

# Cocoon 3.0 ALPHA - Reference Documentation

Apache Cocoon 3.0 ALPHA

Reinhard Pötz (Indoqa Software Design und Beratung GmbH)

Copyright © 2008

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1. Introduction .....	1
1.1. Why Cocoon 3 - Motivation .....	1
1.2. Relationship to previous versions .....	1
1.3. Requirements .....	1
2. Pipelines .....	2
2.1. What is a pipeline? .....	2
2.1.1. Linear pipelines .....	2
2.1.2. Pipelines by example .....	2
2.2. Pipeline implementations .....	4
2.3. Embedding a pipeline .....	4
2.4. SAX components .....	4
2.4.1. Available components .....	4
2.4.2. Writing custom components .....	4
2.5. StAX components .....	5
2.5.1. Available components .....	5
2.5.2. Writing custom components .....	5
2.5.3. Using StAX and SAX components in the same pipeline .....	8
2.5.4. Java 1.5 support .....	9
2.6. Utilities .....	9
3. Sitemaps .....	10
3.1. What is a sitemap? .....	10
3.2. Sitemap evaluation? .....	10
3.3. Expression languages .....	10
3.4. Spring integration .....	10
3.5. Embedding a sitemap .....	10
4. Web applications .....	11
4.1. Usage scenarios .....	11
4.2. Servlet-Service framework integration .....	11
4.3. Sitemaps in an HTTP environment .....	11
4.4. System setup .....	11
4.5. Connecting pipeline fragments .....	11
4.6. RESTful web services .....	11
4.6.1. Sitemap based RESTful web services .....	11
4.6.2. JAX-RS based controllers (JSR311) .....	11
4.7. Caching and conditional GET requests .....	14
4.8. Authentication .....	14
4.9. Testing .....	14
4.10. Profiling support .....	14
4.10.1. Module configuration .....	14
4.10.2. Using Cocoon Profiling .....	14
4.10.3. Using Firebug with Cocoon-profiling .....	15
4.11. Monitoring support .....	16
4.11.1. Module configuration .....	16
4.11.2. Available parts .....	16
4.12. Tutorial .....	21
5. Wicket Integration .....	22
5.1. Introduction .....	22
5.2. Integrate Cocoon into Wicket .....	22
5.2.1. Mount a Cocoon sitemap .....	22
5.2.2. Mount a Cocoon pipeline .....	23
5.2.3. CocoonSAXPipeline Wicket component .....	24
5.3. Integrate Wicket into Cocoon .....	24

5.3.1. Wicket reader ..... 24

---

# Chapter 1. Introduction

## 1.1. Why Cocoon 3 - Motivation

The main idea behind Cocoon is the concept of pipelines. Cocoon 1.x and 2.x applied this idea with a focus on web applications. But sometimes pipelines would be useful although you don't develop a web application. Those former Cocoon versions don't really help you in that case.

In contrast, Cocoon 3 follows a layered approach so that its basic module - the pipeline module - can be used from within any Java environment without requiring you adding a huge stack of dependencies.

On top of this, Cocoon 3 has the goal to make the development of RESTful web services and web applications a simple task.

## 1.2. Relationship to previous versions

Cocoon 3 has been built completely from scratch and doesn't have any dependencies on Cocoon 2.x or 1.x.

## 1.3. Requirements

Cocoon 3 requires Java 5 or higher.

---

# Chapter 2. Pipelines

## 2.1. What is a pipeline?

A Cocoon 3 pipeline expects one or more component(s). These components get linked with each other in the order they were added. There is no restriction on the content that flows through the pipeline.

A pipeline works based on two fundamental concepts:

- The first component of a pipeline is of type `org.apache.cocoon.pipeline.component.Starter`. The last component is of type `org.apache.cocoon.pipeline.component.Finisher`.
- In order to link components with each other, the first has to be a `org.apache.cocoon.pipeline.component.Producer`, the latter `org.apache.cocoon.pipeline.component.Consumer`.

When the pipeline links the components, it merely checks whether the above mentioned interfaces are present. So the pipeline does not know about the specific capabilities or the compatibility of the components. It is the responsibility of the `Producer` to decide whether a specific `Consumer` can be linked to it or not (that is, whether it can produce output in the desired format of the `Consumer` or not). It is also conceivable that a `Producer` is capable of accepting different types of `Consumer` and adjust the output format

### 2.1.1. Linear pipelines

A Cocoon 3 pipeline always goes through the same sequence of components to produce its output. There is no support for conditionals, loops, tees or alternative flows in the case of errors. The reason for this restriction is simplicity and that non-linear pipelines are more difficult (or even impossible) to be cached. In practice this means that a pipeline has to be constructed completely at build-time.

If non-linear XML pipes with runtime-support for conditionals, loops, tees and error-flows are a requirement for you, see the [XProc](#) standard of the W3C. There are several available implementations for it.

### 2.1.2. Pipelines by example

But let's get more specific by giving an example: Cocoon has become famous for its SAX pipelines that consist of exactly one SAX-based XML generator, zero, one or more SAX-based XML transformers and exactly one SAX-based XML serializer. Of course, these specific SAX-based XML pipelines can be build by using general Cocoon 3 pipelines: generators, transformers and serializers are pipeline components. A generator is a `Starter` and a `Producer`, a transformer can't be neither a `Starter`, nor a `Finisher` but is always a `Producer` and a `Consumer` and a serializer is a `Consumer` and a `Finisher`.

Here is some Java code that demonstrates how a pipeline can be utilized with SAX-based XML components:

```
Pipeline<SAXPipelineComponent> pipeline = new NonCachingPipeline<SAXPipelineComponent>(); ❶
pipeline.addComponent(new XMLGenerator("<x></x>")); ❷
pipeline.addComponent(new XSLTTransformer(this.getClass().getResource("/test1.xslt"))); ❸
pipeline.addComponent(new XSLTTransformer(this.getClass().getResource("/test2.xslt"))); ❹
pipeline.addComponent(new XMLSerializer()); ❺

pipeline.setup(System.out); ❻
pipeline.execute(); ❼
```

- ❶ Create a `NonCachingPipeline`. It's the simplest available pipeline implementation. The `org.apache.cocoon.pipeline.Pipeline` interface doesn't impose any restrictions on the content that flows in it.
- ❷ Add a generator, that implements the `org.apache.cocoon.pipeline.component.PipelineComponent` interface to the pipeline by using the pipeline's `addComponent(pipelineComponent)` interface.

The `XMLGenerator` expects a `java.lang.String` object and produces SAX events by using a SAX parser. Hence it has to implement the `org.apache.cocoon.sax.component.SAXProducer` interface.

