Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses | Computer Science Department

5-2-2007

J2Components: A Java Web Application Framework

Samuel Hughes
Trinity University

Follow this and additional works at: http://digitalcommons.trinity.edu/compsci_honors

Part of the Computer Sciences Commons

Recommended Citation
Hughes, Samuel, "J2Components: A Java Web Application Framework" (2007). *Computer Science Honors Theses*. 17.
http://digitalcommons.trinity.edu/compsci_honors/17

This Thesis open access is brought to you for free and open access by the Computer Science Department at Digital Commons @ Trinity. It has been accepted for inclusion in Computer Science Honors Theses by an authorized administrator of Digital Commons @ Trinity. For more information, please contact jcostanz@trinity.edu.

J2Components: a Java Web Application Framework

Samuel Hughes

A departmental thesis submitted to the

Department of Computer Science at Trinity University

in partial fulfillment of the requirements for Graduation.

May 2nd, 2007

Thesis Advisor

Department Chair

Associate Vice President

for

Academic Affairs

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-nd/2.0/>or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford,

California 94305, USA.

Acknowledgments

I would like to thank Dr. John Howland for advising and helping throughout the research process.

J2Components: a Java Web Application Framework

Samuel Hughes

Contents

1 Introduction and motivation **8**

2 History of Java web frameworks **10**

2.1 J2EE 10

2.2 JavaServer Faces 11

2.3 Spring 13

2.4 Tapestry 13

3 The problem **16**

3.1 The problem 16

3.2 XML configurations 17

3.3 Ruby on Rails 18

3.4 An alternate solution 18

4 J2Components introduced **20**

4.1 Overview 20

4.2 A visual run through 21

4.2.1 What to note 21

4.2.2 The example 21

4.2.3 Page configurations 27

5 J2Components **29**

5.1 J2Components structure 29

5.1.1 The `folders` package 29

5.1.2 Page classes 29

5.1.3 Dispatcher 30

5.1.4 Model 31

5.2 Page configurations 31

5.3 Arguments and methods 33

5.3.1 General arguments 33

5.3.2 Arguments passed into a form 34

5.3.3 Arguments passed into links 35

5.3.4 URL methods versus HTML methods 37

5.4 Forms 37

5.4.1 Basic forms 37

5.4.2 Model forms 38

5.5 Template parser 41

6 Conclusion **42**

6.1 Future additions to J2Components 42

6.1.1 Better page-view interaction 42

6.1.2 Larger library 43

List of Figures

2.1 Simple application.xml file. This ties together the separate tiers in a J2EE application 11

2.2 A simple example of JSF components in a JSP page. 12

2.3 A managed bean used in JavaServer Faces 12

2.4 Simple example of springapp-servlet.xml used my its accompanying springapp web application. 14

4.1 HTML files and their corresponding `Page` classes. 22

4.2 `MyPage.html` in detail. 23

4.3 `org.folders.test.MyPage` class in detail. 23

4.4 The output of a request for MyPage. Notice the additional text that comes from the page layout. 24

4.5 An example of a page layout. In this example, the page content is `MyPage.html`. 25

4.6 Configuration methods being called in the `MyPage` class. 26

4.7 The `pageLayout()` configuration method being called in a `Config` class. `Config`'s methods can also be overidden by any of the page classes. 28

4.8 The `pageStylesheet()` method being applied to `TestLayout.html`. 28

5.1 Default.html 32

5.2 Example of a URL request used in J2Components 33

5.3 Example of Types.properties file. 34

5.4 Example of a simple form 35

5.5 Methods for links and redirects. 36

5.6 Form tags listed. 37

5.7 myForm() is an example method in org.folders.test.MyPage sample class. . 38

5.8 This shows form input added and saved to the Honey table. saveHoneyData() is found in the example org.folders.test.TestPage class. 39

5.9 The Honey model used in Figure 5.7 40

Chapter 1

Introduction and motivation

Web application frameworks have quickly escalated into a popular topic among bleeding edge web programmers. With the release of Ruby on Rails in 2004[1], more and more web programmers have abandoned their current "favorite" language for a newer, dynamically typed Ruby language. This new paradigm in the web framework world points out, among other things, how unnecessarily bloated Java web frameworks have become. With an old fashion idea that Java and XML work as a team, it seems that Java web programmers write two programs: one in Java, and a copy in XML. People typically prefer to think one way or the other, especially when XML is actually adding functionality rather than strictly configuration. XML is essentially an additional programming language to learn, and depending on the framework, the developer must learn its XML documentation in order to determine the available attributes that may be used. This on top of trying to learn where to place specific XML files.

This hasn't caused too much disdain among Java programmers, as it is still the number one popular programming language by a large margin, as reported by TIOBE Programming Community in January 2007[13]. But Java's momentum is dwindling[13].

What inspired J2Components is the idea that a Java web framework can be just as flexible as a Ruby framework. Granted, Java is statically typed and Ruby dynamically typed, which makes a strong case for why Java may only survive with the help of a description language such as XML. But this is an old-fashioned bias that has been ingrained in developers' heads over the years. Others have tried to combat this dilemma by making XML configurations easier, as pointed out in the next chapter, but J2Components chooses to start from square one. J2Components is as flexible, easy, and clean as Ruby on Rails, but it still maintains the strength and stability of a statically typed language, i.e. Java.

