The TAO of DSpace Services

Tao of DSpace Services

Introduction

During the last DSpace Committer meeting, I was prodded to start to show the community how one uses the Service Manager with some real world examples. I've decided to start to create a few of these HowTo's to assist you in learning to use the DSpace Services in intelligent ways. I had this little project floating around for some time and I decided it was something I could get down on paper fairly quickly for you. DSpace Services are about using Spring to Simplify your development life. And for our first example showing how to create services we will refactor the DSpace Launcher to no longer utilize the launcher.xml and instead load configured commands from a Spring configuration. The result of this approach will allow for Addons to easily provide additional commands into the launcher without having to physically alter a configuration.xml file in a config directory. To start with we will review the ScriptLauncher and discuss the problems that exist with it. For a full view of the source please see: ScriptLauncher.java

DSpace Legacy ScriptLauncher

First and foremost, we see that `ScriptLauncher` begins the process of execution by bringing the `ServiceManager` into existence. This is important so that services are available to the executed commandline tools — these services currently wire discovery and statistics into dspace.

```
 */
public class ScriptLauncher
{
    /** The service manager kernel */
    private static transient DSpaceKernelImpl kernelImpl;

    /**
     * Execute the DSpace script launcher
     *
     * @param args Any parameters required to be passed to the scripts it executes
     */
    public static void main(String[] args)
    {
        // Check that there is at least one argument
        if (args.length < 1)
        {
            System.err.println("You must provide at least one command argument");
            display();
            System.exit(1);
        }

        // Initialise the service manager kernel
        try {
            kernelImpl = DSpaceKernelInit.getKernel(null);
            if (!kernelImpl.isRunning())
            {
                kernelImpl.start(ConfigurationManager.getProperty("dspace.dir"));
            }
        } catch (Exception e)
        {
            // Failed to start so destroy it and log and throw an exception
            try
            {
                kernelImpl.destroy();
            }
            catch (Exception e1)
            {
                // Nothing to do
            }
            String message = "Failure during filter init: " + e.getMessage();
            System.err.println(message + ":" + e);
            throw new IllegalStateException(message, e);
        }
```

But, here, we just focus on what `ScriptLauncher` itself is doing and how we can improve it with services. So looking further on we see that we parse the XML file with JDOM and use XPath to navigate the configuration, which means if we want to do anything new in the Launcher in the future, we may need to extend the XML file and alter the parsing. So, what we do see is that execution of the commmand is very tightly bound to the parsing and iteration over the XML file — in fact, they are so tightly bound together that new code would need to be written if you wanted to get a command available outside the launcher.

Firstly we need to get commands:

```
        // Parse the configuration file looking for the command entered
        Document doc = getConfig();
        String request = args[0];
        Element root = doc.getRootElement();
        List<Element> commands = root.getChildren("command");
        for (Element command : commands)
        {
            if (request.equalsIgnoreCase(command.getChild("name").getValue()))
            {
```

Then we need to get the steps and process them

```
// Run each step
List<Element> steps = command.getChildren("step");
for (Element step : steps)
{
    ...
    className = step.getChild("class").getValue();
```

decide about passing them on to the steps

```
// Run each step
List<Element> steps = command.getChildren("step");
for (Element step : steps)
{
    // Instantiate the class
    Class target = null;

    // Is it the special case 'dsrun' where the user provides the class name?
    String className;
    if ("dsrun".equals(request))
    {
        if (args.length < 2)
        {
            System.err.println("Error in launcher.xml: Missing class name");
            System.exit(1);
        }
        className = args[1];
    }
    else {
        className = step.getChild("class").getValue();
    }
    try
    {
        target = Class.forName(className,
                               true,
                               Thread.currentThread().getContextClassLoader());
    }
    catch (ClassNotFoundException e)
    {
        System.err.println("Error in launcher.xml: Invalid class name: " + className);
        System.exit(1);
    }
```

decide to pass arguments to the step

```
    // Strip the leading argument from the args, and add the arguments
    // Set <passargs>false</passargs> if the arguments should not be passed on
    String[] useargs = args.clone();
    Class[] argTypes = {useargs.getClass()};
    boolean passargs = true;
    if ((step.getAttribute("passuserargs") != null) &&
        ("false".equalsIgnoreCase(step.getAttribute("passuserargs").getValue())))
    {
        passargs = false;
    }
```

assign args in the child elements of the steps,

```
if ((args.length == 1) || (("dsrun".equals(request)) && (args.length == 2)) || (!passargs))
{
    useargs = new String[0];
}
else
{
    // The number of arguments to ignore
    // If dsrun is the command, ignore the next, as it is the class name not an arg
    int x = 1;
    if ("dsrun".equals(request))
    {
        x = 2;
    }
    String[] argsnew = new String[useargs.length - x];
    for (int i = x; i < useargs.length; i++)
    {
        argsnew[i - x] = useargs[i];
    }
    useargs = argsnew;
}
```