The `SAXProducer` interface extends the `org.apache.cocoon.pipeline.component.Producer` interface. This means that it expects the next (or the same!) component to implement the `org.apache.cocoon.pipeline.component.Consumer` interface. The check that the next pipeline component is of type `org.apache.cocoon.sax.component.SAXConsumer` isn't done at interface level but by the implementation (see the `org.apache.cocoon.sax.component.AbstractXMLProducer` for details which the `XMLGenerator` is inherited from).

Since a generator is the first component of a pipeline, it also has to implement the `Starter` interface.

- ❸ Add a transformer, that implements the `org.apache.cocoon.pipeline.component.PipelineComponent` interface, to the pipeline by using the pipeline's `addComponent(pipelineComponent)` method.

This `XSLTTransformer` expects the `java.net.URL` of an XSLT stylesheet. It uses the rules of the stylesheet to add, change or delete nodes of the XML SAX stream.

Since it implements the `org.apache.cocoon.pipeline.component.Consumer` interface, it fulfills the general contract that a `Consumer` is linked with a `Producer`. By implementing the `org.apache.cocoon.sax.component.SAXConsumer` interface, it fulfills the specific requirement of the previous `XMLGenerator` that expects a next pipeline component of that type.

This transformer also implements the `org.apache.cocoon.sax.component.SAXProducer` interface. This interface extends the `org.apache.cocoon.pipeline.component.Producer` interface which means that the next component has to be a `org.apache.cocoon.pipeline.component.Consumer`. Like the previous `XMLGenerator`, the `XSLTTransformer` inherits from the `org.apache.cocoon.sax.component.AbstractXMLProducer` which contains the check that the next component is of type `org.apache.cocoon.sax.component.SAXConsumer`.

- ❹ Add another transformer to the pipeline. A pipeline can contain any number of components that implement the `Producer` and `Consumer` interfaces at the same time. However, they mustn't be neither of type `Starter` nor `Finisher`.
- ❺ Add a serializer, that implements the `org.apache.cocoon.pipeline.component.PipelineComponent` interface to the pipeline by using the pipeline's `addComponent(pipelineComponent)` interface.

The XML serializer receives SAX events and serializes them into an `java.io.OutputStream`.

A serializer component is the last component of a pipeline and hence it has to implement the `org.apache.cocoon.pipeline.Finisher` interface.

Since it receives SAX events, it implements the `org.apache.cocoon.pipeline.sax.SAXConsumer` interface.

- ❻ A pipeline has to be initialized first by calling its `setup(outputStream)` method. This method expects the output stream where the pipeline result should be streamed.
- ❼ After the pipeline has been initialized, it can be executed by invoking its `execute()` method. The first pipeline component, a `Starter`, will be invoked which will trigger the next component and so on. Finally the last pipeline component, a `Finisher` will be reached which is responsible for the serialization of the pipeline content.

Once the pipeline has been started, it either succeeds or fails. There is no way to react on any (error)

conditions.

**Table 2.1. SAX components and their interfaces**

Component type	Structural interfaces	Content-specific interfaces		
SAX generator	Starter, Producer, PipelineComponent	SAXProducer		
SAX transformer	Producer, Consumer, PipelineComponent	SAXProducer, SAXConsumer		
SAX serializer	Finisher, Consumer, PipelineComponent	SAXConsumer		

## 2.2. Pipeline implementations

TBW: noncaching, caching, async-caching, expires caching, own implementations

## 2.3. Embedding a pipeline

TBW: Passing parameters to the pipeline and its components, finish() method

## 2.4. SAX components

concept, writing custom SAX components, link to Javadocs

### 2.4.1. Available components

Link to Javadocs

### 2.4.2. Writing custom components

#### 2.4.2.1. SAX generator

explain from a user's point of view, what she needs to do to implement one (available abstract classes)

#### 2.4.2.2. SAX transformer

explain from a user's point of view, what she needs to do to implement one

buffering

#### 2.4.2.3. SAX serializer

explain from a user's point of view, what she needs to do to implement one

## 2.5. StAX components

StAX pipelines provide an alternative API for writing pipeline components. Although they are not as fast as SAX, they provide easier state handling as the component can control when to pull the next events. This allows an implicit state rather than have to manage the state in the various content handler methods of SAX.

The most visible difference of StAX components in contrast to SAX is that the component itself has controls the parsing of the input whereas in SAX the parser controls the pipeline by calling the component. Our implementation of StAX pipelines uses just StAX interfaces for retrieving events - the writing interface is proprietary in order to avoid multithreading or continuations. So it is really a hybrid process - the StAX component is called to generate the next events, but it is also allowed to read as much data from the previous pipeline component as it wants. But as the produced events are kept in-memory until a later component pulls for them, the components should not emit large amounts of events during one invocation.

### 2.5.1. Available components

- `StAXGenerator` is a Starter and normally parses a XML from an `InputStream`.
- `StAXSerializer` is a Finisher and writes the StAX Events to an `OutputStream`.
- `AbstractStAXTransformer` is the abstract base class for new transformers. It simplifies the task by providing a template method for generating the new events.
- `StAXCleaningTransformer` is an transformer, which cleans the document from whitespaces and comments.
- `IncludeTransformer` includes the contents of another document.

For further information refer to the [javadoc](#)

### 2.5.2. Writing custom components

#### 2.5.2.1. StAX generator

The `StAXGenerator` is a Starter component and produces `XMLEvents`.

```
import java.io.InputStream;
import java.net.URL;

import javax.xml.stream.FactoryConfigurationError;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.XMLEvent;

import org.apache.cocoon.pipeline.SetupException;
import org.apache.cocoon.pipeline.component.Starter;
public class MyStAXGenerator extends AbstractStAXProducer implements Starter {

    private XMLEventReader reader;

    public MyStAXGenerator(InputStream inputStream) {
        try {
            this.reader = XMLInputFactory.newInstance().createXMLEventReader(inputStream);
        } catch (XMLStreamException e) {
```



```

        throw new SetupException("Error during setup an XMLEventReader on the inputStream", e);
    } catch (FactoryConfigurationError e) {
        throw new SetupException("Error during setup the XMLInputFactory for creating an XMLEventReader", e);
    }
}

public void execute() {
    this.getConsumer().initiatePullProcessing();           ❸
}

public boolean hasNext() {
    return this.reader.hasNext();                         ❹
}

public XMLEvent nextEvent() throws XMLStreamException {
    return this.reader.nextEvent();                       ❺
}

public XMLEvent peek() throws XMLStreamException {
    return this.reader.peek();                             ❻
}
}

```

- ❶ In order to implement an own `StAXGenerator` the easiest approach is to inherit from `AbstractStAXProducer`.
- ❷ The constructor creates a new `XMLEventReader` for reading from the inputstream.
- ❸ The pipeline is started using the `execute` method. As StAX is a pull based approach the last component has to start pulling.
- ❹ This method should return true if the generator has a next Event.
- ❺ Returns the next event from the generator.
- ❻ Returns the next event from the generator, without moving actually to the next event.

