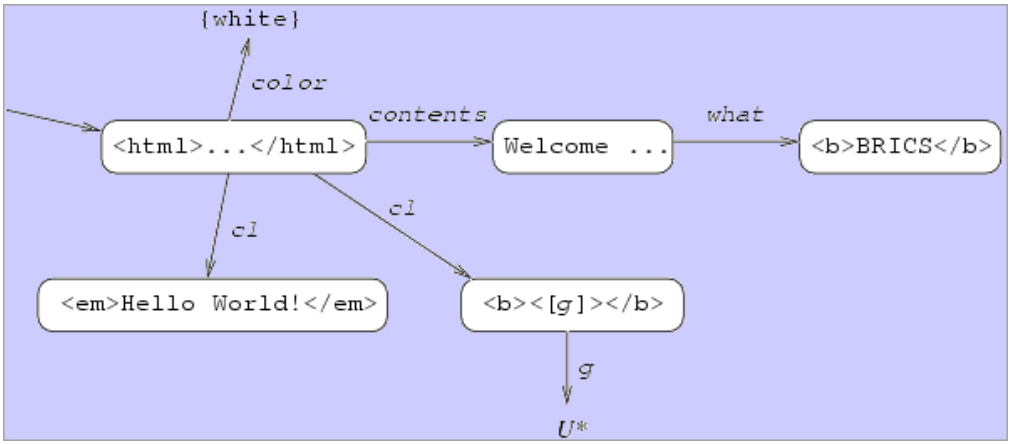
1. single methods
2. code gaps

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 `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 `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 `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	9.7
Guess	94	8	7	11.1
Calendar	133	6	2	10.0
Memory	167	9	6	15.1
TempMan	238	13	3	11.0
WebBoard	766	32	24	13.5
Bachelor	1,078	88	14	131.3
Jaoo	3,923	198	9	39.9

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

(See also the XML schema language tutorial.)

Conclusion

JWIG is a novel alternative to Servlets, JSP, etc.

Through **high-level language design** it provides:

- **explicit sessions**
- **XML templates**
- shared data using standard Java mechanisms
- a powerful API for other Web programming issues

and **specialized program analyses** check at compile-time for

- **XHTML validity**
- matching form fields and **`receive` operations**
- safe use of **plug operations**

More information:

- `www.brics.dk/JWIG` - the JWIG site
- *Extending Java for High-Level Web Service Construction* - a comprehensive article on JWIG
- JWIG User Manual - describing the language and the implementation

The current JWIG implementation and API are initial prototypes - stay tuned for improvements, and contact the JWIG team to contribute to the development!

PowerForms

- declarative input field validation

Input Field Validation

- an inevitable part of interactive Web services.

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

- *the phone number must be 8 digits*
- *only legal e-mail adresses may be written*
- *the password must be at least 5 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, for instance)

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

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

This gives the programmer an unpleasant task:

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

Hacking JavaScript

The **standard solution** is to write lots of JavaScript code to perform the validation:

- Easy introduction
- Form validation tutorial
- Questions to the experts
- Bugs on parade

This is not an ideal solution:

- JavaScript is a **large language** that does much more than validation
- programmers must write the same code **again and again**
- even trivial programs tend to require **testing and debugging**

Immediate reaction of a Computer Scientist:

- **analyze** the problem domain
- create a **domain-specific language**
- construct a **compiler** that will generate JavaScript code automatically

Regular Expressions

Text input fields must generally satisfy **simple formats**:

- dates
- phone numbers
- passwords
- e-mail addresses
- URLs

These can be captured by **regular expressions**, which denote sets of strings:

- `()` is a regular expression denoting the empty set
- `#` is a regular expression denoting the set containing the empty string
- any Unicode character is a regular expressions denoting the singleton set containing that character
- if `R` and `S` are regular expressions, then `R|S` denotes the union of these sets
- if `R` and `S` are regular expressions, then `RS` denotes the concatenation of these sets
- if `R` is a regular expression, then `R*` denotes the set of sequences of strings from that set

As an example, the regular expression `(a|b|c)*de` denotes the strings:

- `de`
- `ade`
- `bde`
- `cde`
- `aade`
- `abde`
- `acde`
- `...`
- `bacaacbbcade`
- `...`
- `bbbbbaaacacabacabaccaaabcacbccabacabacabacabacaaaaacccaaade`
- `...`

In practical use, it is nice to have lots of syntactic sugar.

