**Extensible Component Scanner**

**Version 0.2b**

**User Manual**

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

**2010-06-05**

# 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](#_Toc263512055)

[1. The purpose of Component Scanning 4](#_Toc263512056)

[1.1. What is a Component 4](#_Toc263512057)

[1.2. Why to scan for Components 5](#_Toc263512058)

[1.3. Is Component Scanning equal to Classpath Scanning 5](#_Toc263512059)

[2. The Extensible Component Scanner API 6](#_Toc263512060)

[2.1. The ComponentScanner class 6](#_Toc263512061)

[2.2. The ComponentQuery class 6](#_Toc263512062)

[2.3. A first code example 7](#_Toc263512063)

[3. The Component Query Language 8](#_Toc263512064)

[3.1. The select clause 8](#_Toc263512065)

[3.2. The from clause 8](#_Toc263512066)

[3.3. The andStore clause 9](#_Toc263512067)

[3.4. The returning clause 10](#_Toc263512068)

# The purpose of Component Scanning

## What is a Component

For the matters of the Extensible Component Scanner 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 for the Extensible Component Scanner 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 criteria more narrow 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 class loader using the method

ClassLoader.getResources(String name)

Since it is possible to pass a custom class loader to the Extensible Component Scanner you could retrieve components from just about any place, depending on the implementation of the getResources method of the class loader 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 Component Query Language, an embedded domain specific language (eDSL) you use to create a query defining which components you actually want to find. The Component Query Language 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. It features two public methods, both of which are part of the API.

**ComponentScanner.**

**getClasses(componentQuery: ComponentQuery): Set<Class<?>>**

This method lets you specify a component query defining the criteria components you are interested in must match. It looks up the default class loader and uses it to fulfill the request. It returns a set of classes matching the defined criteria of the query. The default class loader is retrieved via

Thread.*currentThread*().getContextClassLoader();

**ComponentScanner.**

**getClasses(componentQuery: ComponentQuery, classLoader: ClassLoader): Set<Class<?>>**

Along with the component query defining the criteria components you are interested in must match this method lets you specify a custom class loader to be used to fulfill the request. This allows you for example to use the Extensible Component Scanner within a web container like Tomcat and use its WebappClassLoader. This method returns a set of classes matching the defined criteria of the query.

## The ComponentQuery class

This abstract class in the package net.sf.extcos is the base for all component query definitions to be passed to one of the ComponentScanner methods. It defines one abstract method which needs to be implemented by the subclass used to define a specific component query:

**ComponentQuery.query(): void**

This method acts as a container for the actual query. It must contain exactly one Component Query Language query.

Besides the query method this class features a number of eDSL methods. These are explained in chapter 3.

## A first code example

Now that we know about the ComponentScanner and the ComponentQuery classes it’s time for a first code example. You may use it as a template every time you use the Extensible Component Scanner.

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

// the actual query goes here

}

});

This example uses the default class loader. If you want to use a custom class loader, pass it to the getClasses method, like so:

ClassLoader customClassLoader = getCustomClassLoader();

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

// the actual query goes here

}

}, customClassLoader);

Implement the getCustomClassLoader method in whatever manner you like and you’re done.

# The Component Query Language

The Component Query Language (CQL) is an eDSL and the heart and soul of the Extensible Component Scanner. It defines what kind of artifacts to find from which packages matching which criteria and how to return or where to store the resulting classes. The CQL is currently comprised of the four clauses select, from, andStore, and returning.

## The select clause

The select clause defines which kinds of artifacts to find components of. It’s defined as the two methods

**ComponentQuery.select(): BasePackageSelector**

and

**ComponentQuery.select(ResourceType... resourceTypes): BasePackageSelector**

The first acts as a default and calls the second with the resource type for precompiled Java classes. So every time you just want to find precompiled Java classes you can use that method without needing to specify any resource type.

In order to make the CQL easily readable resource types are required to comply with a convention. They need to be classes with a private constructor and a static method returning an instance, rather like singletons. That static method should be called something like javaClasses or groovySources and needs to be imported as a static import. In the case of resource type for precompiled Java classes this method is implemented as

JavaClassResourceType.javaClasses()

in the net.sf.extcos.internal package. The specifics for extensions for other kinds of artifacts are documented with each extension separately.

