JWIG: Yet Another Framework for Maintainable and Secure Web Applications

Anders Møller and Mathias Schwarz
Department of Computer Science, Aarhus University, Denmark
amoeller@cs.au.dk, schwarz@cs.au.dk

Abstract

Although numerous frameworks for web application programming have been developed in recent years, writing web applications remains a challenging task. Guided by a collection of classical design principles, we propose yet another framework. It is based on a simple but flexible server-oriented architecture that coherently supports general aspects of modern web applications, including dynamic XML construction, session management, data persistence, caching, and authentication, but it also simplifies programming of server-push communication and integration of XHTML-based applications and XML-based web services. The resulting framework provides a novel foundation for developing maintainable and secure web applications.

1 Introduction

Web applications build on a platform of fundamental web technologies, in particular, XHTML, HTTP, and JavaScript. Although these are relatively manageable technologies, it is generally regarded as difficult to write and maintain nontrivial, secure applications. The programmer must master not only the fundamental technologies but also techniques for session management, caching, data persistence, form input validation, and rich user interfaces. In addition, most web applications are exposed to all hackers in the world, so the programmer must also consider potential security problems, such as, injection attacks, cross site scripting, and insufficient authentication or encryption.

In recent years, a multiplicity of *web application frameworks* have emerged, all claiming to make web application development easier. Among the most widely known are Struts (McClanahan et al., 2008), Spring MVC (Johnson et al., 2008), Google Web Toolkit (Google, 2008), Ruby on Rails (RoR) (Hansson et al., 2008), PHP (Lerdorf et al., 2008), and ASP.NET (Microsoft, 2008). These frameworks represent different points in the design space, and choosing one for a given task is often based on subjective arguments. However, a general limitation is the lack of a simple unified communication model that supports both the traditional server-oriented structure and the more modern style using AJAX[1], server-push[2], and JSON or XML-based web services (Crockford, 2006; W3C, 2008).

In this paper, we try to take a fresh view on the problem. Based on essential design principles, we exhibit some of the limitations of existing frameworks, propose yet another web application framework, and compare it with the existing ones. The main contributions of this paper can be summarized as follows:

- First, we identify and argue for a small collection of requirements and design principles for creating a web application framework.
- In Section 2 we propose a simple architecture that supports the basic features of receiving HTTP client input and producing XHTML output, in accordance with the design goals.
- In Sections 3–9 we demonstrate the flexibility of the architecture by showing how it can smoothly handle other important aspects that can be challenging to work with in other frameworks, including server-push communication, session state, persistence, and authentication.
- The resulting framework provides a clear structure of both control and data, which makes the application program code amenable to specialized static analysis. For example, one analysis checks—at compile-time—that only valid XHTML 1.0 data is sent to the clients during execution.
- The new framework is evaluated through a series of short but nontrivial example programs and a case study that demonstrate the resulting system in relation to the requirements and design principles, in comparison with alternative frameworks.

Premises and Requirements

To specify the aims of our task and narrow the design space, we begin by formulating the basic premises and requirements:

- We focus on Java, for the simple reasons that this language is well-known to most programmers, especially those developing web applications, and it

[1] `http://en.wikipedia.org/wiki/Ajax_(programming)`

[2] `http://en.wikipedia.org/wiki/Comet_(programming)`

comes with a massive collection of useful, open source libraries that we and the application programmers can build on. An immediate consequence of this choice is that we cannot benefit from, for example, general higher-order functions, closures and continuations, as known from functional languages and frameworks such as Seaside (Ducasse et al., 2007) and the PLT Scheme Web Server (Krishnamurthi et al., 2007). Such features can be emulated in Java, but not elegantly. On the other hand, we can exploit Java's reflection capabilities. We permit simple syntactic extensions of Java, in particular to support XML processing.