Validating with Automata

A simple e-mail address can be specified by the regular expression:

- `[a-z0-9]+\@[a-z0-9]+(\.[a-z0-9]+)+`

This can be translated into an automaton and used to generate validation in the browser:

Submit

The status icons mean:

- the text is a legal string
- the text is a proper prefix of a legal string
- every extension of the text is an illegal string

The XHTML document looks like this:

```
<form>
  <input type="text" name="email" size="30"/>
  <p/>
  <input type="submit" name="test" value="Submit"/>
</form>
```

and the validation is specified as follows:

```
<powerforms>
  <constraint field="email">
    <regexp pattern="[a-z0-9]+\@[a-z0-9]+(\.[a-z0-9]+)+"/>
  </constraint>
</powerforms>
```

Interdependencies

Regular expressions and automata can handle all reasonable formats for single text input fields.

But what about complex interdependencies between texts, buttons, and menus?

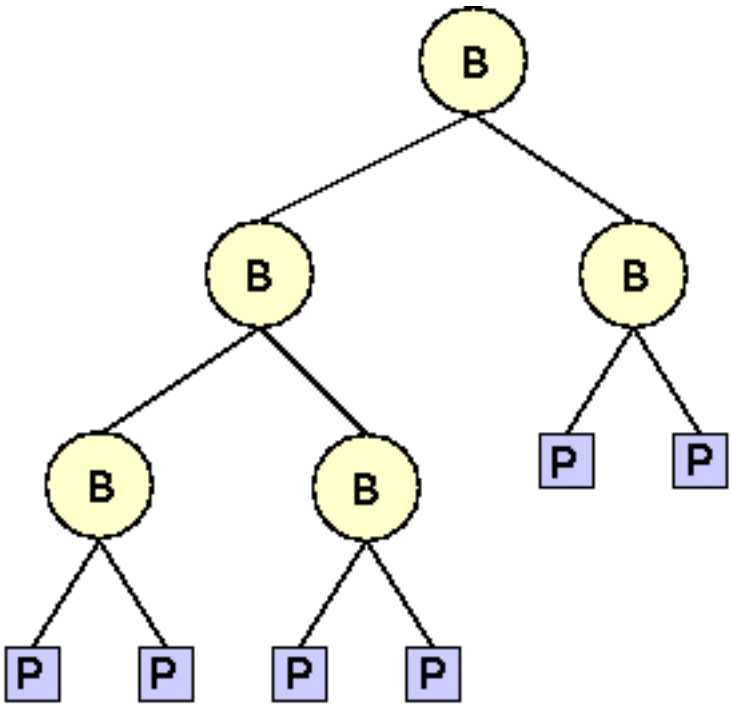
Consider a customer registration form like the following:

Do we need full-scale programming to handle this example?

The Constraint Model

A simple uniform **constraint model** will handle many cases, including the previous example.

Each field is given a binary **decision tree**:



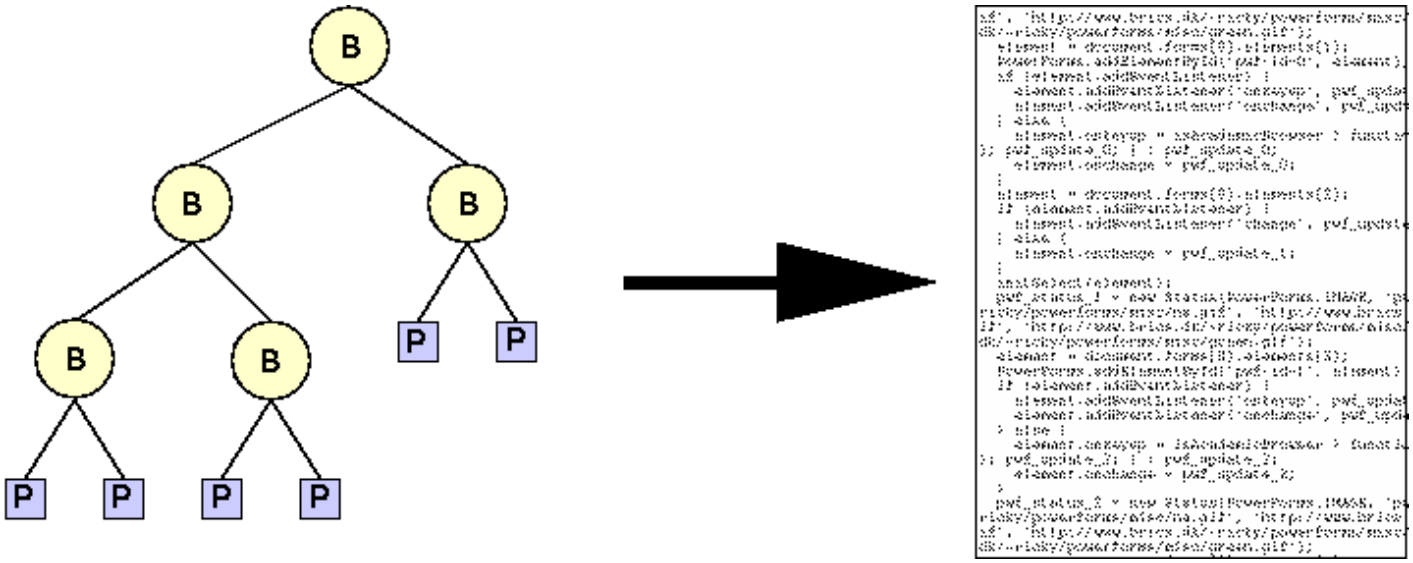
The nodes are boolean expressions combining **predicates** with **and**, **or**, and **not**.

The leaves are **predicates** that are required to hold for the given field.

The Constraint Model

A simple uniform **constraint model** will handle many cases, including the previous example.

Each field is given a binary **decision tree**:



The nodes are boolean expressions combining **predicates** with **and**, **or**, and **not**.

The leaves are **predicates** that are required to hold for the given field.

Constraint will be automatically compiled into the required JavaScript code.

Constraint Syntax

PowerForms constraints are written in XML syntax and look like:

```
<powerforms>
  <constraint field="name">
    body

  </constraint>
  ...
  <constraint field="name">
    body
  </constraint>
</powerforms>
```

A ***body*** is an ***expression***, a ***regexp***, or looks like:

```
<ignore/>
```

or:

```
<if>
  expression
  <then>body</then>
  <else>body</else>
</if>
```

An ***expression*** is a ***predicate*** or looks like one of:

```
<and>expression*</and>
<or>expression*</or>
<not>expression*</not>
```

A ***predicate*** looks like one of:

```
<count min="int max="int"/>
<equal field="name" value="string"/>
<match field="name">regexp</match>
```

A ***regexp*** looks like:

```
<regexp pattern="regexp"/>
<regexp url="URL"/>
```

Applying Constraints

After evaluating all conditions, a final predicate is applied to each field.

Different things happen depending on the kind:

	text	**radio**	**checkbox**	**select**	**textarea**
regexp	status	illegal buttons pop up	illegal buttons pop up	illegal items are filtered	checked at submit
count	checked at submit	checked at submit	checked at submit	checked at submit	checked at submit
equal	checked at submit	checked at submit	checked at submit	checked at submit	checked at submit
ignore	N A status	no effect	no effect	no effect	no effect

These actions take place whenever fields change values.

Evaluating All Constraints

Note that when a constraint is applied to a field, its value may be changed!

