Radeox Developer Guide

Stephan Schmidt
February 2004

Contact:

Stephan J. Schmidt
stephan@mud.de

Radeox
http://www.radeox.org

Contents

1 Introduction

Radeox is a rendering engine which renders text markup like __bold__ to XHTML. Radeox uses the Java programming language. It is used in wiki engines and applications to render wiki markup. To try radeox type (you need to have commons-logging.jar in the same directory for this to work) into your shell or DOS prompt

```
> java -jar lib/radeox.jar
```

And then at the prompt enter

```
> __radeox__
```

This should render the markup to XHTML and show

```
<b class="bold">radeox</b>
```

The latest version and information on how to use it can be found at http://radeox.org/. Radeox is available under the terms and conditions of the GNU Lesser General Public License (see license.txt). *Enjoy Radeox*.

2 Dependencies

Radeox has the following dependencies:

- commons-logging.jar

All other JAR files are needed to run the examples and the unit tests.

3 Radeox Architecture

Radeox uses a two stage architecture. The first stage consists of a set of filters, which take the input and turn it into some output.

For example the BoldFilter takes as input something like

```
This is some __bold__ text.
```

and turns this into XHTML like

```
This is some <b class="bold">bold</b> text.
```

4 Using Radeox

All Examples can be found in the examples/ directory. The examples are wrapped as JUnit tests and can be executed with

```
> ant test
```

Using the Radeox RenderEngine in your Java applications is very simple. You just have to call render() on a RenderEngine object:

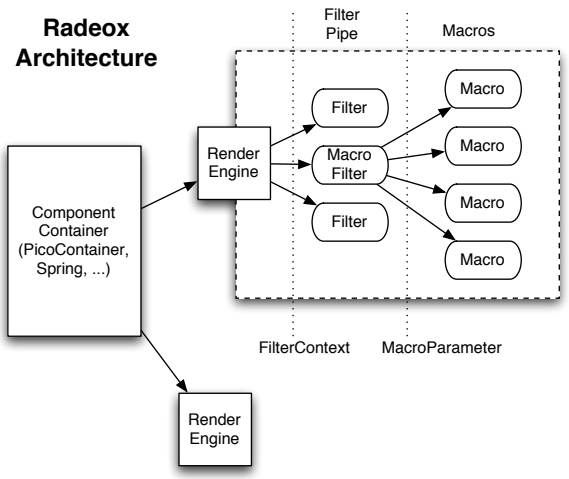

Figure 1: Radeox Render Architecture

```
1 RenderContext context = new BaseRenderContext();
2 RenderEngine engine = new BaseRenderEngine();
3 String result = engine.render("__Radeox__", context);
```

The RenderEngine needs a context object which is of Type RenderContext and contains the enviroment for the RenderEngine. RenderContext can be implemented by yourself to pass additional parameters and enviroment variables to your filters and macros. Usually you only use BaseRenderContext:

```
RenderContext context = new BaseRenderContext();
```

That's it. Place radeox.jar into your classpath and your done.

5 Changing input and output patterns

5.1 Using your own locale

The patterns and the generated output is not hardwired into Radeox. They can easily be changed to represent your own wiki markup. Radeox uses locale files for the regular expression patterns and the generated output (e.g. XHTML). These locale files have key and value pairs

```
key=value
```

To adapt the mapping to your needs you have to create our yown property file with the mappings you want to change. All the mappings we don't want to change are read from the standard Radeox properties locale. The standard locale is named *radeox_markup.properties* and part of the *radeox.jar.* Create a new file *radeox_markup_mywiki.properties* with the content

```
filter.bold.print=\$1<b class=\"mybold\">\$2</b>\$3
```

and place the properties locale file in the classpath. Every filter reads it's output from a property

```
filter.<name>.print
```

where different filters get parameters from the pattern matches. BoldFilter gets three parameters $1, $2 and $3. $1 and $3 are the matches before and after the pattern, $2 is the match between the __.