To fully appreciate J2Components, one must have a basis. The next chapter brushes over a few of the more successful Java web frameworks. Keep in mind that this summary barely scratches the surface of all Java web frameworks in existence today. That notion, in and of itself, helps explain the apparent difficulty in making a successful Java web framework; especially since a majority do so through use of XML. After Java's history, Ruby on Rails will be briefly mentioned. Lastly, J2Components is described in detail, with comparisons to the previous frameworks. In doing so, the topic should hopefully not be foreign, and as a result, emphasize certain values potentially obtained by use of J2Components

Chapter 2

History of Java web frameworks

2.1 J2EE

J2EE (Java 2 Enterprise Edition) is one of the original Java web enterprise frameworks. Developed by Sun Microsystems in 2001, it was released as 1.3 and the JSR 58 standard[6]. J2EE stands out, especially in the blogger community, for it's seemingly "bloated" use of XML as a solution to configuration. In 2006, Sun Microsystems released Java EE 5 to combat these issues[14]. They did this through use of JMI (Java Metadata Interface)[4], which injects XML directly into the Java code. While more convenient than independent XML files, it is still XML configuration.

The philosophy behind J2EE is a three tier model: the client, web, and business tier[2]. Each tier is tied together through XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<application version="5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application_5.xsd">
   <display-name>SampleApp</display-name>
   <module>
      <web>
         <web-uri>SampleApp-war.war</web-uri>
         <context-root>/SampleApp-war</context-root>
      </web>
   </module>
   <module>
      <ejb>SampleApp-ejb.jar</ejb>
   </module>
</application>
```

Figure 2.1: Simple application.xml file. This ties together the separate tiers in a J2EE application

2.2 JavaServer Faces

JavaServer Faces (JSF) is another Java standard (JSR 127 and JSR 252)[5]. JSF attempts to add a simple visual component library that is mixed into JavaServer Pages (JSP), which are a hybrid of Java and Hypertext Markup Language (HTML)[3]. JSF is the recent buzz as far as the direction standard Java Enterprise and web applications are headed. The ultimate goal is to create a distinct separation between model, view, and controller, where model can be an Enterprise Javabean (EJB), the view comes from the JSF components, and the controller is the context manager, which is described shortly.

A downside to JSF is the lack of flexibility that results from using the limited JSF component libraries. Tapestry, a framework mentioned later is somewhat of a solution to this in that their components are attributes injected into HTML through use of id's.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:view>
<h:panelGrid columns="2" rendered="#{poll.showScore}">
<h:outputText value="Total votes:" />
<h:outputText value="#{poll.total}" />
<h:outputText value="It rocks!" />
<h:outputText value="#{poll.vote1Score}%" />
<h:outputText value="It seems okay" />
<h:outputText value="#{poll.vote2Score}%" />
<h:outputText value="Terrible" />
<h:outputText value="#{poll.vote3Score}%" />
</h:panelGrid>
</f:view>
```

Figure 2.2: A simple example of JSF components in a JSP page.

```
<managed-bean>
   <managed-bean-name>personBean</managed-bean-name>
   <managed-bean-class>org.PersonBean</managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Figure 2.3: A managed bean used in JavaServer Faces

Another downside, which was initially introduced as a benefit, is managing the object life cycles through use of *faces-config.xml*. The original benefit is that it adds the ability to determine the life cycle of objects used in the web application. The downside is that it becomes redundant as Figure 2.3 displays. For each getter/setter method that is called inside of a JSP page, it should be described by a managed bean.

JSF is relatively small and not yet capable of being a "complete package". In other words, success with JSF is often gained through its use along with other web frameworks. The reason J2Components framework does not build on JSF to make it more mature is because even when JSF gets to that point, the managed bean philosophy still exists. And, at least relative to frameworks that use a dynamically typed language like Ruby on Rails[15], this will always be a slower process to maintain and build.

2.3 Spring

The Spring Framework[9] uses the Inversion of Control design pattern. To summarize, Spring has XML files that manually inject arguments into methods of classes to be used at runtime. The design pattern itself is helpful in that it disassociates the data input from the class design. The downside is the overwhelming use of XML. Figure 2.4 shows a small sample of a Spring application's XML servlet configuration.

The Spring Framework has also grown exponentially in size, and can be intimidating for developers beginning a new project.

2.4 Tapestry

Tapestry[16] is most similar to J2Components. Tapestry uses a component model that has a class for each HTML page. What especially separates J2Components from Tapestry is

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="springappController" class="web.SpringappController">
    <property name="productManager">
     <ref bean="prodMan"/>
    </property>
  </bean>

  <bean id="prodMan" class="bus.ProductManager">
    <property name="products">
      <list>
        <ref bean="product1"/>
        <ref bean="product2"/>
        <ref bean="product3"/>
      </list>
    </property>
  </bean>

  ...

  <bean id="messageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>messages</value></property>
  </bean>

  <bean id="urlMapping"
  class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/hello.htm">springappController</prop>
      </props>
    </property>
  </bean>

  <bean id="viewResolver"
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
  </bean>
</beans>
```

Figure 2.4: Simple example of springapp-servlet.xml used my its accompanying springapp web application.

Tapestry's use of ".page" files, which are XML configuration files to describe the data being passed from page class to the HTML file.

It has an advantage over JSF in that it uses HTML rather than JSP files, so it is potentially more expressive. But it still makes use of XML, which becomes repetitive like a managed bean in JSF.

Chapter 3

The problem

3.1 The problem

It should be relatively clear that Java web frameworks can be unnecessarily bloated. Unfortunately, the last chapter does not do this statement its justice since it only summarizes the frameworks and points out a few weak spots. But justice is still served upon realization that everything described in the last chapter refers to configurations that are not necessary. Describing the way an object behaves should happen inside the class; that is what a class is for. It should not occur additionally in an XML file as Figure 2.3 illustrated.

