Get More Refcardz! Visit refcardz.com

www.dzone.com

Very First Steps in Flex

CONTENTS INCLUDE:


- About Flex
- Hello, World
- Web Services
- Remoting and Messaging with Java
- Hot Tips and more...


# Very First Steps in Flex

By Bruce Eckel and James Ward

## ABOUT FLEX

Rich Internet Applications and Flex are taking off. Many developers need to learn Flex fast. That is why we wrote the book *First Steps in Flex*. We're giving you just the right information to get you started. This refcard highlights three useful chapters which help readers get started learning Flex. The first is a simple "Hello, World" application which will help familiarize readers with the Flex Builder tool and programming model. Secondly, we'll go into depth about how to connect Flex applications to back-end data through RESTful XML and SOAP. And lastly, "Remoting and Messaging with Java" walks through how to easily connect to a Java back-end with the open source BlazeDS product.

## HELLO, WORLD

### Build Your First Flex Applications

First, download and install the 60-day free trial of Flex Builder from http://www.adobe.com/go/flex_trial. The easiest way to do this is to get the standalone installation (even if you already have Eclipse installed for some other purpose). There is also an Eclipse plugin if you are so inclined.

### Display a Label

- Select **File|New|Other**. This brings up the "Select a wizard" dialog box.
- Select "Flex Project." This brings up the "New Flex Project" Dialog box.
- Type in a project name: **helloWorld**.
- Use the default location (which will already be checked).
- Select "Web application (runs in Flash Player)."
- Leave everything else alone and press "Finish."

Your project will open in the MXML code editor. You will see the file titled **helloWorld.mxml**. Note that this is a valid XML file.

- Now add an XML tag in between the Application tags: **<mx:Label text="Hello, world"/>**

This inserts a **Label** component into your Flex application, which should now look like the following.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application name="helloWorld"
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">
    <mx:Label text="Hello, world"/>
</mx:Application>
```

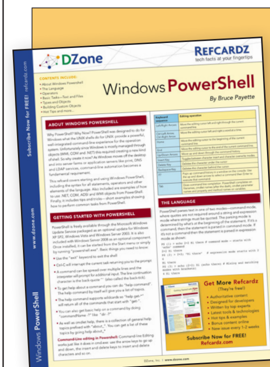
Flex Builder automatically puts in the first line that you see in the listing above; this is the *doctype* line which specifies that this is a standard XML file. However, Flex ignores this line when compiling the program and so it can be omitted.

The **name** property in the **Application** tag is ignored by Flex; we use it to give the name of the file containing program. In this case, the full file name is **helloWorld.mxml**.

Now run the application. Go to the **Run** menu and select "Run." This should launch your web browser and run the application within the browser. You'll see the words "Hello, world" in a field of blue.

Note that "Hello, World" is in the upper left corner. Modify the Application tag by removing the **layout="absolute"** property and re-run the application. You'll see that "Hello, world" is now centered.

The Application component in Flex as well as other components like Panel have a layout property. The valid values for layout are "absolute", "vertical", and "horizontal". This selects the container class which is used inside the component. If unspecified the "vertical" layout is used. The "absolute" layout utilizes the Canvas contains. In a Canvas components are positioned with x and y values or using constraints like top, bottom, left, right, horizontalCenter, verticalCenter. Setting these properties on non-absolutely positioned containers like VBox and HBox has no effect. The "vertical" layout utilizes the VBox container which positions its children in a vertical stack (top to bottom). The "horizontal" layout utilizes the HBox container which positions its children next to one another (left to right).


Get More Refcardz
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

## Build Your First Flex Applications, continued

MXML allows you to describe and configure components in a higher-level, declarative fashion. However, MXML is always translated into the (procedural) ActionScript language, which is then compiled into SWF bytecode to be executed by the Flash Virtual Machine (VM).

## Control Label Text with Data Binding

Create a new application called **helloWorld2**, and remove the **layout** property. Place your cursor within the **Application** body and add a **String** and a **Label** component. Note that when you enter '<' and then start typing, for example, "String," Flex Builder will perform command completion for you. Make your example look like this:

```
<mx:Application name="helloWorld2"
    xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:String id="message">Hello, world</mx:String>
    <mx:Label text="{message}"/>
</mx:Application>
```

The **String** object has an **id** property. The **id** is the name of the object, and is necessary so that other objects can talk to it. You'll use **id** a lot.

Here, the **id** is **message**, and the **Label** component uses this id, but within curly braces. Curly braces have a special meaning, which is "bind to this other object." In this case, the **text** field in the **Label** object is fetched from the **String** object. That's data binding—the data in one object is bound to the data in another.

In the above example the string doesn't change during the execution of the program. Data binding is especially interesting when the bound-to data does change, because the **Label** will be automatically updated. Let's look at a more complex example:

```
<mx:Application name="helloWorld3"
    xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:TextInput id="message" text="Hello, world"/>
    <mx:Label text="{message.text}"/>
</mx:Application>
```

Now we've added another component, a **TextInput** field. The **Label**'s **text** field is bound to **message.text**, so when you modify the **TextInput** the label will automatically change.

**Data Binding Makes Development Simpler**

Under the covers, data binding generates fairly complex code in order to watch for changes and respond to them. But you don't have to think about this; all you need is the curly braces. This is one of many cases where Flex does a lot of work for you in order to keep your life simple.

Switch to *Design View:* in the upper left corner you'll see a button that allows you to toggle between "source" and "design." Experiment by adding components to your layout by dragging and dropping them, then switch back to "source" view and notice that you've just added more MXML components. In "design" view, try configuring the components using the "Flex Properties" pane.

**Flex Communities Online**

There are a number of online communities out there to help answer your questions about Flex:

**Flexcoders Yahoo Group**
http://FirstStepsInFlex.com/go/Flexcoders

**Adobe Forums**
http://FirstStepsInFlex.com/go/AdobeForums

Blogs are also a great way to continue learning about Flex. **http://feeds.adobe.com** has a Flex category which makes finding Flex blogs easy.

# WEB SERVICES

## Flex easily uses web services to get and change data on remote servers.

Flex applications, whether in the browser or on the desktop, run on the client side. Thus, there are no special server-side components necessary when building Flex applications. However, most Flex applications get data from remote servers, make changes to that data and then submit those changes back to the server.

Flex provides many ways to load and manipulate remote data. Most of these methods are simple and can work with any back-end technology such as .Net, Java, Ruby, Python, PHP, etc.—as long as the back-end logic is exposed through some kind of web service. *Web services* is a broad term which can mean SOAP, RESTful, JSON, XML-RPC, and more. Flex can easily connect to any of these web services.

## General Purpose HTTP Networking

**HTTPService** is the easiest way for a Flex application to make network requests. It makes an HTTP request to a server and returns the results. This simple example makes a request for an XML file located at http://FirstStepsInFlex.com/bookmarks.xml

```
<mx:Application name="UsingHTTPService"
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="srv.send()">
    <mx:HTTPService id="srv"
        url="http://FirstStepsInFlex.com/bookmarks.xml"/>
    <mx:DataGrid
        dataProvider="{srv.lastResult.bookmarks.item}"/>
</mx:Application>
```

The **HTTPService url** property is set to the URL of the file we are requesting. After the application has finished initializing, the **applicationComplete** event is dispatched, which goes to the event handler of the same name, which calls **srv.send()**. The meaning of "send" here is "send the request to the server." You can send request parameters as arguments to the **send()** method.

The **DataGrid** displays the results of the request by binding to **srv.lastResult**, which is dynamically set when the response comes back from the server.

By default the **srv.lastResult** comes back as objects, which the **HTTPService** automatically converts from XML. What does this mean? Each XML node becomes an object, which contains other objects. If a node contains multiple subnodes, you end up with an array. If a node is just names and properties, then

## General Purpose HTTP Networking, continued

that node becomes an **Object** with corresponding keys and values. This algorithm repeats to the depth of your XML tree.

You can also set the **HTTPService resultFormat** property to **"e4x"** in which case the results will be kept as native **XML** objects for use in E4X expressions. Or you can set **resultFormat** to "text" which allows for custom deserializers like JSON.

Binding to the **lastResult** is the simplest way to get the results of a network request. You can also write a result event handler and attach it to the **HTTPService**:

```
<mx:Application name="HTTPServiceResultHandler"
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="srv.send()">
    <mx:HTTPService id="srv"
        url="http://FirstStepsInFlex.com/bookmarks.xml">
        <mx:result>
        dg.dataProvider = event.result.bookmarks.item;
        </mx:result>
    </mx:HTTPService>
    <mx:DataGrid id="dg"/>
</mx:Application>
```

Note that the **DataGrid** isn't configured with a **dataProvider**. Instead, this happens in the **result** event handler, which is written as a script block.

**HTTPService** also has a **fault** event for requests that return errors.

## SOAP Web Services

The **WebService** component connects to a remote server using SOAP:

```
<mx:Application name="UsingSOAP"
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="srv.GetInfoByZIP(94103)">
    <mx:WebService id="srv"
        wsdl="http://www.webservicex.net/uszip.asmx?wsdl">
        <mx:operation name="GetInfoByZIP">
            <mx:result>
            dg.dataProvider = event.result.NewDataSet.Table
            </mx:result>
        </mx:operation>
    </mx:WebService>
    <mx:DataGrid id="dg"/>
</mx:Application>
```

> **Note** If you get an error when running this program, close all instances of your browser and try again. For an explanation, refer to the following Networking Limitations section.

SOAP defines operations that are specified by its WSDL (Web Services Description Language). Once we set the **wsdl** property of the **WebService** to the remote WSDL URL, those operations can be called on the **WebService** object as if they were native methods of that object (this is one of the many benefits of using a dynamic language).

Here, the call to the **WebService** method **GetInfoByZip()** takes a single parameter. The **result** event handler puts the data returned from the request into the **DataGrid**. Data binding can also be used with the results of the request by binding to the **srv.GetInfoByZIP.lastResult.NewDataSet.Table** property.

## Networking Limitations

The browser imposes several limitations on the networking capabilities of the Flash Player. The primary limitation is that applications running in the Flash Player can only make requests back to the server from which the application originated.

There are two workarounds:

1. Use a proxy server on the server from which the application originated from.
2. Use cross-domain requests.

Any server can act as a proxy. Apache is easy to configure as a proxy, as is the open source BlazeDS server (a subset of the LiveCycle Data Services product).

To enable cross-domain requests, server administrators must explicitly allow them. Unfortunately, this can open web sites to XSRF types of attacks. During development testing, the Flash Player lets you bypass this security restriction for specific applications. This happens automatically when developing Flex application in Flex Builder. It can also be done manually by modifying the Flash Player's trust file. For more information, see http://FirstStepsInFlex.com/go/FlashPlayerTrustFile

Another limitation of the networking capabilities of Flash Player is that non-200 HTTP response codes cannot be accessed in the Flash Player. This is due to limitations in the browsers' Plug-in networking APIs. Either insure that all responses from your server are 200, or utilize a proxy.

## Further Learning

The open source AS3Corelib library adds JSON support to **HTTPService**: http://FirstStepsInFlex.com/go/as3corelib

Flex Builder 3 supports a new technique for working with SOAP web services which stubs out the ActionScript objects for a given WSDL. This feature is available in Flex Builder from the Data menu: http://FirstStepsInFlex.com/go/WSDLImport

More information on **HTTPService** and **WebService**: http://FirstStepsInFlex.com/go/WebServices

# REMOTING AND MESSAGING WITH JAVA

## The open-source BlazeDS provides simple two-way communication with Java back ends.

The Web Services examples showed how Flex applications can easily communicate with back-end servers through various web services protocols. Those servers can be running Java, ColdFusion, .Net, PHP, Ruby or any number of other server-side technologies. While web services are an easy way to communicate between Flex applications and servers, other options exist which can dramatically increase application performance as well as developer productivity.