To change the locale the RenderEngine is using for patterns and output, you can pass the RenderEngine an InitialRenderContext:

```
InitialRenderContext initialContext =
    new BaseInitialRenderContext();
initialContext.set(RenderContext.INPUT_LOCALE,
                   new Locale("mywiki", "mywiki"));
RenderEngine engineWithContext =
  new BaseRenderEngine(initialContext);
String result = engineWithContext.render(
    "__Radeox__",
    new BaseRenderContext());
```

The string result should now contain

```
<b class="mybold">Radeox</b>
```

5.2 Using different input and output locales

Radeox uses different input and output locales, which can be accessed and set in the InitialRenderContext with

```
RenderContext.OUTPUT_LOCALE
RenderContext.INPUT_LOCALE
```

The input locale patterns follow the same schema as the output locale and are value and key pairs. The input is a regular expression pattern which is used to find matches in the input text. The bold filter contains

```
filter.bold.match=
    (^|>|[\\p{Punct}\\p{Space}]+)__(.*?)__([\\p{Punct}\\p{Space}]+|<|$)
```

All filter keys end with *match* for the regular expression. This regex can be changed to fit a different wiki markup. The OUTPUT_LOCALE and the INPUT_LOCALE can be changed independently.

5.3 Changing the properties file

By default the locales are read from a file named *radeox_markup.properties* in the classpath. You can set the name of the file with

```
initialContext.set(
  RenderContext.INPUT_BUNDLE_NAME,
  "myown_markup");
```

Radeox then tries to load the input patterns from *myown_markup.properties*. The name of the output properties file can be set accordingly with

```
initialContext.set(
  RenderContext.OUTPUT_BUNDLE_NAME,
  "myown_markup");
```

Input and output properties may be loaded from different files.

5.4 Natural language locale

Messages for the user which are generated by macros and filters are read from *radeox_messages.properties*. The language locale is set with the default language locale of the JDK but can be overriden with

```
initialContext.set(
    RenderContext.LANGUAGE_LOCALE,
    new Locale("de", "DE");
```

This forces Radeox to use a German locale. To translate the macros or your own macros to a new language, you should copy *radeox_messages.properties* to e.g. *radeox_messages_de.properties* and then translate the values. The locale looks like

```
#Macros
macro.table.description=Displays a table.
```

The name of the messages bundle file is set with

```
initialContext.set(
    RenderContext.LANGUAGE_BUNDLE_NAME,
    "radeox_messages")
```

Another way is to inherit from BaseInitialRenderContext and create your own InitialRenderContext where you set the appropriate values to the context. For example you can write a class called MyInitialRenderContext

```
1  public class MyInitialRenderContext
2      extends BaseRenderContext
3      implements InitialRenderContext {
4
5    public MyInitialRenderContext() {
6      Locale languageLocale = Locale.getDefault();
7      Locale locale = new Locale("mywiki", "mywiki");
8      set(RenderContext.INPUT_LOCALE, locale);
9      set(RenderContext.OUTPUT_LOCALE, locale);
10     set(RenderContext.LANGUAGE_LOCALE, languageLocale);
11     set(RenderContext.INPUT_BUNDLE_NAME, "my_markup");
12     set(RenderContext.OUTPUT_BUNDLE_NAME, "my_markup");
13     set(RenderContext.LANGUAGE_BUNDLE_NAME, "my_messages");
14   }
15 }
```

6 Radeox and PicoContainer

Sometimes it is desired to make the RenderEngine exchangeable. This can be done with a factory pattern. But it is much more flexible to use a component container which manages the different components of an application. There are several easy to use containers like Spring[3] or PicoContainer[2]. The application puts class and interface mappings into the container and later gets the implementation of an interface from the container without knowing which implementation it gets. This makes it easy to change implementations of components in the application.