### 2.5.2.2. StAX transformer

Implementing a StAX Transformer should be the most common use case. The `AbstractStAXTransformer` provides a foundation for new transformers. But in order to write new transformers even simpler, let's describe another feature first:

#### 2.5.2.2.1. Navigator

Navigators allow an easier navigation in the XML document. They also simplify transformers, as usually transformers need only process some parts of the input document and the navigator helps to identify the interesting parts. There are several implementations already included:

- `FindStartElementNavigator` finds the start tag with certain properties(name,attribute)
- `FindEndElementNavigator` finds the end tag with certain properties(name,attribute)
- `FindCorrespondingStartEndElementPairNavigator` finds both the start and the corresponding end tag.
- `InSubtreeNavigator` finds whole subtrees, by specifying the properties of the "root" element.

For further information refer to the [navigator javadoc](#)

##### 2.5.2.2.1.1. Using navigators

Using a navigator is a rather simple task. The transformer peeks or gets the next event and calls `Navigator.fulfillsCriteria` - if true is returned the transformer should be process that event somehow.

##### 2.5.2.2.1.2. Implementing a navigator

Creating a new navigator is a rather simple task and just means implementing two methods:

```
import javax.xml.stream.events.XMLEvent;

public class MyNavigator implements Navigator {
    public boolean fulfillsCriteria(XMLEvent event) {           ❶
        return false;
    }

    public boolean isActive() {                                 ❷
        return false;
    }
}
```

- ❶ This method returns true if the event matches the criteria of the navigator.
- ❷ Returns the result of the last invocation of fulfillsCriteria.

### 2.5.2.2.2. Implementing a transformer

The next example should show you an transformer featuring navigators and implicit state handling through function calls.

```
public class DaisyLinkRewriteTransformer extends AbstractStAXTransformer {
    @Override
    protected void produceEvents() throws XMLStreamException {
        while (this.getParent().hasNext()) {
            XMLEvent event = this.getParent().nextEvent();
            if (this.anchorNavigator.fulfillsCriteria(event)) {           ❶
                ArrayList<XMLEvent> innerContent = new ArrayList<XMLEvent>();
                LinkInfo linkInfo = this.collectLinkInfo(innerContent);    ❷
                if (linkInfo != null) {
                    linkInfo.setNavigationPath(this.getAttributeValue(event.asStartElement(),
                        PUBLISHER_NS, "navigationPath"));                ❸
                }

                this.rewriteAttributesAndEmitEvent(event.asStartElement(), linkInfo);    ❹

                if (innerContent.size() != 0) {
                    this.addAllEventsToQueue(innerContent);
                }
            }
            /* ... */
        }
        /* ... */
    }

    private LinkInfo collectLinkInfo(List<XMLEvent> events) throws XMLStreamException {
        Navigator linkInfoNavigator = new InSubtreeNavigator(LINK_INFO_EL);    ❺
        Navigator linkInfoPartNavigator = new FindStartElementNavigator(LINK_PART_INFO_EL);
        LinkInfo linkInfo = null;

        while (this.getParent().hasNext()) {
            XMLEvent event = this.getParent().peek();                        ❻

            if (linkInfoNavigator.fulfillsCriteria(event)) {
                event = this.getParent().nextEvent();
                if (linkInfoPartNavigator.fulfillsCriteria(event)) {
                    /* ... */
                    String fileName = this.getAttributeValue(event.asStartElement(), "fileName");
                    if (!"".equals(fileName)) {
                        linkInfo.setFileName(fileName);
                    }
                }
                /* ... */
            } else if (event.isCharacters()) {
                events.add(this.getParent().nextEvent());
            } else {
                return linkInfo;
            }
        }
        return linkInfo;
    }
}
```

```
private void rewriteAttributesAndEmitEvent(StartElement event, LinkInfo linkInfo) ;
}
```

- ❶ The transformer checks for anchors in the XML.
- ❷ If an anchor is found, it invokes a method which parses the link info if there is any. The additional array is for returning any events, which were read but do not belong to the linkinfo.
- ❸ This method finally writes the start tag with the correct attributes taken from the parsed LinkInfo.
- ❹ The events, which were read but not parsed, are finally added to the output of the transformer.
- ❺ The parser for the linkInfo object uses itself also navigators ...
- ❻ ... and reads more events from the parent.

### 2.5.2.3. StAX serializer

The `StAXSerializer` pulls and serializes the `StAX` events from the pipeline.

```
public class NullSerializer extends AbstractStAXPipelineComponent
    implements StAXConsumer, Finisher {

    private StAXProducer parent;                                ❶

    public void initiatePullProcessing() {                       ❷
        try {
            while (this.parent.hasNext()) {                     ❸
                XMLEvent event = this.parent.nextEvent();
                /* serialize Event */
            }
        } catch (XMLStreamException e) {
            throw new ProcessingException("Error during writing output elements.", e);
        }
    }

    public void setParent(StAXProducer parent) {                ❹
        this.parent = parent;
    }

    public String getContentType() ;                             ❺
    public void setOutputStream(OutputStream outputStream) ;
}
```

- ❶ The Finisher has to pull from the previous pipeline component..
- ❷ In case of `StAX` the last pipeline component has to start pulling for Events.
- ❸ The serializer pulls the next Event from the previous component and should as next step serialize it.
- ❹ During the pipeline construction the `setParent` is called to set the previous component of the pipeline.
- ❺ These two methods are defined in the Finisher and allow to set the `OutputStream` (if the Serializer needs any) and to retrieve the content-type of the result..

## 2.5.3. Using StAX and SAX components in the same pipeline

The `StAX` pipeline offers interoperability to `SAX` components to a certain degree. However due their different paradigms only two use cases are currently implemented: Wrapping a `SAX` component in a `StAX` pipeline and a `StAX`-to-`SAX` pipeline, which starts with `StAX` components and finishes with `SAX`.

