Dynamic LiveCycle/Blaze Data Services 2.6 Configuration

October 7, 2009
Eric Garza

Summary

This document provides the reader with a discussion on how to use the LiveCycle/Blaze Data Services run-time configuration APIs to create, modify, and delete services, destinations, and adapters dynamically on the server without the need for any Data Services configuration files.

Introduction

An introduction to endpoints and channels will be presented as a preface to provide some context for dynamic configuration of services. This chapter will also describe how to create LiveCycle Data Services objects from MXML and ActionScript at run-time.

About Channels and Endpoints

Channels are client-side objects that encapsulate the connection behavior between Flex components and the LiveCycle Data Services ES server. Channels communicate with corresponding endpoints on the LiveCycle Data Services ES server. You configure the properties of a channel and its corresponding endpoint in the **services-config.xml** file.

Configuring channels and endpoints

You configure channels in channel definitions in the **services-config.xml** file. The channel definition in the following example creates an `AMFChannel` that communicates with an `AMFEndpoint` on the server:

```
<channels>
        ...
        <channel-definition id="samples-amf"
              type="mx.messaging.channels.AMFChannel">
              <endpoint url="http://servername:8400/myapp/messagebroker/amf" port="8700"
                    type="flex.messaging.endpoints.AMFEndpoint"/>
        </channel-definition>
</channels>
```

The channel-definition element specifies the following information:

- id and channel class type of the client-side channel that the Flex client uses to contact the server
- endpoint element that contains the URL and endpoint class type of the server-side endpoint
- properties element that contains channel and endpoint properties
- server element, which when using an NIO-based channel and endpoint optionally refers to a shared NIO server configuration

The endpoint URL is the specific network location that the endpoint is exposed at. The channel uses this value to connect to the endpoint and interact with it. The URL must be unique across all endpoints exposed by the server. The url attribute can point to the `MessageBrokerServlet` or an NIO server if you are using an NIO-based endpoint.

How channels are assigned to a Flex component

Flex components use channel sets, which contain one or more channels, to contact the server. You can automatically or manually create and assign a channel set to a Flex component. The channel allows the component to contact the endpoint, which forwards the request to the destination.

Alternative to using `-services` compile flag

If you compile an MXML file using the MXML compiler option `-services` pointing to the **services-config.xml** file, the component (`RemoteObject`, `HTTPService`, and so on) is automatically assigned a channel set that contains one or more appropriately configured channel instances. The configuration is based on the channel definition assigned to a destination in a configuration file. Alternatively, if you do not compile your application with the –services option or want to override the compiled-in behavior, you can manually create a channel set in MXML or ActionScript, populate it with one or more channels, and then assign the channel set to the Flex component.

Application-level default channels are especially important when you want to use dynamically created destinations and you do not want to create and assign channel sets to your Flex components that use the dynamic destination. In that case, application-level default channels are used. For more information, see "**Assigning channels and endpoints to a destination**" on page 44 of the LiveCycle DS Development Guide.

When you compile a Flex client application with the MXML compiler `-services` option, it contains all of the information from the configuration files that is needed for the client to connect to the server. To manually create channels, you create your own `channelSet` in MXML or ActionScript, add channels to it, and then assign the channel set to a component. This process is common in the following situations:

- You do not compile your MXML file using the `-services` MXML compiler option. This is useful when you do not want to hard code endpoint URLs into your compiled SWF files on the client.
- You want to use a dynamically created destination (the destination is not in the **services-config.xml** file) with the run-time configuration feature. We will describe this scenario in the next section.
- You want to control in your client code the order of channels that a Flex component uses to connect to the server.

When you create and assign a channel set on the client, the client requires the correct channel type and endpoint URL to contact the server. The client does not specify the endpoint class that handles that request, but there must be a channel definition in the **services-config.xml** file that specifies the endpoint class to use with the specified endpoint URL.

