JWIG—High-Level Web Services in Java
Interactive Web Applications and Services with Java and XML

Lecture 6

Troels C. Damgaard
Department of Theoretical Computer Science (ITU)

4th of October / IWSJ Fall 2005
INTRODUCTION
- JWIG Overview
- Motivating the JWIG Project

JWIG
- A (Slightly) Larger Webservice
- The Components of JWIG

COMPARING JSP/SERVLETS WITH JWIG
- Need for Type-safety in Web-applications?

(NOT COVERED IN THIS LECTURE) POWERFORMS

EXERCISES

RUNNING JWIG

LITERATURE / LINKS
INTRODUCTION

- JWIG Overview
- Motivating the JWIG Project

JWIG

- A (Slightly) Larger Webservice
- The Components of JWIG

Comparing JSP/Servlets with JWIG

- Need for Type-safety in Web-applications?

(Not Covered in This Lecture) PowerForms

Exercises

Running JWIG

Literature / links
Today:

We continue our survey of the technologies developed by Michael Schwartzbach and coworkers for developing safer web applications.

- Last lecture: XACT — an XML technology for type-safe XML programming (i.e. static guarantees of output validity)
- Today: JWIG — a Java technology (building on top of servlets) aiming to improve (in particular) session handling
  (— employs parts of the technology that later became XACT).

Historically, JWIG (and the project that JWIG sprang from <bigwig>) predates XACT.
Important point:

- We are presenting XACT and JWIG to you as examples of how higher-level language design and research may aid web service development;
- – not necessarily, because we think they are the coolest things since sliced bread...
- They show, however, quite a lot of ideas that will probably be provided by next-generation web services architectures (building on Java or not)
- Further, they have fully functional prototype implementations – which allows us to experiment; even though it might take a bit more effort than installing Java SDK or Microsoft Visual Studio.
THE JWIG SYSTEM

In one line:
Another Java-based framework for programming Web applications.

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

- a stronger session concept
- XML templates as first-class values (now XACT)
- shared state is accessed through usual scope mechanisms
- static guarantees about the behavior of running services
There is a JWIG prototype implemented by:

- providing a set of **Java packages** `dk.brices.jwig`
- **extending** the Java syntax using a desugarer as preprocessor
- supplying a special **module for the Apache server**
Recall, how sessions are represented *implicitly* in Servlets/JSP:

- individual interactions are tied together by *action* attributes in the forms
- identity of client in of a session is stored in cookies, hidden fields, or the URL
- local state must be explicitly saved and restored across all interactions with the client
In JWIG, session concept is explicit:

- a `Session` is written as a `sequential` program (like an ordinary Java method)
- interactions with the client are similar to `remote method invocations`
- `local state` is automatically the full state of the thread running the `Session`

Before we look more closely on `state` in JWIG, let’s see a Hello World service 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>
                      <form><input type="submit" /></form>
                   </body></html> ]];
            x = x <[ what = [[ Hello Wonderful World! ]] ];
            show x;
            XML y = [[ <html><head><title>JWIG</title></head><body>
                      Goodbye Cruel World!</body></html> ]];
            exit y;
        }
    }
}

Run Hello World
Note:

- *show* and *exit* statements mark a transfer of control to the client.
- *action* attribute of *form* is missing; a default one is inserted by JWIG to point to the session thread on the server.
- XHTML is produced by plugging XML templates as in XACT (here just with a bit of syntactic sugar)
This table illustrated, how state is stored in servlets:

<table>
<thead>
<tr>
<th>State type</th>
<th>… in Servlets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>ContextObject attributes</td>
</tr>
<tr>
<td>Shared (appl.)</td>
<td>ContextObject attributes(*)</td>
</tr>
<tr>
<td>Session</td>
<td>HttpSessionObject attributes</td>
</tr>
<tr>
<td>Transient</td>
<td>Local vars in do-methods</td>
</tr>
</tbody>
</table>