### 2.5.3.1. Wrapping a SAX component in a StAX pipeline

This allows to use existing `SAX` components in a `StAX` pipeline. Beware the overhead of the conversion of `StAX`->`SAX`->`StAX` - so no performance gains from a `SAX` component can be expected.

```
Pipeline<StAXPipelineComponent> pipeStAX = new NonCachingPipeline<StAXPipelineComponent>(❶);
pipeStAX.addComponent(new StAXGenerator(input));                                ❷
```

```

pipeStAX.addComponent(new SAXForStAXPipelineWrapper(new CleaningTransformer()));
pipeStAX.addComponent(new StAXSerializer());
pipeStAX.setup(System.out);
pipeStAX.execute();

```

- ❶ The pipeline uses a `StAXGenerator` - which produces StAX events.
- ❷ In order to embed a single SAX component in a StAX pipeline, the `SAXForStAXPipelineWrapper` is needed. The constructor argument is the SAX component.
- ❸ Although the `CleaningTransformer` would emit SAX calls - the wrapper converts them back to the appropriate StAX events the `StAXSerializer` can write..

### 2.5.3.2. StAX-to-SAX pipeline

This converter allows to mix StAX and SAX components - but is limited to starting with StAX and then switching to SAX.

```

Pipeline<PipelineComponent> pipeStAX = new NonCachingPipeline<StAXPipelineComponent>();
pipeStAX.addComponent(new StAXGenerator(input));
pipeStAX.addComponent(new StAXToSAXPipelineAdapter());
pipeStAX.addComponent(new CleaningTransformer());
pipeStAX.addComponent(new XMLSerializer());
pipeStAX.setup(System.out);
pipeStAX.execute();

```

- ❶ The pipeline starts with a `StAXGenerator`.
- ❷ The adapter converts the StAX events to SAX method calls.
- ❸ The `CleaningTransformer` is a SAX component.
- ❹ The `XMLSerializer` writes the SAX method calls to a file.

### 2.5.4. Java 1.5 support

In order to use StAX with Java 1.5 an additional dependency is needed in the project's `pom.xml`.

```

<dependency>
  <groupId>org.codehaus.woodstox</groupId>
  <artifactId>wstx-asl</artifactId>
  <version>3.2.7</version>
</dependency>

```

Using woodstox is simpler, as the reference implementation depends on JAXP 1.4, which is not part of Java 1.5.

## 2.6. Utilities

TBW: XMLUtils, TransformUtils

---

## **Chapter 3. Sitemaps**

### **3.1. What is a sitemap?**

TBW

### **3.2. Sitemap evaluation?**

TBW

### **3.3. Expression languages**

TBW

### **3.4. Spring integration**

TBW

### **3.5. Embedding a sitemap**

TBW

---

# Chapter 4. Web applications

## 4.1. Usage scenarios

TBW

## 4.2. Servlet-Service framework integration

TBW: Composition, servlet: protocol, inheritance

## 4.3. Sitemaps in an HTTP environment

TBW: Status codes, Conditional GET requests, Mime-type handling

## 4.4. System setup

TBW: Logging, JNet, Configuration, Spring integration Deployment: Blocks as deployment units AND Creating a web archive (WAR), Deveopment with Eclipse and Maven

## 4.5. Connecting pipeline fragments

TBW

## 4.6. RESTful web services

### 4.6.1. Sitemap based RESTful web services

#### 4.6.1.1. Introduction

TBW: REST controller, Rendering views using StringTemplate, Request-wide transactions (incl. Subrequests)

### 4.6.2. JAX-RS based controllers (JSR311)

#### 4.6.2.1. Introduction

JAX-RS (JSR 311) is the Java standard for the development of RESTful web services. It provides a set of annotations that, when being applied, define resources that are exposed by using Uniform Resource Identifiers (URIs).

The [wiki of the Jersey project](#) that provides the Reference Implementation of JAX-RS contains a lot of useful information about how to define REST resources.

The main pieces of the JAX-RS/Cocoon-integration are

- the `CocoonJAXRSServlet` Servlet-Service, which is responsible for the JAX-RS integration into the Cocoon Servlet-Service framework, and
- the `URLResponseBuilder` class, which allows calling resources provided by other Servlet-Services (usually Cocoon pipelines exposed by sitemaps).

#### 4.6.2.2. Cocoon and JAX-RS by example

Adding support for JAX-RS services to your Cocoon application requires following three steps:

- Add the `cocoon-rest` module as a dependency.
- Add the `CocoonJAXRSServlet` Servlet-Service
- Add at least one JAX-RS root resource

##### 4.6.2.2.1. Cocoon-Rest dependency

The first step is to add the `cocoon-rest` module to your Cocoon application:

```
<dependency>
  <groupId>org.apache.cocoon.rest</groupId>
  <artifactId>cocoon-rest</artifactId>
</dependency>
```

##### 4.6.2.2.2. JAX-RS resource

Then at least one JAX-RS resource is required:

```
@Path("/sample") ❶
public class SampleRestResource { ❷

    private Settings settings;

    @GET ❸
    @Path("/parameter-passing/{id}") ❹
    public Response anotherService(
        @PathParam("id") String id, ❺
        @QueryParam("req-param") String reqParam, ❻
        @Context UriInfo uriInfo, ❼
        @Context Request request) { ❽

        Map<String, Object> data = new HashMap<String, Object>(); ❾
        data.put("name", "Donald Duck");
        data.put("id", id);
        data.put("reqparam", reqParam);
        data.put("runningMode", this.settings.getProperty("testProperty"));

        return URLResponseBuilder.newInstance("servlet:sample:/controller/screen", data) ❿
            .build();
    }

    public void setSettings(Settings settings) {
        this.settings = settings;  (11)
    }
}
```

- ❶ The `@javax.ws.rs.Path` annotation identifies the URI path that this resource class or class method will serve requests for. The path is relative to the mount point of the servlet-service that references this

resource.

- ② A JAX-RS root resource.
- ③ The `@javax.ws.rs.GET` annotations indicates that this method responds to HTTP GET requests.
- ④ Again a `@Path` annotation, but this time at method level. In this example requests for `sample/parameter-passing/NNN` will be handled by the `anotherService()` method.
- ⑤ The `@PathParam` annotation binds the URI template value of `id` to the method parameter `id`.
- ⑥ The `@QueryParam` annotation binds the request parameter `req-param` to the method parameter `reqParam`.
- ⑦ By annotating the `URIInfo` method parameter with the `@Context` annotations, a current instance of `URIInfo` is passed to the method.
- ⑧ By annotating the `Request` method parameter with the `@Context` annotations, a current instance of `Request` is passed to the method.
- ⑨ A map of `String/Object` is collected.
- ⑩ The `URLResponseBuilder` sends the result of the passed URL as response. It allows passing a map of `String/Object` which are available in the called resource (usually a pipeline).

In this case the `servlet:` protocol is used. It allows accessing URLs (that usually expose pipelines) defined by other Servlet-Services.

A setter method to allow injecting the `Setting` bean.