Java has decreased 2.92% in popularity since April of 2006, as recorded in April of 2007; the largest decrease among the top twenty languages. Ruby increased 2.31% since April of 2006, as recorded in April of 2007; the largest increase among the top twenty languages. It could be argued that a change in taste from static languages to dynamic languages is occurring, but as TIOBE also reports, the D programming language, which is statically typed, has increased in popularity by 1.03%, which moves it up five spots to fourteenth since last year[13]. Additionally, in the web application world, it's an unwritten law that

Java web applications play in an entirely different ball park. But is that necessary?

To reiterate, Figure 2.3 showed a managed bean. These are not too often labeled as an annoyance, but they still beg the question: *what for?* It's describing an object of a class. Another way to describe an object of a class is to naturally instantiate it in Java code, i.e. the old fashioned way. A line needs to be drawn for how far configurations can go before they replace the program code altogether. In theory, a program can be entirely composed of configuration files if the framework knows how to handle each particular configuration file. It is a slippery slope that creates ambiguity for where responsibility lies in an application. Is it the programmer or the framework?

3.2 XML configurations

Configuration files are of course still beneficial to an application. For example, within GUI applications, they are a helpful way to adjust layout changes without the need to recompile. A more specific category of configurations should be focused on when regarding them as a problem. That is XML. When used in Java web application frameworks, it has a history of essentially being an alternate way to read Java, but popular opinion would claim it less readable. The Apache Ant Project[11], an Apache Software Foundation[12] standard, is an example of XML's complexity. The task library and its attributes are literally correlated to Ant's Java library, posing the question: *why not use its Java library?* Of course, it does eliminate the need for compilation. In that case, it could be regarded as an inflexible Java interpreter that requires a parser for each set of interpreted XML code. This of course is out of the question.

3.3 Ruby on Rails

Ruby on Rails[15] came on to the scene as the hero[10]. Its philosophy is to remove the constant repetition that, in many ways, has taken over the web application world. For years, GUI applications have stuck closely to the Model-View-Controller paradigm. This has not been the case with web applications. Ruby on Rails' goal was to fix this problem. In addition, it aimed to fix the burden of slow project start up time that is a byproduct of the multiple XML configurations needed to get a Java web application, more specifically, a J2EE application, up and running.

One technique is by use of a `generate` script, which includes calls such as:

```
./script/generate controller my_controller
```

or:

```
./script/generate model my_model
```

These commands write skeleton code for the particular options. In this case, the controller and model. Even without the script, there is not much code connected with either the controller or model alone, but its value becomes more visible when additional customization options are needed.

3.4 An alternate solution

Ruby on Rails is an easy framework to work with. It has taken an ingenious approach to a problem that historically struck out more than once. It'd be natural to assume that much of its success is attributed to the Ruby programming language, a dynamically typed language with simplicity at its core. The goal of J2Components is to prove this is not necessarily the

case. A sense of lost confidence for Java has grown over the years, but the language has changed little in its philosophy besides minor improvements. This bias is partly the result of complicated attempts at web framework models.

Chapter 4

J2Components introduced

4.1 Overview

The philosophy of J2Components is a hybrid of two web frameworks: Ruby on Rails and Tapestry[16]. As previously mentioned, Tapestry uses a component model with each page also represented by a Java class. Ruby on Rails is closer in line with the model-view-controller principle by use of its `ActionRecord, ActionView, and ActionController` class structure. J2Components attempts at bringing the two together.

Each individual page in the web application can be traced back to a sub package of `folders` and a subclass of `org.framework.Page`. This is similar to Ruby on Rails' `:action, :controller` model, where the `:action` and the `:controller` can be visually traced back using the web application's URL. Where it differs is, in Ruby on Rails, the controllers are subclasses of `ActionController::Base`, and its methods represent the HTML pages. J2Components instead makes use of a package, which contains `org.framework.Page` classes. Each `Page` class also has a corresponding HTML page. This is similar to Tapestry's component model, where a class represents each HTML page. It differs from Tapestry in that it

adds the package element, which can still directly be traced back in the URL. For example:

```
http://<your-site>/Dispatcher?folder=test&page=TestPage
```

is directly linked to:

```
org.folders.test.TestPage
```

and its corresponding HTML page:

```
./<your-site-folder>/web/folders/test/TestPage.html
```

More details will be described in the next chapter, which covers the J2Components structure.

4.2 A visual run through

4.2.1 What to note

Making sense of the way J2Components works is best done with a visual run through and explanation. Throughout this section, there are visual representations. This run through only highlights a small subset of what J2Components can do, but it helps give a general basis for understanding the structure and organization involved.

4.2.2 The example

Figure 4.1 shows the basic folder structure of a J2Components web application. There is a web folder that contains HTML files and layout configurations, and there is a source folder that contains the corresponding classes.