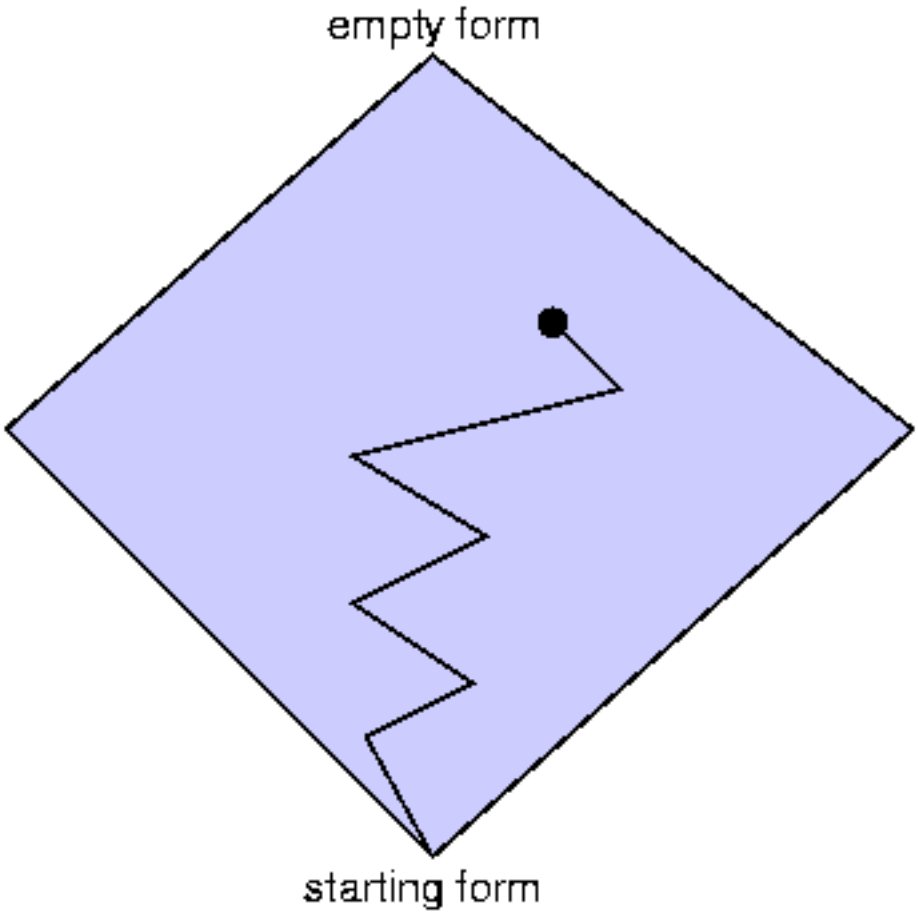
This could potentially lead to problems, since some boolean expressions may consequently change values.

To make sure that the right behavior is obtained, we do the following:

- evaluate all conditions
- apply the resulting constraints
- repeat this, until no more changes take place

How do we know that this doesn't loop or occillate?

Results from lattice theory guarantee that the picture looks as follows:



and the computation always terminates.

Cool Examples

A text field only accepting the constant string `"Hello World!"`:

XHTML	PowerForms	Demo

A text field only accepting numbers between 0 and 100:

XHTML	PowerForms	Demo

Radio buttons filtering the items in a menu:

XHTML	PowerForms	Demo

Radio buttons limiting other radion buttons:

XHTML	PowerForms	Demo

Two text field whose values must be equal:

XHTML	PowerForms	Demo

Customizing the icons:

XHTML	PowerForms	Demo

The NYC example:

XHTML	PowerForms	Demo

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.*;

public class PowerFreebie extends Service {

    public class HowMany extends Session {

        static final int MAX = 5;

        XML templateAsk = [[
            <html><body><form>
              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]> k00l T-shirts any day now...
            </body></html>
        ]];