The Big Problem : The `ScriptLauncher` hardwires XML Configuration to Business Logic

What we see is that the Script Launcher hardwires all Business Logic to XML configuration, and while this is a great prototype, certainly not very extensible. For instance, what if we may want to initialize other details into the command or step? What if we are calling something that uses a different method than a static main to execute? and what if we want to set certain properties on its state beforehand? To do these tasks we would either need to rewrite the class with a main method, or we would either need to extend the launcher to do this, or add those details from the dspace.cfg. So what we will show you in refactoring this code is that you can get a great deal more flexibility out of Spring with a great deal less work on your part. Spring applies the concept of "inversion of control". The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code. In our case, we will be removing a dependency on the hardcoded XML file for configuration and liberating the Command/Step domain model from the `ScriptLauncher`, allowing for possible reuse in other areas of the application.

Spring Based DSpace Script Launcher

Firstly we recognize that we have few domain objects here that we can work with: Command, Step and Arguments are all ripe to be defined as interfaces or implementation classes that capture the domain of the Launcher that we will execute. Perhaps by creating these we can disentangle the Business logic of the Launcher from its configuration in XML. Doing so without Spring, we might still have to bother with parsing the XML file. So with Spring we get a luxury that we no longer need to think about that.

We will do this initial example directly in the dspace-api project so you do not get sidetracked in terms of dealing with Maven poms....

Step 1: Domain model

`Command`

`Command` holds the details about the command we will call. Notice it doesn't parse any XML and most importantly it does not instantiate any `Step` directly; that is decoupled from the `Command` and we will show you later how it is assembled. All a `Command` does when getting created is sit there like a good old Java Bean should. It just "IS".

```
package org.dspace.app.launcher;

import java.util.List;

public class Command implements Comparable<Command> {

    public String name;

    public String description;

    public List<Step> steps;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public List<Step> getSteps() {
        return steps;
    }

    public void setSteps(List<Step> steps) {
        this.steps = steps;
    }

    @Override
    public int compareTo(Command c){

        return name.compareTo(c.name);
    }
}
```

Step

`Step` will be responsible for retaining the details about the step being executed. You will notice it knows nothing about a `Command`, and in our default case it just knows a little bit about how to reflectively execute a Main Class. Again, it knows little about its instantiation nor anything specifically on how it is configured. It just "IS".

```
package org.dspace.app.launcher;

import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Step {


    public List<String> arguments;

    public String className;

    /**
     * User Arguments Defaults to True.
     */
    public boolean passUserArgs = true;
```

```
    public boolean isPassUserArgs() {
        return passUserArgs;
    }

    public void setPassUserArgs(boolean passUserArgs) {
        this.passUserArgs = passUserArgs;
    }

    public List<String> getArguments() {
        return arguments;
    }

    public void setArguments(List<String> arguments) {
        this.arguments = arguments;
    }

    public String getClassName() {
        return className;
    }

    public void setClassName(String className) {
        this.className = className;
    }

    public int exec(String[] args) {

        //Merge the arguments
        List<String> argsToPass = Arrays.asList(args);

        if(this.getArguments() != null)
            argsToPass.addAll(this.getArguments());

        String[] useargs = argsToPass.toArray(new String[0]);


        // Instantiate and execute the class
        try {

            Class target = Class.forName(getClassName(),
                    true, Thread.currentThread().getContextClassLoader());

            Class[] argTypes = {useargs.getClass()};
            Object[] finalargs = {useargs};

            Method main = target.getMethod("main", argTypes);
            main.invoke(null, finalargs);
        }
        catch (ClassNotFoundException e) {
            System.err.println("Error in command: Invalid class name: " + getClassName());
            return 1;
        }
        catch (Exception e) {

            // Exceptions from the script are reported as a 'cause'
            System.err.println("Exception: " + e.getMessage());
            e.printStackTrace();
            return 1;
        }

        return 0;
    }
}
```

Step 2: The Command Service

Ok at this point you're probably itching to combine all these together and get your Launcher to execute. But at this point, well, we still have some more work to do to get ready. And likewise, we are still keeping completely hands off of any configuration detail at this point. Do not worry, we will get there soon enough.