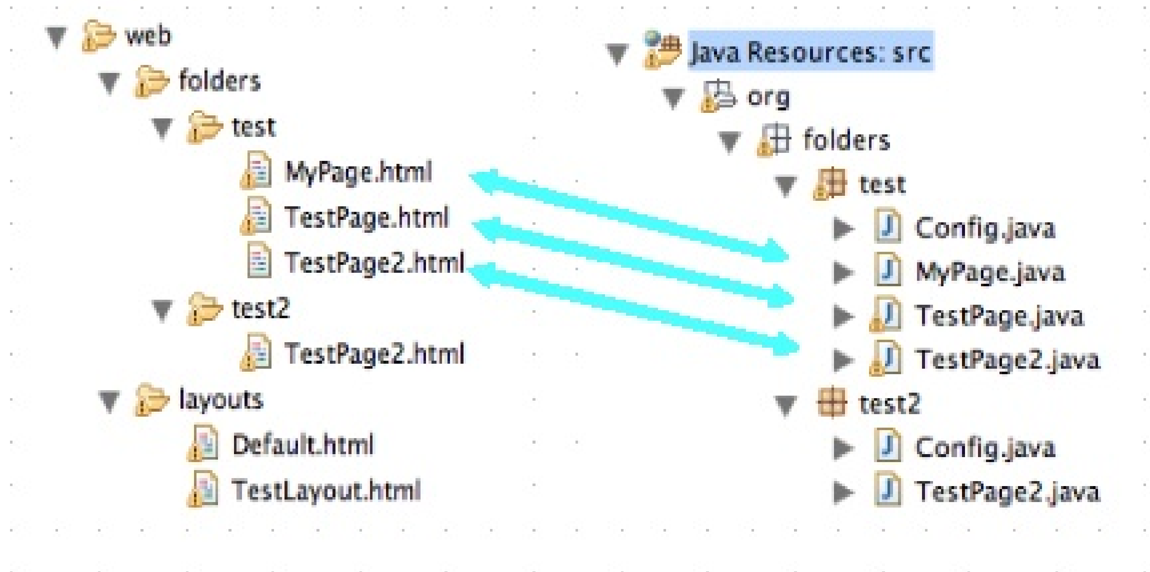

Figure 4.1: HTML files and their corresponding `Page` classes.

In this example, a URL request for `http://⟨your-site⟩/Dispatcher?folder=test&page=MyPage` is made. Thus, `MyPage.html` and the `org.folders.test.MyPage` class are used. Figure 4.2 and 4.3 show these two files in closer detail. The output of the request is displayed in Figure 4.4. But as Figure 4.4 also highlights, there is additional text that comes from neither the `MyPage.html` nor the `org.folders.test.MyPage` class. This is an example of text displayed in the page layout. Figure 4.5 shows this case's particular page layout in detail.

A page layout is helpful when creating a general web application theme. The menus and site font colors can be written once in a page layout, and all pages using this theme can be directed to do so by calling for the particular page layout. Figure 4.6 shows an example of `MyPage` calling for the use of the `TestLayout.html` layout.

```
This is MyPage.html <br>

<% methodOne %><br>
<% methodTwo %><br>
<% methodThree %><br>
```

Figure 4.2: `MyPage.html` in detail.

```
public class MyPage extends Config {

    public String pageLayout() {
        return "TestLayout.html";
    }

    public String pageStylesheet() {
        return "Default.css";
    }

    public void methodOne() {
        out.print("This is method one!");
    }

    public void methodTwo() {
        out.print("This is method two! Ah!!");
    }

    public void methodThree() {
        out.print("This is method three!");
    }

}
```

Figure 4.3: `org.folders.test.MyPage` class in detail.

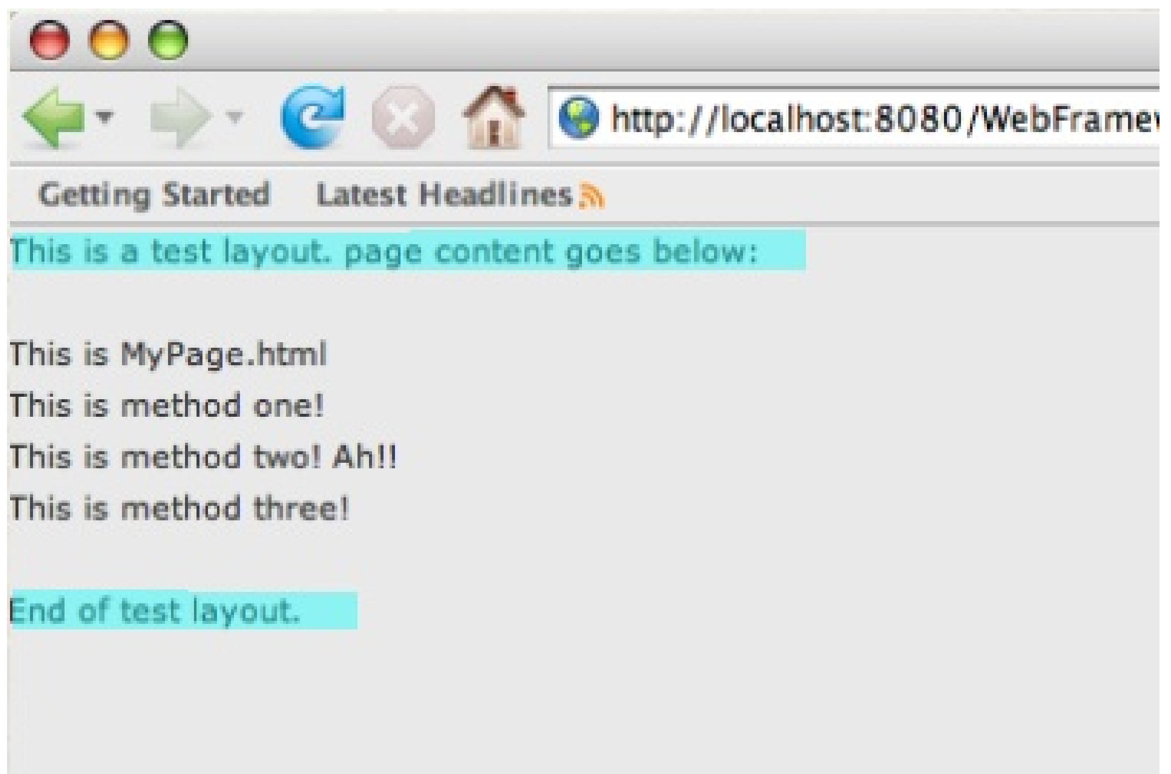

Figure 4.4: The output of a request for MyPage. Notice the additional text that comes from the page layout.

