JWIG User Manual

Anders Møller & Mathias Schwarz
{amoeller,schwarz}@cs.au.dk

October 4 2012

JWIG User Manual

Contents

1 What is JWIG?

JWIG is a Java-based web application programming system. JWIG is being developed to explore high-level language design and program analysis support for web programming.

Highlights of JWIG 2.0:

- The main part of a JWIG application runs on the server, although JWIG also uses a basic JavaScript library for generic client-side functionality.
- JWIG uses the Xact system for flexible and safe dynamic construction of XML (typically XHTML) data.
- The JWIG runtime system provides a simple REST/HTTP-friendly mapping from requests to code and tight coupling between code that generates forms and code that receives form field values.
- JWIG helps managing session state, caching, XHTML events, push-based page updates, and post-redirect-get.
- The JWIG program analyzer checks a range of correctness properties at compile time, for example ensuring that all XML data being generated is valid XHTML 1.0.

This document describes the design and capabilities of JWIG and is intended to make it possible for others to play with the system. JWIG has many interesting connections to other web programming frameworks; however those connections are *not* the focus of this document. The JWIG 2.0 implementation builds upon Tomcat and other widely used Java tools, such as Ant, Hibernate, log4j, and JavaMail. The version of JWIG described here supersedes all previous releases of JWIG and is not designed to be directly backward compatible (although the main ideas from previous releases are still present). For more information about the design of JWIG, see the paper *JWIG: Yet Another Framework for Maintainable and Secure Web Applications*.

The javadoc **JWIG API** documentation supplements this user manual by describing more details of all the classes and method in the JWIG API.

2 Hello World!

For instructions on how to **install** JWIG, go to the JWIG web page at
`http://www.brics.dk/JWIG/`

and download either the kickstart file (which contains a basic JWIG application) or the full JWIG source package (open source, LGPL license), and then read the `INSTALL` file.

We naturally begin with the JWIG variant of the ubiquitous *Hello World* program:

`Main.xact` - a tiny JWIG program

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

public class Main extends WebApp {

  public XML hello() {
    return [[
      <html>
        <head><title>JWIG</title></head>
        <body>Hello World</body>
      </html>
    ]];
  }
}
```

(In the following examples, we omit the `import` declarations.) Compile and deploy, for example with `ant compile` and `ant deploy` using the Ant build file from `kickstart.zip`. Note that this source file contains Xact syntax, so it must be compiled with the Xact compiler (see Section 4). Then direct your browser to

`http://`*`HOST`*`:`*`PORT`*`/`*`TOMCAT-APP`*`/hello`

where *`HOST`*, *`PORT`*, and *`TOMCAT-APP`* should be be replaced by the name and port of your web server and the Tomcat application name, according to the configuration in `jwig.properties` and `build.xml`. You should now see the `Hello World` message. This tiny example application demonstrates a number of basic features in JWIG, as explained in the following sections.

3 Web apps, web methods, and web sites

Any program written in JWIG has a class named `Main` (default package). In the example above, `Main` extends `WebApp` and is called a **web app**. When the JWIG runtime system starts, a single instance of the web app is created.

Each public method in the web app class is a **web method** that can be accessed via a HTTP GET request, as in the Hello World example above. Since web methods are associated to GET requests, they should be *safe* and *idempotent*, as required by the HTTP specification. Web methods generate values that are returned to the client. Depending on the return type of the web method, the behavior of the web method may be to send an XHTML page (Section 4.1) to the client, to redirect a client to another page (Section 4.2) or to send plain text (Section 4.3).

The `Main` class can alternatively extend `WebSite`, which is used for a **web site** that consists of *multiple* web apps. Example:

`Main.java` - a web site consisting of two web apps

```java
public class Main extends WebSite {

  public void init() {
    add(new examples.GuessingGame());
    add(new examples.QuickPoll());
  }
}
```

When a web site is started, its `init` method is invoked. In this method, each web app is created and added to the web site using the `add` method. Web methods are, using the default configuration, invoked as

`http://`*`HOST`*`:`*`PORT`*`/`*`TOMCAT-APP`*`/`*`WEBAPP-CLASS`*`/`*`WEBMETHOD`*

where ***`HOST`***, ***`PORT`***, and ***`TOMCAT-APP`*** are as before. ***`WEBAPP-CLASS`*** is the web app class name with '/' instead of dots (e.g. `examples/GuessingGame` in the example above), and ***`WEBMETHOD`*** is the web method name. The mapping from URLs to web methods can be configured as explained in Section 5.

4 Generating responses

Web methods can return values of various types, which affects the behavior of the clients. We now describe the basic return types and the corresponding client behavior.

4.1 Returning XML generated using Xact

Web methods typically produce XHTML output, which is sent to the client to be shown in a browser window. A key constituent of JWIG is the Xact XML transformation language. This section provides a short introduction to Xact.

Xact is a general tool for transforming XML data. We here focus on its capabilities for constructing XHTML documents dynamically.

For more information about **Xact**, see `http://www.brics.dk/Xact/`.

An **XML template** is a fragment of an XML document that may contain unspecified pieces called *gaps*. These gaps can be filled with strings or other templates to construct larger templates and complete XML (typically XHTML) documents. XML templates are represented in the JWIG program by values of the `XML` type. XML values are immutable, that is, all operations that manipulate XML templates create new values without modifying the input values.

XML template constants are written as plain XML text enclosed in double square brackets. For example, the following line declares a variable to contain XML template values and initializes it to a simple constant value:

An XML variable and a simple template constant

```
XML hello = [[Hello <i>World</i>!]];
```

XML template constants must be well-formed, that is, the tags must be balanced and nest properly. Implicitly, JWIG sets the default namespace to `http://www.w3.org/1999/xhtml` (the XHTML 1.0 namespace). This can be changed with ordinary XML namespace declarations.

A **code gap** in an XML template is written using the syntax <{*code*}> where the content is a Java expression:

XML template with code gaps

```
String title = "JWIG";