        XML format = [[
            <powerforms>
              <constraint field="amount">
                <regexp pattern=[pattern]/>
              </constraint>
            </powerforms>
        ]];

        public void main() {
            show templateAsk powerforms format<[pattern="<1-"+MAX+">"];
            int amount = Integer.parseInt(receive amount);
            exit templateReply<[amount=amount];
        }
    }
}
```

Notable points:

- the PowerForms specification is constructed *dynamically* with XML templates

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

Web Services

- application-to-application services using SOAP, WSDL, UDDI

Web Services

A classification of Web services:

- **interactive** - where the client is a human being using a browser
  - developed using Servlets/JSP/JWIG/...

- **application-to-application** - where the client is some program
  - the topic of this tutorial...

Web services combine the **HTTP** communication protocol and the **XML** data format - and thereby get all the benefits from both worlds.

HTTP+XML

We know how to communicate using the HTTP protocol.

We know how to store and manipulate data using the XML format.

- what's more to it?

WSDL, SOAP, UDDI, ...

The Web service platform:

SOAP

- a framework for exchanging XML-based information in a network
- SOAP used to be an acronym: Simple Object Access Protocol
- "This is no longer the case." (it is neither simple nor has anything to do with objects)
- the currently most hyped XML/Web service technology
- mostly just hot air...

WSDL (Web Service Description Language)

- an XML-based language for describing network services
- WSDL descriptions of capabilities and locations of services
- like an interface description language for Web services
- communication using SOAP or direct HTTP

UDDI (Universal Description, Discovery, and Integration)

- provides a registry mechanism for clients and servers to find each other
- uses SOAP for communication

- this presentation will mostly focus on WSDL.

Examples

- Google's Web Service - access the Google search engine
- Amazon's Web Service - access Amazon's product information
- XMethods - collection of information about existing Web services
- SalCentral - WSDL / SOAP Web services search engine

WSDL - Web Service Description Language

WSDL is an XML-based language that allows formal descriptions of the **interfaces of Web services**:

- which **interactions** does the service provide?
- which **arguments and results** are involved in the interactions?
- which network addresses are used to **locate** the service?
- which communication **protocol** should be used?
- which **data formats** are the messages represented in?

So what are the benefits?

- an interface description is a **contract** between the server developers and the client developers
- having **formal** descriptions allows **tool support**, e.g. code template generators
  (see the Apache AXIS project, the alphaWorks Emerging Technologies Toolkit, and CapeClear's WSDL tools)

The Cover Pages collects information about WSDL.

WSDL 1.2 is under development.

A WSDL Example

A WSDL description of a "stock quote" service:

```
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd1="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
         <complexType>
           <all>
             <element name="price" type="float"/>
           </all>
         </complexType>
      </element>
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
```

```
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>

</definitions>
```

- some **data types** are defined using XML Schema
- some simple **message types** are defined from the data types
- the **port type** describes a single **operation** `GetLastTradePrice`, which uses the message types for input/output
- the **binding** tells that the communication is through SOAP
- the **port** associates the binding with the URI `http://example.com/stockquote` where the running service can be accessed

- this is all we need to write clients for the service!

The WSDL Language

The language can be described as having two layers:

- the **service definition layer** describes **abstract** properties:
  - data types
  - message types
  - operations
  - services

- the **binding layer** describes **concrete** properties:
  - protocols
  - data formats

  (using SOAP, HTTP, MIME)

An actual WSDL document consists of a set of `definitions` of the following kinds:

- `types` - containing XML Schema element and type definitions

- `message` - consisting of either
  - a number of named `parts` typed by XML Schema elements, or
  - a single `part` typed by a XML Schema type

- `portType` - describing a set of `operations`, each being either
  - *one-way*: receiving an `input` message,
  - *request-response*: receiving an `input` message and then responding with an `output` message (like Remote Procedure Calls),
  - *solicit-response*: sending an `output` message and then receiving an `input` message, or
  - *notification*: sending an `output` message

- `binding` - selects communication protocol and data formats for each operation and message

- `service` - describes a collection of named `ports`, each associated with a binding and a network address

An `import` mechanism allows modularization of definitions.

(Unfortunately, the binding layer is quite loosely specified...)

WSDL Bindings

- **SOAP**
  - described later...

- **HTTP**
  - selects GET or POST method
  - absolute URI for each port
  - relative URI for each operation
  - optionally, encoding of request message parts (URL encoding, URL replacement)

- **MIME**
  - specifies MIME types for message parts (`text/xml`, `multipart/related`, ...)
  - only describes data formats, needs SOAP/HTTP binding to specify communication protocol

An example using HTTP and MIME bindings (without SOAP):