- We assume that the browsers are capable of rendering XHTML and executing JavaScript code, as in Internet Explorer 8 and Firefox 3, but the framework cannot use browser plugins. This means that the applications will be readily deployable to all users with modern browsers.
- We postulate that a majority of web applications do not require massive scalability and have less than a thousand concurrent uses. Our framework design should focus on this category.
- The framework must uniformly support both XHTML applications (where the clients are browsers) and XML-based web services (where the clients are other types of software). It should be possible to statically check that only valid XML output is produced, relative to a given XML schema. This requires a clear flow of control and data in the code. For the back-end, the framework must integrate with existing object-relational mapping (ORM) systems, e.g. Hibernate (O'Neil, 2008), such that data persistence can be obtained with minimal effort to the programmer, but without being tied to one particular system.
- Representational state transfer (REST) is a collection of network architecture principles that the Web is based on (Fielding and Taylor, 2002). We focus on its use of URLs for addressing resources with generic interfaces as HTTP methods (GET, POST, etc.) and caching. In contrast, the remote procedure call (RPC) approach encourages the use of URLs for addressing operations, not resources, which sometimes fits better with the abstractions of the programming language in use. Both approaches should be uniformly supported by the framework.

Design Principles

To guide the design of the framework, we adhere to the following key principles:

High cohesion and low coupling Cohesion is a measure of how strongly related and focused the responsibilities of a software unit is. The related concept of coupling is a measure of how strongly dependent one software unit is on other software units. It is common knowledge in software engineering that high cohesion and weak coupling are important to maintainability and reliability of the software (Stevens et al., 1974). Nevertheless, many web programming frameworks fail in these measures when considering, for example, handling of form data and asynchronous client–server communication. Examples of this are shown in Sections 4 and 5.

Secure by design Buffer overruns are an example of a security concern that has largely been eliminated by the use of languages that are secure by design, for example Java or C# instead of C or C++. However, injection attacks, cross-site scripting, insecure direct object references, broken session management, and failure to restrict URL access remain among the most serious classes of web application vulnerabilities (OWASP, 2007). We believe that the principle of 'secure by design' should be applied at the level of web application frameworks to address those classes of vulnerabilities.

Convention over configuration All configuration should have sensible defaults, in order to minimize the number of detailed decisions that developers need to make for the common cases. This principle was popularized by Ruby on Rails.

In the following sections, we give examples of how existing frameworks violate these principles and explain our proposal for a solution. We omit discussions of how our framework handles input validation and XHTML extensions for making catchy user interfaces, and we only briefly touch upon the relations to other frameworks.

2 Architecture

Web clients in general cannot be trusted, and essentially all web applications contain sensitive data, so according to the 'secure by design' principle and for simplicity our starting point is a classical *server-oriented* approach where the application code is executed on the server rather than on the client. This means that we avoid many of the security considerations that programmers have if using a client-oriented architecture, as e.g. GWT. (A concrete example of this advantage is shown in section 10.2.) One of the typical arguments in favor of a client-oriented approach is the opportunity to create rich user interfaces—however, such effects are possible

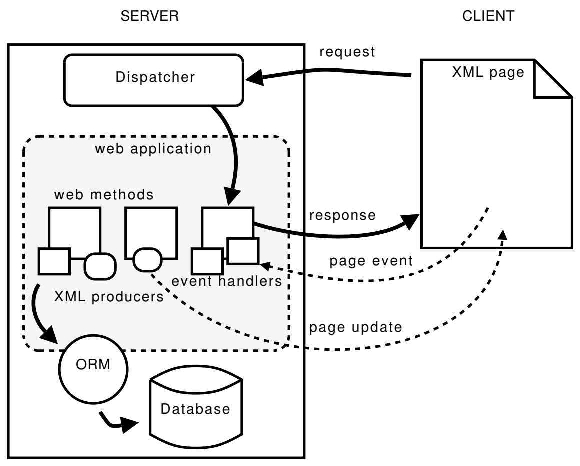


Figure 1: The framework architecture. The boxes and circles represent the main static components, and the arrows show the dynamic interactions between the components.

also in server-oriented frameworks using tag libraries containing JavaScript code. Also, pushing computation to the client-side may imply a decreased load on the server, but, on the other hand, it often conflicts with the use of ORM systems: It is difficult to ensure high performance if the application code is physically separated from the ORM system.

Figure 1 illustrates the basic architecture. As in many other newer server-oriented frameworks, a *dispatcher* on the server receives the client HTTP requests and invokes the appropriate server code, here called *web methods*. Many frameworks rely on separate configuration files for mapping from request URLs to application code. To decrease coupling, our web methods are instead discovered using introspection of the web application code, as in e.g. CherryPy. The web methods have formal parameters for receiving arguments from the clients and return values to be sent as response.