Next we have the `CommandService`. This is a "Controller" which you will use to execute a `Command`. It doesn't have a main method, because, well, you can execute your commands entirely independent of a CLI. Wouldn't that be nice in Curation Tasks and Filter Media? Yes, sure it would, but I digress, lets get back to the topic at hand. The Command Service...

All it will do is contain the business logic to call a command. In this case, you will see, it just has no clue at all how it gets those commands, but it surely just understands how to execute them, pretty smart little service, it just does what its told. Doesn't think about much else.

```
package org.dspace.app.launcher;

import org.dspace.services.RequestService;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CommandService {

    public List<Command> getCommands() {
        return commands;
    }

    public void setCommands(List<Command> commands) {
        this.commands = commands;
    }

    public List<Command> commands;

    public void exec(String[] args){


        // Check that there is at least one argument
        if (args.length < 1)
        {
            System.err.println("You must provide at least one command argument");
            return;
        }

        for (Command command : commands)
        {
            // Parse the configuration file looking for the command entered
            if (args[0].equalsIgnoreCase(command.getName()))
            {
                // Run each step
                for (Step step : command.getSteps())
                {
                    int result = 0;

                        if(step.isPassUserArgs())
                            /* remove command from Args */
                            result = step.exec(Arrays.copyOfRange(args,1,args.length));
                        else
                            /* send empty args */
                            result = step.exec(new String[0]);


                    return result;
                }
            }
        }
    }
}
```

Step 3: The Main Function

Finally, we want a way to run this from the commandline. Note I've simplified it a little to get the DSpace Services Kernel instantiation out of the way, but its still pretty basic here.

```
package org.dspace.app.launcher;

...
public class ScriptLauncher
{

private static transient DSpaceKernelImpl kernelImpl;

public static void main(String\[\] args)
{

...

CommandService service = kernelImpl.getServiceManager().getServiceByName(CommandService.class.getName(),
CommandService.class);

...

try {
            service.exec(Arrays.copyOfRange(args,1,args.length));
        } catch (Exception e) {
            ...
        }
 
...
}

}
```

Ok, at this point you're saying, "but, wait a second, how did you configure the `CommandService`, Commands and Steps so they could be available?" That's the majic of Spring, and there's a couple ways we do this, but I will take the most explicit approach that still will allow you to wire in your own Commands later.

Step 4: The Spring Configuration

We place the Spring Configuration in a special place for the DSpace Service Manager, and the Service Manager is pretty smart about finding these. It expects us to place these files in a couple different places. But in our example I will show the current places in DSpace 1.7.x. In all cases we want to be placing these files under *[dspace-module]/src/main/resources/spring*

spring-dspace-core-services.xml

This location allows us to "augment the existing services without actually overriding them". We generally reserve this for the core DSpace services like `RequestService`, `ConfigurationService`, `SessionService`, etc.

spring-dspace-addon-[a unique name]-services.xml

This location allows us to "override" the XML loading a specific service by overwriting its configuration in one of our own.

Now to show you our Launcher Service. The trick here is we will use a feature in Spring called autowire byType. It will collect all the Commands and wire them together for us, no matter the type. I'll save you having to view the whole file — you can see it here if you like .

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

    The contents of this file are subject to the license and copyright
    detailed in the LICENSE and NOTICE files at the root of the source
    tree and available online at

    http://www.dspace.org/license/

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

    <!-- allows us to use spring annotations in beans -->
    <context:annotation-config/>


    <!-- Instantiate the Launcher Service and Autowire its dependencies -->
    <bean class="org.dspace.app.launcher.ScriptLauncher" autowire="byType"/>

    <bean class="org.dspace.app.launcher.Command">
        <property name="name" value="checker"/>
        <property name="description" value="Run the checksum checker"/>
        <property name="steps">
            <list>
                <bean class="org.dspace.app.launcher.Step">
                    <property name="className" value="org.dspace.app.checker.ChecksumChecker"/>
                </bean>
            </list>
        </property>
    </bean>
```

You'll notice we now have created the `ScriptLauncher Service` via a bean definition

```
 <!-- Instantiate the Launcher Service and Autowire its dependencies -->
<bean class="org.dspace.app.launcher.ScriptLauncher" autowire="byType"/>
```

And that we have created the `Command` for the Checksum Checker here...

```
<bean class="org.dspace.app.launcher.Command">
   <property name="name" value="checker"/>
   <property name="description" value="Run the checksum checker"/>
   <property name="steps">
     <list>
       <bean class="org.dspace.app.launcher.Step">
         <property name="className" value="org.dspace.app.checker.ChecksumChecker"/>
       </bean>
     </list>
    </property>
</bean>
```

You also recall that with DSpace we have a DSRun command that gives us a different behavior that normal commands: it rather wants to receive the class, rather than defining it itself, so we exended `Step` and introduce a special `Step` in DSpace that just knows how to do that.