Every server technology can easily speak XML since it is a text-based protocol. XML is perfect when protocol transparency is necessary. For instance, Flickr's web services use RESTful-style XML over HTTP. This allows any developer using any technology to easily interact with Flickr by sending text-based requests. Flickr then responds with simple XML. One downside to text-based protocols like XML is that the additional layer of

## Remoting and Messaging with Java, continued

data abstraction is usually cumbersome to write and maintain. In addition, this data abstraction layer consumes resources on the server- and client-side when the data is serialized and deserialized.

For some time, Flash Player has supported a transport protocol that alleviates the bottlenecks of text-based protocols and provides a simpler way to communicate with servers. Called *Action Message Format* (AMF), this binary protocol for exchanging data can be used over HTTP in place of text-based protocols that transmit XML. Applications using AMF can eliminate an unnecessary data abstraction layer and communicate more efficiently with servers. To see a demonstration of the performance advantages of AMF, look at the *Census RIA Benchmark* at http://www.jamesward.org/census.

**Open Source AMF Implementations**

The AMF protocol has numerous open source implementations:

**Java: BlazeDS**
http://opensource.adobe.com

**.Net: FlourineFx**
http://FirstStepsInFlex.com/go/FlourineFx

**PHP: AMFPHP**
http://FirstStepsInFlex.com/go/AMFPHP

**Python: PyAMF**
http://FirstStepsInFlex.com/go/PyAMF

**Ruby: RubyAMF**
http://FirstStepsInFlex.com/go/RubyAMF

The open source BlazeDS project includes a Java implementation of AMF which is used for remotely communicating with server-side Java objects as well as for passing messages between clients. BlazeDS remoting technology allows developers to easily call methods on Plain old Java objects (POJOs), Spring services, or EJBs. Developers can use the messaging system to easily send messages from the client to the server or from the server to client. BlazeDS can also be linked to other messaging systems such as JMS or ActiveMQ. Because the remoting and messaging technologies use AMF over HTTP they gain the performance benefits of AMF as well as the simplicity of fewer data abstraction layers. BlazeDS works with a wide range of Java-based application servers, including Tomcat, WebSphere, WebLogic, JBoss, and ColdFusion. In addition, BlazeDS can be easily used in Flex applications for the web (running in the Flash Player) and the desktop (running in Adobe AIR).To get started, simply deploy the **blazeds-samples.war** file in any servlet container. This web application contains a number of preconfigured sample applications that can be accessed at http://localhost:8080/blazeds-samples (The port may vary depending on your application server and configuration).

## BlazeDS Remoting

To use the BlazeDS Remoting Service:

1. Create a new POJO Java class which exposes the methods you want to access from a Flex application.
2. Configure a BlazeDS remoting destination in the **remoting-config.xml** file.
3. Create a Flex application which uses the **RemoteObject** class.

Let's walk through those steps in more detail using Eclipse and Flex Builder. You will need the following software installed:

- Eclipse 3.3 Classic, from http://www.eclipse.org
- The Flex Builder 3 plugin for Eclipse, from http://www.adobe.com (For building this example, you must use the plugin rather than the standalone Flex Builder installation).
- Any Java Application Server (Tomcat, JBoss, WebLogic, WebSphere, etc.)
- BlazeDS from http://opensource.adobe.com

To create a simple Remoting Application:

1. Unjar the **blazeds.war** file from BlazeDS into your application server's deployment folder. For instance, on JBoss use **<JBOSS_HOME>/server/default/deploy/blazeds.war**
2. Start Eclipse + Flex Builder.
3. Create a new Java project that you can use to configure BlazeDS and add Java classes to your web application.
   a. Use a project name like **blazeds_server**
   b. Create the project from existing source; use the path of the WEB-INF directory of your deployed BlazeDS WAR, such as: **<JBOSS_HOME>/server/default/deploy/blazeds.war/WEB-INF**
   c. Add the **src** directory to the build path.
   d. Use the **WEB-INF/classes** directory as the output folder.
4. Create a new Java file called **HelloWorld.java** with the following code (this is your POJO):