XML getMessage() {
  return [[Hello <b>World</b>]];
}

XML hello = [[
  <html>
    <head><title><{title}></title></head>
    <body>
      <{getMessage()}>
    </body>
  </html>
]];
```

These gaps can be placed within text and tags, as if they were themselves tags. Gaps placed like this are called *template gaps*. When executed, the code in the code gaps is evaluated, and the resulting values are inserted in place of the gaps.

Code gaps can also be placed as the values of attributes. For example, `<a href={foo()}>` is an anchor start tag whose `href` attribute has a value obtained by invoking the method `foo`. Gaps placed like this are called *attribute gaps*.

A **named gap** is written using the syntax <[*name*]>. The gap name must be a legal Java identifier. Named gaps can also be placed inside tags, as the values of attributes.

For example, `<a href=[LINK]>` is an anchor start tag whose `href` attribute is a gap named `LINK`.

Named gaps are filled using the `plug` operator:

Building XML templates using `plug`

```
XML hello1 = [[<p align=[ALIGNMENT]>Hello <[WHAT]>!</p>]];
XML hello2 = hello1.plug("WHAT", [[<i><[THING]></i>]]);
XML hello3 = hello2.plug("THING", "World").plug("ALIGNMENT", "left");
```

After executing this code, the values of the variables will be:

```
hello1:  <p align=[ALIGNMENT]>Hello <[WHAT]>!</p>
```

```
hello2:  <p align=[ALIGNMENT]>Hello <i><[THING]></i>!</p>
hello3:  <p align="left">Hello <i>World</i>!</p>
```

As can be seen from the example, both strings and XML templates can be plugged into template gaps. However, only strings may be plugged into attribute gaps. When a string is plugged into a gap, characters that have a special meaning in XML (e.g. < and &) are automatically escaped.

The plug method allows any object to be plugged into a gap. XACT will convert the value into an XML fragment in the following way:

- If the object implements the `ToXMLable` interface then, the `toXML()` method on the object is invoked and the return value is plugged into the gap.
- Otherwise, the `toString()` method on the object is called and the value is treated as a text string without markup.

When a web method returns an XML template, all remaining template gaps are plugged with empty strings, and all remaining attribute gaps are removed.

The **JWIG program analyzer** (see Section 12) can check - at compile time - that the XML documents being generated dynamically using XACT are always **valid XHTML 1.0**.

A common use of named gaps in XML templates is for avoiding redundancy in dynamic XML construction, for example when many XHTML pages use the same overall structure:

XHTML wrapper

```
XML my_xhtml_wrapper = [[
  <html>
    <head><title><[TITLE]></title></head>
    <body>
      <h1><[TITLE]></h1>
       <[BODY]>
    </body>
  </html>
]];

XML page1 = my_xhtml_wrapper.plug("TITLE", "one")
  .plug("BODY", [[This is <b>one page</b>]]);

XML page2 = my_xhtml_wrapper.plug("TITLE", "two")
  .plug("BODY", [[This is <b>another page</b>]]);
```

Instead of inlining a template in a JWIG program, an XML template constant can be placed in an external file and loaded into the program at runtime using, for example, `XML.parseTemplateResource(`*`class`*`,`*`name`*`)` where *`class`* is a java class object used to locate the resource and *`name`* is a resource name. See the documentation of `Class` for more information on how class resources are located.

4.1.1 Compiling Xact programs

Because of the non-Java syntax, files containing XML template constants `[[...]]` must be desugared using the XACT compiler. Note that such files typically have extension `.xact` instead of `.java`. The compiler can be invoked from the command line using the following command:

```
java -classpath xact.jar dk.brics.xact.compiler.Main *.xact
```

See also the Ant build file `build.xml` in the JWIG kickstart file.

4.2 Redirecting the client

One alternative to returning XML data is to redirect to another web resource by returning a URL. As an example, when invoking the following web method, the response is a redirection to the given URL:

Response redirection

```
public URL redirect() {
  return new URL("http://www.brics.dk/JWIG/");
}
```

4.3 Returning plain text

Web methods can return plain text (media type text/plain) instead of XML or redirecting, simply by returning a string instead of an XML value or a URL:

`HelloWorld2.java` - returning plain text

```
public class HelloWorld2 extends WebApp {

  public String hello() {
    return "Hello␣World";
  }
}
```

4.4 Returning binary data using low-level servlet input/output

Raw binary data can be sent via the low-level Servlets API. JWIG gives access to the underlying `HttpServletRequest` or `HttpServletResponse` object using the `getServletRequest` and `getServletResponse` methods. A web method that uses the low-level framework should be defined as a high priority filter (see Section 5.5) to avoid interfering with the JWIG system. The `HttpServletResponse` object provides further access to an `OutputStream` object, which makes it possible to write raw data directly to the client. As an example, the following (very insecure!) web method allows the client to read a file from the server:

`ReadFile.java` - writing binary data

```
public class ReadFile extends WebApp {

  @Priority(PRE_CACHE)
  public void read(String fileName) throws IOException {
    getResponse().setContentType("audio/mid");
    FileInputStream in = new FileInputStream(fileName);
    HttpServletResponse res = getServletResponse();
    OutputStream out = res.getOutputStream();
    byte[] buffer = new byte[1024];
    int length;
    while ((length = in.read(buf)) != -1) {
      out.write(buf,length);
    }
    in.close();
    out.close();
  }
}
```

The output stream must be closed when the web method has finished writing the data, as in the example, to prevent JWIG from writing additional data to the client.

5 Requests and parameters

Web methods can receive parameters, such as form data fields, that arrive using the most common encodings (`application/x-www-form-urlencoded` and `multipart/form-data`). Parameters often occur in connection with form submission, as explained in Section 5.4.

The following example shows a web method with a single parameter named `x`:

`HelloWorld3.java` - web method parameters