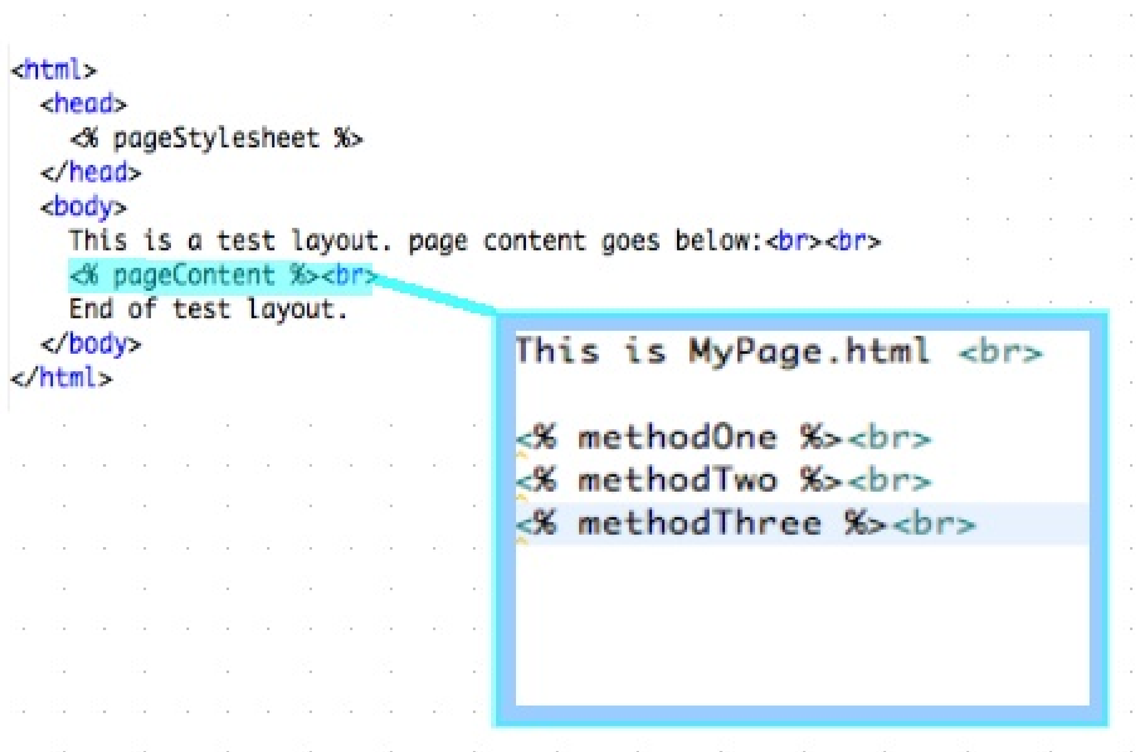

Figure 4.5: An example of a page layout. In this example, the page content is `MyPage.html`.

```java
public class MyPage extends Config {

    public String pageLayout() {
        return "TestLayout.html";
    }

    public String pageStylesheet() {
        return "Default.css";
    }

    public void methodOne() {
        out.print("This is method one!");
    }

    public void methodTwo() {
        out.print("This is method two! Ah!!");
    }

    public void methodThree() {
        out.print("This is method three!");
    }

}
```

Figure 4.6: Configuration methods being called in the `MyPage` class.

4.2.3 Page configurations

The `pageLayout()` method is an example of a configuration method. By J2Components design, all methods prefixed with "page" are J2Components configurations. Again, as Figure 4.6 highlights, configurations can be called in a `Page` class. But if for example, the `TestLayout.html` layout is used for all pages in the `test` folder/package, this can be configured in the `folders.test.Config` class, as Figure 4.7 displays. By J2Components design, every sub package of the `folders` package contain a `Config` class for configurations that apply to all classes in the package.

Another configuration method that can be configured in a `Page` class (or `Config` class), is `pageStylesheet()`. This method returns the stylesheet to be used for the given `Page` class. While the actual configuration is described in the `Page` class (or `Config` class), it also must be directed for use in the page layout. Figure 4.8 shows the `pageStylesheet()` method being called in the `TestLayout.html` layout.

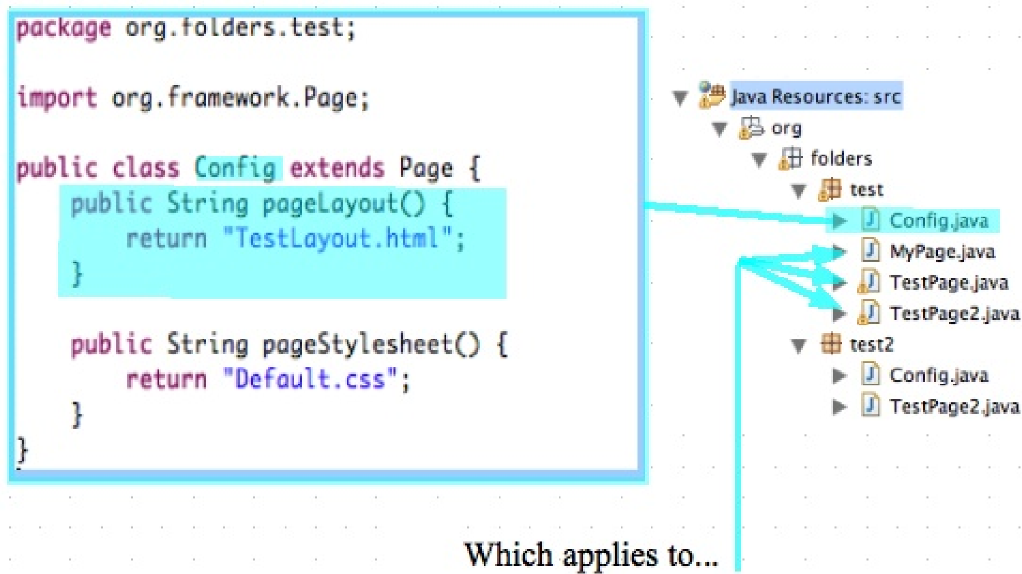

Figure 4.7: The `pageLayout()` configuration method being called in a `Config` class. `Config`'s methods can also be overidden by any of the page classes.