The following example shows a `RemoteObject` component that defines a channel set and channel inline in MXML:

```
...
<RemoteObject id="ro" destination="Dest">
      <mx:channelSet>
          <mx:ChannelSet>
```

```
                <mx:channels>
                    <mx:AMFChannel id="myAmf" uri="http://myserver:2000/myapp/messagebroker/amf"/>
                </mx:channels>
            </mx:ChannelSet>
        </mx:channelSet>
</RemoteObject>
...
```

The following example shows ActionScript code that is equivalent to the MXML code in the previous example:

```
...
private function run():void {
        ro = new RemoteObject();
        var cs:ChannelSet = new ChannelSet();
        cs.addChannel(new AMFChannel("myAmf","http://servname:2000/eqa/messagebroker/amf"));
        ro.destination = "Dest";
        ro.channelSet = cs;
}
...
```

Important: *When you create a channel on the client, you must still include a channel definition that specifies an endpoint class in the services-config.xml file. Otherwise, the message broker cannot pass a Flex client request to an endpoint*

Assigning channels and endpoints to a destination

Settings in the LiveCycle Data Services ES configuration files determine the channels and endpoints from which a destination can accept messages, invocations, or data, except when you use the run-time configuration feature. The channels and endpoints are determined in one of the following ways:

Most destinations across services using the same channels

If most of the destinations across all services use the same channels, you can define application-level default channels in the `services-config.xml` file, as the following example shows.

Note: *Using application-level default channels is a best practice wherever possible*

```
<services-config ...>
        ...
        <default-channels>
            <channel ref="my-http"/>
            <channel ref="my-amf"/>
        </default-channels>
...
```

In this case, all destinations that do not define channels use a default channel. Destinations can override the default channel setting by specifying their own channels, and services can also override it by specifying their own default channels.

Most destinations in a service using the same channels

If most of the destinations in a service use the same channels, you can define service-level default channels, as the following example shows:

```
<service ...>
       ...
       <default-channels>
            <channel ref="my-http"/>
            <channel ref="my-amf"/>
       </default-channels>
       ...
```

In this case, all destinations in the service that do not explicitly specify their own channels use the default channel.

Channel inline in a destination

The destination definition can reference a channel inline, as the following example shows:

```
<destination id="sampleVerbose">
       <channels>
            <channel ref="my-secure-amf"/>
       </channels>
       ...
</destination>
```

Choosing an endpoint

The two types of endpoints in LiveCycle Data Services ES are servlet-based endpoints and NIO-based endpoints that use Java New I/O APIs. The J2EE servlet container manages networking, IO, and HTTP session maintenance for the servlet-based endpoints. NIO-based endpoints are outside the servlet container and run inside an NIO-based socket server. NIO-based endpoints can offer significant scalability gains. Because they are NIO-based, they are not limited to one thread per connection. Far fewer threads can efficiently handle high numbers of connections and IO operations.

If the web application is not servicing general servlet requests, you can configure the servlet container to bind

non-standard HTTP and HTTPS ports. Ports 80 and 443 are then free for your NIO-based endpoints to use. Because LiveCycle Data Services ES is a superset of BlazeDS, you still have access to the servlet-based endpoints if you want to use them instead.

The servlet-based endpoints are part of both BlazeDS and LiveCycle Data Services ES. Reasons to use servlet-based endpoints when you have LiveCycle Data Services ES are that you must include third-party servlet filter processing of requests and responses or you must access data structures in the application server `HttpSession`.

The NIO-based endpoints include RTMP endpoints as well as NIO-based AMF and HTTP endpoints that use the same client-side channels as their servlet-based counterparts.

The following situations prevent the NIO socket server used with NIO-based endpoints from creating a real-time connection between client and server in a typical deployment of a rich Internet application:

- The only client access to the Internet is through a proxy server.
- The application server on which LiveCycle Data Services ES is installed can only be accessed from behind the web tier in the IT infrastructure.

Servlet-based streaming endpoints and long polling are good alternatives when NIO-based streaming is not an option. In the worst case, the client falls back to simple polling. The main disadvantages of polling are increased overhead on client and server machines, and increased network latency.

For complete endpoint and channel selection information, see "**Assigning channels and endpoints to a destination**" on page 46-48 of the LiveCycle DS Development Guide.

Runtime Service Configuration

Using run-time configuration provides server APIs that let you create, modify, and delete services, destinations, and adapters dynamically on the server without the need for any Data Services configuration files.

Runtime configuration overview

Using run-time configuration provides server-side APIs that let you create and delete data services, adapters, and destinations, which are collectively called components. You can create and modify components even after the server is started.

There are many reasons why you might want to create components dynamically. For example, consider the following use cases:

- You want a separate destination for each doctor's office that uses an application. Instead of manually creating destinations in the configuration files, you want to create them dynamically based on information in a database.

- You want a configuration application to dynamically create, delete, or modify destinations in response to some user input.

There are two primary ways to perform dynamic configuration. The first way is to use a custom bootstrap service class that the `MessageBroker` calls to perform configuration when the BlazeDS server starts up. This is the preferred way to perform dynamic configuration. The second way is to use a `RemoteObject` instance in a Flex client to call a remote object (Java class) on the server that performs dynamic configuration.

