Keywords: Web programming, Web services, Java, XML, XHTML, AJAX, REST.

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, server-push/Comet, 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 can be summarized as follows:

• First, we identify and argue for a small collection of requirements and design principles for creating a Java-based 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 3–8 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, which is evolved from the 2003 version of JWIG (Christensen et al., 2003), provides a clear structure of both control and data. This 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 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 users. Our framework design should focus on this category.

The framework must uniformly support both XHMTL 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. 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:

\textbf{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.

\textbf{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. Due to the limited space, many features of our resulting system are omitted in this paper; for an extended version see (Møller and Schwarz, 2009).

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
public class Main extends WebApp {
    public XML hello(String what) {
        return [[
            <html>
            <head><title>My Pink Web App</title></head>
            <body>
            Hello {{ what }}
            </body>
            </html>]
    }
}

Figure 2: A “hello world” web application.

means that we avoid many of the security considera-
tions that programmers have if using a client-oriented
architecture, as e.g. GWT. A concrete example of this
advantage is shown in (Møller and Schwarz, 2009).
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
also in server-oriented frameworks using tag libraries
containing JavaScript code. Also, pushing computa-
tion 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 dis-
patcher on the server receives the client HTTP re-
quests and invokes the appropriate server code, here
called web methods. Many frameworks rely on sepa-
rates structure files for mapping from request
URLs to application code. To decrease coupling, our
web methods are instead discovered using introspec-
tion of the web application code, as in e.g. CherryPy.
The web methods have formal parameters for receiv-
ing 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 is separated from
the main web application code. However, we propose
a less rigid division between the view and the con-
troller 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 require-
ments we defined, which includes introducing the no-
tions of XML producers and event handlers and the
page event and page update actions appearing in Fig-
ure 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 com-
ponents 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 vul-
nerabilities.

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

49

that gets inserted. The syntax extension reduces the
burden of working with XML data inside Java pro-
grams 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 fol-
lowing 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>
}]}
This wrapper can then be used whenever a complete page needs to be generated, for example in this web method:

```javascript
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).

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 page 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. Due to the limited space, 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 sec-
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 [[
                                    <$ul>
                                    <![<li><[MSG]></li>]]
                                    .plugWrap("MSG", messages) |
                                    <$ul>]];
                        else
                            return [[]];
                    }">
                </form>]
                .plug("SEND", new SubmitHandler() {
                    void run(String msg) {
                        messages.add(msg);
                        update(messages);
                    }
                });
            </body></html>]];

        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. 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 (Møller and Schwarz, 2009).

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. 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:

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

```java
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.

For persistent data in the web applications, 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.

9 CONCLUSIONS

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. Example programs and a larger case study are presented in the technical report (Møller and Schwartz, 2009).

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


