# Cocoon2

## Your first Cocoon application using Maven 2

In this tutorial, you will:

* Create a Cocoon block (the application resources and logic)
* Import the block as a project in Eclipse
* Start the block as a web application and access it from your browser

## Creating a Cocoon block

Create a new directory which will be the root directory of your Cocoon application and change into it. For this tutorial, let's name it getting-started-app.The next step is to create a Cocoon block to contain your custom application. The development of any Cocoon web application should be done within one or more  blocks.  Cocoon blocks are little different to other Maven blocks except they have a particular expected [directory and file structure](http://cocoon.apache.org/2.2/core-modules/core/2.2/1263_1_1.html).

**Note:** You could manually create your blocks, but a simpler and less error-prone process is to use Maven's 'archetype' plugin.  This allows you to select from a list of block types and have Maven create them for you.  By default, the list of Maven archetypes is quite large.  For convenience, a catalog of Cocoon specific archetypes is available on the Cocoon web site and you will make use of this in the following instructions.

Run the following Maven command:

**mvn archetype:generate -DarchetypeCatalog=http://cocoon.apache.org**

Maven will generate some output and then interactively ask you for some information:

Choose archetype:

1: remote -> cocoon-22-archetype-block-plain (Creates an empty Cocoon block; useful

 if you want to add another block to a Cocoon application)

2: remote -> cocoon-22-archetype-block (Creates a Cocoon block containing some small

 samples)

3: remote -> cocoon-22-archetype-webapp (Creates a web application configured to

host Cocoon blocks. Just add the block dependencies)

Choose a number: (1/2/3): **2**

Maven first asks which archetype you want to create.  Item 2 in this list is the standard Cocoon block with some small sample files already in place.

Define value for groupId: : **com.mycompany**

Define value for artifactId: : **myBlock1**

Define value for version: 1.0-SNAPSHOT: : **1.0.0**

Define value for package: : **com.mycompany.myBlock1**

Maven then asks a few more questions.  Just enter the details as shown above.  The groupId is a label that you can use to group all the blocks in your application.  The artifactId is the unique label that will identify this block from all others.  It is also the name of the directory that will be created to contain the block's files.  The package is the Java package that any Java files in the block will be assigned to.

## Import the block in Eclipse

Change into the myBlock1 directory and call

**mvn eclipse:eclipse**

This will create the necessary project descriptors for Eclipse. In Eclipse you can import the project into your workspace: *File >  Import... > Existing Projects into Workspace*

## Run the block as Java web application

After creating the Cocoon block you probably want to run it. For this purpose there is a Maven plugin that generates a minimal web application that loads your block. The pom.xml of your block already contains the necessary settings.

**mvn jetty:run**

and point your browser at http://localhost:8888/myBlock1/ and get a hello world page

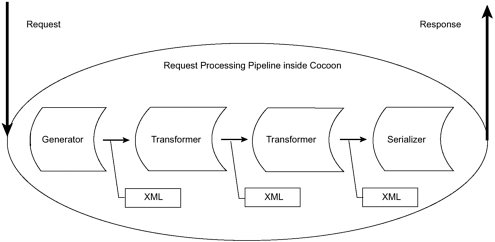
**Note:** The mentioned minimal web application is automatically created, when mvn jetty:run is invoked. This happens because the rcl goal of the Cocoon plugin is bound to the Maven build lifecycle which is invoked too, when the jetty:run goal is executed. See the block's pom.xml for details.

## Your first XML pipeline (publishing)

In this tutorial you will

* get an overview of what Cocoon pipelines and sitemaps are
*  create your first pipeline that emits XML
* enhance this first pipeline by adding an XML transformation step based on XSLT that produces HTML as output
* write a pipeline that creates a PDF document

### Overview



Cocoon is based around the concept of pipelines: If a request comes in, a [**generator**](http://cocoon.apache.org/2.2/core-modules/core/2.2/850_1_1.html) produces [SAX events](http://www.saxproject.org/) that represent the XML, [**tranformers**](http://cocoon.apache.org/2.2/core-modules/core/2.2/851_1_1.html) can alter the stream of SAX events and a [**serializer**](http://cocoon.apache.org/2.2/core-modules/core/2.2/849_1_1.html) finally creates an output stream which is sent to the client. **Sitemaps** are used to define those pipelines and connect them with requests. These connections are defined by some kind of rules. In the context  of Cocoon  those rules are named matchers. Whenever one of those rules (*matcher*) matches, a pipeline is executed.

### Create a pipeline in the sitemap

First, open src\main\resources\COB-INF\sitemap.xmap in your favorite XML editor. That's your block's base sitemap. There are already some pipelines there (/sitemap/pipelines) but for the sake of this tutorial, let's create your own pipeline.

<map:pipelines>

**<map:pipeline>**

**<map:match pattern="myFirstPipeline">**

**<map:generate src="myXmlFile.xml" type="file"/>**

**<map:serialize type="xml"/>**

**</map:match>**

**</map:pipeline>**

[...]

