CONTENTS INCLUDE:

Getting Started with LiveCycle Data Services ES

By Michael Slinn

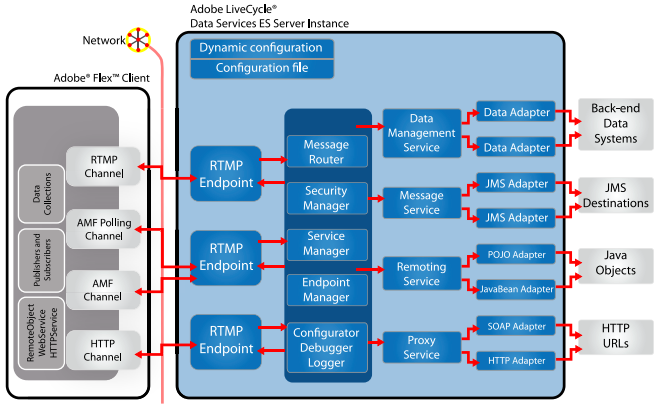
ABOUT LIVECYCLE DS

Adobe® LiveCycle® Data Services ES (LCDS) is a high-performance, scalable, and flexible framework that streamlines the development of Rich Internet Applications (RIAs) using the Flash Platform.

LCDS facilitates the creation of client-server applications and supports a rich set of features to create real-time and near real-time solutions, included support for occasionally connected AIR clients. LCDS provides numerous options for remote procedure calls, proxy support, lazy loading and server push, including publish/subscribe messaging, data synchronization, data paging and conflict resolution.

LCDS supports Java EE web applications. Versions are available for Windows, Linux, AIX, HP_UX and Solaris. LCDS v2.6.1 is not officially supported under Mac O/S X, however the v3.0 beta does provide support for Mac.

This Refcard briefly mentions the major features, discusses a few key concepts and shows how to get started with LCDS. The book entitled "**Flex Data Services, Hibernate and Eclipse**" contains much more information, complete code examples, and the LCDS version of the free software tool mentioned in this Refcard.



Installation

You can download the current version of LCDS from http://www.adobe.com/products/livecycle/dataservices/. You need to sign in with your Adobe ID to get the download. IDs are free, so register if you need one. The download page displays a serial number which you need to save in order to install the product.

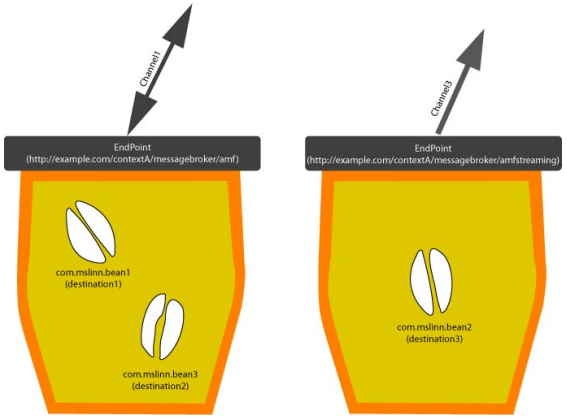
Unlike BlazeDS, LCDS has an installation script. By default, LCDS installs into `C:\Program Files\Adobe\lcds` on Windows.

Channels and Endpoints

Data moves between client and server in a manner prescribed by the data services configuration. Both the client and the server must be configured in a compatible manner. Adobe provides samples of five XML files that are parsed by Flex Builder and the server-side Data Services runtime at startup. You must modify some or all of these files in order to configure the data services transport for each project. The configuration files define client channels ("pipes"), server endpoints (the URL and properties) and destinations.

A client channel formats and translates messages into a network-specific form and delivers them to an endpoint on the server; channels define message formats, network protocols and network behaviors. An endpoint unmarshals messages in a protocol-specific manner. A destination is the server-side object or service that the endpoint connects to. Destinations are defined by channels and adapters; adapters connect directly with the server object or service. The LCDS message broker routes requests arriving at endpoints to the appropriate service destination.

The following diagram illustrates how client channels, server endpoints and destinations are related.



The message broker uses the last token of the URL to route the incoming request to the appropriate service for handling. The preceding diagram shows two endpoints, with routing tokens `amf` and `amfstreaming`. Endpoints are URLs that are responded to by the LCDS message broker.