#### 4.6.2.2.3. JAX-RS resource as Spring bean

This resource has to be configured as Spring bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="org.apache.cocoon.sample.rest.resource.one"
    class="org.apache.cocoon.sample.jaxrs.SampleRestResource">
    <property name="settings"
      ref="org.apache.cocoon.configuration.Settings" />
  </bean>

</beans>
```

- ① The `SampleRestResource` is a usual Spring bean and in this example it gets the `org.apache.cocoon.configuration.Settings` bean injected.

#### 4.6.2.2.4. Servlet-Service integration

Finally the Spring bean has to be exposed by the `CocoonJAXRSServlet`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:servlet="http://cocoon.apache.org/schema/servlet"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://cocoon.apache.org/schema/servlet
    http://cocoon.apache.org/schema/servlet/cocoon-servlet-1.0.xsd">

  <!-- A servlet-service that exposes JAX-RS REST endpoints. -->
  <bean id="org.apache.cocoon.sample.rest.servlet"
    class="org.apache.cocoon.rest.jaxrs.container.CocoonJAXRSServlet">
    <servlet:context mount-path="/jax-rs"
      context-path="blockcontext:/cocoon-sample/">
      <servlet:connections>
        <entry key="sample" value-ref="org.apache.cocoon.sample.servlet" />
      </servlet:connections>
    </servlet:context>

    <property name="restResourcesList">
      <list>
        <ref bean="org.apache.cocoon.sample.rest.resource.one" />
      </list>
    </property>
  </bean>
```



```
</list>
</property>
</bean>
</beans>
```

- ❶ The `CocoonJAXRSServlet` exposes REST resources.
- ❷ The mount path of this Servlet-Service is `/jax-rs`.
- ❸ Connections to other Servlet-Services.
- ❹ A list of JAX-RS resources, which also have to be Spring beans, is exposed.

Alternatively a `<map>` of resources can be injected by the `restResourceMap` property.

## 4.7. Caching and conditional GET requests

TBW

## 4.8. Authentication

TBW

## 4.9. Testing

TBW: Integration tests

## 4.10. Profiling support

A Cocoon request goes through many components; while performing a Cocoon request, servlets, sitemaps and pipeline components are being executed. It is also quite common that Cocoon requests are cascaded which makes it sometimes difficult to understand what exactly was happening. Cocoon Profiling enables you to profile any request to your website.

### 4.10.1. Module configuration

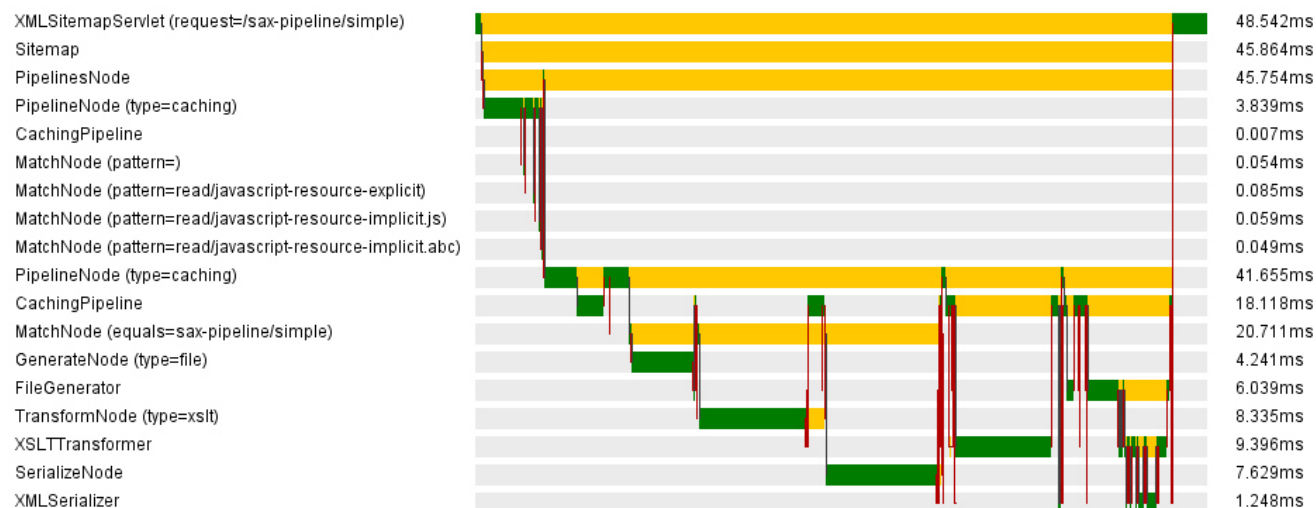
In order to use Cocoon-profiling, you simply have to include the `cocoon-profiling` jar in your classpath. Cocoon-profiling uses Spring AOP, so no further configuration is needed.

### 4.10.2. Using Cocoon Profiling

Cocoon-profiling provides several generators for xml and graphical output, which is used by the Firebug plugin. However, you can directly access the data using

- `yourdomain.com/controller/profiling/{id}` for the xml file (xml schema see `profilingSchema.xsd` in `cocoon-profiling`)
  - `yourdomain.com/controller/profiling/{id}.png` for the graphical representation (example see below)
- where `{id}` is the id of the request. This id (`X-Cocoon-Profiling-ID`) can be found in the header of the original request.

This is an example for the graphical representation:



#### 4.10.2.1. How can I enable/disable cocoon-profiling on the fly?

Cocoon-profiling is enabled per default. If you just want to use it, you can skip this section.

If you want to start or stop profiling while the server is running, you can use the management bean provided by cocoon-profiling. Cocoon-monitoring automatically exposes this MBean; just make sure that cocoon-monitoring is in your classpath. You can then connect to the server process using jconsole and call the "start" and "stop" method of the org.apache.cocoon.profiling MBean.

Keep in mind that cocoon-profiling uses Spring AOP to intercept method calls, which cannot be enabled or disabled at runtime. Therefore, a disabled cocoon-profiling still affects the performance (at least a little bit). Only removing cocoon-profiling from the classpath and restarting the server guarantees maximum performance.

#### 4.10.2.2. I need cocoon-profiling to profile my custom cocoon component, how can I do that?

Generally, you don't have to change anything. Cocoon-profiling uses Spring AOP to profile all cocoon components and servlets as well as any subclass or cocoon interface implementation.

Advanced users might want to add their own Profiler class to cocoon-profiling if they have a specific component with specific parameters or other bits of data they want cocoon-profiling to collect.

