

BUILDING A FLEX APPLICATION INTEGRATED WITH BLAZE DS MESSAGING

BY AARON WEST

201

The Internet was built on the stateless nature of HTTP decades ago and it hasn't changed much over the years. Today, when you request a web page in your browser, every element on the page results in a client request to the server and a server response to the client. While there isn't anything particularly wrong with this behavior, it limits our ability to satisfy the demands of today's users. Rich Internet Applications (RIAs) are becoming more prevalent because they promote user collaboration, they deliver richer experiences, and they're simply more fun to use.

One way to create these types of applications with Flex is to bypass the request/response model of HTTP and build on technology included in Adobe's BlazeDS. This open source server platform for delivering messaging and remoting services enables developers to wire up applications with real-time distributed data.

BlazeDS has three key components:

- **Remoting** - If you've used the **HTTPService** or **RemoteObject** classes in Flex you have already made RPC (Remote Procedure Calls) connections to a back-end server. If you want to connect to BlazeDS with remoting, you'll continue to use these components to access resources via the BlazeDS server.
- **Messaging** - BlazeDS supports a publish/subscribe model allowing Flex or AIR to send messages over an asynchronous channel. BlazeDS describes clients that send messages as producers and clients that receive messages as consumers. Don't confuse this terminology with the client/server model. Both server and client can function as either message producer or message consumer.
- **Proxy** - BlazeDS includes a proxy service that allows you to build applications typically governed by a cross-domain policy (crossdomain.xml). News feeds, for example, work flawlessly when you access their URL in your browser, but display security errors when you attempt to access the same URL from Flex or AIR. BlazeDS conquers this problem by acting as the middleman, sending all your requests to the remote service for you.

In this article, I'll talk about using the messaging component of BlazeDS to build a one-way communication application. I'll describe a real-world application in use today at Dealerskins where I work. It allows members of the IT (Information Technology) team to send critical messages to the entire company. The messages sent could deal with email issues or network latency, or even contain a reminder about a meeting. Both remote sales staff and office workers can receive the messages in real time, creating communication employees have come to rely on. To build our application we married three technologies: AIR, BlazeDS, and ColdFusion. For simplicity, in this article I'll discuss building the application with Flex instead of AIR.

This article will be published in Flex Authority Volume 2 Issue 1.

<http://www.flex-authority.com>

BLAZE DS INSTALLATION AND CONFIGURATION

First, install and configure the BlazeDS server. I've created a video-based walkthrough that shows how to install BlazeDS on an existing ColdFusion 8 server. You can follow my instructions at <http://www.trajiklyhip.com/blog/index.cfm/2008/8/28/Integrating-BlazeDS-with-ColdFusion-8> or you can install BlazeDS on a Java server such as Tomcat. Once you have BlazeDS installed and working properly, you'll need to make a few configuration changes.

But before we move forward, I want to provide some background on the different ways to configure BlazeDS messaging.

Message producers and consumers communicate with BlazeDS over channels. On the surface, channels are nothing more than a few settings wrapped around an endpoint URI. These include polling settings, caching settings, serialization settings, and more. Channels in BlazeDS can be configured in one of three ways: polling, long polling and streaming.

POLLING

A channel with basic polling has consumers ask BlazeDS for messages at a specified polling interval. If BlazeDS has a message for the consumer, it responds immediately with the message. If it doesn't have a message for the consumer, it still responds immediately but does not include a message in the response. This series of requests and responses continues indefinitely as long as the consumers can connect to the BlazeDS server. Because of all this chatter over the wire, channels with basic polling do not scale well.

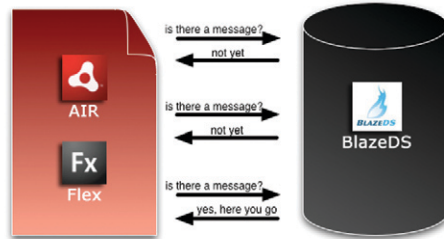


Figure 1: Basic Polling

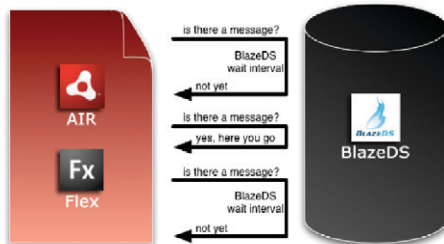


Figure 2: Long Polling

LONG POLLING

Long polling takes basic polling one step further. Consumers still ask BlazeDS for messages at a specified interval, but BlazeDS holds the initial request for messages for a period. If a message arrives before the end of that period of time, the server sends it immediately. If the wait interval completes before a message arrives, the server simply responds without a message. The BlazeDS wait interval provides the real beauty of long polling, since the technique creates the user experience of real time data messaging while reducing traffic. The messaging won't technically happen in real time, but most users won't notice the difference. Channels using long polling will scale better than basic polling channels, but still have a chatty quality because of the requests for new messages.