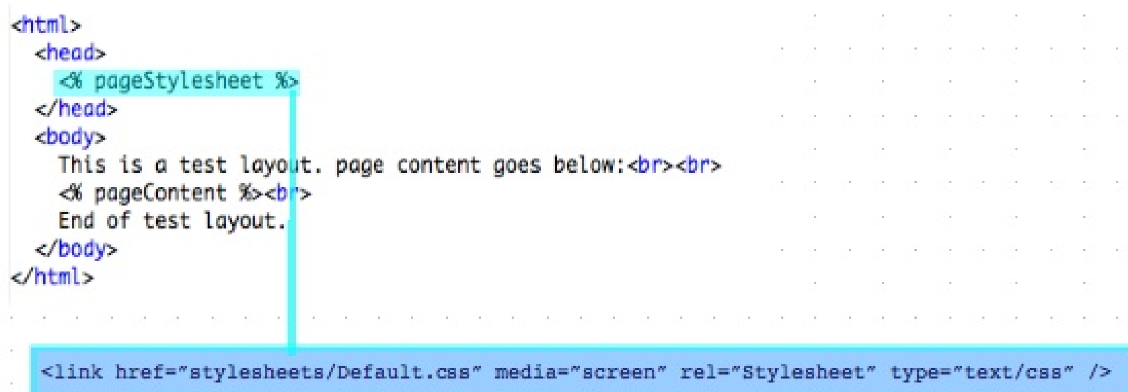

Figure 4.8: The `pageStylesheet()` method being applied to `TestLayout.html`.

Chapter 5

J2Components

5.1 J2Components structure

5.1.1 The `folders` package

There is one package named `folders` where packages and Java classes for a J2Components web application are located. Every subpackage of `folders` is what gets passed into the URL. If a request for:

```
http://<your-site>/Dispatcher?folder=test&page=TestPage
```

is made, J2Components knows to use the package `folders.test` as instructed by `folder=test` in the URL request.

5.1.2 Page classes

In each sub package of the `org.folders` package, reside the page classes. In the earlier request:

```
http://<your-site>/Dispatcher?folder=test&page=TestPage
```

TestPage is a class inside `org.folders.test`. To put the pieces together, when the URL request is made, J2Components knows to instantiate the `org.folders.test.TestPage` class. The page class is a subclass of `org.framework.Page`. More details of what is involved with the Page class will be described in a later chapter.

In addition to the Page classes, there are HTML files that directly correlate each Page class. So for:

```
org.folders.test.TestPage
```

there is also:

```
./<your-site-folder>/web/folders/test/TestPage.html
```

Any method that is referenced in the HTML file is, by default, a method of it's correlating class. Again, more details will be described in a later chapter.

5.1.3 Dispatcher

The Dispatcher class receives every request made to the J2Components web application. It must decide the steps needed based on the URL request. In a Java web application:

```
javax.servlet.http.HttpServlet
```

handles client requests made to the server. The HttpServlet is also in charge of creating threads for each client's session. Since the Dispatcher inherits from HttpServlet, it comes with this functionality by default. In addition, the Dispatcher class initializes the Page class environment, which includes passing:

```
static javax.servlet.http.HttpServletRequest
```

and:

```
static javax.servlet.http.HttpServletResponse
```

into the current Page class.

The Dispatcher class is also in charge of parsing the correlating Page class HTML files, which were described in the last section. This is done by use of:

```
org.framework.TemplateParser
```

which parses the HTML file in search of Page class methods in between `<%...%>` tags.

5.1.4 Model

In keeping in line with the Model-View-Controller principle, a data model should be described as part of the J2Components structure. This is the area of J2Components that needs the most expansion, but a foundation has still been set. Similar to Ruby on Rails, J2Components's model is self sufficient. Each database table is represented by a subclass of:

```
org.framework.Model
```

which inherits data search methods and data saving/updating methods.

5.2 Page configurations

In Figure 5.1, both inserted methods are prefixed with "page". These are methods specifically reserved for J2Components configurations. For example, TestPage needs "LayoutX.css" rather than "Default.css", TestPage override the method `pageStyleSheet` in:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
<% pageStylesheet %>
<title></title>
</head>
<body>
<% pageContent %>
</body>
</html>
```

Figure 5.1: Default.html

```
org.folders.test.TestPage
```

having it return the string, "LayoutX.css". This act of overriding for a specific page also works for all classes in a sub package of `folders`. This is possible because all classes in a sub package of `folders` are a subclass of that current package's Config class, which is a subclass of the Page class. For example:

```
org.folders.test.TestPage
```

is a subclass of:

```
org.folders.test.Config
```

So, any "page" method overridden in the Config class will apply to all pages/classes of that package, and any "page" method overridden in a page class will only apply to that particular page.

5.3 Arguments and methods

5.3.1 General arguments

