

AGETOR®

ServiceRunner User Guide

Contents

1	Preface	1
1.1	Reloading classes.....	1
2	Configuration	2
2.1.1	Element: services	2
2.1.2	Services.....	2
3	Service types	3
3.1	Native program	3
3.2	Java type.....	4
3.3	ORB type	4
3.4	ServiceRunner client type	5
4	Configuration values.....	6
5	Properties	6
6	Command line arguments	6
7	Classpath	8
8	Run status	8
9	Status	8
10	Message.....	9
11	Stop services.....	9
12	Shutting down the ServiceRunner.....	9
13	Log	10
14	Troubleshooting	10

1 Preface

The ServiceRunner is able to start the services on the local machine from a configuration file. Both Java programs and other programs can be started allowing easy configuration of services as well as startup and shutdown of the entire AGETOR system.

The ServiceRunner uses the JVM's mechanism for starting external processes and this may support remote execution.

Besides this the ServiceRunner will execute multiple Java programs in one Java Virtual Machine, which gives a significant memory advantage.

Even though these Java programs are running in the same Virtual Machine they will not be able to interfere with each other, because the ServiceRunner provides a separate environment for each of them. Any Java program is run within its own thread group and have separate ClassLoaders. Java programs have their classes reloaded before execution. The ServiceRunner itself is a service and may be contacted through the broker.

The ServiceRunner may be configured to start the Broker as if the broker was any other ordinary Java program (which in fact it is).

The ServiceRunner is not an application server that will spawn extra services when load is high, and it does currently not restart crashed services, even though it will know if any service has crashed.

1.1 Reloading classes

The ServiceRunner is able to restart a single service reloading all necessary classes without starting a new JVM. This is achieved by a specialized ClassLoader, which is able to reload classes.

Classes that are not directly related to the service can be loaded by the default class loader and you can use the property `agetor.servicerunner.excludedpackages` to indicate which classes should be excluded from the specialized class loader and loaded by the default. The value of the property is a semicolon separated list of classname prefixes.

Example entry in a property file:

```
agetor.servicerunner.excludedpackages=com.sun;sun;org;COM
```

This entry will cause all classes like `com.sun.java.util.collections.List` , `org.javalobby.News` and `COM.mycom.Util` to be loaded by the default classloader.

2 Configuration

The command `services-start` starts the ServiceRunner with the configuration files specified in `conf.cfg`. These configuration files specify the programs on the machine that the ServiceRunner can start and stop and are simply XML files.

The XML configuration files may reference properties by using `${property-name}`, which will be replaced by the value of the property (environment variable defined during setup).

This configuration file consists of program elements.

The file must have the following template (see also Example configuration) :

```
<SERVICES LOGBASE="...">

    <!-- services -->
</SERVICES>
```

2.1.1 Element: services

This is the root element for the XML configuration file. It accepts the attribute `logbase` which is the base directory for log output from the services.

A typical definition will be:

```
<SERVICES LOGBASE="${AGETOR_HOME}/logs">
```

2.1.2 Services

The services configuration is described in the following together with the service types explanation.

3 Service types

The ServiceRunner supports 4 service types:

- Native programs
- Java classes
- ORB services
- ServiceRunner client services

3.1 Native program

The type key `external` specifies a native program. This kind of service is executed by a shell command. This means that a native program can only be started and killed.

The command is specified in the `PROGRAM` attribute (the `NAME` attribute from previous versions can still be used).

The service accepts a working directory specified by the attribute `WORKINGDIR`. This directory can be used by the native program as its working directory. In addition, it is possible to specify command line arguments and system properties (see below).