6.1 Basic PicoContainer usage

The prefered way to use Radeox is to treat the RenderEngine as a component in PicoContainer. Using Radeox this way, you can replace the RenderEngine in Radeox easily with few changes to your code .

```
DefaultPicoContainer dc = new DefaultPicoContainer();
try {
  // Register BaseRenderEngine as an Implementation
  // of RenderEngine
  dc.registerComponentImplementation(
      RenderEngine.class,
      BaseRenderEngine.class);
} catch (Exception e) {
  System.err.println("Could not register component.");
}

// now only work with container
PicoContainer container = dc;

// Only ask for RenderEngine, we automatically
// get an available object
// that implements RenderEngine
RenderEngine engine = (RenderEngine)
  container.getComponentInstance(RenderEngine.class);
RenderContext context = new BaseRenderContext();
String result = engine.render("__SnipSnap__", context);
```

6.2 Example with another locale

If we want to change the locale that the RenderEngine gets at startup time, we have to tell PicoContainer which InitialRenderContext to use. Basically the same as above, but we add an InitialRenderContext to the PicoContainer. PicoContainer then automatically configures RenderEngine with the available InitialRenderContext object.

```
DefaultPicoContainer dc = new DefaultPicoContainer();
try {
  InitialRenderContext initialContext =
      new BaseInitialRenderContext();
```

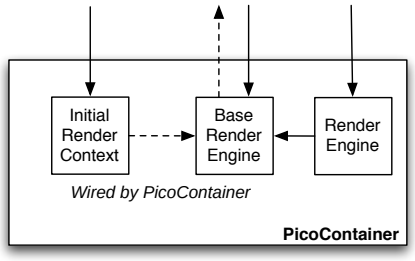

Figure 2: InitialRenderContext in PicoContainer

```
  initialContext.set(RenderContext.OUTPUT_LOCALE,
                     new Locale("mywiki", "mywiki"));
  dc.registerComponentInstance(InitialRenderContext.class,
                               initialContext);
  dc.registerComponentImplementation(RenderEngine.class,
                                     BaseRenderEngine.class);
} catch (Exception e) {
  System.err.println("Could not register component.");
}
```

7 Extending Radeox

7.1 Writing Macros

There are several ways to extend and customize Radeox with your own ideas. The easiest and most powerful way to extend Radeox is to write macros. Macros are implemented as a filter (MacroFilter). A macro in Radeox is a command that does something, like show the number of users, search for a string, display a list of recently changed wiki pages, render source code or change the font color. Macros have the form {macroname} and can have none or several arguments. For example {user-count} will display the number of registered readers in SnipSnap. This tutorial teaches you the basics about macros. Macros may surround content like

```
{code}
  Example code
{code}
```

is rendered as

```
Example code
```

7.1.1 Macro Class

The easiest way to write a macro is to inherit from *org.radeox.macro.BaseMacro* and implement

```
public abstract void execute(Writer writer, MacroParameter params)
      throws IllegalArgumentException, IOException;
```

Writer is a java.io.Writer object. To output something from your macro just use

```
writer.write("hello world");
```

If you have to output other objects than Strings then you can encapsulate the writer with java.io.PrintWriter which supports print and println for several other types. Especially do not use write(i); with an int.

As an example let's write a HelloWorld macro.

```
1  package examples;
2
3  import org.radeox.macro.BaseMacro;
4  import org.radeox.macro.parameter.MacroParameter;
5
6  import java.io.IOException;
7  import java.io.Writer;
8
9  public class HelloWorldMacro extends BaseMacro {
10   public void execute(Writer writer, MacroParameter params)
11     throws IllegalArgumentException, IOException {
12     writer.write("hello world");
13   }
14
```

This macro just writes "hello world" to the output stream. Usually it's a good idea to write XHTML, for example

```
writer.write("<b>Hello World</b>");
```

7.1.2 Deployment

To make this macro run you have to first give your macro a command name. We take "hello" for this one. Add a getName method to the macro:

```
public String getName() {
    return "hello";
}
```

so you can call the macro with {hello} from your input text. Create a jar file with two files:

```
META-INF/services/org.radeox.macro.Macro
examples/HelloWorldMacro.class
```

META-INF/services/org.radeox.macro.Macro should contain lines with the class names of your macros, for example *examples.HelloWorldMacro*

Put this jar somewhere in the classpath and Radeox should find your macro.