An example of a server endpoint is http://localhost:8080/test/messagebroker/amf. The message broker uses the last token of the URL to route the incoming request to the appropriate service for handling. The preceding diagram shows two endpoints, with routing tokens amf and amfstreaming. The message broker is normally configured to pass requests arriving at these endpoints to services that handle remote procedure calls (RPCs) and streaming, respectively.

LCDS scales to orders of magnitude more clients per CPU than BlazeDS because it offers NIO-based server endpoints in addition to the servlet-based endpoints that BlazeDS offers. You can take a web application developed using BlazeDS and get much greater throughput simply by installing LCDS in place of BlazeDS, and editing `services-config.xml` to use NIO endpoints.

At a minimum, client channels must have an id, a class and an endpoint. The id allows the channel to be referenced. The class defines the scope of the channel's potential behavior. The endpoint defines the URL for a remote client to access the channel. Channels have default properties, which can be overridden. The available properties vary for each class of channel. The client-side channel classes are all defined in the `mx.messaging.channels` ActionScript package. The server-side endpoint classes are all defined in the `flex.messaging.endpoints` Java package.

The following server-side endpoints are all implemented using servlet-based blocking I/O, requiring one thread per connection. This table has many entries – defining a channel gets really confusing when all the options are considered. The free Flex Data Services Channel Designer described next, is intended to clarify the options and write the XML for the channels that your application needs.

Servlet-based endpoints and their client channels	
Binary AMF data	`AMFChannel / AMFEndpoint` Used for RPC when default properties are not overridden; used for messaging and data management service when polling properties are configured. Transmits data in the binary **AMF** format. Usually uses the `http` protocol.
	`SecureAMFChannel / SecureAMFEndpoint` Secure subclasses of `AMFChannel / AMFEndpoint`. Uses the `https` protocol.
	`StreamingAMFChannel / StreamingAMFEndpoint` Uses the HTTP 1.1 'chunked' server push model instead of polling for data from the server. An internal HTTP connection to the server is held open so the server can stream data to the client with little overhead. Cannot be compressed or proxied.
	`SecureStreamingAMFChannel / SecureStreamingAMFEndpoint` Subclasses of `StreamingAMFChannel / StreamingAMFEndpoint` which use the https protocol.
Plain Text data (AMFX - an XML Format)	`HTTPChannel / HTTPEndpoint` Like `AMFChannel`, but transmits data in plain text. Not recommended for production sites.
	`SecureHTTPChannel / SecureHTTPEndpoint` Like `HTTPChannel / AMFEndpoint` but uses `https` protocol.
Plain Text data (AMFX - an XML Format)	`StreamingHTTPChannel / StreamingHTTPEndpoint` Like `StreamingAMFChannel / StreamingAMFEndpoint` but transmits data in plain text instead of the binary **AMF** format. Not recommended for production sites.
	`SecureStreamingHTTPChannel / SecureStreamingHTTPEndpoint` Like `SecureStreamingAMFChannel / SecureStreamingAMFEndpoint`, but transmits data in plain text.

All of the above servlet-based endpoints have Java NIO-based counterparts, as shown in the next table. The RTMP protocol is also NIO-based. All of the NIO-based server endpoints support many threads per connection because they do not use blocking I/O calls. Unlike HTTP-based protocols, RTMP is full-duplex, requiring half as many channels for two-way traffic. RTMP also immediately informs the server of client disconnects, allowing the server to release resources without waiting for the client to time out. BlazeDS does not support NIO-based server endpoints, just LCDS.

Notice that server endpoints might be implemented using NIO or blocking I/O, however the client channel is the same regardless of the type of server endpoint – with the exception that RTMP channels must use matching RTMP endpoints.