```
http://<your-site>/Dispatcher?folder=test&page=TestPage&m_setName_1=Samuel
```

Figure 5.2: Example of a URL request used in J2Components

The method of passing input into the page variables is surprisingly cleaner than the method used in Ruby on Rails. With Ruby on Rails, arguments are passed into the URL request as query strings, and they are dealt with through use of the `@params`[8] hash variable, which is automatically filled with header data by the Rails dispatcher. This is relatively easy to deal with, but it does not make use of the natural, object-oriented paradigm, which uses methods to get/set variables. With J2Components, like Ruby on Rails, client input is passed in by the URL request query strings, but the Dispatcher treats the query string variables as methods.
Figure 5.2 shows an example of:

```
org.folders.test.TestPage
```

being instantiated with the object calling the method:

```
public void setName(java.lang.String name)
```

The prefix, "m_" and suffix, "_1" seen in Figure 5.2 tell the Dispatcher that it's dealing with a method and of type:

```
java.lang.String
```

```
1=java.lang.String
2=java.lang.Integer
3=java.lang.Double
```

Figure 5.3: Example of Types.properties file.

This could be misunderstood as an eyesore, made to cause confusion. But it allows for a potentially great deal of power when using a statically typed language. the number 1 refers to the type of `setName`'s argument. This is customized in a file called "Types.properties", which the Dispatcher automatically knows to look for when dealing with a URL query string that portrays a method. In "Types.properties", the developer lists any type used in a URL request with a reference number. So when the Dispatcher reads:

```
m_setName_1=Samuel
```

it's signaled that the particular query string variable deals with the method:

```
setName(java.lang.String name)
```

in:

```
org.folders.test.TestPage
```

5.3.2 Arguments passed into a form

In order for a web application to be interactive, there should be a way to pass in data. J2Components uses `startFormTag()` and `endFormTag()`. Figure 5.4 shows an example of a simple working form. Upon submit, `myForm()` would instantiate:

```
org.folders.test.TestPage
```

and display:

```
./<your-site>/web/folders/test/TestPage.html
```

The `java.lang.String[]` passed into `startFormTag` contain the arguments where `arg1` is `argument 1`. Having arguments in a String[] array is relatively rudimentary, but it allows for an indefinite amount of arguments, and it's quick to implement. A future "todo" of J2Components would have the Dispatcher determine typing for arguments. Figure 5.4 shows argument data being passed in a form, but that is clearly not necessary since it's in the URL request. The last argument of `startFormTag` in Figure 5.4 is the database table ID. This associates the form with a database. It says that the form input fields are also fields in a table, which is helpful when a form is submitting data directly into the database. If the database table ID is left null, then the form is unrelated to a database table. More details are discussed in the next section.

```
public void myForm() {
  startFormTag("test", "TestPage", new String[] {"arg1=",
    "argument 1"}, null);
  submitTag("Submit");
  endFormTag();
}
```

Figure 5.4: Example of a simple form

5.3.3 Arguments passed into links

Section 5.3.1 explains the actual URL address of the link. But there are a couple of methods in the Page class that make the process of writing links easier.

```
1. public void linkTo(String folder, String page, String value,
      String[] args, String[] attr) throws IOException

2. public void linkTo(String value, String[] args, String[] attr)
      throws IOException

3. public void redirect(String folder, String page,
      String[] args, String[] attr) throws IOException

4. public void redirect(String[] args, String[] attr)
      throws IOException
```

Figure 5.5: Methods for links and redirects.

Number 1 of Figure 5.5 shows a Page class that represents a link where folder and page are passed in. The `String[] args` contains any extra URL methods that are needed. URL methods are described further in Section 5.3.4. So for example:

```
linkTo("test", "TestPage", "Go to TestPage",
       new String[] {"m_arg1_1", "argument 1"},
       new String[] {"id", "myId"});
```

would write the link:

```
<a href="<your-site>/Dispatcher?folder=test&page=TestPage&m_arg1_1=argument
          1" id="myId">To to TestPage</a>
```

Number 2 and 3 of Figure 5.5 work like the `linkTo` methods, but the moment they're called, the link is requested. Rather than waiting to be pressed on the client side.

5.3.4 URL methods versus HTML methods

An advantage J2Components has over Ruby on Rails is the two distinct ways methods can be instantiated. As described in Section 5.3.1, arguments are not only be passed in through the header, but also they use the Page class methods to do so. In Ruby on Rails, arguments are passed into a `@params` hash variable, which is not bad, but it's not a suggested design pattern of object-oriented programming.

5.4 Forms

5.4.1 Basic forms

As previously mentioned, forms begin with `startFormTag` and end with `endFormTag` methods. More detail is shown in Figure 5.6. These methods are used in Page classes, and can be referenced in the Page class's corresponding HTML file. For forms not dealing with model data, upon submitting, the client input is passed to the class's methods as described in Section 5.3.1.

1. `public void endFormTag()`
2. `public void input(String fieldName)`
3. `public void startFormTag(String folder, String page, String[] args, Model model)`
4. `public void submitTag(String value)`

Figure 5.6: Form tags listed.

5.4.2 Model forms

If an `org.framework.Model` object is passed into `startFormTag`, then the form specifically applies to a model. In such a case, input variables must apply specifically to database table fields.