7.1.3 Description

Every macro has to document itself. There is a method called getDescription. Add

```
public String getDescription() {
    return "Say hello.";
}
```

to give the HelloWorldMacro a description. This description is used by {list-of-macros}. The MacroListMacro displays all known macros with their name and their description.

7.1.4 Parameters

The execute method of Macro gets a MacroParameter object. The MacroParameter object is created within Radeox. The RenderContext is passed from your appication to the MacroParameter object in your macro call.

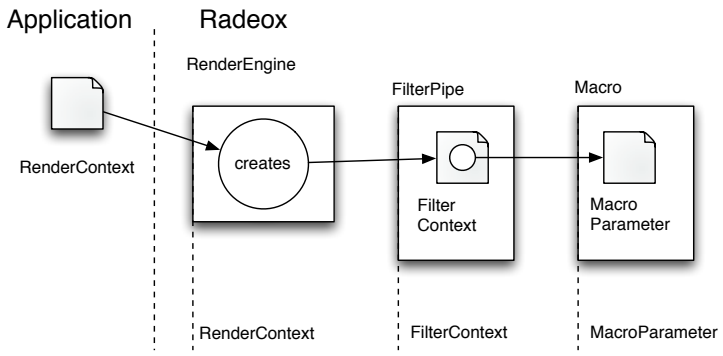

Figure 3: MacroParameter creation

This object encapsulates useful information about the execution context. With params.get("0"); you can get the first un-named parameter from the macro call. To extend the macro to read a parameter and output the parameter after the "hello" instead of "world" rewrite execute to:

```
1  public void execute(Writer writer, MacroParameter params)
2      throws IllegalArgumentException, IOException {
3    if (params != null && params.getLength() == 1) {
4      writer.write("Hello <b>");
5      writer.write(params.get("0"));
6      writer.write("</b>");
7    } else {
8      throw new IllegalArgumentException(
9          "Number of arguments does not match");
10   }
11 }
```

The macro can now be used with a parameter: {hello:Stephan} which produces

```
Hello Stephan
```

7.1.5 Named parameters

Arguments can be without a name, so {image:Stephan} displays the image with the name "Stephan" and {image:img=Stephan} will do the same. The named version is useful when there are several optional arguments. If a macro is given several arguments, they are sperated with a " | " like in {image:img=Stephan | align=left}. To make the previous HelloWorld example a named parameter, just use

```
writer.write(params.get("name"));
```

which makes the macro usable as {hello:name=Stephan}

7.1.6 Content block

The content between two macro tags as in

```
{hello}
  Stephan is so funny.
{hello}
```

is also part of the MacroParameter object and can be accessed with

```
params.getContent();
```

This might be *Null* when there is no content. A ContentHelloWorldMacro could read the name from the content and turn

```
{hello}stephan{hello}
```

into

```
hello stephan
```

The code is almost the same but reads the name from the content block.

```
1 public void execute(Writer writer, MacroParameter params)
2   throws IllegalArgumentException, IOException {
3   writer.write("hello " + params.getContent());
4 }
```

7.1.7 Reading the InitialRenderContext

If you want to read initial parameters in your Macro you need access to the InitialRenderContext object. The Macro interface has a method setInitialContext which is implemented in BaseMacro like this

```
public void setInitialContext(InitialRenderContext context) {
  this.initialContext = context;
}
```

The same method signature is defined in the Filter interface. At creation time the RenderEngine implementation in Radeox calls setInitialContext on all Macros and Filters. You can then read a value from your own InitialRenderContext. We extend the HelloWorldMacro to read the name from the InitialRenderContext. First we have to create a InitialRenderContext and set the name as a value.

```
initialContext.set(
    "hello.name",
    "stephan");
```