```
<definitions .... >
  <types>
    ...
  </types>

  <message name="gadgetID">
    <part name="id" type="xsd:string"/>
  </message>

  <message name="gadgetInfo">
    <part name="info" type="tns:GadgetInfo"/>
  </message>

  <message name="gadgetIDandInfo">
    <part name="id" type="xsd:string"/>
    <part name="info" type="tns:GadgetInfo"/>
  </message>

  <message name="status">
    <part name="code" type="xsd:int"/>
  </message>
```

```
<portType name="widgetPortType">
  <operation name="getGadgetInfo">
    <input message="tns:gadgetID"/>
    <output message="tns:gadgetInfo"/>
  </operation>
  <operation name="setGadgetInfo">
    <input message="tns:gadgetIDandInfo"/>
    <output message="tns:status"/>
  </operation>
</portType>

<service name="widgetService">
  <port name="port" binding="tns:binding">
    <http:address location="http://widget.org/"/>
  </port>
</service>

<binding name="binding" type="tns:widgetPortType">
  <http:binding verb="POST"/>
  <operation name="getGadgetInfo">
    <http:operation location="getGadgetInfo"/>
    <input>
      <mime:content type="application/x-www-form-urlencoded"/>
    </input>
    <output>
      <mime:mimeXml/>
    </output>
  </operation>
  <operation name="setGadgetInfo">
    <http:operation location="setGadgetInfo"/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <mime:content part="id" type="text/plain"/>
        </mime:part>
        <mime:part>
          <mime:mimeXml part="info"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
    <output>
      <mime:content type="text/plain"/>
    </output>
  </operation>
```

```
  </binding>

</definitions>
```

- `http://widget.org/getGadgetInfo` is a Web service operation that
  - receives a "gadget ID" string using HTTP POST and
  - send back an XML document of type `GadgetInfoType`
- `http://widget.org/setGadgetInfo`
  - receives a "gadget ID" string and a `GadgetInfoType` document and
  - sends back a "status code" string
- `mime:mimeXml` means: text/xml that conforms to the schema type
- `mime:multipartRelated` aggregates a set of message parts

Problems with WSDL...

- the HTTP and MIME bindings are too loosely specified (so not all WSDL documents "make sense")
  - which extension elements may/must be specified?
  - how are message parts ordered and identified in a response?
  - can XML input be transmitted with `urlReplacement`?
- WSDL is too closely tied with SOAP
  - SOAP is big and complicated without offering much (SOAP is described next)
- *solicit-response* and *notification* are not supported by any existing binding
- more complex interaction patterns cannot be described (e.g. transactions)
- WSDL is too closely tied with XML Schema (and there are other schema languages)
- none of these issues seem to be solved with WSDL 1.2...

- still, the ideas behind WSDL are great :-)

SOAP

"SOAP is fundamentally a stateless, one-way message exchange paradigm for XML-based information"

It's big (500 page spec and still growing):

- Part 1: Messaging Framework
- Part 2: Adjuncts
- Email Binding
- Attachment Feature
- ...

Typical message exchange styles:

- **document-style** (one-way XML messages)
- **RPC** (Remote Procedure Call, request-response)

Typical protocol bindings:

- HTTP (not the same as the HTTP binding for WSDL!)
- SMTP (Simple Mail Transport Protocol)

- there are simpler alternatives to SOAP, e.g. XML-RPC

Using SOAP

Using SOAP typically means:

- when sending messages, put them into a SOAP envelope
- when receiving messages, take them out of the SOAP envelope

An **envelope** is a wrapper containing:

- a **header** - information to intermediaries (network nodes on the message path)
- a **body** - the actual contents (depending on the application)

Example document-style SOAP message (from the SOAP primer):

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
 <env:Header>
    <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
                   env:role="http://www.w3.org/2002/06/soap-envelope/role/next"
         env:mustUnderstand="true">
      <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
      <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
    </m:reservation>
    <n:passenger xmlns:n="http://mycompany.example.com/employees"
                 env:role="http://www.w3.org/2002/06/soap-envelope/role/next"
        env:mustUnderstand="true">
      <n:name>John Q. Public</n:name>
    </n:passenger>
  </env:Header>
 <env:Body>
    <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
      <p:departure>
        <p:departing>New York</p:departing>
        <p:arriving>Los Angeles</p:arriving>
        <p:departureDate>2001-12-14</p:departureDate>
        <p:departureTime>late afternoon</p:departureTime>
        <p:seatPreference>aisle</p:seatPreference>
      </p:departure>
      <p:return>
        <p:departing>Los Angeles</p:departing>
        <p:arriving>New York</p:arriving>
        <p:departureDate>2001-12-20</p:departureDate>
        <p:departureTime>mid-morning</p:departureTime>
        <p:seatPreference/>
      </p:return>
    </p:itinerary>
    <q:lodging xmlns:q="http://travelcompany.example.org/reservation/hotels">
      <q:preference>none</q:preference>
    </q:lodging>
  </env:Body>
```