As an example, the tiny web application shown in Figure 2 accepts HTTP GET requests of the form `http://example.org/Main/hello?what=World` and returns a small XHTML page as response. The class `Main` is a web application that contains a single web method `hello`. We explain the XML construction and the `[[...]]` notation in Section 3. Following the 'convention over configuration' principle, no application specific configuration is required to make this run—unlike the situation in, for example, Struts or RIFE (Bevin, 2007).

The architecture is a variant of the Model-View-Controller pattern that many other frameworks also apply. Most importantly, the model (the database) is sep-

```
public class Main extends WebApp {
  public XML hello(String what) {
    return [[
      <html>
        <head><title>Example</title></head>
        <body>
         <p>Hello <{ what }></p>
        </body>
      </html>
    ]];
  }
}
```

Figure 2: A "hello world" web application.

arated from the main web application code. However, we propose a less rigid division between the view and the controller than used in other frameworks, as we explain in Section 3.

In the following sections, we go into more details and extend this basic architecture to fulfill the requirements we defined, which includes introducing the notions of *XML producers* and *event handlers* and the *page event* and *page update* actions appearing in Figure 1.

3 Generating XML Output

A common approach to generating XHTML output dynamically in web application frameworks is to use a template system, either where templates contain code for dynamic construction of content (as in e.g. ASP.NET, PHP, or RoR) or where templates contain placeholders where the code can insert content (as in e.g. RIFE). Another approach is to use GUI components that are assembled programatically (as e.g. GWT or Wicket). Web services, on the other hand, need the ability to also receive and decompose XML data, which is often done completely differently in a DOM-style fashion.

All those systems lack the ability to ensure, at compile time, that output is always well-formed and valid XML data. This is known to affect reliability and may lead to, for example, cross site scripting vulnerabilities.

We propose a solution that (1) unifies the template approach and the DOM-style approach, (2) permits static validation analysis, and (3) avoids many security issues by design.

We build on a new version of the XACT system (Kirkegaard et al., 2004). XML data is represented as well-formed XML fragments that are first-class values, meaning that they can be stored in variables and passed between methods (unlike most other template systems). Figure 2 shows an XML value constant (enclosed by `[[...]]`, using a simple syntax extension to Java) that contains a snippet of code (enclosed by `<{...}>` in XML content or `{...}` in attribute values), which at runtime evaluates to a value that gets inserted. The syntax extension reduces the burden of working with XML data inside Java programs and is desugared to ordinary Java code.

XML values may also contain named *gaps* where other XML values or strings can be inserted, which makes it easy to reuse them. As an example, the following code defines a wrapper for XHTML pages and stores it in a variable `w`:

```
XML w = [[
 <html>
  <head><title>My Pink Web App</title></head>
  <body bgcolor="pink"><[BODY]></body>
 </html>
]];
```

This wrapper can then be used whenever a complete page needs to be generated, for example in this web method:

```
XML time() {
  return w.plug("BODY", [[
   <p>The time is: <b><{ new Date() }></b></p>
  ]] );
}
```

To obtain separation of concerns between programmers and web page designers, XML values can also be stored in separate files. In fact, contracts can be established to formalize and verify this separation, as explained in the article by Böttget et al. (2006).

XML values can also be decomposed and transformed with operations inspired by JDOM and XPath. By design, XML values are always well-formed (i.e. tags are balanced, etc.). When inserting text strings into fragments, special characters are automatically escaped, which eliminates a significant class of security vulnerabilities. In addition, variables can be annotated with XML Schema type information, and a program analysis can perform type checking to ensure that output is always valid according to a given schema (Kirkegaard and Møller, 2005).