NIO-based server-side endpoints and their client channels	
Binary AMF data	`AMFChannel / NIOAMFEndpoint` Used for RPC when default properties are not overridden; used for messaging and data management service when properties are configured. Transmits data in the binary **AMF** format. Usually uses the `http` protocol.
	`SecureAMFChannel / SecureNIOAMFEndpoint` Secure subclasses of `AMFChannel / NIOAMFEndpoint`. Uses the https protocol.
	`StreamingAMFChannel / StreamingNIOAMFEndpoint` Uses the HTTP 1.1 'chunked' server push model instead of polling for data from the server. An internal HTTP connection to the server is held open so the server can stream data to the client with little overhead. Cannot be compressed or proxied.
	`SecureStreamingAMFChannel / SecureStreamingNIOAMFEndpoint` Subclass of StreamingAMFChannel which uses the https protocol.
	`RTMPChannel / RTMPEndpoint` Used for messaging; does not route well. Uses the rtmp protocol, which provides best support for real-time applications.
	`SecureRTMPChannel / SecureRTMPEndpoint` Used for secure messaging; does not route well. Uses the `rtmps` protocol.
Plain Text data (AMFX - an XML Format)	`HTTPChannel / NIOHTTPEndpoint` Like `AMFChannel / NIOAMFEndpoint`, but transmits data in plain text. Not recommended for production sites.
	`SecureHTTPChannel / SecureNIOHTTPEndpoint` Like `HTTPChannel` but uses `https` protocol.
	`StreamingHTTPChannel / StreamingNIOHTTPEndpoint` Like `StreamingAMFChannel` but transmits data in plain text instead of the binary **AMF** format. Not recommended for production sites.
	`SecureStreamingHTTPChannel / SecureStreamingNIOHTTPEndpoint` Like `SecureStreamingAMFChannel / SecureStreamingNIOAMFEndpoint`, but transmits data in plain text.

Channel Sets

Channel definitions are combined into channel sets, and these are assigned to destinations; default channel sets can also be defined for each service class and the overall web application. The service classes common to BlazeDS and LCDS are `RemotingService`, `MessageService` and `HTTPProxyService`. LCDS adds the `DataService`. Channel sets provide for graceful and transparent fallback should a channel not be available for a specific client. Each time a client connects to a destination on a server, the destination's channels are tried, in the order

listed in the `channels` element. Should the first channel specify a protocol or endpoint that is not available, or fails during usage, the other channels are tried in sequence.

If the destination specifies a `channels` element, only those channels are tried when a client connects to that destination. Service classes can each have their own default channel set, defined by the service's `default-channels` element; if specified, those channels are tried in sequence if the service has not specified a `channels` element. An application-level default channel set may be specified in the top-level `default-channels` element, and its channels are tried in sequence if the service class and the destination do not specify `default-channels` or `channels` elements, respectively.

For example, messaging applications need an ongoing dialog, such as provided by polling and streaming protocols. All of the channel types described in the previous section could work, if they were suitably configured. For the sake of this example, let us assume that a messaging application needs a secure transport. One might opt to support the following channel set. This `ChannelSet` falls back from a secure RTMP channel to a secure NIO streaming channel, then a secure streaming AMF channel, then to a secure AMF channel with polling enabled to work around network components such as web server connectors, HTTP proxies, or reverse proxies that could buffer chunked responses incorrectly. Recall that a channel also defines an associated server endpoint; the table shows the channel and endpoint classes for each channel in the channel set.

SecureRTMPChannel / SecureRTMPEndpoint	Provides best performance; guarantees the order of messages; scales well. RTMP is bidirectional, so half as many channels are required compared to HTTP-based channels. RTMP also immediately notifies the server when a client disconnects, so resources can be released. Firewalls or proxies might block RTMP.
SecureStreamingAMFChannel / SecureStreamingNIOAMFEndpoint	Good performance; scales well; difficult to configure if you want to use the same port as HTTP. Nonstandard ports might cause firewall or proxies to block it.
SecureStreamingAMFChannel / SecureStreamingAMFEndpoint	Routes well; good performance; does not scale well.
SecureAMFChannel / SecureAMFEndpoint configured for piggybacking	Routes well; moderate performance; does not scale well.

Channel Designer

LCDS provides sample channel and endpoint definitions for common configurations in `{LCDS}/resources/config/services-config.xml`.

The free Flex Data Services Channel Designer provided with the book is a handy tool for visually designing client channels and server endpoints. The Channel Designer is intended to clarify the options and write the XML for the channels that your application needs.

Java to ActionScript Type Mapping

Data exchanged between Flex client and Java server must be converted between the ActionScript representation and the Java representation. Regardless of the transport protocol used, the same rules are used for the data conversion. The following two tables describe the general rules for the conversions.