(*): Or servlet class fields
The corresponding one for JSP looked (not surprisingly) quite similar:

**State scope in JSP**

<table>
<thead>
<tr>
<th>State type</th>
<th>...in JSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>ContextObject attributes</td>
</tr>
<tr>
<td>Shared (appl.)</td>
<td>ContextObject attributes&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>Session</td>
<td>HttpSessionObject attributes</td>
</tr>
<tr>
<td>Transient (&lt;em&gt;page&lt;/em&gt;)</td>
<td>PageContext attributes&lt;sup&gt;(2)&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<sup>(1)</sup>: or class variables – declared in JSP declarations

<sup>(2)</sup>: or local variables – declared in JSP statements
In **JWIG** the corresponding diagram would look like this:

### State Scope in JWIG

<table>
<thead>
<tr>
<th>State type</th>
<th>...in JWIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>public fields of Service class</td>
</tr>
<tr>
<td>Shared (appl.)</td>
<td>fields of Service class</td>
</tr>
<tr>
<td>Session</td>
<td>fields of Session inner class</td>
</tr>
<tr>
<td>Transient (page)</td>
<td>local variables methods in Session</td>
</tr>
</tbody>
</table>
In JWIG we use the **standard scoping mechanisms** for representing state.

Of course, that means that we have to handle e.g. **synchronization** ourselves;

But as we saw earlier (*in the JSP lecture*), I claim, that we typically need to handle this anyhow.
In Servlets, XML values are **implicit**:  
- values appear as Java `String` output from `print` statements  
- 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 – provides some separation of functionality and presentation
In JWIG (as in XACT) XML values are explicit:

- they are all instances of an XML class
- they are first-class values, just like String values

In JWIG only the constructive operations on XML templates that we saw in XACT are implemented.

In short, that means that we have:

- Instances of XML that are templates: XML fragments containing named gaps.
- Gaps may at runtime be plugged with other templates or strings:
We saw last time how plugging a template works.

Only thing to note is, that in JWIG we typically use the syntactic sugar to express plugging:
1 Introduction
   - JWIG Overview
   - Motivating the JWIG Project

2 JWIG
   - A (Slightly) Larger Webservice
   - The Components of JWIG

3 Comparing JSP/Servlets with JWIG
   - Need for Type-safety in Web-applications?

4 (Not Covered in This Lecture) PowerForms

5 Exercises

6 Running JWIG

7 Literature / links
Let’s compare a slightly larger Hello World web service written as a servlet and as a JWIG service:

```java
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>";
    }
}
```
**HELLO SERVLET**

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>";
    }
}
**GOODBYE SERVLET**

```java
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 (Slightly) Larger Webservice (cont.)

- a session in this service consists of a sequence of interactions: Enter, Hello, Goodbye
- the Servlets are coupled together by the action attributes in the forms
- this is fragile:
  - we must explicitly encode (allowed) page flow
  - we must explicitly check that
  - no compile-time checks that request parameter names match submitted form input parameters
- no compile-time checks that valid XHTML is produced
- quite a lot of low-level detail...
A (Slightly) Larger Webservice (cont.)

An equivalent service written in JWIG looks like this:

**Hello Goodbye Service in JWIG**

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

public class ExampleService extends Service {
    int counter = 0; // shared state
    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; // local (page) state
            show x<[contents=greeting<[who=name, count=next()]>];
            exit wrapper<[body=goodbye<[who=name]>];
        }
    }
}
```

Run Example Service in JWIG
A (Slightly) Larger Webservice (cont.)

Points to Note about the JWIG Implementation

- 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 variable is shared state.
- Name is local (or page) state.
- Interactions are performed by the show or exit statements.
- Form fields are received by the receive expression.
- XML value plugging is expressed by `<[ ... ]>`.

You can see some more demos at [http://www.brics.dk/JWIG/demo.html](http://www.brics.dk/JWIG/demo.html) (when they fix the freewig.brics.dk server, which appears to be down at the moment...).
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
A session is specified as a subclass of the `Session` class.