As an example (from the CourseAdmin system described in Section 11) in the RPC-style web service category, the following web method uses the features of XACT to produce an XML document describing the status of hand-in exercises for a given student:

```
XML<c:handins> getHandins(XML<c:student> s) {
 String sid =
  db.getStudents().get(s.getAttribute("id"));
 List<Handin> hs = db.getHandins().get(sid);
 if (hs == null)
  throw new NotFoundException();
 XML x = [[ <c:handins> <[H]> </c:handins> ]];
 for (Handin h : hs)
  x = x.plug("H", [[
   <c:handin number={ h.getNumber() }
             status={ h.getStatus() } />
   <[H]>
  ]] );
 return x;
}
```

We here assume that the underlying model is represented by a data structure `db`. By default, web methods react only on HTTP GET requests. This can be changed using an annotation, for example `@POST` for web methods that are unsafe in the HTTP sense.

4 XML Producers and Page Updates

The response of a web method is generally a view of some data from the underlying database. When the data changes, the view should ideally be updated automatically while only affecting the relevant parts of the pages to avoid interfering with form data being entered by the user. With many frameworks, this is a laborious task that requires many lines of code and insight into the technical details of JavaScript and AJAX, so many web application programmers settle with the primitive approach of requiring the user to manually reload the page to get updates.

We propose a simple solution that hides the technical details of the server-push techniques (aka. Comet) and integrates well with the XACT system. An *XML producer* is an object that can produce XML data when its `run` method is called. When the object is constructed, dependencies on the data model are registered according to the observer pattern. An XML producer can be inserted into an XHTML page such that whenever it is notified through its dependencies, `run` is automatically called and the resulting XML value is pushed to all clients viewing the page. All the technical details involving AJAX and Comet are hidden inside the framework, and it scales well to hundreds of concurrent uses.

Typically, the XML producer is created as an anonymous inner class within the web method that generates the XHTML page. This ensures high cohesion for that software component and low coupling by only depending on the model of the data that is shown to the client. An example is shown in Section 6.

5 Forms and Event Handlers

Forms constitute the main mechanism for obtaining user input in web applications. With most frameworks, the code that generates the form is separate from the code that reacts on the input being submitted, and these pieces of code are connected only indirectly via the URLs being generated and the configuration mapping from URLs to code. This violates the principle of high cohesion and low coupling, and the flow of control and data becomes unclear.

Our solution is to extend the architecture with *event handlers* as a general mechanism for reacting to user input. We here focus on forms although the mechanism also works for other kinds of DOM events. The technique is related to the use of action callbacks in Seaside.

Forms are created by building XML documents that contains a `form` tag with relevant input fields. A specialized event handler, called a submit handler, is plugged into the `action` attribute. The submit handler contains a method that can react when the form is submitted and read the form input data. (An example is shown in Section 6.) As with XML producers, event handlers are typically anonymous classes located within the web methods.

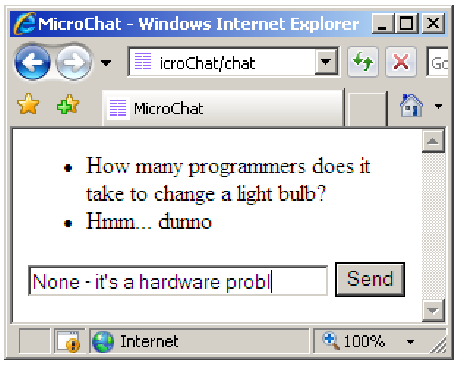
The consequence of this structure is a high degree of code cohesion. Also, it becomes possible by static analysis to verify consistency between the input fields occurring in the form and the code that receives the input field data.

6 Example: MicroChat

The following example shows a tiny chat application that uses the features explained in the previous sections. It shows a list of messages and a text field where users can write new messages (see Figure 3).

```
public class MicroChat extends WebApp {
 List<String> messages =
  new ArrayList<String>();

 public XML chat() {
  return [[
   <html>
    <head><title>MicroChat</title></head>
    <body>
     <{ new XMLProducer(messages) {
       XML run() {
       if (!messages.isEmpty())
         return [[
          <ul>
```



- How many programmers does it take to change a light bulb?
- Hmm... dunno

None - it's a hardware probl

Figure 3: The running MicroChat application.

```
          <{ [[<li><[MSG]></li>]]
             .plugWrap("MSG", messages) }>
         </ul>
        ]];
       else
        return [[]];
    }
    } }>
    <form method="post" action=[SEND]>
     <p>
      <input type="text" name="msg"/>
      <input type="submit" value="Send"/>
     </p>
    </form>
   </body>
  </html>
 ]]
 .plug("SEND", new SubmitHandler() {
  void run(String msg) {
   messages.add(msg);
   update(messages);
  }
 });
}}
```