Java Type	ActionScript (AMF3/AMFX)
`java.lang.String`	String
`java.lang.Boolean`	Boolean

`java.lang.Integer`
`java.lang.Short`
`java.lang.Byte`
	int

`java.lang.Double`
`java.lang.Long`
`java.lang.Float`
	Number

`java.util.Calendar` `java.util.Date`	Date
`java.lang.Character` `java.lang.Character[]`	String
`java.lang.Byte[]`	ByteArray
`java.util.Collection`	ArrayCollection
`java.lang.Object[]`	Array
`java.util.Map` `java.util.Dictionary`	Object
`java.lang.Object`	Typed Object
`null`	null

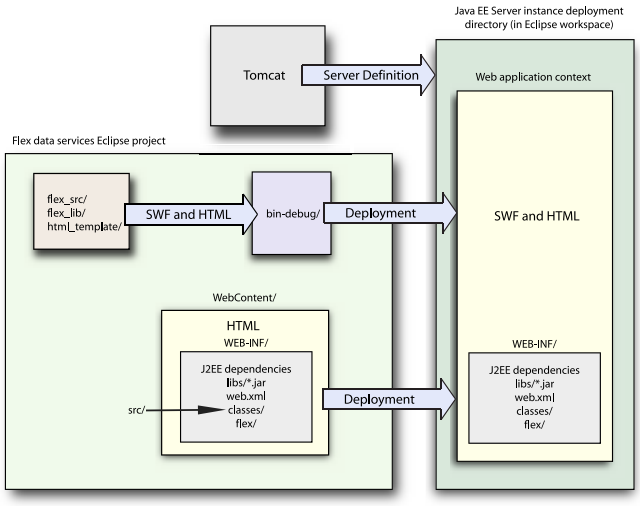
ActionScript to Java Type Mappings

ActionScript type (AMF3)	Java Interface	Supported Java type binding
Array (dense)	`java.util.List`	`java.util.Collection`, `Object[ ]` (native array) If the type is an interface, it is mapped to the following interface implementations • `List` becomes `ArrayList` • `SortedSet` becomes `TreeSet` • `Set` becomes `HashSet` • `Collection` becomes `ArrayList` A new instance of a custom Collection implementation is bound to that type.
Array (sparse)	`java.util.Map`	`java.util.Collection`, native array
String containing "true" or "false"	`java.lang.Boolean`	`Boolean, boolean, String`
flash.utils.ByteArray	`byte []`	
flash.utils.IExternalizable	`java.io.Externalizable`	
Date	`java.util.Date` (formatted for Coordinated Universal Time (UTC))	`java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Time, java.sql.Date`
int/uint	`java.lang.Integer`	`java.lang.Byte, java.lang.Double, java.lang.Float, java.lang.Long, java.lang.Short, java.math.BigDecimal, String,` primitive types of byte, double, float, long, and short
null	`null`	primitives
Number	`java.lang.Double`	`java.lang.Byte, java.lang.Double, java.lang.Float, java.lang.Long, java.lang.Short, java.math.BigDecimal,` String, 0 (zero) if null is sent, primitive types of byte, double, float, long, and short
Object (generic)	`java.util.Map`	If a Map interface is specified, Flex Data Services creates a new java.util.HashMap for java.util.Map and a new java.util.TreeMap for java.util.SortedMap.
String	`java.lang.String`	`java.lang.String, java.lang.Boolean, java.lang.Number`
typed Object	typed `Object` when you use [`RemoteClass`]	typed `Object`
undefined	`null`	`null` for `Object`, default values for primitives
XML	`org.w3c.dom.Document`	`org.w3c.dom.Document`

Eclipse Projects

Eclipse WST is used to develop LCDS server-side projects. You can create integrated Flex / Java Eclipse projects or separate Flex and Java projects. Integrated projects are much easier to work with. Use the Flex Builder plugin with Eclipse Ganymede SR2 for best results. This project also requires Tomcat 6 to be installed. You can use the version of Tomcat provided with LCDS, but that has been tweaked to make the demo programs work. Just so there is no magic, download a fresh copy of Tomcat 6 core – we will refer to the directory that you place Tomcat into as `{CATALINA_HOME}`.

The following diagram shows how client-side Flex ActionScript and MXML source code are built into a SWF with an HTML wrapper, and then deployed to a Tomcat instance. The server-side Java code is compiled into a staging area called `WebContent`, which is also deployed to the Tomcat instance.



With an integrated Flex/Java project you can debug both client and server simultaneously. It is cool to see the client and server debug stacks next to each other.