The InitialRenderContextHelloWorldMacro then reads the name from the InitialRenderContext

```
1  private String name;
2
3  public void setInitialContext(InitialRenderContext context) {
4    super.setInitialContext(context);
5    name = (String) context.get("hello.name");
6  }
7
8  public void execute(Writer writer, MacroParameter params)
9    throws IllegalArgumentException, IOException {
10   writer.write("hello "+name);
11 }
```

This macro, called with {hello} returns

```
hello stephan
```

7.1.8 Macros with XML closing syntax

7.2 Writing Filters

While Macros are commands, filters instead replace special text markup with XHTML, for example the BoldFilter replaces __bold__ with bold. You should also be familiar with regular expressions[4] to follow this trail.

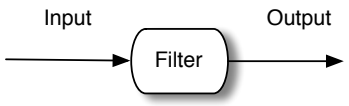

Figure 4: Filter principle

7.2.1 Filter Interface

The basis for all filters is the Filter interface. All classes that want to act as filters have to implement Filter.

```
public interface Filter {
  public String filter(String input, FilterContext context);

  public String[] before();
}
```

The important method is filter(). This method gets a string as an input and a FilterContext object. It takes the input string, applies its filtering and then returns the filtered string. FilterContext holds attributes and references, e.g. to the rendering engine.

7.2.2 Example Filter

To write your own filter and implement Filter there are some support classes available. The most basic is FilterSupport which does not much, but implement the before() method so you do not have to care. Most of the time though you want to use regular expressions (regex) to filter your content. For this there are two classes you can inherit from

- RegexReplaceFilter
- RegexTokenFilter

RegexReplaceFilter takes a regex and replaces it with some output like sed or perls s/.../.../g does. RegexTokenFilter calls a subroutine for every match.

Here is a smily replace filter as an example. Suppose you want to replace every Frowny *:-(* with a smiley *:-)*. The only thing you have to do is to supply the regular expressions:

```
1 public class SmileyFilter extends RegexReplaceFilter {
2   public SmileyFilter() {
3     super(":-\\(", ":-)");
4   };
5 }
```

The two backslashes are for quoting the (in the regular expression. The SmileyFilter now replaces every occurence of *:-(* in the input with *:-)*.

7.2.3 Caching

Radeox supports caching. To tell the engine that the output from your filter is cachable (= produces the same output for the same input which is usually the case with static filters) add the marker interface CacheFilter to your filter:

```
public class SmileyFilter
   extends RegexReplaceFilter
   implements CacheFilter {
...
```

You can then ask the RenderContext after a render() call, if the render result is cacheable

```
public boolean isCacheable();
```

and cache the result. As a rule of thumb, if the input contained macros, the result will not be cacheable, if it only contained filters, the result will be cacheable.

7.2.4 Deployment

To make this filter work you have to create a jar file with

```
META-INF/services/org.radeox.filter.Filter
examples/SmileyFilter.class
```

META-INF/services/org.radeox.filter.Filter should contain lines with the class names of your filters, for example *examples.SmileyFilter*. Put this jar into the classpath and Radeox should find your filters.

7.2.5 Using RegexTokenFilter

Suppose we want to do something more complicated with our filter than just replace something statically from the input. We want to write a filter which searches for a $ sign followed by a number in its input e.g.

```
$3
```

and then replaces this with the square of the number, e.g. 9. RegexReplaceFilter fails to do this, but luckily there is a Filter which exactly does this. RegexTokenFilter takes the input and splits it into tokens. For every found match, handleMatch() is called. In the handleMatch() method you can do whatever you like with the match and write something to the output. The SquareFilter example takes the input, converts it to an int and returns the square.

```
1  public class SquareFilter extends RegexTokenFilter {
2    public SquareFilter() {
3      super("\\$[0-9]+", true);
4    };
5
6    public void handleMatch(StringBuffer buffer,
7                            MatchResult result,
8                            FilterContext context) {
9      // group(0) will e.g. return "$3" so we have to cut the $
10     int number = Integer.parseInt(
11         result.group(0).substring(1));
12     buffer.append(number*number);
13   }
14 }
```

The second parameter in the super() call is whether the match is singleline or not.