```
public class HelloWorld3 extends WebApp {

  public XML hello(String x) {
    return [[
      <html>
        <head><title>JWIG</title></head>
        <body>
          The value of the parameter <tt>x</tt> is: <{ x }>
        </body>
      </html>
    ]];
  }
}
```

We can invoke this web method by e.g.

`http://`*`HOST`*`:`*`PORT`*`/`*`TOMCAT-APP`*`/HelloWorld3/hello?x=John+Doe`

Other data types than strings can also easily be received:

Any class that implements a static `valueOf(String)` method (this includes wrapper classes such as `Integer` and `Boolean`) - where `valueOf` is then used for creating a value from its string representation. An absent field results in the value `null`.

Primitive types (`int`, `boolean`, etc.)
: - parsed by invoking the standard Java functions, e.g. `Integer.parseInt`.

`FileField`
: - for file uploads (which requires `method="post"` and `enctype="multipart/form-data"` in the `form` tag), see Section 5.4.

`XML`
: - XACT values, parsed via `XML.parseDocument`.

`Persistable`
: - see Section 10.

Arrays (of the above kinds)
: - for receiving multiple parameters of the same name (the array may be empty but is never `null`).

`Collection<E>`, or subclasses, where `E` is of the above kinds.
: - for receiving multiple parameters of the same name

`Parameters`
: - matching *any* collection of parameters. If a web method declares an argument of type `Parameters` all request parameters that are not matched by the preceding arguments on the list of formal method parameters are represented using the `Parameters` object.

To make the parameter names of web methods accessible by the JWIG runtime system, either the code must be compiled with debug information, i.e. with option `-g`, or the `@ParamName` annotation must be used on *each* parameter of the method.

5.1 Mapping from URLs to code

The mapping from URLs to web apps, web methods and parameters can be configured using the `@URLPattern` annotation, which allows "REST-style" URLs:

Configuring URL to code mapping

```
@URLPattern("app4")
public class HelloWorld4 extends WebApp {

  @URLPattern("h/$x")
  public XML hello(String x) {
    return [[
      <html>
        <head><title>JWIG</title></head>
        <body>
          The value of the parameter <tt>x</tt> is: <{ x }>
        </body>
      </html>
    ]];
  }
}
```

The `hello` method can now be invoked by e.g.

`http://`*`HOST`*`:`*`PORT`*`/`*`TOMCAT-APP`*`/app4/h/117`

whereas the default mapping described in Section 3 would require

`http://`*`HOST`*`:`*`PORT`*`/`*`TOMCAT-APP`*`/HelloWorld4/hello?x=117`

A URL pattern is generally a set of *choices*, separated by '`|`'. Each choice is a sequence of *steps*, separated by `/`. A step can be:

a string (not containing a separator)
: - matching that string.

`*`
: - matching any string not containing '`/`'.

`**`
: - matching any string (which may contain '`/`').

`$`*`foo`*
: - matching any string not containing '`/`' and binding the string to the request parameter named *`foo`*.

Request parameters specified using `$`*`foo`* are merged with those supplied using the ordinary query string and request body mechanisms.

The URL patterns on `WebApp` classes must begin with a constant string, and they cannot overlap within the same `WebSite`.

5.1.1 @Priority

Multiple web methods may match a request URL. In this case, the value of the `@Priority` annotation determines the order. A default priority is assigned to web methods that have no `@Priority` annotation.

5.2 Generating links to web methods

Often, redirection is to another web method within the same web site. The family of `makeURL` methods makes it easy to construct such URLs:

Response redirection to another web method

```
    public URL redirect2() {
      return makeURL("hello");
    }

    public XML hello() {
      return [[
      <html>
        <head><title>JWIG</title></head>
        <body>Hello World</body>
      </html>
    ]];
  }
```

The string parameter to `makeURL` is the name of the web method in the URL. The `makeURL` method will inspect the web method and generate a URL with the format given in the `@URLPattern` (or the default pattern if no annotation is given for the web method). This makes it straightforward to change the URL of a web method without changing all the places that link to the web method.

Parameters (see Section 5) can be added simply as extra arguments to the `makeURL` method. JWIG will then automatically serialize the parameters depending on the type of the object. See Section 5 for more information about which types are supported.

The following example modifies the Hello World example to take an additional parameter `who`, which is plugged into the greeting returned. The `hello` web method declares this parameter to be of type `String`. The `redirect3` web method generates a URL to `hello` with the `who` argument simply by passing the value as a parameter to `makeURL`:

Response redirection to another web method including a parameter

```
  public URL redirect3() {
    return makeURL("hello", "World");
  }

  public XML hello(String who) {
    return [[
      <html>
        <head><title>JWIG</title></head>
        <body>Hello <{who}></body>
      </html>
    ]];
  }
```

See Section 7 for another example where `makeURL` is used in combination with parameters of other types.

The `makeURL` method takes an optional `Class` parameter identifying the web app containing the target web method. If no such argument is given, `makeURL` assumes that the web method is located within the same web app as the method that generates the URL.

Finally, `makeURL` takes an optional boolean parameter determining whether the generated link will point to a secure (HTTPS) or insecure (HTTP) URL. If no boolean value is given, `makeURL` will generate a secure URL if the current request is to a secure URL.

5.3 Web app parameters

Parameters can also be given to web apps, not only to the individual web methods. This is controlled using `@URLPattern` annotations (see Section 5.1) on the web app classes. A web app parameter is passed as part of any link created by `makeURL` from one web method to another within the same web app. This gives a convenient way to manage a context that needs to be available to multiple web methods in a web app.

In the following example, the `who` parameter has been changed to a web app parameter. If the user sends a request to `myapp/World/redirect4`, the `redirect` method will redirect the client to `myapp/World/hello`.

Response redirection to another web method including a parameter