You can start make a combined Flex / Java Eclipse project as follows:

1. In Eclipse, press Control / Command N to open the New Wizard.
2. Open the Flex Builder folder, select Flex Project and press the **Next** > button.
3. Give your project a name and change the **Application server type** to J2EE. Uncheck **Use remote object access service**. Leave the other options at their default settings. Press the **Next** > button.
4. Select Apache Tomcat 6.0 as the value for the Target runtime. Change the Context root to a single short word. Leave the other options at their default settings. Press the **Finish** button.
5. Copy `{LCDS}/tomcat/webapps/lcds/WEB-INF/web.xml` to `WebContent/WEB-INF`.
6. Copy `{LCDS}/resources/lib/*` to `WebContent/WEB-INF/lib`
7. Ensure that **Publish module contexts to separate XML files** is selected.
8. Select the Servers tab at the bottom of the screen and double-click on the server you just created.
9. Ensure that Publish module contexts to separate XML files is selected.

(Optional) If you are not using dynamically defined channels and destinations and are instead defining them in the WEB-INF/flex XML files:

1. Copy `{LCDS}/resources/config/*` to `WebContent/WEB-INF/flex` and edit as required. You may also want to include these files so that you can customize the LCDS configuration.
2. Add a Flex Compiler argument pointing at the XML file. To do this, open the **Properties / Flex Compiler** dialog and add the following to **Additional compiler arguments**:

```
-services ../WebContent/WEB-INF/flex/services-config.xml
```

Sample Project

I made a Flex Builder project using the source code for the Data Management Service (DMS) Test Drive sample program provided with LCDS, and added all the dependencies. You should be able to import this project into Eclipse Ganymede SR2 with the Flex Builder Plugin and compile it without any editing. To import the DMS Test Drive Eclipse project:

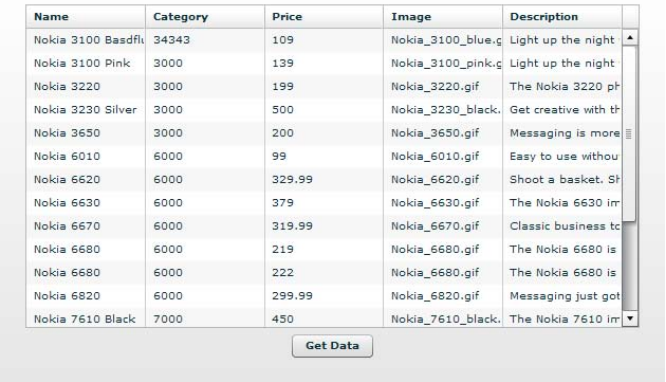
1. Start Eclipse
2. Select **File / Import... / General / Existing Projects into Workspace**
3. Browse to the directory where you unzipped the Test Drive project into.
4. Press TAB and notice that the project is now listed and selected.
5. Click on the **Finish** button.

The LCDS libraries were not written using Java generics. I converted as much of the Java source code as possible to use generics. Subclasses of the LCDS types could not be genericized, so I added the `@SuppressWarnings("unchecked")` annotation where required. I also added `@Override` where required. No other changes were made to Adobe's source code other than cleaning up some formatting issues and removing unused cruft. I simplified the data services configuration files in `WebContent/WEB-INF/flex` so that undefined references for the other sample programs delivered with LCDS were removed.

Before you can run the project, the web application must be associated with a Tomcat instance. To do that, open the **Servers** panel at the bottom of the Java EE perspective and right-click on a Tomcat instance. Select the **Add and Remove Projects...** menu item, then press the **Add All >>** button.

Next you need to start the HSQLDB database. For Windows, double-click on `{LCDS}/sampledb/startdb.bat`; for Linux and Mac, double-click on `{LCDS}/sampledb/startdb.sh`.

You can now start the web application by clicking on the server instance and then clicking on bug icon at the right side of the Servers panel. By default, the web application is automatically built and deployed prior to starting Tomcat. Point your browser to http://localhost:8080/dmsTestdrive/dmsTestdrive.html in two different web browser windows. Here is what you should see:



If you have problems, increase LCDS logging verbosity by editing WebContent/WEB-INF/flex/services-config.xml and changing the level from "Warn" to "Debug" in the following line, then restart the server:

```
<target class="flex.messaging.log.ConsoleTarget" level="Warn">
```

If you *really* need more logging verbosity, add the following filter pattern and restart the server:

```
<pattern>*</pattern>
```

Click on the **Get Data** button in each browser window. Double-click on a cell in one browser, change a value and notice that the value is propagated to the application running in the other browser. Because this simple application has no form validation, an error will be thrown if you enter a non-numeric value into a cell.