STREAMING

Streaming channels scale much better than polling and long polling channels. Consumers and producers initiate a connection with the BlazeDS server by asking for a stream. BlazeDS responds with a new stream. Once it has established a stream,

BlazeDS attempts to keep it alive by sending pings over the wire. All streaming uses standard HTTP ports with the pings designed to keep HTTP timeouts, firewalls, and proxy servers from interfering with the connection between client and server.

Since our application needs to deliver a real-time data experience, we will create a long polling channel. To do this, open the `services-config.xml` configuration file located in the BlazeDS installation directory. The `services-config.xml` file is the main configuration file for the BlazeDS server. Several channel definitions should already exist but you'll need to add a new one that looks like this:

Listing 1: Configuring a Long Polling Channel

```
<channel-definition id="cf-longpolling-amf"
  class="mx.messaging.channels.AMFChannel">
  <endpoint uri="http://{server.name}:{server.port}/
    {context.root}/flex2gateway/cfamflongpolling"
    class="flex.messaging.endpoints.AMFEndpoint"/>
  <properties>
    <polling-enabled>true</polling-enabled>
    <polling-interval-seconds>20</polling-interval-seconds>
    <wait-interval-millis>60000</wait-interval-millis>
    <client-wait-interval-millis>1</client-wait-interval-millis>
    <max-waiting-poll-requests>200</max-waiting-poll-requests>
  </properties>
</channel-definition>
```

Every channel on BlazeDS must have a unique ID, in this case `cf-longpolling-amf`. This channel uses the **AMFChannel** class. This means that all messages sent to and from BlazeDS will use Adobe's lightweight Action Message Format (AMF) protocol. This protocol sends binary data over HTTP and provides AMF serialization of data packets, which translates server data structures and client data structures into their native types. It's super fast and perfect for this type of messaging application.

Next the code defines an endpoint URI, the URI that producers and consumers ultimately access when communicating with the channel. Several variables in the URI get translated to server values at runtime.

We complete our long polling channel definition with a few properties that dictate how our channel will behave. Since it will use long polling, we must first enable polling. Next, we set the interval our Flex application will wait (twenty seconds) between polls of the BlazeDS server. These simple requests, sent by Flex (acting as a consumer of our channel), ask if the server has any messages that Flex needs to know about.

The next property, `wait-interval-millis`, sets how long the BlazeDS server will hold requests from consumers before responding to them. In the example above, BlazeDS will wait a maximum of sixty seconds and will respond immediately if a producer creates a message during that period. The next property determines how long consumers will wait between receiving an acknowledgement from BlazeDS and polling the server again. The value of 1 above means that consumers will immediately poll the server for messages. These four properties together will create the feel of real-time

messaging in our Flex application, since consumers will either be in the “waiting for a message” state or starting a new wait period after requesting new messages.

With our long polling channel defined, now we need to create a destination for our consumers and producers, wrapping multiple channels into one channel set. To create a new destination you edit either the `remoting-config.xml` file or the `messaging-config.xml` file, depending on the component of BlazeDS that Flex is using. Both files are located where BlazeDS was installed. Since our application uses the messaging component of BlazeDS, we’ll edit `messaging-config.xml`.

Open it and add the following code for the destination:

Listing 2: Configuring a BlazeDS Destination

```
<destination id="cfflexmessaging">
  <properties>
    <gatewayid>FlexMessaging</gatewayid>
  </properties>
  <channels>
    <channel ref="cf-longpolling-amf"/>
    <channel ref="cf-polling-amf" />
  </channels>
</destination>
```

Just like channels, destinations must have a unique ID, in this case `cfflexmessaging`. The properties block that follows can include several different settings. In this application, ColdFusion produces all messages from user text entered into a simple HTML form. Since ColdFusion is the message producer, we need to define the ColdFusion gateway—associated with a ColdFusion Component (CFC)—that will send the messages to BlazeDS. We’ll cover the creation of the CFC in a bit.