```
@URLPattern("myapp/$who")
public class HelloApp extends WebApp {
  public URL redirect4() {
    return makeURL("hello");
  }

  public XML hello() {
    String who = getWebAppParameter("who");
    return [[
      <html>
        <head><title>JWIG</title></head>
        <body>Hello <{who}></body>
      </html>
    ]];
  }
}
```

A special version of the `makeURL` method takes a map from parameter names to values, which makes it easy to pass web app parameters between web methods.

5.4 XHTML forms

Ordinary web methods respond to HTTP GET requests, as explained earlier. The response can be an XHTML page containing a form. The JWIG `SubmitHandler` class allows the application to receive values from form submissions. This promotes a programming style where the program code that produces the form becomes tightly coupled with the code that receives the field values, which makes it easy for readers of the code to follow the application flow.

Here is an example of a web method that produces a form and receives a text field and an uploaded file:

`Upload.xact` - a file upload form

```
public class Upload extends WebApp {

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

  public XML upload() {
    return wrapper.plug("BODY", [[
      <form method="post" enctype="multipart/form-data" action=[ACTION]>
        Enter some text: <input type="text" name="t"/> <br/>
        Upload a file: <input type="file" name="f"/> <br/>
        <input type="submit" value="Go!"/>
      </form>
    ]].plug("ACTION",
      new SubmitHandler() {
        public XML run(String t, FileField f) {
          return wrapper.plug("BODY", [[
            Your text: <{ t }> <br/>
            Size of your file: <{ f.getSize() }>
          ]]);
        }
    });
}
```

This `SubmitHandler` class contains a `run` method that processes the form field values. Similarly to web methods, the response of this method can have various types:

`XML`
: - (as in the example above) an XML value that is sent directly back to the client.

`URL`
: - resulting in a redirection to the given location using the POST-redirect-GET pattern.

`void`
: - resulting in a redirection to the URL of the web method that led to the `SubmitHandler` class.

The latter case is useful in combination with the ability of *modifying* an existing web page, as shown in the example application GuessingGame.

`SubmitHandler` objects are subject to regeneration; see Section 6.3.

5.5 Filters and user authentication

A filter is a web method with return type `void`, that is, a web method that may do something on the server but generates no value to return to the client. The default priority of a filter is higher than that of the `cache` method, which means that filters are processed before the caching mechanism explained in Section 6 is applied. By default, filters match GET and POST requests, but they may be annotated to match to any request type.

Filters return no value and therefore they must pass control on to another web method, which can then in turn generate the response. The `next` method matches the request URL with the next web method on the prioritized list of matching web methods.

While filters do not return values, they may still interrupt the request processing using exceptions. This is convenient for authentication and other security checks. Section 11 describes the use of exceptions (and also cookies, which may be used for another kind of user authentication). The following filter matches all request URLs (using `@URLPattern`) and returns an error if an insecure connection is used (that is, without SSL/TLS) and a HTTP Basic authentication "authorization required" message if the right user credentials are not provided:

HTTP Basic authentication

```
@URLPattern("**")
public void accessBasic() {
  if (!isSecure)
    throw new AccessDeniedException("Connection is insecure!")
  User u = getUser();
  if (u == null || !u.getUsername().equals("smith") ||
      !u.getPassword().equals("42"))
    throw new AuthorizationRequiredException("JWIG examples");
  next();
}
```

5.6 XML web service programming

JWIG support all HTTP methods for JWIG web methods, and XACT can easily be used to generate any kind of XML. This allows web methods to implement REST-style web services in a straightforward manner. Web methods are simply annotated with `@GET`, `@POST`, `@PUT`, `@DELETE` etc. to indicate which request kinds the web method should respond to. More than one HTTP method annotation may be set on a single web method. The default is `@GET`. Consider the following example:

Web service example

```
@PUT
public URL store(String s) {
    // implementation omitted
}
```

JWIG will invoke this web method only if a matching PUT request is sent from the client.

6 Caching

JWIG defines a web method that automatically caches all responses generated by web methods with lower priority than the `CACHE_PRIORITY`. HTTP `ETag` and `Last-Modified` headers are automatically inserted to support client-side caching. Additionally, a built-in web method (`cache`) handles server caching and conditional GET requests.

6.1 Invalidating cached pages

Occasionally, the programmer needs to invalidate cached pages because some data has been changed. JWIG uses an explicit representation of the relationship between data objects and cached pages to enable such invalidation.

JWIG maintains a dependency map from data objects to pages that depend on them. To add a dependency for the current page on an object, use the `addResponseInvalidator` method. When the data object is changed, the `update` method must be called for JWIG to invalidate all cached pages that depend on the object.

The database system (Section 10) integrates with this mechanism. When a data object is used as a parameter to a web method, `addResponseInvalidator` is automatically called with this object. Also, when an object is updated in the database, the `update` update method is called automatically.

The example applications in Section 13 show how to use `addResponseInvalidator`.

6.2 Disabling the cache

The cache can be disabled for a web method by assigning a priority to the web method that is higher than the priority of the `cache` filter. The `WebContext` class defines such a constant value `PRE_CACHE`.

See also Section 5.5 about filters. Filters always have a priority above the cache.

6.3 Handler regeneration

Handlers (that is, `SubmitHandler` objects, see Section 5.4, `XMLProducer` objects, see Section 8.1, and `EventHandler` objects, see Section 8.2) may be removed from the server as a result of cache eviction or restart of the web server. Therefore, JWIG may need to regenerate a handler in order to process a request.

A handler can be regenerated in two different ways: recreation or rediscovery.

Recreation:

> JWIG recreates a handler by sending a new GET request to the URL that originally created the evicted handler. Consequently, the application programmer must make sure that such a GET request results a valid recreation of the handler. A handler `a` is a valid recreation of an evicted handler `b` if `a.equals(b)` is true. The implementation of all subclasses of `AbstractHandler` satisfies this property, provided that the property also holds for each of the objects in its list of dependencies (see `AbstractHandler`).

Rediscovery:

> Handlers can be rediscovered if they are constructed in a web method that creates a `Session` object or takes one as a parameter (see Section 7). Such handlers are called session bound handlers. See Section 13.3 for an example of a session bound handler.

Unless session bound, a handler can only be regenerated if:

- it is recreated as a result of sending a request to the original URL, and
- it is not returned as part of an XML document created by another handler (since such a recreation would require a POST request with possible side effects).

7 Session state

The class `Session` makes it easy to encapsulate session state and transparently pass the data to web methods that are later invoked by the client.

A subclass of `Session` contains the session data. Session objects can be passed to other web methods through parameter passing. JWIG automatically provides a session ID to each session object. The following example illustrates a typical use:

Sessions

