**Extensible Component Scanner**

**Version 0.2b**

**User Manual**

**Matthias Rothe (**[**mimarox@users.sourceforge.net**](mailto:mimarox@users.sourceforge.net)**)**

**2010-03-25**

# Before it all starts

It has been pointed out that the Extensible Component Scanner project lacks decent user documentation. In recognition of that fact this manual is put together with the best hopes of solving the issue.

As this project is currently run as a one man show and I wanted to go by releasing early, releasing often I wanted to get the code out first. The documentation took a while to come. But finally here it is! ;-) So keep reading and enjoy!

Table of Contents

[Before it all starts 2](#_Toc257466797)

[1. The purpose of Component Scanning 4](#_Toc257466798)

[1.1. What is a Component 4](#_Toc257466799)

[1.2. Why to scan for Components 5](#_Toc257466800)

[1.3. Is Component Scanning equal to Classpath Scanning 5](#_Toc257466801)

[2. 6](#_Toc257466802)

# The purpose of Component Scanning

## What is a Component

For the matters of the Extensible Component Scanner project a component is some artifact that can be loaded and used as a Java class and conforms to some restrictions. It could either be a precompiled Java class or any class defined in source code of a programming language targeting the Java Virtual Machine (JVM) for which there is an extension available.

Currently only precompiled Java classes can be used, an extension for Groovy is under development. To be considered a component a Java class needs to conform to the following restrictions:

1. It must be a **class**: Interfaces, annotations and enums cannot be components
2. It must be a **primary** class: Nested or inner classes of any kind cannot be components
3. It must be an **instantiable** class: Abstract classes cannot be components

Note:

A class does not need to be public to be considered a component. It could also be package private. However, to be able to instantiate a package private class it must have at least one declared public constructor. This constructor must then also be made accessible before calling it. Supposing you want to work with a default constructor without any parameters you would have to instantiate that class this way:

Constructor<?> constructor = clazz.getConstructor();

constructor.setAccessible(true);

constructor.newInstance();

instead of just using the standard approach of:

clazz.newInstance();

This last way only works with public classes, since they are accessible per se. Needless to say that public classes also need a defined public no-args constructor or no defined constructor at all for this to work.

Any further advice on instantiating the classes retrieved by the Extensible Component Scanner, or on using them in any other way, is beyond the scope of this manual.

Besides conforming to the restrictions just given, components should also be annotated with any kind of annotation, implement some interface or extend some class, other than java.lang.Object, or have a combination of these characteristics. This ensures that they have a special meaning and purpose in your application and can be found by matching them against narrower criteria than extending java.lang.Object.

## Why to scan for Components

There are two primary cases in which you’d probably like to use component scanning. The first one is to replace configuration files e.g. telling your application which classes to use as plug-ins or your framework or container which classes to use for a particular application. For example in the Spring framework ([www.springframework.com](http://www.springframework.com)) component scanning is used as an alternative to defining all the different beans of an application in configuration XML files. As these can become quite large this reduces the amount of configuration code quite a lot.

So if you are developing a new application featuring plug-ins, a new framework or container and want to make the lives of the developers targeting your system as painless as possible use component scanning instead of configuration files.

Even though in the first case component scanning is quite useful, you could do without. Not so in the second case. Suppose you wanted to know all the classes in a particular package that are serializable. In this case you cannot configure anything, as you simply don’t know all those classes. There is nothing short of manually looking through all classes in that package to find those that match, except component scanning.

In short component scanning is good for two things: saving configuration code and saving time.

## Is Component Scanning equal to Classpath Scanning

As component scanning refers to **what** you want to find and classpath scanning refers to **where** you want to find something, those terms are strictly speaking not equal. Furthermore with the Extensible Component Scanner you are not restricted to scanning the classpath that is known to the JVM at the moment of scanning to find the components you are looking for.

The components are retrieved as resources from a classloader using the method

ClassLoader.getResources(String name)

Since it is possible to pass a custom classloader to the Extensible Component Scanner you could retrieve components from just about any place, depending on the implementation of the getResources method of the classloader you are using.

# The Extensible Component Scanner API

The API of the Extensible Component Scanner is split into two parts. The first one is comprised of the classes and methods you use to process a scanning run. This is described in this chapter. The second one is the embedded domain specific language (eDSL) you use to create a query defining which components you actually want to find. That eDSL is described in chapter 3.

## The ComponentScanner class

The main entry point to the Extensible Component Scanner is the class ComponentScanner in the package net.sf.extcos.