To finish defining our destination, we list any channels we want to associate with it. In this example, we include a long polling channel and a basic polling channel. Stacking channels in this way provides a fallback mechanism in case BlazeDS cannot connect to the client with a specific channel definition. In this case, if BlazeDS can’t connect to clients using the long polling channel, it will attempt to connect with the regular polling channel.

ColdFusion Configuration and Event Gateway

Messages displayed in the Flex application have to come from somewhere. The most straightforward approach is to create an HTML form with a textarea where users can submit messages. When a member of the IT staff types a message into the form and presses submit, the form submits to the original page. ColdFusion then creates an instance of a CFC named `FlexMessaging.cfc`, reads the message string out of the FORM scope, and calls the `createMessage()` function of the CFC, passing in the message string. The code to perform these tasks might look like this:

```
<cfset variables.messagingCFC = CreateObject("component", FlexMessaging)>
<cfset variables.messagingCFC.createMessage(FORM.message)>
```

The FlexMessaging.cfc component has two functions: `createMessage()` and `onIncomingMessage()`. The `createMessage()` function receives the message string entered in the HTML form, creates a BlazeDS message and then sends the message through the event gateway. The BlazeDS message contains two basic properties: body and destination. The body of the message has two properties, the date and time the message was sent and the message itself. The destination property tells BlazeDS which of the destinations in `messaging-config.xml` to send the message to. Finally, the `SendGatewayMessage()` function provides communication with the event gateway. The first argument of the function call is the name of the event gateway followed by the name of the BlazeDS message structure we created.

Listing 3: ColdFusion Component / Event Gateway

```
<cfcomponent displayname="FlexMessaging"
    hint="Links CF and BlazeDS for messaging" output="false">
    <cffunction name="createMessage" access="remote" returnType="void">
        <cfargument name="message" type="string" required="true">

        <cfset var messageBody = ArrayNew(1)>
        <cfset var blazeMessage = StructNew()>

        <cfset messageBody[1] = StructNew()>
        <cfset messageBody[1].sentAt = DateFormat(Now(), "mm-dd-yyyy") &
            " " & TimeFormat(Now(), "hh:mm tt")>
        <cfset messageBody[1].message = arguments.message>
        <cfset blazeMessage.body = messageBody[1]>
        <cfset blazeMessage.Destination = "cfflexmessaging">
        <cfset SendGatewayMessage("FlexMessaging", blazeMessage)>
    </cffunction>

    <cffunction name="onIncomingMessage" access="public" returnType="struct">
        <cfargument name="messageArg" type="struct" required="true">
        <cfset var msg = StructNew()>
        <cfset msg.body = arguments.messageArg.data.body>
        <cfset msg.destination = arguments.messageArg.data.destination>
        <cfset msg.headers['gatewayid'] =
            arguments.messageArg.data.headers.gatewayid>
        <cfreturn msg>
    </cffunction>
</cfcomponent>
```

If we want messages to make their way to BlazeDS and eventually to all consumers, we need to create the ColdFusion event gateway instance we called FlexMessaging. To do this, we need access to the ColdFusion Administrator. Once we log in, we'll navigate to the Gateway Instances page located under Event Gateways and add a new gateway. To do this, enter "FlexMessaging" in Gateway ID, select "DataServices Messaging" under Gateway Type and either type or choose the path to the FlexMessaging.cfc we created earlier. After creating the gateway instance, make sure it is running.

With this event gateway instance in place, all messages sent though the `createMessage()` function wind up calling the `onIncomingMessage()` function of our FlexMessaging CFC. This function accepts one argument, the BlazeDS

This article will be published in Flex Authority Volume 2 Issue 1.

<http://www.flex-authority.com>

message structure. The body of the function parses the two properties of the message, `body` and `destination`, determines the gateway ID from the message headers and then simply returns a new structure. When the new structure is returned, the message is sent to all consumers by BlazeDS.

FLEX APPLICATION

Now that we have messages being produced we need to create a Flex application that will serve as a consumer of the `cfflexmessaging` BlazeDS destination.