```
class UserState extends Session {
  List<Item> shopping_cart; // contains user state
}

public URL entry() {
  UserState s = new UserState();
  ... // initialize the new session
  return makeURL("shopping", s);
}

public XML shopping(UserState s) {
  ... // s contains the session state
}
```

This mechanism avoids the pitfalls of traditional String-to-Object session maps that can be difficult to maintain by the application programmer.

A session management thread keeps track of the mapping from session IDs to session objects (via `toString` and `valueOf`; see Section 5) and automatically garbage collects session objects that have not been accessed in some time (as configured in the `Session` constructor). If `jwig.auto_refresh_sessions` is set (see Section 9), session objects are automatically being refreshed regularly (using some JavaScript code) as long as some user is viewing a page holding the session object. This means that it is in most cases unnecessary to write things like "this session will timeout after N minutes, so make sure to submit your form data in time".

The GuessingGame example demonstrates the use of session state.

8 Client side programming

While the JWIG framework is server-based and all computation of the web apps takes place on the server side, it is possible to create dynamic applications in JWIG by using two primitives: the ability to automatically push updates of data to the clients, and the ability to capture XHTML events. This model often makes it unnecessary for the programmer to write any JavaScript code.

8.1 Automatic view updates

JWIG makes it possible to update the XML contents of a page while it is being shown at the client side. These updates are automatically propagated to the clients using a built-in Comet/AJAX system.

An `XMLProducer` is responsible for generating a fragment of the XML data that may be updated and propagated to the client at any time. The XMLProducer must implement a run method that is used to generate its content. Data objects that the XMLProducer depend on are given as arguments to the constructor. When those data objects are changed (see the `update` method in Section 6), the `run` method is invoked, and the resulting XML data is sent to all clients viewing the data.

The following example mimics a page with a highscore for a game. Whenever the highscore is changed, the database system notifies the XMLProducer to invalidate its current value. The XMLProducer is then executed again and the new value is sent to the client.

XMLProducer example

```
public void ajaxy() {
  XML page = ...;
  page = page.plug("NAME", new XMLProducer() {
    public XML run() {
      HighScore highscore = Game.getCurrentGame().getHighScore();
      addResponseInvalidator(highscore);
      return highscore.getName();
    }
  });
}
```

For `XMLProducer` to work in Tomcat, the NIO connector must be used. In `server.xml` make sure that the 'protocol' is set to `org.apache.coyote.http11.Http11NioProtocol`

Connector configuration example for Tomcat

```
<Connector port="8080" connectionTimeout="20000" redirectPort="8443"
           protocol="org.apache.coyote.http11.Http11NioProtocol" />
```

8.2 XHTML events

The `EventHandler` class is used to respond to client side XHTML events. It can be plugged into any XHTML attribute where an XHTML event handler is expected. The EventHandler must contain a run method that is responsible for handling the event on the server side. This run method is able to take parameters just like a SubmitHandler. The following example shows an EventHandler that responds to onClick events:

EventHandler example

```
XML x = [[...<input type="button" value="Click me" onClick=[HANDLER] >...]]
  .plug("HANDLER", new EventHandler() {
    public void run() {
      log.info("User clicked the button");
    }
});
```

`EventHandler` objects are subject to regeneration; see Section 6.3.

9 Configuration

JWIG uses a common configuration system that allows properties to be specified declaratively in a separate file and operationally at runtime. First, properties can be written in the file `jwig.properties`, as this example:

```
jwig.base_url = http://services.brics.dk/java
jwig.base_url_secure = https://services.brics.dk/java
jwig.hibernate = true
hibernate.connection.username = jdoe
hibernate.connection.password = ToPsEcReT
hibernate.connection.url = jdbc:mysql://mysql.cs.au.dk/jdoe
mail.transport.protocol = smtp
mail.host = smtp.nfit.au.dk
```

Second, properties can be set (typically during initialization of the `WebSite` or `WebApp`) using `setProperty`. Properties for the `WebSite` are shared for all web apps and can be overridden locally in each `WebApp`. To read a property, use `getProperty`. The names of JWIG specific properties all begin with `jwig`. Note that reasonable defaults have been selected for all properties, so simple JWIG applications can be created without explicitly setting any configuration.

The `jwig.properties` file must be placed in the base folder of the class loader at runtime (i.e. in the folder containing the class files that belong to the default package).

A complete list of **JWIG configuration properties** is available in the API documentation of the `WebSite` class.

10 Persistence with Hibernate

This section is by no means a complete guide to the Hibernate framework, but merely as a description of how to enable Hibernate for use in JWIG programs.

While Hibernate is a powerful framework that requires some work to master, the basics can be learned quickly. See the Hibernate Getting Started document at `http://hibernate.org/quick-start.html`

Most web apps need to store data externally. For this purpose JWIG integrates closely with the Java ORM framework Hibernate. To enable Hibernate you will need a database server and a JDBC driver for the database. Furthermore, mappings between objects and database tables must be specified. Based on these mappings, Hibernate generates SQL for updating and querying the database. All popular databases have their own SQL dialect so Hibernate needs to be set up to use the right one. Make sure that your database in on the list of list of supported databases.

10.1 Setting up your database connection and dialect

JWIG defaults to MySQL, so if you use MySQL as your database, you will not need to set the dialect. The dialect classes for the most popular databases can be found on this list:

Database	**Dialect**
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9iDialect`

For the full list, please the refer to the Hibernate documentation

To set the database dialect, use the JWIG configuration property '`hibernate.dialect`'. Please refer to Section 9 for details on configuration.

Second you will have to set the class name for your JDBC driver. Again this is done for MySQL. If you use another database, please set the property '`hibernate.connection.driver_class`' in the same way as above to the qualified name of you driver class. This class name can be found on the web site of the JDBC driver.

10.2 Setting up username and password

Finally you will need to set the following properties in the JWIG configuration (see Section 9):

Property	Contents
`hibernate.connection.username`	Your database username
`hibernate.connection.password`	Your database password
`hibernate.connection.url`	Connection URL for you database, for example jdbc:mysql://mysql.
`jwig.hibernate`	Set this to 'true' to enable Hibernate support

10.3 Querier

The `Querier` is central to the Hibernate bridge as it allows JWIG to query objects from the database. The `Querier` can be obtained through `WebSite.getQuerier()`. It can be used directly by the web app code to query objects from the database:

Object querying
```
User u = getWebSite().getQuerier().getObject(User.class, 1);
```

This will return the `User` from the database with `ID` 1 or null if no such user exists. In principle, any ORM framework could be supported by implementing appripriate `Querier` classes. JWIG provides a single implementation of the interface called `HibernateQuerier`. This class allows access to objects mapped in the Hibernate framework and it is enabled if the `jwig.hibernate` property is set.

10.4 Persistable objects

The `Persistable` interface allows JWIG to serialize and query persisted objects. This interface requires the object to have an ID property of type `Integer`. An implementation is provided as `AbstractPersistable`. The ID must be unique for the given object within the database. JWIG refers to persistable objects by their IDs in e.g. URLs.

In Hibernate the class must be mapped and added to the Hibernate framework by accessing the the `Configuration` object through the `HibernateQuerier` during initialization of the web site:

Loading a mapping
```
HibernateQuerier.getConfig().addClass(User.class);
```

More advanced queries can be made using the standard Hibernate framework. The session factory can be obtained using the static method `HibernateQuerier.getSessionFactory()` and Hibernate sessions can be obtained through the `getCurrentSession()` method of this object.

10.5 Persistable parameters

Persistable objects can be used as parameters to web methods. Persistable objects are serialized through their `getId()` method, and JWIG deserializes the objects using the `Querier` interface described above.

Example of persistable parameter passing