7.2.6 Using the locale versions of RegexTokenFilter and RegexReplaceFilter

Suppose someone does not agree with your former definition of a Smiley and thinks the smiley should look like

```
:->
```

To change the smiley he has to modify your Java code, recompile and deploy the jar. It's much easier for him if the smiley would be defined in a properties file. Simply add the lines for a matcher and a output format to the *radeox_markup_mywiki.properties* file (or the default Radeox properties file)

```
filter.smiley.match=:-\\(
filter.smiley.print=:->
```

The standard RegexReplaceFilter does not read from a locale. While you can write the code to read from the locale by hand, there is a helper Filter class which does this for you. LocaleRegexReplaceFilter reads the input and output from locales. Write a LocaleSmileyFilter which extends LocaleRegexReplaceFilter:

```
1 public class LocaleSmileyFilter
2     extends LocaleRegexReplaceFilter {
3
4   protected String getLocaleKey() {
5     return "filter.smiley";
6   }
7 }
```

The only thing you have to do is supply a key for the locale properties file. The key is then expanded to *<key>.match* and *<key>.print*. If you add the locale to your InitialRenderContext, the SmileyFilter should now render *:-(* to *:->*

```
InitialRenderContext context = new BaseInitialRenderContext();
Locale locale = new Locale("mywiki", "mywiki")
context.set(RenderContext.INPUT_LOCALE, locale );
context.set(RenderContext.OUTPUT_LOCALE,  locale);
```

Similar to LocaleRegexReplaceFilter there is a LocaleRegexTokenFiler which reads its parameters from the locale. This filter only reads a match key.

8 Using Radeox in your Wiki

8.1 Wiki architecture

A Wiki system usually consists of:

- Storage Backend (JDBC, Files)
- Some Wiki logic to determine if a page exists
- Some glue that takes input form the user, stores pages to the backend etc.

- A HTML frontend, often with a web framework (WebWork, Struts, WebMacro)
- A render engine that turns wiki markup into XHTML / XML

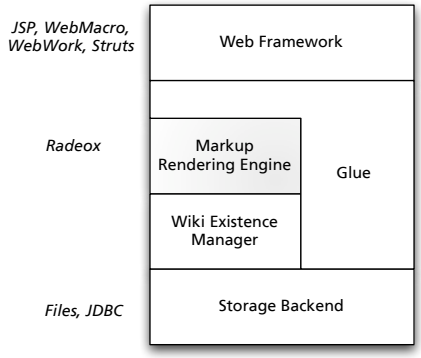

Figure 5: Wiki Architecture

Although people use frameworks for the backend and the frontend, they mostly do not yet use a framework for the rendering engine. Radeox fills this gap.

8.2 Creating wiki links

Radeox is a wiki render engine which can be used out of the box. There is a render engine called BaseRenderEngine which can be used for rendering. This engine though does not know about you wiki backend, so you have to write your own engine. This backend tells Radeox if a wiki page exists, if it should link to a wiki page or instead a creation form and so on.