```
<bean class="org.dspace.app.launcher.Command">
   <property name="name" value="dsrun"/>
   <property name="description" value="Run a class directly"/>
   <property name="steps">
      <list>
         <bean class="org.dspace.app.launcher.DSRUNStep"></bean>
      </list>
   </property>
</bean>
```

So yes again, we made sure that our DSRUNStep is smart and simple minded. It just knows how to do one thing very well, that's execute a class passed as an option on the commandline. It does it smartly reusing a generic `Step` and setting all those values it needs within it.

```
package org.dspace.app.launcher;

import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.List;

public class DSRUNStep extends Step {


    public int exec(String[] args) {

        Step ds = new Step();

        ds.setArguments(this.getArguments());

        System.out.println(args.length);
        System.out.println(args[0]);


        if (args.length < 1)
        {
            throw new RuntimeException("Error in launcher: DSRun Step Missing class name.");
        }

        /* get the classname from arguments */
        ds.setClassName(args[0]);


        ds.setPassUserArgs(true);

        /* Execute the requested class with the appropriate args (remove  classname) */
        return ds.exec(Arrays.copyOfRange(args,1,args.length));
    }

}
```

Adding Commands in other Addon Modules

One of the luxuries we get from using the autowire byType in our spring configuration within the ServiceManager, is that now we can allow others to toss in their commands regardless of what they are named — we just need to know that they implement our `Command` interface and provide Steps we can execute. For instance, to introduce a command that will allow us to index discovery or operate on the statistics indexes, we can, in our Discovery and Statistics Addons add additional *src/main/resources/spring/spring-dspace-addon-discovery-services.xml*. Do the following:

Discovery (dspace/trunk/dspace-discovery/dspace-discovery-provider/src/main/resources/spring)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-2.5.xsd">

   <bean class="org.dspace.app.launcher.Command">
      <property name="name" value="update-discovery-index"/>
      <property name="description" value="Update Discovery Solr Search Index"/>
      <property name="steps">
        <list>
          <bean class="org.dspace.app.launcher.Step">
             <property name="className" value="org.dspace.discovery.IndexClient"/>
          </bean>
        </list>
      </property>
    </bean>
</beans>
```

Statistics (dspace/trunk/dspace-statistics/src/main/resources/spring)

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

    The contents of this file are subject to the license and copyright
    detailed in the LICENSE and NOTICE files at the root of the source
    tree and available online at

    http://www.dspace.org/license/

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


    <bean class="org.dspace.app.launcher.Command">
        <property name="name" value="stats-log-converter"/>
        <property name="description" value="Convert dspace.log files ready for import into solr statistics"/>
        <property name="steps">
            <list>
                <bean class="org.dspace.app.launcher.Step">
                    <property name="className" value="org.dspace.statistics.util.ClassicDSpaceLogConverter"/>
                </bean>
            </list>
        </property>
    </bean>

    <bean class="org.dspace.app.launcher.Command">
        <property name="name" value="stats-log-importer"/>
        <property name="description" value="Import previously converted log files into solr statistics"/>
        <property name="steps">
            <list>
                <bean class="org.dspace.app.launcher.Step">
                    <property name="className" value="org.dspace.statistics.util.StatisticsImporter"/>
                </bean>
            </list>
        </property>
    </bean>

    <bean class="org.dspace.app.launcher.Command">
        <property name="name" value="stats-util"/>
        <property name="description" value="Statistics Client for Maintenance of Solr Statistics Indexes"/>
        <property name="steps">
            <list>
                <bean class="org.dspace.app.launcher.Step">
                    <property name="className" value="org.dspace.statistics.util.StatisticsClient"/>
                </bean>
            </list>
        </property>
    </bean>

</beans>
```

Summary

So in this tutorial, I've introduced you to the ServiceManager, how to rethink your approach using Spring and not hardwire your application by binding its business logic to the parsing of a configuration file or other source of configuration, and how to lightly wire it together so that others can easily extend the implementation on their own. Likewise, I've show you the power of how to separate the configuration of the Script Launcher to a file that can be added to DSpace Maven Addon Module so that you can write your own commands without having to do anything at all in the dspace.cfg or launcher.xml in the DSpace config directory to enable them. A solution that will no doubt make your maintenence of those code changes much much easier as DSpace releases new versions and you need to merge yout config files to stay in line. The final take home mantra for you to repeat 1000 times tonight before going to sleep... "If I don't have to alter configuration to add my changes into DSpace, I wouldn't need to merge differences when I upgrade."