Listing 4: Flex Application Code

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  creationComplete="initApp()" cornerRadius="10" fontFamily="Verdana"
  color="#000000" backgroundImage="images/what.jpg" width="337" height="546"
  verticalScrollPolicy="off" horizontalScrollPolicy="off">

  <mx:Script>
    <![CDATA[
      import mx.messaging.messages.IMessage;
      import mx.collections.ArrayCollection;
      [Bindable]
      private var messageCollection:ArrayCollection = new ArrayCollection;

      private function initApp():void {
        messageCollection.addItem("...Now listening for messages...");
        consumer.subscribe();
      }

      private function messageHandler(message:IMessage):void {
        messageCollection.addItemAt(message.body.SENTAT +
          "\n" + message.body.MESSAGE, 0);
      }
    ]]>
  </mx:Script>

  <mx:Consumer id="consumer" destination="cfflexmessaging"
    message="messageHandler(event.message)"/>

  <mx:Canvas x="0" y="41" width="335" height="499" id="mainCanvas">
    <mx:List id="messageList" styleName="messageText"
      dataProvider="{messageCollection}" x="8" y="0"
      width="317" height="499" alpha="1.0" backgroundAlpha="0.6"
      wordWrap="true" variableRowHeight="true"
      color="#FFFFFF" borderColor="#000000" themeColor="#41060A"
      cornerRadius="5" fontFamily="Verdana"
      fontSize="12" backgroundColor="#000000"/>
  </mx:Canvas>
</mx:Application>
```

The MXML and ActionScript for our consumer is pretty basic. ActionScript handles received messages and subscribes to our BlazeDS destination as a consumer. Then a bit of MXML handles the display of messages with a **List** component. When the

Flex application fires up the `creationComplete` event executes, which causes the `initApp()` function to run. This function adds a generic message to an **ArrayCollection** that holds all received messages.

Next, our consumer—as defined in the MXML Consumer class—registers itself with BlazeDS in order to receive any messages produced on the `cfflexmessaging` destination. The **Consumer** class sets up an event handler in the `message` attribute. When the Flex application receives messages, the `messageHandler()` function executes and receives the sent message as an event of type `IMessage`. The `messageHandler()` function adds a new string element at index zero of the `messageCollection` **ArrayCollection**. The string element added to the **ArrayCollection** contains both the date and time the message was sent and the message string itself, separated by a newline character. Since the `messageCollection` **ArrayCollection** is bound to the `messageList` **List** component, every new message will automatically display at the top of the list.

The HTML form, ColdFusion CFC, Flex application, and underlying BlazeDS server all come together to create something that looks like Figure 3. Messages are submitted in the textarea and ColdFusion functions as a message producer, handing the messages off to BlazeDS. BlazeDS sends the messages to the Flex application, functioning as a message consumer, through a long polling channel. New messages appear at the top of the **List**

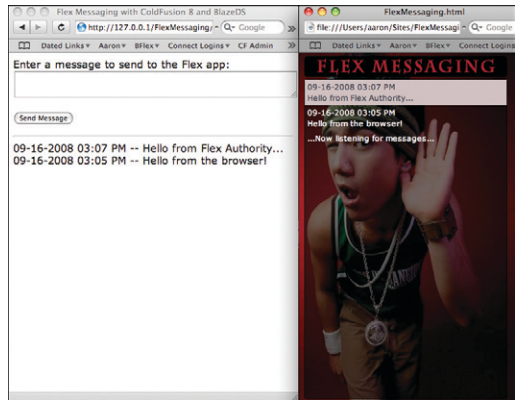


Figure 3: Final Application

component in Flex as well as in the browser. The entire browser-to-server-to-Flex client roundtrip occurs almost instantaneously, creating an experience that feels much like instant messaging (except it's one-way).

While this type of application might not amount to a full-fledged RIA, it certainly offers an improvement over logging into a back-end system to check for messages or pinging your mail server over and over to see if there's something new. In addition, the application does not depend on the email service; when it goes down, you can still let people know. If you wanted to take the idea further, you could spend 5-10 minutes converting the Flex application into an AIR application deployed on employee desktops. Then everyone, from customer support to remote sales, remains in the loop on system status.

AARON WEST is the Development Manager at Dealerskins in Nashville, TN and has worked with ColdFusion since 2000 and Flex since version 1.5. He's been the manager of the Nashville ColdFusion User Group for three years and has served the ColdFusion, Flash, and Flex community as an Adobe Community Expert since the program's inception. Aaron participates in several online ColdFusion communities and blogs at www.trajiklyhip.com/blog.