</map:pipelines>

The map:match element has an attribute @pattern. If the request http://localhost:8888/myBlock1/**myFirstPipeline** comes in, this matcher *matches* and the request processing continues with the generation of myXmlFile.xml. This file has to be in the same directory as the sitemap:

After saving the sitemap and the XML file, you can point you browser to http://localhost:8888/myBlock1/myFirstPipeline and you will see the output of your first own pipeline serialized as XML (<map:serialize type="xml"/>).

### Adding an XSLT transformation step

The first pipeline wasn't really exciting, just reading some XML file and serializing it without changing anything. The second example will add a transformation step based on [XSLT](http://www.w3.org/TR/xslt):

<map:pipelines>

<map:pipeline>

 <map:match pattern="myFirstPipeline">

<map:generate src="myXmlFile.xml" type="file"/>

<map:serialize type="xml"/>

</map:match>

**<map:match pattern="mySecondPipeline">**

**<map:generate src="myXmlFile.xml" type="file"/>**

**<map:transform src="myXsltFile.xslt" type="xslt"/>**

**<map:serialize type="html"/>**

**</map:match>**

</map:pipeline>

[...]

</map:pipelines>

The XSLT file is very simple and creates an XML following the structure of XHTML (though without using any namespaces in order to keep things simple):

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">

<html>

<head>

<title>My second XML Pipeline</title>

</head>

<body>

My second XML Pipeline:

<xsl:value-of select="/content"/>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

After saving the sitemap and the XSLT file, you can point you browser to http://localhost:8888/myBlock1/mySecondPipeline and you will see the output, this time its HTML because of the use of the HTML serializer.

**Note:** In this tutorial you have already implemented two pipelines but there is only one pipeline element used in the sitemap. Given this is confusing when you use Cocoon sitemaps the first time. However, there is a reason for this: The <pipeline> element indicates which *pipeline implementation* is used. If there is no @type attribute, the generators, tranformers and serializers are invoked from a caching pipeline implementation.   
When Cocoon developers talk about pipelines, they mean a series of a matcher that includes at least a generator and a serializer. Transformers and other sitemap components are optional.

### Creating a PDF

The first example created XML and the second one HTML. One of the strength of Cocoon is that it makes it very simple to support many different output formats. The result of this third example will be a PDF, again based on *myXmlFile.xml*:

<map:pipelines>

<map:pipeline>

<map:match pattern="myFirstPipeline">

<map:generate src="myXmlFile.xml" type="file"/>

<map:serialize type="xml"/>

</map:match>

<map:match pattern="mySecondPipeline">

<map:generate src="myXmlFile.xml" type="file"/>

<map:transform src="myXsltFile.xslt" type="xslt"/>

<map:serialize type="html"/>

</map:match>

**<map:match pattern="myThirdPipeline">**

**<map:generate src="myXmlFile.xml" type="file"/>**

**<map:transform src="myXml2PdfFile.xslt" type="xslt"/>**

**<map:serialize type="fo2pdf"/>**

**</map:match>**

</map:pipeline>

[...]

</map:pipelines>

This pipeline needs a different XSLT file which transforms the XML of myXmlFile.xml to some XML that can be used by [FOP](http://xmlgraphics.apache.org/fop/) to create PDF:

<?xml version="1.0"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="/">

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

<fo:layout-master-set>

<fo:simple-page-master master-name="page"

page-height="29.7cm"

page-width="21cm"

margin-top="1cm"

margin-bottom="2cm"

margin-left="2.5cm"

margin-right="2.5cm">

<fo:region-before extent="3cm"/>

<fo:region-body margin-top="3cm"/>

<fo:region-after extent="1.5cm"/>

</fo:simple-page-master>

<fo:page-sequence-master master-name="all">

<fo:repeatable-page-master-alternatives>

<fo:conditional-page-master-reference

master-reference="page" page-position="first"/>

</fo:repeatable-page-master-alternatives>

</fo:page-sequence-master>

</fo:layout-master-set>

<fo:page-sequence master-reference="all">

<fo:static-content flow-name="xsl-region-after">

<fo:block text-align="center"

font-size="10pt"

font-family="serif"

line-height="14pt">page <fo:page-number/></fo:block>

</fo:static-content>

<fo:flow flow-name="xsl-region-body">

<fo:block font-size="36pt" space-before.optimum="24pt"

text-align="center">

My second XML Pipeline

</fo:block>

<fo:block font-size="12pt" space-before.optimum="12pt"

text-align="center">

<xsl:value-of select="/content"/>

</fo:block>

</fo:flow>

</fo:page-sequence>

</fo:root>

</xsl:template>

</xsl:stylesheet>

You also have to use a different serializer for this example, a serializer that can use the XML created by the stylesheet and produce PDF based on it. Cocoon ships this serializer as part of the FOP block. Since the block that you created probably doesn't have the FOP block added as dependency, this has to be done before you can test your pipeline. Adding a block requires editing the pom.xml of your block:

<dependency>

<groupId>**org.apache.cocoon**</groupId>

<artifactId>**cocoon-fop-impl**</artifactId>

<version>**1.0.0**</version>

</dependency>

Now you can point your browser at http://localhost:8888/myBlock1/myThirdPipeline and you will get a PDF as result.