For simplicity, the application state is here represented as a field `messages` in the application class; we discuss persistence in Section 8. The XHTML document being created when the user invokes the `run` web method contains an instance of an `XMLProducer` that declares itself to be an observer of `messages` and writes the messages in the document. The `plugWrap` method here builds a list of `<li>` items containing the messages. Secondly a submit handler is plugged into the `action` attribute of the form. The handler method reads the form field `msg`, adds it to the list of messages, and then invoke `update` to notify all observers of the list, in this case the XML producer. The effect is that whenever a user posts a new message, the list of messages is automatically updated in all browsers viewing the page.

Notice the high degree of cohesion within the `chat` web method. In most other frameworks (with Seaside and as a notable exception) the equivalent code would be more fragmented and hence less maintainable.

7 Parameters and References to Web Methods

Web methods and event handlers can be parameterized, as shown in the previous examples. Any Java class that contains a static method `valueOf` can be used for such parameters for deserializing values, as known from the basic Java library. Conversely, serialization for output is performed using `toString` (for strings) or `toXML` (for XML values). The dispatcher then transparently handles the serialization and deserialization. For example, JSON data is trivial to transfer with this mechanism.

The URL format used in requests to web methods can be controlled by an annotation, to support REST-style addressing of resources. As an example, the following annotation could be placed on the `hello` web method from Figure 2 to override the default format, such that it can be invoked by URLs like `http://example.org/foo/World`:

```
@URLPattern("foo/$what")
```

Links between pages can be created using the method `makeURL`:

```
XML my_link = [[
 <a href={ makeURL("hello", "John Doe") } >
  click here
 </a>
]];
```

The web method name is passed as a constant string (since methods are not first-class values in Java), but it is straightforward to statically type check that it matches one of the web methods within the application. Parameters are serialized as explained above, and the resulting URL is generated according to the URL pattern of the web method.

This approach ensures that the coupling between web pages becomes explicit in the application source code, in contrast to frameworks that involve URL mappings in separate configuration files. As we explain in the next section, it additionally provides a novel foundation for managing session state and shared state.

8 Session State and Persistence

A classical challenge in web application programming is how to represent session state on top of the inherently stateless HTTP protocol. Although REST prescribes the use of stateless communication, without ever storing session state on the server, we believe that the benefits of allowing session state on the server outweigh the disadvantages—a concrete example is shown in section 10.2.

Many frameworks track clients using, for example, cookies or URL rewriting and provide a string-to-object map for each client for storing session state. Using cookies in this way conflicts with the REST principle that resources should be addressable by URIs: With cookie-based session management, one client cannot participate simultaneously in several sessions of the same web application, and it is difficult to transfer a session from one client to another.

Our approach is to store session state in objects derived from an abstract class named `Session`. Using a class instead of a map means that the ordinary Java type checker will check that expected properties are present when a web method accesses the session state. The class defines the `valueOf` and `toString` methods so the objects can be given as parameters to web methods using the mechanism described in Section 7. Serialization assigns a long random key to each session object, and deserialization finds the session object using this key.

The following example extends the “Hello World” example from Section 2 so it now saves the `what` parameter in a newly created session object and then redirects to the `sayHi` method, which reads the session data and embeds it in its XML output:

```
URL hello(String what) {
 return makeURL("sayHi", new HelloSession(what));
}

class HelloSession extends Session {
 String name;
 public HelloSession(String s) { name = s; }
}
public XML sayHi(HelloSession s) {
 return [[
  <html>
   <head><title>Example</title></head>
   <body><p>Hello <{ s.name }></p></body>
  </html> ]];
}
```

Session state is here encapsulated in the `HelloSession` objects, and it is clear from the signature of the `sayHi` web method that it requires the client to provide a session key. This way of tracking clients is effectively a variant of URL rewriting that is integrated with the dispatching mechanism of the framework.

A session garbage collector thread takes care of removing session objects that are not accessed by a client within a certain period of time. As a convenient extra feature, the generated XHTML pages automatically (using AJAX) inform the server that the relevant session objects are alive as long as the pages are being viewed in a browser.