The Java classes that are configurable are `MessageBroker`, `AbstractService` and its subclasses, Destination and its subclasses, and `ServiceAdapter` and its subclasses. For example, you use the `flex.messaging.services.HTTPProxy` service class to create an HTTP proxy service, the `flex.messaging.services.http.HTTPProxyAdapter` class to create an HTTP proxy adapter, and the `flex.messaging.services.http.HTTPProxyDestination` class to create an HTTP proxy destination. Some properties (such as `id`) cannot be changed when the server is running. The API documentation for these classes is included in the public BlazeDS Javadoc documentation.

Configuring components with a bootstrap service

To dynamically configure components at server startup, you create a custom Java class that extends the `flex.messaging.services.AbstractBootstrapService` class and implements the `initialize()` method of the abstract `BootstrapService` class. You can also implement the `start()`, and `stop()` methods of the `AbstractBootstrapService` class; these methods provide hooks to server startup and shutdown in case you need to do special processing, such as starting or stopping the database as the server starts or stops.

Method	Description
`public abstract void initialize(String id, ConfigMap properties)`	Called by the MessageBroker after all of the server components are created, but just before they are started. Components that you create in this method are started automatically. Usually, you use this method rather than the start() and stop() methods because you want the components configured before the server starts. The id parameter specifies the ID of the AbstractBootstrapService. The properties parameter specifies the properties for the AbstractBootstrapService.

Method	Description
`public abstract void start()`	Called by the MessageBroker as server starts. You must manually start components that you create in this method.
`public abstract void stop()`	Called by the MessageBroker as server stops.

You must register custom bootstrap classes in the services section of the **services-config.xml** file, as the following example shows. Services are loaded in the order specified. You generally place the static services first, and then place the dynamic services in the order in which you want them to become available.

```
<services>
      <service-include file-path="remoting-config.xml"/>
      <service-include file-path="proxy-config.xml"/>
      <service-include file-path="messaging-config.xml"/>
      <service class="dev.service.MyBootstrapService1" id="bootstrap1"/>
      <service id="bootstrap2" class="my.company.BootstrapService2">
      <!-- Bootstrap services can also have custom properties that can be
          processed in initialize method -->
          <properties>
              <prop1>value1</prop1>
              <prop2>value2</prop2>
          </properties>
      </service>
</services>
```

***Note**: The resources/config/bootstrapservices folder of the BlazeDS installation contains sample bootstrap services.*

Configuring components with a remote object

Java class that calls methods directly on components, and you expose that class as a remote object (Remoting Service destination) that you can call from a `RemoteObject` in a Flex client application. The component APIs you use are identical to those you use in a bootstrap service, but you do not extend the `AbstractBootstrapService` class.

The following example shows a Java class that you could expose as a remote object to modify a Message Service destination from a Flex client application:

```
package runtimeconfig.remoteobjects;
/*
* The purpose of this class is to dynamically change a destination that
* was created at startup.
```

```
*/
import flex.messaging.MessageBroker;
import flex.messaging.MessageDestination;
import flex.messaging.config.NetworkSettings;
import flex.messaging.config.ServerSettings;
import flex.messaging.config.ThrottleSettings;
import flex.messaging.services.MessageService;
public class ROMessageDestination
{
       public ROMessageDestination()
       {
       }

       public String modifyDestination(String id)
       {
            MessageBroker broker = MessageBroker.getMessageBroker(null);

            //Get the service
            MessageService service = (MessageService) broker.getService("message-service");

            MessageDestination msgDest = (MessageDestination)service.createDestination(id);
            NetworkSettings ns = new NetworkSettings();
            ns.setSessionTimeout(30);
            ns.setSharedBackend(true);
            ...
            msgDest.start();
            return "Destination " + id + " successfully modified";
       }
}
```

The following example shows an MXML file that uses a `RemoteObject` component to call the `modifyDestination()` method of an `ROMessageDestination` instance:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="run()">
       <mx:RemoteObject destination="ROMessageDestination" id="ro"fault="handleFault(event)"
       result="handleResult(event)"/>
       <mx:Script>
            <![CDATA[
                 import mx.rpc.events.*;
                 import mx.rpc.remoting.*;
                 import mx.messaging.*;
                 import mx.messaging.channels.*;
                 public var faultstring:String = "";
                 public var noFault:Boolean = true;
                 public var result:Object = new Object();
                 public var destToModify:String = "MessageDest_runtime";

                 private function handleResult(event:ResultEvent):void {
                      result = event.result;
                      output.text += "-> remoting result: " + event.result + "\n";
                 }

                 private function handleFault(event:FaultEvent):void {
                      //noFault = false;
```