### 4.10.3. Using Firebug with Cocoon-profiling











Getting the Firebug Plugin able to work is really easy. You'll need to install Firebug 1.4X from <http://getfirebug.com/releases/firebug/1.4X> and then install the Firebug .xpi from cocoon-profiling-firebug (opening it with Firefox should suffice). The generate the xpi from the sources, simply switch to the cocoon-profiling-firebug folder and type `mvn install`. This will automatically build a new xpi. After installing the plugins, you should have a little bug on the bottom right corner of your Firefox. Clicking will open it, then you can navigate to the net-panel (you might want to be sure its activated) and there open up a request (you might need to refresh). In the request, there should be a tab called "Cocoon 3 Profiling", if you click it, and the Profiling Service works, you should now see the generated profiling data.

The data itself is presented in two textboxes. The left one is the tree which resembles the .xml that the profiling-component generates, the right one displays the elements, properties, return-values and the profiler of a given (= selected) row in the left tree. You can navigate through the left tree, and based on which row you

currently have selected, the right textbox will display the names and values. To add some visual help, pictures are being displayed, based on which kind of element is displayed.

The underline is:

**Table 4.1. Icons**

 Argument	 Component	 Exception
 Invocation	 Node	 Profiler
 Property	 Return-value	 Servlet
 Sitemap		

You also have two options which can change what you'll be able to see in the Firebug Plugin. These are located on the net Tab Option Panel, which is a little triangle. The options are marked {c3p} and allow you to Show / Hide the Sitemap and to change if the invocations element of the tree is closed or opened. These changes will require a reload, either by reloading your browser, or by clicking the "Reload"-Button in the Profiling Panel.

The other button on the panel will open a new tab in your Firefox in which a graphical outline will be shown which lists your components and the execution time of each of these components. Be aware, though, that the first profiling cycle will present you a kind of wrong picture (it takes a lot longer), because of the java-intern class-loading and other administrative stuff. You'll need to reload the request, not the picture, to change this.

#### 4.10.3.1. Customizing your Firebug Plugin

To customize your Firebug Plugin, go to the plugin folder in your Firefox, select the Profiling plugin, then navigate to `chrome\skin\classic\cocoon3profiling.css`. In this css, you can change the height and width of the trees and of their columns. Simply navigate to the element you want to change, and change the value in there.

## 4.11. Monitoring support

This module gives you the possibility of monitoring Cocoon 3 applications. It expose a simple API via `Spring JMX MBeans` and it can be user via `jconsole` or other JMX console.

### 4.11.1. Module configuration

All module configurations are defined in `META-INF/cocoon/spring/cocoon-monitoring.xml`. Each part of module can be enabled or disabled by commenting (or removing) appropriate entry in configuration file, by default all parts are enabled.

### 4.11.2. Available parts

#### 4.11.2.1. Inspect logging settings and reconfigure them

##### 4.11.2.1.1. Inspect all configured loggers

The operation `getLoggers()` returns a list of all configured Log4j loggers in the application, it consists of pairs: `<class or package name> (category) => <logging level>`

##### 4.11.2.1.2. Permanent change of a logging level

For doing permanent changes of logging level for particular class or package use the operation `setLoggingLevel(String category, String newLogLevel)`. Where `Category` is name of class or package with you want to change logging level and `newLogLevel` is one of logging level: OFF, INFO, WARN, ERROR, FATAL, TRACE, DEBUG, ALL (this parameter isn't case sensitive, so you can also use lower case names).

##### 4.11.2.1.3. Temporal change of a logging level

For doing temporal changes of logging level for particular class or package use the operation `setLoggingTempoporalLevel(String category, String temporalLogLevel, String timeOut)`. First two parameters are same as in `setLoggingLevel()`, last one determinate how long this logging level should be used (after that amount of time logging level would be set back to old level). `timeOut` parameter should match regular expression: `^[0-9.]+[dhm]?$`, for example if value of `timeOut` is set for `1,5h` that means that new logging level would be active for one and half hour.

##### 4.11.2.1.4. Load a new configuration file

For loading a completely new configuration file use the operation `loadNewConfigurationFile(String path)`. This method is capable for both XML and `properties` configuration files. There is only one parameter `path` that should contain absolute path to new configuration file located locally on the server. Before performing any action that file is validated. First of all the file extension (there are only two appropriate extensions: `*.xml` and `*.properties`) is checked. The next validation step is different for both files, for XML files its content is validated against the Log4j DTD or schema then all output log files are checked that they exist and they are writable. `Properties` configuration files are validated that they contain at least one appender and each output file directory exists and is writable. These validations are done to prevent Log4j to get into an inconsistent state.

#### 4.11.2.2. Inspect available Servlet-Services

Every single Servlet-Service makes his own node in JXM MBean tree. Such node contains below above functions.

##### 4.11.2.2.1. Get path of Servlet-Services

Function `getServletServiceMountPaths()` returns Servlet-Service mount path.

##### 4.11.2.2.2. Get all connections for Servlet-Service

Function `getServletServiceConnections()` returns an array of `String` contains all connections names for given Servlet-Service. Every connection is represented by: `<short name> => <full qualified bean name>`

##### 4.11.2.2.3. Get informations about Servlet-Service

Function `getServletServiceInfo()` returns information about Servlet-Service, such as author, version, and copyright.

#### 4.11.2.2.4. List Servlet-Service parameters

Function `getServletServiceInitParameters()` list all init parameters provided for that Servlet-Service.

#### 4.11.2.3. Overview of cache entries

This module contains three smaller submodules:

[CacheMonitor](#)

[CacheBurstActions](#)

[CacheEntriesMonitor](#)

##### 4.11.2.3.1. CacheMonitor

This submodule exposes all configured caches on with basic operations on every cache.

###### 4.11.2.3.1.1. Clear cache

Operation `clear()` remove all cache entry's from this particular cache.

###### 4.11.2.3.1.2. List all cache key

Operation `listKey()` returns list of all `CacheKeys` that are stored in this particular cache.

###### 4.11.2.3.1.3. Removing cache entry

If you want remove single cache entry you should use operation `removeKey(String)`, where parameter is `CacheKey` name taken from `listKey()` result. This operation returns `true` if it was successes otherwise it return `false`.

###### 4.11.2.3.1.4. Checking size of cache

It is also possible to check size of particular cache using function `size`. It will return human readable size of this cache.

##### 4.11.2.3.2. CacheBurstActions

This module add set of operations that can be performed on all caches. You can find and remove all cache entry's which meet certain requirements, specified in operation parameters.

###### 4.11.2.3.2.1. Find all cache entry's that size is greater then specified value

Operation `listAllGreatherThen(long)` finds all cache entry's in all configured cache's that size (in bytes) is greather then value passed as a parameter.