```
public void myForm() {
  startFormTag("test", "TestPage", new String[] {"arg1=",
    "argument 1"}, new Honey());
  input("Name");
  submitTag("Submit");
  endFormTag();
}
```

Figure 5.7: myForm() is an example method in org.folders.test.MyPage sample class.

Figure 5.7 shows an example of a simple form. For the method:

```
startFormTag("test", "TestPage", new String[] {"arg1=",
             "argument 1"}, new Honey());
```

upon submitting, the form changes to TestPage. Since `startFormTag` is passed in a model object, the form relates specifically to the model:

```
org.models.Honey
```

and the ”Honey” database table.

As previously mentioned, a model object directly portrays a database table. So with:

```
input("Name");
```

the input text field applies to the "Name" field of the Honey database. When the form is submitted, the page switches to TestPage, which has a method that saves the Honey data in the database. Figure 5.8 shows how model data would be saved in this situation.

The method:

```
public void save(final HttpSession session)
```

used in Figure 5.8 is inherited from:

```
org.framework.Model
```

It's passed the `HttpSession` object of the current session as its argument. It contains the form's input names as keys, and the client user's input as values. When a form's data is submitted, it automatically gets stored in the session. So the `session` variable comes directly from the static HttpSession variable, which is passed to all `Page` classes by the Dispatcher. After it's saved from the HttpSession variable into the database, it's deleted from the session to avoid accidental reuse.

```
public void saveHoneyData() {
  Honey h = new Honey();
  h.save(session);
}
```

Figure 5.8: This shows form input added and saved to the Honey table. saveHoneyData() is found in the example org.folders.test.TestPage class.

```
public class Honey extends Model {
  private Integer id;
  private String name;
  private String taste;

  public Honey(){

  }
  public Integer getId() {
    return id;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getTaste() {
    return taste;
  }
  public void setTaste(String taste) {
    this.taste = taste;
  }
}
```

Figure 5.9: The Honey model used in Figure 5.7

5.5 Template parser

In order for methods displayed in the HTML page files to be called, they must be pulled out by the Dispatcher at runtime. This is made possible by the basic parser:

```
org.framework.TemplateParser
```

J2Components's parser, which parses `folders` HTML files for `<%...%>` tags, is relatively basic. An old methodology is to have code intermixed into the HTML (JSP[3], PHP[7], RHTML[15], etc). This is technically a separation of HTML and programming code because only program code may live in their specified brackets, i.e. `<?php...?>`[7]. But it gets messy quickly. In order to make a clear separation of the view from the controller in the model, view, controller principle, the view should be limited on how much "controlling" it can do. The three examples above have essentially no limit to their "controlling". The approach here is to only allow methods of that class be visible.

Chapter 6

Conclusion

J2Components is still very young. But it has the room to grow exponentially. This is a point that should be highlighted. Many failure stories are the result of poor initial planning. The failure comes as ”special exceptions” are added to try and compensate, which ultimately lead to ambiguity. Java has been going in a bad direction with web application programming, but there is still hope. With J2Components, there are still things that could be done differently.

6.1 Future additions to J2Components

6.1.1 Better page-view interaction

The distinct separation of an HTML page from its corresponding page class is what makes J2Components very clean. A designer who only knows HTML is not left parsing through bits and pieces of Java code to find HTML. This becomes a problem in other frameworks that don't enforce a clear distinction, i.e. frameworks using JSP. But the side effect of J2Components from such distinction shows up in cases where data in a database should be

displayed in a table. The data retrieval occurs in the page's Java class, but each row of data is structured by use of HTML. In this case, HTML must be printed inside of the page's class rather than the page's HTML file. This is a problem when the designer wants to customize the way the table looks, but the HTML for the table is inside the Java class, which is off limits to the designer. Currently, the way to combat this is by use of Cascading Style Sheets. A designer can do as much customization as needed to a table without touching the table. If they are customizing the `TABLE, TR, TD` elements, then there is no problem. If they are customizing an ID or class attribute to be used by those elements, then they must communicate to the programmer to have the table (located in the Java code) contain the specified ID or class attributes. This could potentially create a communication problem.

6.1.2 Larger library

Specifically, more convenience classes. J2Components tackles the features needed to make the framework work as designed and organized. But a current advantage to a framework like Ruby on Rails or Tapestry, is their level of maturity. J2Components is in its infant stage of maturity. But what gives it hope is its clean, highly organized structure that allows for immense scalability.

Bibliography

[1] Edd Dumbill. Ruby on rails: An interview with david heinemeier hansson. http://www.oreillynet.com/pub/a/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html, 2005.

[2] J2ee blueprints digest. http://java.sun.com/developer/technicalArticles/J2EE/DesignEntApps/, 2006.

[3] Java 2 platform, enterprise edition (j2ee) faq. http://java.sun.com/javaee/overview/faq/j2ee.jsp, 2006.

[4] Java metadata interface (jmi). http://java.sun.com/products/jmi/, 2002.

[5] Jsr 127: Javaserver faces. http://www.jcp.org/en/jsr/detail?id=127, 2004.

[6] Jsr 58: Java 2 platform, enterprise edition 1.3 specification. http://jcp.org/en/jsr/detail?id=58, 2001.

[7] Php. http://www.php.net/, 2007.

[8] Rails best practices. http://wiki.rubyonrails.org/rails/pages/RailsBestPractices, 2007.

[9] Spring framework. http://www.springframework.org, 2007.

[10] Darryl K. Taft. Ruby on rails: Making programmers happy. http://www.eweek.com/article2/0,1895,1880280,00.asp, 2005.

[11] The apache ant project. http://ant.apache.org/, 2007.

[12] The apache software foundation. http://www.apache.org/, 2007.

[13] Tiobe programming community index for january 2007. http://www.tiobe.com/tpci.htm, 2005.

[14] Update: An introduction to the java ee 5 platform. http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/, 2006.

[15] Web development that doesn't hurt. http://www.rubyonrails.org/, 2007.

[16] Welcome to tapestry. http://tapestry.apache.org/, 2007.