```
                    faultstring = event.fault.faultString;
                    output.text += "-> remoting fault: " + event.fault.faultString +
                    "\n";
                }

                private function run():void {
                    ro.modifyDestination(destToModify);
                }
            ]]>
        </mx:Script>

        <mx:TextArea id="output" height="200" percentWidth="80" />
</mx:Application>
```

Accessing dynamic components with a Flex client application

Using a dynamic destination with a client-side data services component, such as an `HTTPService`, `RemoteObject`, `WebService`, `Producer`, or `Consumer` component, is essentially the same as using a destination configured in the **services-config.xml** file.

It is a best practice to specify a `channelSet` and `channel` when you use a dynamic destination, and this is required when there are not application-level default channels defined in the **services-config.xml** file. If you do not specify a `channelSet` and `channel` when you use a dynamic destination, BlazeDS attempts to use the default application-level channel assigned in the default-channels element in the **services-config.xml** file. The following example shows a default channel configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
        <services>
            ...
            <default-channels>
                <channel ref="my-polling-amf" />
            </default-channels>
            ...
        </services>
        ...
</services-config>
```

You have the following options for adding channels to a `ChannelSet`:

Create channels on the client

You can create channels on the client, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
        "http://servername:8400/messagebroker/amfpolling");
```

```
cs.addChannel(pollingAMF);
...
```

Retrieve compiled channel definitions

If you have compiled your application with the **server-config.xml** file, use the `ServerConfig.getChannel()` method to retrieve the channel definition, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = ServerConfig.getChannel("my-amf");
cs.addChannel(pollingAMF);
...
```

Usually, there is no difference in the result of either of these options, but there are some properties that are set on the channel and used by the client code. When you create your channel using the first option, you should set the following properties depending on the type of channel you are creating: `pollingEnabled` and `pollingIntervalMillis` on `AMFChannel` and `HTTPChannel`, and `connectTimeoutSeconds` on `Channel`. So, when you create a polling `AMFChannel` on the client, you should set the `pollingEnabled` and `pollingInterval` properties, as the following example shows:

```
...
var cs:ChannelSet = new ChannelSet();
var pollingAMF:AMFChannel = new AMFChannel("my-amf",
      "http://servername:8400/eqa/messagebroker/amfpolling");
pollingAMF.pollingEnabled = true;
pollingAMF.pollingInterval = 8000;
cs.addChannel(pollingAMF);
...
```

The second option, using the `ServerConfig.getChannel()` method, retrieves these properties, so you do not need to set them in your client code. You should use this option when you use a configuration file to define channels with properties. For components that use clustered destinations, you must define a `ChannelSet` and set the `clustered` property of the `ChannelSet` to true.

The following example shows MXML code for declaring a `RemoteObject` component and specifying a `ChannelSet` and `Channel`:

```
<RemoteObject id="ro" destination="Dest">
      <mx:channelSet>
          <mx:ChannelSet>
              <mx:channels>
```

```
                    <mx:AMFChannel id="myAmf" uri="http://myserver:2000/myapp/messagebroker/amf"/>
                </mx:channels>
            </mx:ChannelSet>
        </mx:channelSet>
</RemoteObject>
```

The following example shows equivalent ActionScript code:

```
private function run():void {
       ro = new RemoteObject();
       var cs:ChannelSet = new ChannelSet();
       cs.addChannel(new AMFChannel("myAmf",
            "http://{server.name}:{server.port}/eqa/messagebroker/amf"));
       ro.destination = "RemotingDest_runtime";
       ro.channelSet = cs;
}
```

One slight difference is that when you declare your `Channel` in MXML, you cannot have the dash (-) character in the value of `id` attribute of the corresponding channel that is defined on the server. For example, you would not be able to use a `channel` with an `id` value of 'message-dest'. This is not an issue when you use ActionScript instead of MXML.

References

This guide has been compiled from the following sources:

LiveCycle Blaze Data Services - **http://opensource.adobe.com/wiki/display/blazeds/Developer+Documentation**

LiveCycle Data Services 2.6 Development Guide - **http://livedocs.adobe.com/livecycle/8.2/programLC/programmer/lcds**