```
  public XML viewUser(User user) {
    // User (as mapped above) is automatically queried
    // from the database and passed as parameter
  }

//Furthermore, makeURL can create URLs to such methods
public URL gotoUser(User user) {
    return makeURL("viewUser", user);
}
```

11 Miscellaneous convenience features

11.1 Error reporting and logging

A built in web method takes care of catching all exceptions thrown by the ordinary web methods and reporting errors to the clients. The following exceptions are particularly relevant:

`JWIGException`
- base class for all JWIG runtime exceptions (results in "500 Internal Server Error")

`AccessDeniedException`
- thrown when the client is denied access to a requested resource ("403 Forbidden")

`AuthorizationRequiredException`
- thrown when the client should resend the request with HTTP Basic authentication information ("401 Unauthorized"), see Section 5.5

`BadRequestException`
- thrown when the HTTP request contains errors ("400 Bad Request")

`ServerBusyException`
- thrown when the server is overloaded ("503 Service Unavailable")

`PersistenceException`
- thrown if the persistence system is misconfigured ("500 Internal Server Error"), see Section 10

`SessionDefunctException`
- thrown when trying to fetch a non-existing session from the session state manager ("410 Gone")

JWIG uses log4j for logging, and the JWIG runtime system automatically logs many internal events. A `Logger` object for logging application specific events can be obtained by using the factory method on Logger like this:

Obtaining a logger

```
Logger log = Logger.getLogger(WEBAPP.class);
```

where *WEBAPP* is the name of your web app class. See also the configuration properties (Section 9) whose name start with `log4j`. The following example writes a message to the log with level `INFO`:

Writing to the log

```
log.info("10-4 Rubber Duck");
```

In some cases it can be expensive to create the string that is printed to the log. In that case, test if the log is enabled for the relevant level before generating the string:

Testing log level

```
if (log.isInfoEnabled())
    log.info(getComplicatedStringThatTakesALongTimeToCompute());
```

Log4j can give fine-grained control of what loggers print. It can be configured at class or package (including sub-package) level so that different log levels are used for different program parts. The standard JWIG configuration is to set the log level to print all log statements of at least level INFO to `System.out`. Log4j can be initialized as part of the `init` method of the `WebSite` for more fine-grained control of the loggin behavior. See this Log4j tutorial for more information.

11.2 Cookies

The method `getCookies` returns a (non-null) array of cookies occurring in the request, and `addCookie` adds a cookie to the response.

The following example shows how to use cookies for authentication:

Cookie authentication

```
  @URLPattern("restricted/**")
  public void accessCookie() {
    String key = null;
    for (Cookie c : getCookies())
      if (c.getName().equals("key"))
        key = c.getValue();
    if (key == null || !key.equals("87"))
      throw new AccessDeniedException("You shall not pass!")
    next();
  }
```

Here, `accessCookie` is a filter web method that checks in every request matching `restricted/**` whether a cookie named `key` with value `87` is provided.

11.3 Sending emails

JWIG uses JavaMail for sending emails:

Emails

```
import javax.mail.*;
import javax.mail.internet.*;
import dk.brics.jwig.*;

public class EmailTest extends WebApp {

  public String send() throws MessagingException {
    Email e = new Email();
    e.setText("This is a test!");
    e.setSubject("JWIG email test");
    e.setSender(new InternetAddress("amoeller@cs.au.dk"));
    e.addRecipients(Message.RecipientType.TO, "amoeller@cs.au.dk");
    SendFailedException ex = sendEmail(e);
    if (ex == null)
      return "Email sent!";
    else
      return ex.toString();
  }
}
```

Configuration properties (see Section 9) whose name start with "`mail.`" are used by JavaMail. In `jwig.properties` set e.g.:

```
mail.transport.protocol = smtp
mail.host = smtp.nfit.au.dk
```

11.4 Debugging applications

You can attach a remote debugger to your JWIG program by starting Tomcat with the following options:

```
-Xdebug
-Xnoagent -Xrunjdwp:transport=dt_socket,address=127.0.0.1:8000,server=y,suspend=n
```

(On Windows, edit the Java Options in 'configure Tomcat'.) You may want to use a different value of `address`. Set your IDE to use the same address for remote debugging (in Eclipse start the debugger with a Remote Java Application), and you can now use the debugger while Tomcat is running.

12 The JWIG program analysis suite

FiXme Fatal: TO DO: update this section with the newest XACT/JWIG analyses

The design of JWIG allows some specialized program analyses to be performed, such that the programmer can check - at compile time - that certain kinds of errors cannot occur at runtime. The JWIG analyzer considers the following properties for a given program:

- **show validity:** that all pages being shown are valid XHTML 1.0,
- **plug consistency:** that gaps are present when subjected to the plug operation and XML templates are never plugged into attribute gaps,
- **form field consistency:** that the parameters of form submit handlers and event handlers match the form names that occur in the generated XHTML pages,

- **URL construction:** that values of @URLPattern and makeURL are meaningful, and
- **method safety:** that web methods and XML producers do not have side effects that may influence behavior of the web app.
- **regeneration correctness:** that the referer page can be reconstructed correctly after system reboot and emptying of the cache.

Additionally, the program analysis can produce a global control-flow graph of a web app to visualize its structure.

13 Examples

13.1 Example: QuickPoll

The QuickPoll example illustrates a small application where it is possible for a client to ask a question to which other clients can answer either 'yes' or 'no'. The web app has four web methods, index, init, vote and results, and a single filter, authenticate, that filters requests to the init method and requires the client to authenticate using a username and a password. An inner class State holds the question and the votes. Finally, the field wrapper holds an XML fragment which is used by all the web methods.

The web methods follow a straightforward pattern returning an XHTML page to the client upon request. The init and vote methods allow changes to the state of the web app. Both methods define a SubmitHandler to receive parameters from the client on form submission and update the question and the votes accordingly. Finally, the results method illustrates how XML generation can be wrapped in an XMLProducer to allow server push of values as they change.

Example: QuickPoll

```
package quickpoll;

import dk.brics.jwig.*;
import dk.brics.xact.*;

@URLPattern("quickpoll")
public class QuickPoll extends WebApp {

  XML wrapper = [[
    <html>
      <head><title>QuickPoll</title></head>
      <body>
        <h1>QuickPoll</h1>
        <[BODY]>
      </body>
    </html>
  ]];

  class State {
    String question;
    int yes;
    int no;
  }

  State state = new State();

  @URLPattern("")
  public XML index() {
    return wrapper.plug("BODY", [[
      <ul>
        <li><a href={makeURL("init")}>Initialize</a> (access control)</li>
        <li><a href={makeURL("vote")}>Vote</a></li>
        <li><a href={makeURL("results")}>View results</a></li>
      </ul>
    ]]);
  }

  @URLPattern("init")
  public void authenticate() {
    User u = getUser();
    if (u != null && u.getUsername().equals("jdoe") &&
        u.getPassword().equals("42"))
      next();
    else
      throw new AuthorizationRequiredException("QuickPoll");
  }

  public XML init() {
    return wrapper.plug("BODY", [[
      <form method="post" action=[INIT]>
        What is your question?<br/>
        <input name="question" type="text" size="40"/>?<br/>
        <input type="submit" value="Register_my_question"/>
      </form>
    ]]).plug("INIT", new SubmitHandler() {
      XML run(String question) {
        synchronized (state) {
          state.question = question;
          state.yes = state.no = 0;
        }
        update(state);
        return wrapper.plug("BODY", [[
          Your question has been registered.
          Let the vote begin!
        ]]);
      }
    });
  }

  public XML vote() {
    if (state.question == null)
      throw new AccessDeniedException("QuickPoll_not_yet_initialized");
    addResponseInvalidator(state);
    return wrapper.plug("BODY", [[
      <{state.question}>?<p/>
      <form method="post" action=[VOTE]>
        <input name="vote" type="radio" value="yes"/> yes<br/>
        <input name="vote" type="radio" value="no"/> no<p/>
        <input type="submit" value="Vote"/>
      </form>
    ]]).plug("VOTE", new SubmitHandler() {
      XML run(String vote) {
        synchronized (state) {
          if ("yes".equals(vote))
            state.yes++;
          else if ("no".equals(vote))
            state.no++;
        }
        update(state);
        return wrapper.plug("BODY", [[
          Thank you for your vote!
        ]]);
      }
    });
  }

  public XML results() {
    return wrapper.plug("BODY", new XMLProducer(state) {
      XML run() {
        synchronized (state) {
          int total = state.yes + state.no;
          if (total == 0)
            return [[No votes yet...]];
          else
            return [[
              <{state.question}>?<p/>
              <table border="0">
                <tr>
                  <td>Yes:</td><td><{drawBar(300*state.yes/total)}></td>
                  <td><{state.yes}></td>
                </tr>
                <tr>
                  <td>No:</td><td><{drawBar(300*state.no/total)}></td>
                  <td><{state.no}></td>
                </tr>
              </table>
            ]];
        }
      }
    });
  }

  private XML drawBar(int length) {
    return [[<table><tr>
                <td bgcolor="black" height="20" width={length}></td>
              </tr></table>]];
  }
}
```

13.2 Example: MicroChat

The MicroChat example is an application where clients are able to communicate with each other through a simple interface containing an input box and a list of messages. The list of messages is stored in the `messages` field using the `Messages` class which simply contains a list of strings.

A single web method `run` presents the list of messages. The generated XHTML list is wrapped in an `XMLProducer` to allow the view to be updated at all clients as the list changes. The `XMLProducer` listens for updates at the `messages` object and updates the view if the list is updated. Furthermore, the `run` method contains a `SubmitHandler` that receives messages written by clients. The handler adds the new message to the list of messages, and the `add` method on `Messages` calls `update` to notify the `XMLProducer`. Finally, the example uses an `EventHandler` to allow clients to clear the list of messages. The EventHandler is executed when the `reset` button is clicked and the change is pushed to all clients as a consequence of the call to `update`.

Note that the use of the XACT syntax means that values from the clients are escaped by construction. This means that it is not possible for a malicious client to inject HTML or JavaScript into another client's browser in this example.

Example: MicroChat