Example:

```
<PROGRAM
    PROGRAM="C:/Program Files/MyApplic.exe"
    TYPE="external"
    WORKINGDIR="{AGETOR_HOME}/data/NativeServiceData"
    DESCRIPTION="NativeService"
    LOGFILE=" NativeService.log"
>
    <PROPERTY NAME="AGETOR_HOME" VALUE="{AGETOR_HOME}" />
    <PARAM NAME="input" VALUE="c:/mydata/catalog.mdb" />
</PROGRAM>
```

XML tag	XML attribute	Explanation
PROGRAM	PROGRAM	Excecutable
PROGRAM	TYPE	ServiceRunner service type
PROGRAM	DESCRIPTION	Service name
PROGRAM	LOGFILE	Redirected standard output
PROGRAM	WORKING	Working dir
PROPERTY	NAME	Additional system property name
PROPERTY	VALUE	Additional system property value
PARAM	NAME	Command line parameter name
PARAM	VALUE	Command line parameter value

3.2 Java type

The type key `Java-method` specifies Java type services. They are Java programs that can be started and killed. In addition, if the program terminates with an exception then this can be caught. The Java class is specified by the `CLASS` attribute (`NAME` can still be used).

The service configuration accepts command line arguments, system properties and the class path. The system properties are appended to the current properties while the class path is placed in front of the current class path.

Example:

```
<PROGRAM
    CLASS="dk.mycomp.myapplic.firstservice.Service"
    TYPE="Java-method"
    CLASSPATH="c:/MyProjects/SrvA/classes;c:/MyProjects/Lib1/classes"
    DESCRIPTION="JavaService"
    LOGFILE=" JavaService.log"
>
    <PROPERTY NAME="agetor.orb.timeout" VALUE="120" />
    <PARAM NAME="input" VALUE="c:/mydata/catalog.mdb" />
</PROGRAM>
```

XML tag	XML attribute	Explanation
PROGRAM	CLASS	Java class name
PROGRAM	TYPE	ServiceRunner service type
PROGRAM	DESCRIPTION	Service name
PROGRAM	LOGFILE	Redirected standard output
PROGRAM	CLASSPATH	Additional classpath
PROPERTY	NAME	Additional system property name
PROPERTY	VALUE	Additional System property value
PARAM	NAME	Command line parameter name
PARAM	VALUE	Command line parameter value

3.3 ORB type

The type key `orb` specifies ORB type services. They are Java programs, which runs a server ORB (ServerORB or WebAnswerORB). They can be started, shut down and killed. It is possible to get a custom service status if the service implements it and the ORB support this. In addition, if the program terminates with an exception then this can be caught.

The Java class is specified by the `CLASS` attribute while the port is defined by `PORT`.

The service configuration accepts command line arguments, system properties and the class path. The system properties are appended to the current properties while the class path is placed in front of the current class path.

```
<PROGRAM
```

```

        CLASS="dk.mycomp.myapplic.firstservice.Service"
        RPORT="15"
        TYPE="orb"
    CLASSPATH="c:/MyProjects/SrvA/classes;c:/MyProjects/Lib1/classes"
        DESCRIPTION="ORBService"
        LOGFILE="ORBService.log"
    >
        <PROPERTY NAME="agetor.socket.status.cnnpool" VALUE="false" />
        <PARAM NAME="input" VALUE="c:/mydata/catalog.mdb" />

</PROGRAM>

```

XML tag	XML attribute	Explanation
PROGRAM	CLASS	Java class name
PROGRAM	TYPE	ServiceRunner service type
PROGRAM	DESCRIPTION	Service name
PROGRAM	PORT	Service absolute port number
PROGRAM	RPORT	Service relative port number
PROGRAM	LOGFILE	Redirected standard output
PROGRAM	CLASSPATH	Additional classpath
PROPERTY	NAME	Additional system property name
PROPERTY	VALUE	Additional System property value
PARAM	NAME	Command line parameter name
PARAM	VALUE	Command line parameter value

3.4 ServiceRunner client type

The type key `client` specifies ServiceRunner Client type services. They are Java classes, which extend the `AbstractServiceClient` class. They can be started, shutdown and killed. It is possible to get a custom service status if the service implements and it is possible to send messages to the service form the `scmd` command. In addition if the program terminates with an exception then this can be caught.