A `Session` object corresponds to a 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
  - **access control**, using HTTP Authentication
THE XML CLASS

...much as we saw it for XACT, but

- without the deconstructive operations,
- and supported by syntactic sugar (also in XACT actually)

A TEMPLATE CONSTANT

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

- where js, name, and rest are gaps.
Plug templates by writing, for instance:

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

...
This is different from the approach e.g. in JDOM, in that:

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

- and different from Servlets in that:
  - documents need not be constructed **linearly**
  - **well-formedness** is easily guaranteed
  - **escaping** of special characters is automatic
  - analysis is actually feasible
JWIG can emulate JSP “page-centered” style with embedded code using code gaps and a special case of the Session class called Page:

```java
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(); }}
                You are visitor number {{ return next(); }}
            </body></html> ]];
        }
    }
}
```

(Corresponds to JSP example seen in lecture 3.)

- 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)

Note:

- efficiency, because of threads is mainly a problem, because JWIG implementation builds on J2SE.
- future work – to integrate JWIG with a J2EE server

Seslet

- intended for interaction with non-XHTML clients, e.g. applets
- input/output is supplied using an InputStream and OutputStream
The JWIG classes have a fully documented API.

Available packages in JWIG

- `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

Most functionality is available through syntactic sugar or is provided implicitly (like desugaring).
A fully functional JWIG prototype implementation is available.

At runtime, each session is allocated:

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

The index.html page always contains the most recent response shown to the client.

A garbage collector takes care of removing dead session directories.
This is quite different from the JSP/Servlet solutions:

- session id is not encoded (using cookies, URL rewriting, or hidden fields)
- the URL functions as an identity of the session
- hence, sessions can be bookmarked (suspended and later resumed)
- the history buffer of the browser is not filled with references to obsolete requests
JWIG is a session-centered alternative to Servlets, JSP, etc.

JWIG provides:

- explicit sessions
- XML templates as first-class values
- shared data using standard Java mechanisms
- API for other Web programming issues

- specialized program analyses check at compile-time
  - XHTML validity
  - matching form fields and receive operations
  - safe use of plug operations
OUTLINE

1. Introduction
   - JWIG Overview
   - Motivating the JWIG Project

2. JWIG
   - A (Slightly) Larger Webservice
   - The Components of JWIG

3. COMPARING JSP/SERVLETS WITH JWIG
   - Need for Type-safety in Web-applications?

4. (Not Covered in This Lecture) PowerForms

5. Exercises

6. Running JWIG

7. Literature / links
In JSP, Servlets, and similar technologies interaction with the client can be risky.

Interaction with clients is normally completely unchecked:
- client may receive invalid XML documents (e.g. XHTML, WML)
- 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
  ...

The Web service may fail in either of these cases.
In JWIG, client interaction is viewed abstractly as a remote method invocation:
- any 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 received are as expected
Many websites in fact generate invalid (X)HTML (or whatever they claim to generate):

Let’s check: Validator at w3.org

...AND SO WHAT?

- It should be obvious, that this can be at the least annoying (read: when your not using a very forgiving browser like IE).
- But for some Web Services such as, application-to-application services it might make the service
  - malfunction
  - or introduce errors

in some occasions.
(Which BTW is worse than performing badly consistently!)
Claim: Using JSP and Servlets there is a tendency that programmers and graphical designers fight for control over the web appl.

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.

To some extent technologies such as tags and frameworks such as Struts alleviate this.
Claim: The central JWIG template approach – supported by the static analysis is actually a step in the right direction for expliciting contracts between designers and programmers.

E.g.: “There are these 33 templates with these specific (named) gaps and (named) input fields”

Within these constraints, programmers and designers can work independently. The static analysis helps in checking at compile-time that the contracts have been upheld.
1. **Introduction**
   - JWIG Overview
   - Motivating the JWIG Project

2. **JWIG**
   - A (Slightly) Larger Webservice
   - The Components of JWIG

3. **Comparing JSP/Servlets with JWIG**
   - Need for Type-safety in Web-applications?