8.2.1 WikiRenderEngine

For your own renderer engine you can extend BaseRenderEngine from Radeox. Then you have to implement the interface WikiRenderEngine. This tells the renderer in Radeox that your use the renderer in a wiki enviroment and not for another application.

```
public interface WikiRenderEngine {
  public boolean exists(String name);
  public boolean showCreate();
  public void appendLink(StringBuffer buffer,
    String name,
    String view,
    String anchor);
  public void appendLink(StringBuffer buffer,
    String name,
    String view);
  public void appendCreateLink(StringBuffer buffer,
```

```
    String name,
    String view);
}
```

- *exists* takes the name of a wiki page and returns true if the wiki page exists
- *showCreate* returns true if radeox should render links to a creation form. This can be used to only link 'create wiki' pages if the user is logged in
- *appendLink* appends e.g. the <A HREF> HTML code for linking to a wiki page with the given name to StringBuffer. View is the text that should be shown to the user.
- *appendCreateLink* appends the HTML code for linking to the 'create wiki' page

8.2.2 Making it work

A small WikiRenderEngine, which only knows 'SnipSnap' and 'stephan' as wiki entries could look like this:

```
1  public class MyWikiRenderEngine
2      extends BaseRenderEngine
3      implements WikiRenderEngine {
4
5    public boolean exists(String name) {
6      // make a lookup in your wiki if the page exists
7      return name.equals("SnipSnap") || name.equals("stephan");
8    }
9
10   public boolean showCreate() {
11     // we always want to show a create link, not only e.g.
12     // if a user is registered
13     return true;
14   }
15
16   public void appendLink(StringBuffer buffer,
17                          String name,
18                          String view) {
19     buffer.append("<a href=\"/show?wiki=");
20     buffer.append(name);
21     buffer.append("\">");
22     buffer.append(view);
23     buffer.append("</a>");
24   }
25
26   public void appendLink(StringBuffer buffer,
27                          String name,
28                          String view,
29                          String anchor) {
30     buffer.append("<a href=\"/show?wiki=");
31     buffer.append(name);
32     buffer.append("#");
```

```
33      buffer.append(anchor);
34      buffer.append("\">");
35      buffer.append(view);
36      buffer.append("</a>");
37    }
38
39    public void appendCreateLink(StringBuffer buffer,
40                                 String name,
41                                 String view) {
42      buffer.append(name);
43      buffer.append("<a href=\"/create?wiki=");
44      buffer.append(name);
45      buffer.append("\">");
46      buffer.append("?</a>");
47    }
48
49    public String getName() {
50      return "my-wiki";
51    }
52  }
```

Then use the render engine as described before. You have to set your engine in BaseRenderContext though, so Radeox does use your engine to determine if wiki pages exist. The LinkFilter in Radeox takes a look at your RenderEngine and if it implements WikiRenderEngine, Radeox uses your WikiRenderEngine to look for existing wiki pages and for creating XHTML links.

```
RenderEngine myEngine = new MyWikiRenderEngine();
RenderContext context = new BaseRenderContext();
context.setRenderEngine(myEngine);
String result = myEngine.render("My String", context);
```

Alternativly you could create your own RenderContext that returns your engine. That's it.

8.3 Changing the linking style to WikiLinkingStyle

8.4 Porting to Radeox

For example, WikiLand uses Cocoon for rendering. To change their architecture to use Radeox they would have to insert the Radeox API between their application and their wiki markup renderer.

9 Writing your own RenderEngine

You want to write a completely different RenderEngine for Radeox (or SnipSnap). Perhaps you want to render XML to XHML using XSLT or you want to support other wiki markup and your goal cannot be achived with writing macros or filters. Perhaps you want to use a parser instead of regular expressions.