</env:Envelope>

- boldface parts are SOAP data, the rest is application-specific
- the `next` role means that the header blocks are targeted at the next SOAP node encountered en route
- `mustUnderstand="true"` means that the header blocks cannot be ignored

The big question: What value does SOAP add?

Using SOAP in WSDL

The SOAP binding in WSDL:

- selects **`document`** or **`rpc`** style (`rpc` wraps message parts)
- selects **HTTP/SMTP**/... protocol
- selects **encoding** (typically, the "SOAP encoding")
- places message parts in **header** or **body** parts of the envelope

Example: Google

The Google Web APIs service package contains

- **GoogleSearch.wsdl** - the WSDL description
- `googleapi.jar` - a client API library (no source, unfortunately)

and provides three operations for the Google database:

- `doGoogleSearch` (example request/response - note the useless wrapper and type attributes)
- `doGetCachedPage`
- `doSpellingSuggestion`

(register to get a licence key, allows 1000 queries per day)

- see also the XML.com article

Buzzword compliance: the Google Web service uses SOAP without any benefits compared to using straight XML+HTTP.

Web Services with Java

The most essential Java tools for Web service development:

- xml.apache.org/axis - Apache **AXIS** (Apache Extensible Interaction System)
  - a Java-based implementation of SOAP+WSDL
  - largely allows the programmer to forget these technologies
  - typically used together with Tomcat
  - highly recommended!

- www.alphaworks.ibm.com/tech/ettk - alphaWorks's **EETK** (Emerging Technologies Toolkit)
  - support for SOAP, WSDL, UDDI and much more...

- java.sun.com/webservices/downloads/webservicespack.html - Sun's **Java WSDP** (Web Services Developer Pack)
  - support for SOAP, WSDL, UDDI, ...
  - JAX-RPC maps SOAP/WSDL to RMI (Java Remote Method Invocations)

Example: Amazon

Amazon's Web Service

Functionality:

- search for product information
- shopping carts

Two interfaces:

- **SOAP** - described in WSDL: `AmazonWebServices.wsdl`
- **XML over HTTP** (also called REST) - a simple, straightforward, and more widely used alternative!

An example "XML over HTTP" operation: *Author Search*

- *Request format:*
  `http://xml.amazon.com/onca/xml2?t=webservices-20&dev-t=`[developer's ID goes here]`&AuthorSearch=`[author name goes here]`&mode=`[product line goes here: must be books]`&type=`[lite or heavy]`&page=`[page # goes here]`&sort=`[sort type goes here (optional)]`&f=xml`
- *Response:*
  The AuthorSearch request returns a ProductInfo node. The ProductInfo node contains an array of Detail nodes. (These nodes are XML elements described by a schema)

The example Java clients build on Apache AXIS.