All web applications and web services, beyond the level of toy examples, encompass a database for data persistence. As mentioned in the introduction, we focus on ORM systems. To this end, we utilize the same pattern as for session state: Persistable data must implement the `Persistable` interface, which requires it to define an ID string for each object. This ID can be used to query the object from the database during deserialization. It can be passed around in the URLs in a call-by-reference style via the `makeURL` mechanism or hidden inside session objects.

Among the currently most serious web application vulnerabilities is *insecure direct object references*, i.e. situations where malicious users can modify data references passed in URLs with insufficient authentication on the server (OWASP, 2007). Employing the 'safe by default' principle, our framework requires that permission must explicitly be given to use an ID of a persistable object. This is done coherently by defining a method named `access` that returns true when the use of the given ID is allowed in the context of the HTTP request concerned. Other issues related to authentication are discussed in the next section.

9 Caching and Authentication

We propose a notion of *filters* that uniformly handles caching, authentication, and logging following the design principles from Section 1. The dispatcher permits a given request URL to match multiple web methods, which are then processed in turn until one produces a response. The order can be controlled by a `@Priority` annotation. A filter is a web method with return type `void` and by default has higher priority than ordinary web methods.

The class `WebApp`, from which all web applications are derived, has a filter named `cache` with a URL pattern that matches all requests. This filter transparently performs server-side caching by storing a number of responses generated by the ordinary web methods. It also handles conditional GET requests to support client-side caching.

The cache can be connected by the observer pattern to the underlying data for removal of stale pages. This currently requires the programmer to specify the dependencies: For example, invoking `addPageInvalidator(x)` ensures that the current page gets invalidated when `update(x)` is invoked. Obviously, this violates the ‘secure by default’ principle since the programmer may forget to specify all dependencies, in which case the users may get stale data. To our knowledge, no existing web programming framework solves this problem; we currently investigate the use of static dependency analysis to address this.

Filters can also be used for decoupling authentication from the application logic:

```
@URLPattern("restricted/**")
public void authenticate() {
  if (!isSecure())
    throw new AccessDeniedException();
  User u = getUser();
  if (u == null ||
      !u.getUsername().equals("jdoe") ||
      !u.getPassword().equals("42"))
    throw new
    AuthorizationRequiredException("MyRealm");
  else
    next();
}
```

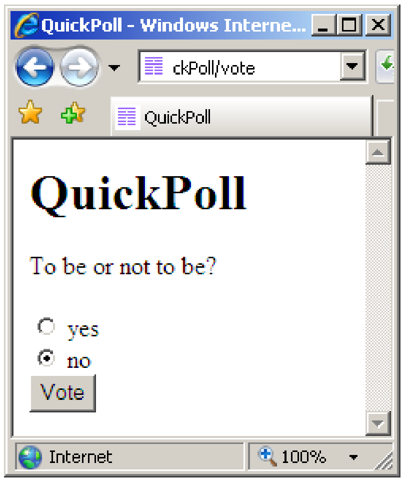
This filter restricts access to all resources with URLs matching the pattern `restricted/**` relative to the base URL of the web application. If the connection is insecure (i.e. not using SSL/TLS), a 403 Forbidden response is made. The `getUser` method returns the credentials provided via HTTP Basic authentication. If the user is not authorized, a 401 Unauthorized response is made to request the client for acceptable credentials. Otherwise, control is passed to the next web method in line using the `next` method.

10 Additional Examples

This section provides a two examples showing how the techniques are used to build an small web application. The QuickPoll example demonstrates the use of the server push techniques and the GuessingGame example demonstrates how sessions can be used to curb the control flow in a web application. The source code of these two examples can be found in the appendix.

10.1 QuickPoll

The first example, QuickPoll, is a poll system. The web method `main` produces a menu page; `init` is used by the poll administrator to set the question to be asked; `vote` produces a page containing the question and a yes/no form and processes the



QuickPoll

To be or not to be?