```
package microchat;

import java.util.*;
import dk.brics.jwig.*;
import dk.brics.xact.*;

@URLPattern("microchat")
public class MicroChat extends WebApp {

  class Messages {
    List<String> ms = new ArrayList<String>();

    void add(String msg) {
      ms.add(msg);
      update(this);
    }

    void reset() {
      ms.clear();
      update(this);
    }
  }

  Messages messages = new Messages();

  @URLPattern("")
    public XML run() {
      return [[
        <html>
          <head>
            <title>MicroChat</title>
          </head>
          <body>
            <{
              new XMLProducer(messages) {
                    XML run() {
                        if (!messages.ms.isEmpty())
                            return [[
                                <ul>
                                    <{ [[<li><[MSG]></li>]]
                                        .plugWrap("MSG", messages.ms) }>
                                </ul>
                            ]];
                        else
                            return [[]];
                    }
                }
            }>
            <form method="post" action=[SEND]>
                <input type="text" name="msg"/>
                <input type="submit" value="Send!"/>
                <p/>
                <input type="button" value="Reset" onclick=[RESET]/>
            </form>
        </body>
    </html>
    ]]
    .plug("SEND", new SubmitHandler() {
        void run(String msg) {
        messages.add(msg);
      }
    })
    .plug("RESET", new EventHandler() {
        void run() {
        messages.reset();
      }
    });
  }
}
```

13.3 Example: GuessingGame

The GuessingGame example is a small game that requires the client to guess a number between 0 and 100. If the client makes an incorrect guess, he is informed whether the number to guess is higher or lower than the number he provided. The application keeps track of the number of guesses he has made so far as well as the global record held by the client who has guessed the number in the least number of guesses.

The GuessingGame example illustrates the use of a `Session` class to control the web app control flow. The web method `start` creates a new `Session` object and redirects the user to the `play` method, which simply returns the value of the `page` field of the session.

The flow of the application is controlled by the `GameSession` class. When a `GameSession` instance is created, an initial page is stored in its `page` field. This page contains a `SubmitHandler` that receives the guess from the client. The `SubmitHandler` updates the `page` value to contain either a message that the number should be higher or lower or a confirmation that the number has been guess as well as possibly a form with a new `SubmitHandler` allowing the client to type his name if he has broken the record.

Notice that the number is never made available on the client side and that the client cannot progress to entering his name for the highscore unless he has broken the record.

Example: GuessingGame

```
package guessinggame;
```

```
import java.net.URL;
import java.util.*;
import dk.brics.jwig.*;
import dk.brics.xact.*;
import dk.brics.jwig.persistence.HibernateQuerier;

@URLPattern("guessinggame")
public class GuessingGame extends WebApp {

  public GuessingGame() {
    HibernateQuerier.getConfig().addClass(GameState.class);
  }

  XML wrapper = [[
    <html>
      <head><title>The Guessing Game</title></head>
      <body bgcolor="aqua"><[BODY]></body>
    </html>
  ]];

  Random rnd = new Random();

  class UserState extends Session {

    int number;
    int guesses;
    XML page;

    UserState() {
      number = rnd.nextInt(100)+1;
      guesses = 0;
      page = wrapper.plug("BODY", [[
          <div>Please guess a number between 1 and 100:</div>
          <form method="post" action=[GUESS]>
            <input name="guess" type="text" size="3" maxlength="3"/>
            <input type="submit" value="continue"/>
          </form>
        ]]).plug("GUESS", new SubmitHandler() {
          void run(int guess) {
            guesses++;
            if (guess != number) {
              page = page.setContent("/xhtml:html/xhtml:body/xhtml:div", [[
                That is not correct. Try a
                <b><{(guess>number)?"lower":"higher"}></b> number:
              ]]);
            } else {
              page = page.setContent("/xhtml:html/xhtml:body", [[
                You got it, using <b><{guesses}></b> guesses.<p/>
              ]]);
              final XML thanks = [[Thank you for playing this exciting game!]];
              GameState game = GameState.load();
              if (game.getRecord() > 0 && guesses >gt;= game.getRecord())
                page = page.appendContent("/xhtml:html/xhtml:body", thanks);
              else
                page = page.appendContent("/xhtml:html/xhtml:body", [[
                  That makes you the new record holder!<p/>gt;
                  Please enter your name for the hi-score list:
                  <form method="post" action=[RECORD]>
                    <input name="name" type="text" size="20"/>
                    <input type="submit" value="continue"/>
                  </form>
                ]]).plug("RECORD", new SubmitHandler() {
                  void run(String name) {
                    synchronized (GuessingGame.class) {
                      GameState game = GameState.load();
                      if (guesses < game.getRecord())
                        game.setRecord(guesses, name);
                    }
                    page = page.setContent("/xhtml:html/xhtml:body", thanks);
                    update(UserState.this);
                  }
                });
            }
            update(UserState.this);
          }
        });
    }
  }

  public URL start() {
    GameState.load().incrementPlays();
    return makeURL("play", new UserState());
  }

  public XML play(UserState s) {
    return s.page;
  }

  public XML record() {
    final GameState game = GameState.load();
    return wrapper.plug("BODY", new XMLProducer(game) {
      XML run() {
        synchronized (game) {
          if (game.getHolder() != null)
            return [[
              In <{game.getPlays()}> plays of this game,
              the record holder is <b><{game.getHolder()}></b> with
              <b><{game.getRecord()}></b> guesses.
            ]];
          else
            return [[<{game.getPlays()}> plays started.
                     No players finished yet.]];
        }
      }
    });
  }

}
```