UDDI - Universal Description, Discovery, and Integration

- a step higher than WSDL in the protocol stack

UDDI (400 page spec): a Web service **registry** mechanism
- "a **meta service** for **locating** Web services by enabling robust queries against rich metadata"

- UDDI **business registration**: XML files used to describe business entities and their Web services
  - *white pages* - business address, contact info, etc.
  - *yellow pages* - industrial categorizations based on standard taxonomies (allows search within particular industry, product category, geographical region, ...)
  - *green pages* - more technical information, for instance **WSDL** descriptions

- UDDI also provides a SOAP+WSDL-based registry API for **registering** ("publish") and **discovering** ("inquire") Web services.

Some tools:

- a cool UDDI browser (try searching for "amazon" in the XMethods registry)
- UDDI4J - open-source Java implementation of UDDI (included in WSTK)
- Microsoft's free UDDI registry

UDDI Examples

A UDDI **businessEntity** describes a business and its services:

```
<businessEntity businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
                operator="www.ibm.com/services/uddi"
                authorizedName="0100001QS1">
  <discoveryURLs>
    <discoveryURL useType="businessEntity">http://www.ibm.com/services/uddi/uddiget?
businessKey=BA744ED0-3AAF-11D5-80DC-002035229C64</discoveryURL>
  </discoveryURLs>
  <name>XMethods</name>
  <description xml:lang="en">Web services resource site</description>
  <contacts>
    <contact useType="Founder">
      <personName>Tony Hong</personName>
      <phone useType="Founder" />
      <email useType="Founder">thong@xmethods.net</email>
    </contact>
  </contacts>
  <businessServices>
    <businessService serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
                     businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Delayed Stock Quotes</name>
      <description xml:lang="en">20-minute delayed stock quotes</description>
      <bindingTemplates>
        <bindingTemplate bindingKey="d594a970-3e16-11d5-98bf-002035229c64"
                         serviceKey="d5921160-3e16-11d5-98bf-002035229c64">
          <description xml:lang="en">SOAP binding for delayed stock quotes service</
description>
          <accessPoint URLType="http">http://services.xmethods.net:80/soap</
accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo tModelKey="uuid:0e727db0-3e14-11d5-98bf-
002035229c64" />
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
```

- Key attributes locate various data structures
- a `businessService` describes a particular service (or family of services)
- a `bindingTemplate` describes where and how a service is accessed
- a `tModel` describes compliance with a specification (e.g. a WSDL description)

This **tModel** for the single service described above refers to bindings in a WSDL description:

```
<tModel tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
        operator="www.ibm.com/services/uddi"
        authorizedName="0100001QS1">
  <name>XMethods Simple Stock Quote</name>
  <description xml:lang="en">Simple stock quote interface</description>
  <overviewDoc>
    <description xml:lang="en">wsdl link</description>
    <overviewURL>http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl</overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
                    keyName="uddi-org:types"
                    keyValue="wsdlSpec" />
  </categoryBag>
</tModel>
```

- the WSDL document has no `service` part - the service address is specified in the `bindingTemplate`
- UDDI is much more general (and more complicated) than these examples suggest...

The following SOAP message could be sent to a UDDI registry to inquire about services named "delayed stock quotes":

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_service businessKey="*" generic="1.0" xmlns="urn:uddi-org:api">
      <name>delayed stock quotes</name>
    </find_service>
  </Body>
</Envelope>
```

- alternatively, we could search by category codes of various kinds
- the `find_service` meta-service operation is specified in the WSDL for the UDDI API
- of course, the UDDI registries are also registered services :-)

Conclusion

Constructing application-to-application Web services is simple, in principle.

SOAP, WSDL, and UDDI are

- unavoidable frameworks for XML communication and service description
- surprisingly complicated and their benefits are not always obvious, although the basic ideas behind them are great
- still under development...

Links

Selected links:

webservices.xml.com

XML.com's Web service section

webservices.xml.com/pub/a/ws/2002/02/12/webservicefaqs.html

"Top Ten FAQs for Web Services"

www.w3.org/2002/ws

W3C's Web Services activity

ws.apache.org

Apache's Web Service project

www.onjava.com/pub/a/onjava/excerpt/java_xml_2_ch2

chapter on SOAP from the book "Java and XML"

wsindex.org

Web service links and resources