◯ yes
◉ no
Vote

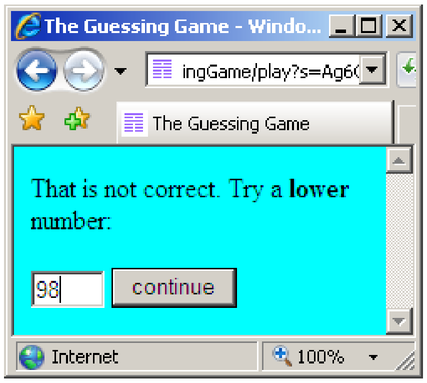
Figure 4: Voting in QuickPoll.

user's choice; and finally, `results` shows an overview of the votes. It uses all the framework features covered by Sections 2–7—in particular, two submit handlers and one XML producer. The application state could easily be made persistent, as explained in Section 8, and authentication could be added as in Section 9.

10.2 GuessingGame

The second example, GuessingGame, is the classical toy where each user must guess a number between 1 and 100 in as few attempts as possible. The `start` web method creates a new session object, corresponding to a user starting a new game, and immediately redirects to `play`. The session object contains the secret number to be guessed, the number of attempts, and also a snapshot of the current XHTML page. The `play` web method simply shows the page of the given session. The main functionality is located in the constructor of the session object: it first creates the initial XHTML page and then uses submit handlers together with the features of XACT (that we shall not explain in detail here) to modify the page contents when the users provides input. Finally, `record` shows the total number of plays and the current record holder. The global game state is stored in a persistable `GameState` object. The source code for the `GameState` class it not included here, since it belongs to the data model and does not contain any web-specific functionality.

Note that all the web methods react on GET requests whereas the submit handlers use POST, reflecting the fact that the former are safe (in the HTTP sense)



That is not correct. Try a **lower** number:

98 continue

Figure 5: Playing the GuessingGame.

unlike the latter which have side-effects. In particular, a URL for the `play` method, which contains a session key as parameter, can always be used to get the current page of the session.

We here omit a thorough comparison of how web applications with similar functionality could be programmed with other frameworks. However, we claim that the benefits discussed in the previous sections regarding code structure and maintainability also apply to these examples. In particular, the functionality of GuessingGame is cumbersome to express without storing session state on the server since it is crucial that the users are not able to manipulate it or backtrack during a game.

11 Case Study: CourseAdmin

To evaluate the framework in a more realistic setting and to test it for programming applications beyond toy examples we have used it to develop the application CourseAdmin. This is a medium sized web application (more than 20,000 lines of code) for handling course administration at our institute. The application includes management of assignments, a webboard, and attendance counting. The application has close to 1200 users with about 200 daily unique users. A part of the system is written using the XML syntactic sugar while the rest is written in pure Java, and even without the extra convenience of the syntactic extension, the XML model remains usable.

CourseAdmin is structured as four distinct web applications with a common model layer: the public web pages of classes and students, the course configuration application for course staff, an application for students to view their status and upload assignments, and finally a configuration application for creating new courses. High cohesion exists in this architecture because the web applications share common web methods, including authentication filters, by inheriting from common abstract super classes, and the division of the system into multiple web applications results in low coupling between unrelated program parts. This indicates that the patterns presented in the small example programs in this paper can be used to build much larger systems.

The main XML templates are located in separate files, and layout is controlled using CSS, which makes the page design consistent and easy to modify.

The 'convention over configuration' approach means that CourseAdmin only needs a handful of configuration lines for database credentials and email server names.

The 'secure by default' principle is important in CourseAdmin, since user provided content in the webboard is shared between the clients, and the system contains confidential data about student grades. By the use of the `access` mechanism for protecting data in the model layer as explained in Section 8 and authentication filters as in Section 9, all necessary security checks are collected in one cohesive component, and new functionality can be added to the system without risking violations of the existing policies.

12 Conclusion

We have presented a novel minimalistic Java-based framework that provides uniform support for common tasks in web programming. By the use of a reflection-based *dispatcher*, XACT for XML processing, *XML producers* for server-push communication, *event handlers* for user input processing, and *filters* for caching and authentication, we obtain a framework for making maintainable and secure web applications with high cohesion, low coupling, and security-by-design.

As ongoing and future work, we aim to provide a more detailed analysis of the capabilities and limitations of other frameworks and their relation to the one we have presented here. Also, we remain to show how the framework can handle user input validation through a combination of XML producers and event handlers and rich user interfaces by integrating existing JavaScript libraries using tag-like XHTML extensions.