The Java class is specified by the `CLASS` attribute. The service configuration accepts command line arguments, system properties and the class path. The system properties are appended to the current properties while the class path is placed in front of the current class path.

```

<PROGRAM
    CLASS="dk.mycomp.myapplic.firstservice.Service"
    TYPE="client"
    CLASSPATH="c:/MyProjects/SrvA/classes;c:/MyProjects/Lib1/classes"
    DESCRIPTION="SRClientService"
    LOGFILE="SRClientService.log"
>
    <PROPERTY NAME="agetor.orb.protocol.encoding" VALUE="UTF-16" />
    <PARAM NAME="input" VALUE="c:/mydata/catalog.mdb" />

```

</PROGRAM>

XML tag	XML attribute	Explanation
PROGRAM	CLASS	Java class name
PROGRAM	TYPE	ServiceRunner service type
PROGRAM	DESCRIPTION	Service name
PROGRAM	LOGFILE	Redirected standard output
PROGRAM	CLASSPATH	Additional classpath
PROPERTY	NAME	Additional system property name
PROPERTY	VALUE	Additional System property value
PARAM	NAME	Command line parameter name
PARAM	VALUE	Command line parameter value

4 Configuration values

All the configuration values (XML attribute values) can contain system property from the ServiceRunner system properties (the system properties the ServiceRunner had when it started up). The syntax for those special values is `${propname}`.

5 Properties

It is possible to set system properties for all the kinds of services. The service is then capable of seeing the standard properties plus the properties added in the configuration. Properties added in the configuration will override the standard properties.

As you can see from the example, the property is configured with the XML element `PROPERTY`:

```
<PROPERTY NAME="propname" VALUE="propvalue" />
```



The properties, you configure for a specific service, are not accessible from other services. The start system properties used are the ServiceRunner system properties.

6 Command line arguments

A command line argument has the form:

```
<PARAM NAME="name" VALUE="value" />
```

The ServiceRunner passes the arguments on to the service in a String array that is composed from the arguments in a special way. To read the arguments from the service you should use the utility class `dk.bording.inside.util.CommandLine`.

If you wish to read the arguments manually, the table below shows how the ServiceRunner transforms the name-value pairs into strings:

	Name	Value	Resulting string	Recommended
	<char>	<value>	-<char><value>	✓
Example:	p	20010	-p20010	✓
	-<char>	<value>	-<char><value>	
Example:	-p	20010	-p20010	
	<string>	<value>	--<string>:<value>	✓
Example:	rport	10	--rport:20010	✓
	--<string>	<value>	--<string>:<value>	
Example:	--port	20010	--port:20010	
	<char>	<nothing>	-<char>	✓
Example:	v		-v	✓
	<string>	<nothing>	--<string>	✓
Example:	verbose		--verbose	✓
	<nothing>	<char value>	<char value>	✓
Example:		v	v	✓
	<nothing>	<string value>	<string value>	✓
Example:		verbose	verbose	✓

7 Classpath

All the Java-based services accept a class path. The class path is added in front of the ServiceRunner class path. Each class path is specific for the particular service that has configured it and cannot be read from other services.

The items in the classpath are separated by the symbol “;” (semicolon).

8 Run status

When the services are listed from the ServiceRunner Commander, *scmd*, using its *show* command it reports the run status (see the ARE_guide). The run status can be: *running*, *stopping* and *stopped*.

The status is related to the running thread in the service. Only when all the threads are terminated then the service has the *stopped* status.

Status types:

- *Running*
The service is not stopped and the ServiceRunner is not trying to stop it.
- *Stopping*
The ServiceRunner is trying to stop the service (see that section for more information) but the service is not stopped yet.
- *Stopped*
All the threads belonging to the service are terminated.

The *stopped* status is very important for the services, which use limited resources (like DB connections). Because when the run status is *stopped* then you can be sure that all the threads are terminated and all the allocated resources are free again.

9 Status

With the run status you can discover a defunct service that will not stop or will not free important resources. But the run status does not mean that the service healthiness is OK. The run status indicates only that some threads are still running, but not what they are doing. The *status* command of *scmd* will show you this information (see the ARE_guide).


It is possible to ask the services directly about their status. This is possible for `orb` and `client` type services. If the service implements the functionality then you will get more detailed information about the healthiness of the service.