###### 4.11.2.3.2.2. Find all cache entry's that size is smaller then specified value

Operation `listAllSmalledThen(long)` finds all cache entry's in all configured cache's that size (in bytes) is smaller then value passed as a parameter.

###### 4.11.2.3.2.3. Find all cache entry's that are older then specified date

Operation `listAllOlderThen(String)` finds all cache entry's in all configured cache's that are older then value specified in a parameter. Parameter value must match regular expression: `^\d+[smhd]$\` where each letter stands for:

s - second  
m - minutes  
h - hours  
d - days

#### **4.11.2.3.2.4. Find all cache entry's that are younger then specified date**

Operation `listAllYoungerThen(String)` finds all cache entry's in all configured cache's that are younger then value specified in a parameter. For parameter description see `listAllOlderThen(String)`.

#### **4.11.2.3.2.5. More flexible cache entry's search**

If you want to get more flexibility searching you can use `list(long, long, String, String, Sting, Sting)`. First two `long` parameters limits cache entry's size, their minimum and maximum size. Second two `String` parameters limits cache entry minimum and maximum age (syntax is same as in `listAllOlderThen(String)` and `listAllYoungerThen(String)` functions). Last two `String` parameters are applicable only to `ExpiresCacheKey` instances and they limit result entry's on minimum and maximum experience time of entry. Every result entry meets all limitations. For excluding one (or more) limitation you must pas value -1 for `long` parameters and `null` or empty `String` for rest parameters.

#### **4.11.2.3.2.6. Remove all cache entry's that size is greater then specified value**

Operation `clearAllGreatherThen(long)` perform same search action as `listAllGratherThen(long)` but it removes cache entry's that fulfilled requirements instead of listing them. It will return `true` if every cache entry was successfully removed, after first failure of removing cache entry this operation will stop and return `false`.

#### **4.11.2.3.2.7. Remove all cache entry's that size is smaller then specified value.**

Operation `clearAllSmallerThen(long)` perform same search action as `listAllSmallerThen(long)` but it removes cache entry's that fulfilled requirements instead of listing them. It will return `true` if every cache entry was successfully removed, after first failure of removing cache entry this operation will stop and return `false`.

#### **4.11.2.3.2.8. Remove all cache entry's that are older then specified value.**

Operation `clearAllOlderThen(String)` perform same search action as `listAllOlderThen(String)` but it remove cache etry's that fulfilled requirements instead of listing them, this operation also require same parameter schema as in `listAllOlderThen(String)`. It will return `true` if every cache entry was successfully removed, after first failure of removing cache entry this operation will stop and return `false`.

#### **4.11.2.3.2.9. Remove all cache entry's that are younger then specified value.**

Operation `clearAllYoungerThen(String)` perform same search action as `listAllYoungerThen(String)` but it remove cache etry's that fulfilled requirements instead of listing them, this operation also require same parameter schema as in `listAllYoungerThen(String)`. It will return `true` if every cache entry was successfully removed, after first failure of removing cache entry this operation will stop and return `false`.

#### **4.11.2.3.2.10. More flexible cache entry's removing**

Operation `clear(long, long, String, String, String, String)` perform same search action and takes same parameters as described above `list(long, long, String, String, String)` please see it if you are looking for detailed description.

#### **4.11.2.3.2.11. Extending BurstCacheAction module**

You can add your own burst actions. It is very simple, just obtain instance of

`org.apache.cocoon.monitoring.cache.CacheBurstActions` from the container and you can perform any action on cache entry's using `performActionOnCaches(long, long, String, String, String, String, CacheAction)` method. Meaning of each parameter in this method is same as in `list(long, long, String, String, String, String)` and `clear(long, long, String, String, String, String)` methods, but there is one additional parameter, it is implementation of `CacheAction` interface. This interface has only one method `performAction(Cache, CacheKey)` and it will be executed on every cache entry that match passed parameters.

#### 4.11.2.3.3. CacheEntriesMonitor

This module enables overview of cache entry's that are connected with specified pipeline. This module will only publish cache entry's for those pipeline's that had `jmx-group-id` parameter set for unique value/name. This module also require additional refresh action, it can be performed by user or it can be executed in some period of time eg every one minute. Behavior of refresh action can be configured by `CacheEntriesMonitorInitializer` constructor parameter or on JMX.

##### 4.11.2.3.3.1. Configuring refresh action

`AutoRefresh` action can be enabled and disabled during runtime. This action will register new `MBeans` that are connected with new cache entry's and unregister old `MBeans` that was connected with expired or removed cache entry's.

###### 4.11.2.3.3.1.1. Enable `AutoRefresh` action

For enable `AutoRefresh` action call operation `enableAutoRefresh(long)` where `long` parameter is period time (in millisecond) between each refresh. This operation will stop previously configured `AutoRefresh` action and run new with new time period.

###### 4.11.2.3.3.1.2. Disable `AutoRefresh` action

Operation `disableAutoRefresh()` will stops actual running `AutoRefresh` action.

###### 4.11.2.3.3.1.3. Manually perform refresh action

Operation `performRefreshAction()` will immediately refre's published cache entry's.

##### 4.11.2.3.3.2. Cache entry's operations

On each published cache entry you can perform that set of actions:

###### 4.11.2.3.3.2.1. Obtain `CacheKey` for that entry

Operation `getCacheKey()` will return `CacheKey` connected with that entry.

###### 4.11.2.3.3.2.2. Obtain cache value

Operation `getCacheValue()` will return value of this cache entry.

###### 4.11.2.3.3.2.3. Set cache value

Operation `setCacheValue(String)` will set new value of this cache entry.

###### 4.11.2.3.3.2.4. Obtain size of entry

Operation `getSize()` will return human readable size of current entry.

## 4.12. Tutorial

TBW



---

# Chapter 5. Wicket Integration

## 5.1. Introduction

Apache Wicket has become one of the most popular web frameworks of these days. Especially developers with a strong Java background benefit from its Java-centric approach because all object-oriented features can be applied. This results in highly reusable code.

On the other side Cocoon implementing the pipe/filter pattern has its merits in the field of generating resources in different output formats.

The Cocoon-Wicket integration module bridges between those two web application frameworks in order to use the strengths of both. This integration supports the integration of Cocoon into Wicket as well as the integration of Wicket into Cocoon.

*Note:* This is *not* an introduction into Apache Wicket. This documentation explains to the experienced Wicket user what needs to be done to integrate Cocoon 3 into a Wicket application.