REFERENCES

Bevin, G. (2007). RIFE. `http://rifers.org/`.

Crockford, D. (2006). JavaScript Object Notation (JSON). RFC4627. `http://tools.ietf.org/html/rfc4627`.

Ducasse, S., Lienhard, A., and Renggli, L. (2007). Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150.

Google (2008). Google Web Toolkit. `http://code.google.com/webtoolkit/`.

Hansson, D. H. et al. (2008). Ruby on Rails. `http://www.rubyonrails.org/`.

Johnson, R. et al. (2008). Spring MVC. `http://www.springframework.org/`.

Kirkegaard, C. and Møller, A. (2005). Type checking with XML Schema in XACT. Technical Report RS-05-31, BRICS. Presented at Programming Language Technologies for XML, PLAN-X '06.

Kirkegaard, C., Møller, A., and Schwartzbach, M. I. (2004). Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192.

Krishnamurthi, S., Hopkins, P. W., McCarthy, J. A., Graunke, P. T., Pettyjohn, G., and Felleisen, M. (2007). Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460.

Lerdorf, R. et al. (2008). PHP. `http://www.php.net/`.

McClanahan, C. R. et al. (2008). Struts. `http://struts.apache.org/`.

Microsoft (2008). ASP.NET. `http://www.asp.net/`.

O'Neil, E. J. (2008). Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD '08*.

OWASP (2007). The ten most critical web application security vulnerabilities. `http://www.owasp.org/index.php/Top_10_2007`.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

W3C (2008). Web services activity. `http://www.w3.org/2002/ws/`.

APPENDIX

This appendix contains the source code for the GuessingGame and QuickPoll example web applications programmed with our framework. A couple of screenshots are shown in Fig. 4 and 5. Our primary interest here is code maintainability, so we settle for a primitive design of the XHTML pages. (As explained in Section 3, separation of concerns between XHTML page design and Java code can be accomplished by moving the XML templates into separate files.)

Example: QuickPoll

```
public class QuickPoll extends WebApp {

 private 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();

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

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

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]>
   <p>
    <input name="vote" type="radio" value="yes"/>
    yes<br/>
    <input name="vote" type="radio" value="no"/>
    no<p/>
    <input type="submit" value="Vote"/>
   </p>
  </form>
 ]]).plug("VOTE", new SubmitHandler() {
  XML run(String vote) {
   if ("yes".equals(vote))
    state.yes++;
   else if ("no".equals(vote))
    state.no++;
   update(state);
   return wrapper.plug("BODY", [[
    <p>Thank you for your vote!</p>
   ]]);
```

```
    }
   });
  }

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

Example: GuessingGame

```
public class GuessingGame extends WebApp {

 private XML wrapper = [[
  <html>
   <head><title>The Guessing Game</title></head>
   <body style="background-color: 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", [[
    <p id="MSG">
      Please guess a number between 1 and 100:
    </p>
    <form method="post" action=[GUESS]>
     <p>
      <input name="guess" type="text" size="3"/>
      <input type="submit" value="continue"/>
     </p>
    </form>
   ]]).plug("GUESS", new SubmitHandler() {
    void run(int guess) {
     guesses++;
     if (guess != number)
      page = page.setContentOfID("MSG", [[
       That is not correct. Try a
       <b><{guess>number?"lower":"higher"}></b>
       number:
      ]]);
     else {
      boolean record =
         game.getHolder() != null &&
         guesses >= game.getRecord();
      final XML thanks = [[
       <p>
       Thank you for playing
       this exciting game!
      </p>
     ]];
     page = wrapper.plug("BODY", [[
      <p>
       You got it, using
       <b><{guesses}></b> guesses.
      </p>
      <{ !record ? thanks : [[
        <p>
         That makes you the new
         record holder!
        </p>
        <p>
         Please enter your name for
         the hi-score list:
        </p>
        <form method="post" action=[RECORD]>
         <p>
          <input name="name" type="text" size="20"/>
          <input type="submit" value="continue"/>
         </p>
        </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 = wrapper.plug("BODY", thanks);
        }
       })
      }>
     ]]);
    }
  };
 });
}

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

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

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