The status request runs in its own thread in the service thread pool. A timeout is applied to its execution. The timeout is specified by the property: `agetor.servicerunner.timeout`. The value is specified in seconds.

10 Message

It is possible to send text messages to `client` type services. This means that you can communicate with your service without having an ORB. For example, you could send message like: `pause`, `reloadconfig`, `restart`, etc. The message can be send using `scmd`'s message command (see the `ARE_guide`).





-  The status request runs in its own thread in the service thread pool. A timeout is applied to its execution. The timeout is specified by the property: `agetor.servicerunner.timeout`. The value is specified in seconds.

11 Stop services

The stop command sends a stop or kill signal to a service that extends `AbstractServiceClient`. The stop command will wait for the service to terminate while the kill command will wait only for a specified number of seconds thereafter if the service has not terminated then the `ServiceRunner` tries to kill the service by stopping all the running threads.

In order to terminate the service in a consistent status then services has to stop any other threads, which has been started when they receive a stop or kill signal. Stop and kill signal are sent to the client and orb service types.

The kill signal has the time left back before the threads are interrupted by the `ServiceRunner`. The service could use the time to decide if the operation can be completed or aborted.

-  The threads in the `java services` type will be interrupted without notification while the external services are destroyed directly. Therefore in order to terminate the service in a consistent status then you should choice to implement either an `orb` or `client` service type.
-  Java cannot stop a thread. A waiting thread can be interrupted by an `InterruptedException`. Therefore it is important that the service catches and handles the `InterruptedException`.
-  Some operations like IO operations cannot be interrupted therefore the `ServiceRunner` calls the `onKill` method before start killing a service. The services, which extend the `AbstractServiceClient`, should implement the `onKill` method in order to abort any running operation.
-  When the `ServiceRunner` is killed (by the `scmd` command `shutdown -kill:<time>`) then the `ServiceRunner` runs the kill process for the running services thereafter the `ServiceRunner` JVM terminates after the specified number of seconds. The services **cannot** block this process. Therefore it is very important that you use the `kill` and `onKill` method in order to terminate the service in a consistent status.

12 Shutting down the ServiceRunner

As the stop command, the shutdown command sends a shutdown signal to the `ServiceRunner`, which then sends a stop signal to the services. The `ServiceRunner` will wait for all the services to

terminate, and then exit. However, you can still manage the ServiceRunner in the while it is shutting down, e.g., by sending a status command.

If a number *n* is specified in the shutdown command the ServiceRunner will exit the JVM after *n* seconds. This means that if the service is not terminated after *n* second then it is killed immediately. There is no way for a service to block this process. Therefore it is very important that you use the kill and onKill method in order to terminate the service in a consistent status.

The shutdown command in the scmd will not return until the services are terminated. If you know that you have a service which take some time at stopping then you can call the “shutdown &” command. This sends the shutdown signal and returns immediately. In this way you can continue to work with the scmd (e.g. ask for the status). If you use this command then you will get a ConnectionBrokenException when you run a command after the ServiceRunner has terminated.

13 Log

The ServiceRunner is able to redirect the standard output and error from a service to a file. However, it is important to note that the whole standard output and error is redirected. Many sub components in a service could write to those streams. Therefore, if you want a clean log file containing only the relevant information, you should create your own file and assign it to the Log system. Then all the others sub components, which are not writing in your log file, will use the log file created from the ServiceRunner.

 Log files can get very big, therefore it is important to write only the relevant information.

14 Troubleshooting

In some rare occasions the simultaneous load of multiple classes, which is performed when services start with individual class loaders, has been known to cause a deadlock situation where the service runner cannot be contacted. This problem has only been seen with ADK 2.0.3 when using SSL socket support (JDK 1.4.x+). The problem has been solved by having the servicerunner startup before any services are launched. In case of conflict between services on classload you may set the `agetor.servicerunner.servicestartdelay` property to a value that ensures each service has time enough to load its classes. E.g.:

```
agetor.servicerunner.servicestartdelay=20000
```

which will cause a delay of 20 seconds before each service is launched.