4. **(Not Covered in This Lecture) POWERFORMS**

5. **Exercises**

6. **Running JWIG**

7. **Literature / links**
We won’t have time to cover **Powerforms** which is a `<bigwig>` technology which is integrated into JWIG.

**In brief:**

**POWERFORMS**

- is a technology for declarative input field validation.
- allows us to extending HTML forms with **declaratively** specified input validation.
- validation requirements are specified using XML and translates into:
  - client-side validation checks requirements in the browser (using JavaScript)
  - server-side validation checks requirements on the server (in Java)
Integrated into JWIG using:

```java
show template powerforms powerforms-template statement.
```

**See:**

http://www.brics.dk/bigwig/powerforms/

and

http://www.brics.dk/~amoeller/WWW/powerforms/examples.html

for interactive tutorials and demos!
OUTLINE

1 Introduction
   - JWIG Overview
   - Motivating the JWIG Project

2 JWIG
   - A (Slightly) Larger Webservice
   - The Components of JWIG

3 Comparing JSP/Servlets with JWIG
   - Need for Type-safety in Web-applications?

4 (Not Covered in This Lecture) PowerForms

5 EXERCISES

6 Running JWIG

7 Literature / links
EXERCISE 6

If you have not succeeded in installing XACT and/or doing the XACT exercises last week, I suggest you look at them again:


Jens Lærkedal found a syntax error in the batch-file that I provided for updating the CLASSPATH, that might have prevented you from getting XACT to run. He kindly provided a substitute script, and I’ve updated the XACT HowTo webpage accordingly.

Any other general problems?

For JWIG I suggest you focus your attention on the exercises 9.7 and 9.11 in the course compendium.

(Next slide : How to compile and run JWIG programs.)
Introduction
- JWIG Overview
- Motivating the JWIG Project

JWIG
- A (Slightly) Larger Webservice
- The Components of JWIG

Comparing JSP/Servlets with JWIG
- Need for Type-safety in Web-applications?

(Not Covered in This Lecture) PowerForms

Exercises

Running JWIG

Literature / links
The JWIG website contains source code together with an installation guide for Linux. Installing under Windows is not currently supported as the installation requires recompiling a module into the Apache Web Server.

To allow you to compile and run JWIG programs nonetheless, a JWIG server bigwig.itu.dk has been set up on ITU. This will allow you to experiment with JWIG. The server is only accessible via ssh.

Henning wrote a note last year on how to compile and install your programs on bigwig.itu.dk, and according to my testing, it should work without problems (famous last words...):

- HowTo on using JWIG at ITU
- The JWIG server is running at http://bigwig.itu.dk
1 Introduction
   • JWIG Overview
   • Motivating the JWIG Project

2 JWIG
   • A (Slightly) Larger Webservice
   • The Components of JWIG

3 Comparing JSP/Servlets with JWIG
   • Need for Type-safety in Web-applications?

4 (Not Covered in This Lecture) PowerForms

5 Exercises

6 Running JWIG

7 LITERATURE / LINKS
These slides are an adapted version of the course slides for last years course. They are available as part of the recommendable online tutorial “Interactive Web Services with Java” at http://www.brics.dk/~amoeller/WWW/index.html.

More information - including a prototype implementation of JWIG:
http://www.brics.dk/JWIG/

Main publication –

▶ “Extending Java for High-Level Web Service Construction” by Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach

– gives a thorough overview of and motivation for JWIG and a critique of the servlet and JSP approach to web services.

Sections 1 and 2 focus mainly on these comparisons and the JWIG language, and are recomended reading. Sections 3, 4, and 5 delve into the static analysis that is a core part of JWIG, while section 6 focuses on the implementation.

Also see, the JWIG user manual at http://www.brics.dk/JWIG/manual/