For precompiled Java classes the following two listings are equivalent:

select()

and

import static net.sf.extcos.internal.JavaClassResourceType.javaClasses;

select(javaClasses())

## The from clause

The from clause defines where to find the components. More precisely it defines the base package or base packages in which components will be found. It’s defined as the method

**BasePackageSelector.from(String... basePackages): ForwardingBuilder**

You may specify various base packages. Make sure that you pass only valid package names as defined by the Java Language Specification. Otherwise an IllegalArgumentException will be thrown. Although technically possible it is not permissible to pass no base package at all. Doing so will also result in an IllegalArgumentException.

A base package is a package that contains classes and / or other packages. If it contains other packages those subpackages will also be scanned and matching components will also be returned. Supposing you’ve got two packages foo.bar1 and foo.bar2 and you pass foo as the base package to the from clause, then all matching components from foo, foo.bar1 and foo.bar2 will be returned.

Now that we learned about the select clause and the from clause, it’s time for the first complete example. If you want to retrieve all the precompiled Java class components from the foo package, this is the way to do it:

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”);

}

});

## The andStore clause

The andStore clause allows you to retrieve several different sets of components with just one query. It’s an optional clause, so if you just need to get one set of components you may not use it. The andStore clause is defined as

**ForwardingBuilder.andStore(StoreBinding... bindings): ReturningSelector**

and takes one or more StoreBinding objects as its parameters. Although it is technically possible to pass no StoreBinding object at all into the andStore clause, this is not permissible and will result in an IllegalArgumentException.

The CQL way of obtaining the StoreBinding objects to pass into the andStore clause is to use matching clauses defining component filters. Only matching components will be stored. These matching clauses are defined in the ComponentQuery class and described in detail in sections 3.5 to 3.10. The following table gives an overview.

|  |  |
| --- | --- |
| **Matching clause** | **Described in section** |
| thoseAnnotatedWith(Class<? extends Annotation> annotation) | 3.5 |
| thoseAnnotatedWith(Class<? extends Annotation> annotation, ArgumentsDescriptor arguments) | 3.6 |
| thoseBeing(TypeFilterJunction filter) | 3.7 |
| thoseExtending(Class<T> clazz) | 3.8 |
| thoseImplementing(Class<?>... interfaces) | 3.9 |
| thoseImplementing(Class<T> interfaze) | 3.10 |

Each matching clause returns either a TypelessStoreBindingBuilder or a TypedStoreBindingBuilder. Both classes define an into clause which specifies where the components matching the given matching clause will go to. The difference in the definition of the into clause is in the type of Set it takes. The into clause in the TypelessStoreBindingBuilder is defined as

**TypelessStoreBindingBuilder.into(Set<Class<?>> store): StoreBinding**

and takes a Set which can contain any kind of class object. In contrast the into clause in the TypedStoreBindingBuilder is defined as

**TypedStoreBindingBuilder<T>.into(Set<Class<? extends T>> store): StoreBinding**

and takes a Set which can only contain classes extending or implementing the type T which has originally been defined in the matching clause. That way the type information given via the matching clause is used to require the type of the Set to be as specific as possible. This gives you additional type safety. Obviously this is only possible if the matching clause defines one specific type to match components against. This applies only to the thoseExtending and thoseImplementing clauses, the thoseImplementing clause taking just one parameter.

As you can see the into clauses return the StoreBinding the andStore clause requires. So each StoreBinding object is created by using a matching clause followed by an into clause.

You say all well and good, but can I see an example? Here you go: Suppose you want to get all components implementing the java.io.Serializable interface in one set and all components annotated with some SampleAnnotation in another set, all components being located in the foo package. Then all you need is

final Set<Class<? extends Serializable>> serializables = new ...;

final Set<Class<?>> samples = new HashSet<Class<?>>();

ComponentScanner scanner = new ComponentScanner();

Set<Class<?>> classes = scanner.getClasses(new ComponentQuery() {

protected void query() {

select().from(“foo”).andStore(

thoseImplementing(Serializable.class).into(serializables),

thoseAnnotatedWith(SampleAnnotation.class).into(samples));

}

});

Note that the classes set is empty in this case, as there is no returning clause given and the default behavior in this case is to return an empty set.

## The returning clause