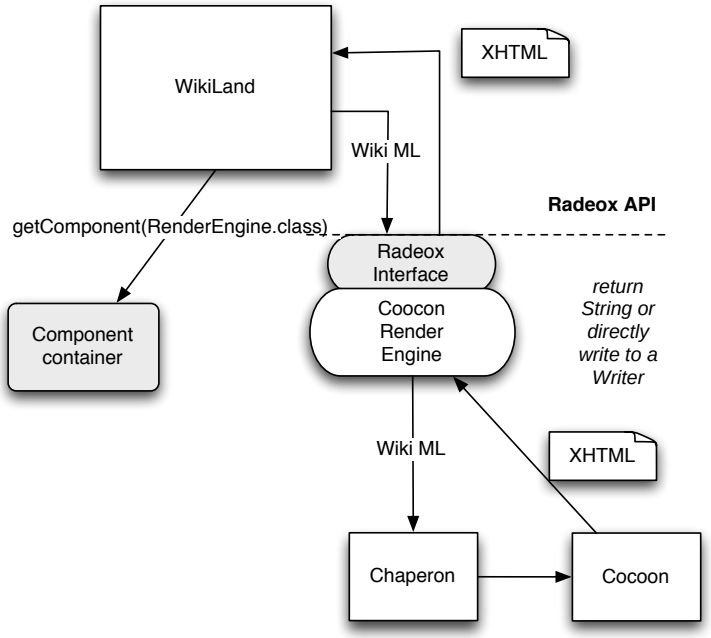

Figure 6: WikiLand and Radeox integration

9.1 Write a Radeox RenderEngine

The RenderEngine interface looks like this:

```
public interface RenderEngine {
  public String getName();
  public String render(String content, RenderContext context);
}
```

You write your own Implementation of RenderEngine, say MyRenderEngine where getName() returns "my" and

```
render(content, context);
```

does the actual rendering, for example replace all "X" with "Y":

```
1 public class MyRenderEngine implements RenderEngine {
2   public String getName() {
3      return "my";
4   }
5   public String render(String content, RenderContext context) {
6      return content.replace('X', 'Y');
7   }
```

10 Using Radeox from other languages

10.1 Writing Macros with Groovy

Groovy[1] is a scripting language which directly compiles to byte code for the Java VM. There are two ways to use Groovy with Radeox

- compiling Macros to .class files
- loading Groovy Macros from Java

```
1  package examples;
2
3  import java.io.Writer
4  import org.radeox.macro.parameter.MacroParameter
5
6  class GroovyMacro extends org.radeox.macro.BaseMacro {
7    void execute(Writer writer, MacroParameter params) {
8      writer.write("Yipee ay ey, schweinebacke")
9    }
10   String getName() {
11     return "groovy"
12   }
13 }
```

You can use ant to compile the Groovy Macro to Java code. It's easier first compile all Groovy scripts to a destination directory and after that your Java files, because otherwise the Java compiler won't find the Groovy .class files.

```
<target name="compile-groovy" depends="prepare">
  <groovyc destdir="${pre-out}" srcdir="${src}" listfiles="true">
      <classpath refid="classpath"/>
  </groovyc>
</target>

<target name="compile" depends="prepare, compile-groovy">
  <javac srcdir="${src}" destdir="${out}" classpathref="classpath"/>
</target>
```

The *pre-out* directory is in the classpath, so the javac task finds the compiled Groovy files. See the build.xml in the examples directory for a working example. To make the groovyc task working, you jave to first add

```
<taskdef name="groovyc" classname="org.codehaus.groovy.ant.Groovyc" classpath
```

to your build.xml

10.1.1 Loading Macros with Groovy at runtime

A Java class which compiles this macro source code to a Macro is GroovyMacroCompiler

```
public class GroovyMacroCompiler  {
 public Macro compileMacro(String macroSource) {
    Macro macro = null;
    try {
      GroovyClassLoader gcl = new GroovyClassLoader();
      InputStream is = new ByteArrayInputStream(
          macroSource.getBytes());
      Class clazz = gcl.parseClass(is, "Macro.groovy");
      Object aScript = clazz.newInstance();
      macro = (Macro) aScript;
    } catch (Exception e) {
      System.err.println("Cannot compile groovy macro.");
    }
    return macro;
  }
}
```

References

[1] Groovy programming language, http://groovy.codehaus.org

[2] Component container, http://www.picocontainer.org/

[3] Spring Framework with component container, http://www.springframework.org/

[4] Jeffrey E. F. Friedl, Mastering Regular Expressions, ISBN: 0596002890