## 5.2. Integrate Cocoon into Wicket

The integration of Cocoon into Wicket is available in several ways:

- A *Cocoon sitemap* can be mounted as [IRequestTargetUrlCodingStrategy](#)
- A single *Cocoon pipeline* can be mounted as [IRequestTargetUrlCodingStrategy](#) (not implemented yet)
- A *CocoonSAXPipeline Wicket component* can be added to a [WebPage](#)

Whatever approach is chosen, the first step is adding `cocoon-wicket` and all its transitive dependencies to your project's classpath:

```
<dependency>
  <groupId>org.apache.cocoon.wicket</groupId>
  <artifactId>cocoon-wicket</artifactId>
  <version>3.0.0-alpha-2</version>
</dependency>
```

### 5.2.1. Mount a Cocoon sitemap

Mounting in the context of Wicket means a class implementing [IRequestTargetUrlCodingStrategy](#) is added to a Wicket web application. This interface is implemented by `CocoonSitemap`:

```
import org.apache.cocoon.wicket.target.CocoonSitemap;
import org.apache.wicket.protocol.http.WebApplication;

public class SomeWebApplication extends WebApplication {

    @Override
    protected void init() {
        ...
        this.mount(new CocoonSitemap("/sitemap", "/sitemap.xmap.xml"));
        ...
    }
}
```

- ❶ The first parameter is the mount path which is a part of the request URI that should be handled by `CocoonSitemap`. The second parameter is the location of the sitemap relativ to the servlet context.

Additionally you have to make sure that all Spring bean definitions provided by the Cocoon modules are loaded into the web application's Spring application context. Cocoon's own bean definitions are located in `META-INF/cocoon/spring/*.xml`.

The simplest solution for this task is referring to the Cocoon Spring Configurator in your main Spring application context, which is usually located in `[servlet-context-base-directory]/WEB-INF/applicationContext.xml`. It will automatically load all bean definitions located in `META-INF/cocoon/spring/*.xml` of all libraries on the classpath. The [Cocoon Spring Configurator documentation](#) contains further details.

Note that the Spring Configurator is one of the transitive dependencies of `cocoon-wicket`.

That's it! Everything else is the same as using Cocoon 3 outside of Wicket except that the `servlet:/` protocol won't work in this environment.

## 5.2.2. Mount a Cocoon pipeline

*NOTE: This hasn't been implemented yet!*

Mounting a Cocoon pipeline follows the same idea as mounting a whole sitemap. However, it's only a single pipeline that is added to Wicket's URI path and that this can be done without having to write any XML.

All that needs to be done is subclassing from `org.apache.cocoon.wicket.AbstractCocoonPipeline` and implementing its `addComponents` method:

```
import com.mycompany.MyCocoonPipeline;
import org.apache.wicket.protocol.http.WebApplication;

public class SomeWebApplication extends WebApplication

    @Override
    protected void init() {
        ...
        this.mount(new MyCocoonPipeline("/my-pipeline"));
        ...
    }
}
```

- ❶ The only parameter is the path where the pipeline should be mounted by Wicket.

In `MyCocoonPipeline` all that needs to be done is subclassing from `org.apache.cocoon.wicket.AbstractCocoonPipeline` and implementing its abstract method `addComponents`:

```
package com.mycompany;
import org.apache.cocoon.wicket.AbstractCocoonPipeline;
import org.apache.wicket.protocol.http.WebApplication;

public class MyCocoonPipeline extends
    org.apache.cocoon.wicket.AbstractCocoonPipeline<SAXPipelineComponent>

    @Override
    protected void addComponents() {
        this.addComponent(new FileGenerator(this.getClass().getResource("test.xml")));
        this.addComponent(new XSLTTransformer(this.getClass().getResource("test.xsl")));
        this.addComponent(new XMLSerializer());
    }
}
```

- 1 Add all pipeline components that are required.

### 5.2.3. CocoonSAXPipeline Wicket component

The third alternative of using Cocoon in Wicket is adding a Cocoon pipeline as [WebComponent](#). This is as simple as instantiating `CocoonSAXPipeline` and adding all generators and transformers that are required:

```
import org.apache.cocoon.pipeline.NonCachingPipeline;
import org.apache.cocoon.sax.SAXPipelineComponent;
import org.apache.cocoon.sax.component.StringGenerator;
import org.apache.cocoon.sax.component.XSLTTransformer;
import org.apache.cocoon.wicket.CocoonSAXPipeline;
import org.apache.wicket.markup.html.WebPage;

public class Homepage extends WebPage {

    public Homepage() {
        CocoonSAXPipeline pipeline = new CocoonSAXPipeline("cocoon-pipeline-component", ❶
            new NonCachingPipeline<SAXPipelineComponent>());
        pipeline.addComponent(new StringGenerator("<b>hello, Cocoon!</b>")); ❷
        pipeline.addComponent(new XSLTTransformer(
            this.getClass().getResource("transform.xslt"));
        this.add(pipeline);
    }
}
```

- ❶ Instantiate the component
- ❷ Adding SAX pipeline components

The pipeline's result is added to the HTML produced by this page. This is the reason why only generators and transformers can be added to this component because the pipeline is always serialized as XHTML. An `XHTMLSerializer` is added implicitly to each pipeline.

## 5.3. Integrate Wicket into Cocoon

*Note:* The integration of Wicket into Cocoon is mostly a proof of concept. It is experimental and has neither been optimized nor tested yet.

The integration of Wicket into Cocoon means that the output of Wicket is added to the content stream of a pipeline. The currently available solution is a reader but alternatively a generator or a transformer would offer an even more alternatives.

As pointed out for the Wicket-Cocoon integration, the first step in every case is adding `cocoon-wicket` and all its transitive dependencies to your project's classpath:

```
<dependency>
  <groupId>org.apache.cocoon.wicket</groupId>
  <artifactId>cocoon-wicket</artifactId>
  <version>3.0.0-alpha-2</version>
</dependency>
```

### 5.3.1. Wicket reader

By using the Wicket reader a servlet request is referred to a Wicket web application. The current implementation expects exactly one Wicket web application being available as Spring bean:

```
<beans>
  <bean id="wicketWebapp"
    class="com.mycompany.MyWicketWebapp"/>
```

```
</beans>
```

By adding the bean definition file as resource into `META-INF/cocoon/spring` the Wicket web application bean will be loaded automatically.

Then the Wicket reader has to be used in the sitemap:

```
<map:sitemap>
  <map:pipelines>
    <map:pipeline type="noncaching">
      <map:match wildcard="my-wicket-app/**">
        <map:read type="wicket" base-path="/my-wicket-app" />
      </map:match>
    </map:pipeline>
  </map:pipelines>
</map:sitemap>
```

❶  
❷

- ❶ Use a `**` wildcard to match all requests that start with `my-wicket-app`.
- ❷ The name of the reader is `wicket`. It's also required to define the base path so that Wicket can calculate relative URLs correctly.