```
public class HelloWorld {
     public String sayHello(String name) {
        return "hello, " + name;
     }
}
```

5. Configure BlazeDS to allow remoting requests to the **HelloWorld** class by adding a destination to the **remoting-config.xml** file found in the **WEB-INF/flex** directory. Use the following destination configuration:

```
<destination id="HelloWorld">
<properties>
     <source>HelloWorld</source>
</properties>
</destination>
```

6. Start your application server and verify that your web application is configured by going to the following URL (The port may vary depending on your application server and configuration): http://localhost:8080/blazeds
   (If your server is not configured to display directory contents, you might see a 404 error. This is OK.)

## BlazeDS Remoting, continued

7. Create a new Flex Project
   a. For the project name, type **testHelloWorld**.
   b. Select "J2EE" as the Application Server Type.
   c. Select "Use remote object access service" and LiveCycle Data Services.
   d. Specify the Root folder to be the location of your deployed WAR file.
   e. Specify the Root URL to be http://localhost:8080/blazeds (Your port name may be different depending on your application server and configuration).
   f. Specify the Context Root to be: /**blazeds**.
   g. Verify the configuration and click "Finish."
8. Create the Flex application by updating the **testHelloWorld.mxml** file with the following source code:

```
<mx:Application name="testHelloWorldUpdate"
    xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:RemoteObject id="ro" destination="HelloWorld"/>
    <mx:TextInput id="n" change="ro.sayHello(n.text)"/>
    <mx:Label text="{ro.sayHello.lastResult}"/>
</mx:Application>
```

9. Run the application and test it by typing your name into the **TextInput** box. You should see "hello, <your name>" displayed beneath the **TextInput**.

The Flex application uses the **RemoteObject** library to communicate with the BlazeDS-enabled server. When the user enters text in the **TextInput** box, the change event causes the **RemoteObject** to make a request to the server. The server then makes a request to the Java Class specified in the remoting destination configuration. This could call a Spring service or EJB session bean, but this example just calls a POJO. The POJO's return value is simply "hello," with the name argument appended.

When the POJO returns a value, that value is serialized into AMF and returned to the Flex application. The **RemoteObject** library then sets the **ro.<method name>.lastResult** property to the value that was returned. (In this case **ro.sayHello.lastResult**.) The result can also be obtained through a result event on the **RemoteObject**. Data binding triggers the **Label** to display the result.

BlazeDS also supports passing typed Java objects back and forth.

## BlazeDS Messaging

To use the BlazeDS Messaging Service:

1. Create a messaging destination in the **messaging-config.xml** file.
2. Create a Flex application that uses the **Producer** and **Consumer** classes to send and receive messages.
3. Begin listening for messages by subscribing to the **Consumer**'s message feed.

Let's build an application that uses the BlazeDS messaging system.

1. Start by adding a messaging destination to the **messaging-config.xml** file found in the **WEB-INF/flex** directory. Add the following destination:

```
<destination id="chat"/>
```

A messaging destination allows the messaging system to relay messages to clients listening on that destination, and it allows messages to be sent to the destination. Messaging destinations can have durability and network parameters, and they can also be connected to other messaging systems like JMS.

2. Restart your application server so that BlazeDS configures the new messaging destination.
3. Create a new Flex Project
   a. For the project name, type **testChat**.
   b. Select "J2EE" as the Application Server Type.
   c. Select "Use remote object access service" and LiveCycle Data Services.
   d. Specify the Root folder to be to location of your deployed WAR file.
   e. Specify the Root URL to be http://localhost:8080/blazeds (Your port name may be different depending on your application server configuration).
   f. Specify the Context Root to be: /**blazeds**.
   g. Verify the configuration and click "Finish."
4. To create a simple chat application that uses the messaging system, update **testChat.mxml** with the following code:

```
<mx:Application name="testChatUpdate"
    xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="cons.subscribe()">
    <mx:Script>
    import mx.messaging.messages.AsyncMessage;
    </mx:Script>
    <mx:Producer id="prod" destination="chat"/>
    <mx:Consumer id="cons" destination="chat"
    message="c.text += event.message.body.msg + '\n'"/>
    <mx:TextArea id="c" width="300" height="300"/>
    <mx:TextInput id="m"/>
    <mx:Button label="Send"
    click="prod.send(new AsyncMessage({msg: m.text}))"/>
</mx:Application>
```

5. Run the application. Enter a message in the lower **TextInput** box and click "Send." Verify that the message is displayed in the upper box. Also notice that messaging works across multiple browser windows.

When the user clicks the "Send" button, a new message is created using an anonymous object to set the **msg** property on the body of the message to the value in the **TextInput**. Because the message type is an **AsyncMessage**, that class must be imported.

The **Consumer** object allows Flex applications to listen for messages. When the application has initialized, it subscribes to the message system. Then, when a message is received, the event handler on the **Consumer** takes the chat message out of the body of the message and appends it to the **TextArea**.

→

## BlazeDS Messaging, continued

The Producer object allows Flex applications to send messages into the message system. There is also a Java API (not used in this example) which allows messages to be sent into the message system on the server.

With a custom adapter or the out-of-the-box JMS adapter you can connect the message system to other messaging systems, but by default the message system runs standalone.

## Further Learning

Downloading LiveCycle Data Services and taking the built-in Test Drive can be a great way to learn more about LCDS: http://FirstStepsInFlex.com/go/LCDS

BlazeDS also has a Test Drive: http://FirstStepsInFlex.com/go/BlazeDS

LCDS and BlazeDS both have two WAR files which help get developers started. The first is a samples war file containing demos and source code explaining the product features. There is also a plain WAR file with just the basics needed to start from scratch.

## ABOUT THE AUTHORS

**Bruce Eckel**
Bruce Eckel specializes in languages and rapid-development tools and techniques, and provides consulting, workshops and conferences. He is the author of the multi-award-winning Thinking in Java and Thinking in C++, among others. Visit his web site at www.MindViewInc.com.

**James Ward**
James Ward is a Technical Evangelist for Flex at Adobe. He travels the globe speaking at conferences and teaching developers how to build better software with Flex. Visit his web site at www.jamesward.com.

## RECOMMENDED BOOK

*First Steps in Flex* will give you just enough information, and just the right information, to get you started learning Flex. Enough so that you feel confident in taking your own steps once you finish the book. For more information visit http://www.firststepsinflex.com.

**BUY NOW**
**books.dzone.com/books/adobeflex**

## Get More FREE Refcardz. Visit refcardz.com now!

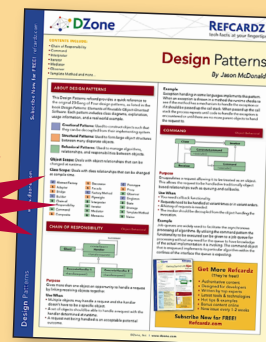
**Upcoming Refcardz:**

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

**Available:**

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server
Silverlight 2

Visit **refcardz.com** for a complete listing of available Refcardz.




Design Patterns
**Published June 2008**

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

**Refcardz Feedback Welcome**
refcardz@dzone.com

**Sponsorship Opportunities**
sales@dzone.com

ISBN-13: 978-1-934238-20-2
ISBN-10: 1-934238-20-1
50795
9 781934 238202
$7.95

Copyright © 2008 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.