How the program works

The LCDS configuration is usually the best place to start when you want to understand how a program built with LCDS works. WebContent/WEB-INF/flex/services-config.xml defines a channel called my-rtmp, and within the channel definition, an endpoint that uses the RTMP protocol to push messages between client and server:

```
<channel-definition id="my-rtmp" class="mx.messaging.channels.
RTMPChannel">
    <endpoint url="rtmp://{server.name}:2037" class="flex.
messaging.endpoints.RTMPEndpoint"/>
    <properties>
    <idle-timeout-minutes>20</idle-timeout-minutes>
    </properties>
</channel-definition>
```

The `data-management-config.xml` file contains the following definition for the default channels by which the client and server communicate using data management services:

```
<default-channels>
   <channel ref="my-rtmp" />
</default-channels>
```

The same configuration file also defines a destination called `inventory`.

```
<destination id="inventory">
    <properties>
        <use-transactions>false</use-transactions>
        <source>flex.samples.product.ProductAssembler</source>
        <scope>application</scope>
        <metadata>
            <identity property="productId"/>
        </metadata>
        <network>
            <paging enabled="false" pageSize="10" />
        </network>
    </properties>
</destination>
```

The destination's `source` property identifies the server-side class that implements CRUD operations. Several default properties are implied, including `auto-sync-enabled` (set true), which automatically propagates changes between client and server in both directions.

The public methods found in `ProductAssembler.java` are:

```
public Collection fill(List fillArgs);
public Object getItem(Map identity);
public void createItem(Object item);
public void updateItem(Object newVersion, Object prevVersion,
List changes);
public void deleteItem(Object item);
```

LCDS creates ActionScript equivalents for each of the public methods in `ProductAssembler`. When called from ActionScript, the `fill()` method populates a client-side Flex `ArrayCollection` with data items from the server. The server-side `fill()` method calls `ProductService.getProduct()`, which runs a JDBC query and the result is returned to the client. `ProductService` references `ConnectionHelper` to make the JDBC connections, and throws `DAOException` if a problem occurs. The other methods in `ProductAssembler.java` are not used by this simple application, because it does not fully implement all CRUD functionality.

Shifting our attention to the Flex client for a moment, the following line in `dmsTestdrive.mxml` specifies that the `inventory` destination should be used by a `DataService` called `ds` to transport CRUD data by the following line:

```
<mx:DataService id="ds" destination="inventory" />
```

The next line defines an `ArrayCollection` called products to act as a client-side data buffer for the data streaming between client and server:

```
<mx:ArrayCollection id="products" />
```

The next line defines an anonymous ActionScript variable of type Product. The variable is anonymous because its value is unimportant. The side effect of this line is that the ActionScript to Java mapping of the Product classes defined in `Product.as` and `Product.java` is incorporated into the Flex program:

```
<local:Product />
```

The data streaming between client and server is of type `Product`. On the client, `Product` is defined as follows:

```
package {
    [Managed]
    [RemoteClass(alias="flex.samples.product.Product")]
    public class Product {
        public var productId:int;
        public var name:String;
        public var description:String;
        public var image:String;
        public var category:String;
        public var price:Number;
        public var qtyInStock:int;
    }
}
```

LiveCycle DS supports the concept of managed classes for projects that must transmit complex trees of hierarchical data between subscribed clients and a server. The `[Managed]` class-level annotation signifies that the entire object graph should not be retransmitted to all subscribers when a property of a subordinate object in the graph changes value. Instead, the responsibility for keeping the remote object in sync is the responsibility of the subordinate class. In this case, all of the public properties are primitives, so `[Managed]` is not really doing anything in this regard. The annotation is required for

Data Management Services to manage the data.
The `[RemoteClass]` class-level annotation provides strong typing information for serialization and deserialization operations. Without the annotation, LCDS does not have knowledge of the public properties of the class, and the `update()` method would throw an error. The mapped Java class has the same public properties as the equivalent ActionScript class.

There is one more important line in `dmsTestdrive.mxml`, which defines the Flex client's button as follows:

```
<mx:Button label="Get Data" click="ds.fill(products)" />
```

The click handler, defined inline in ActionScript, causes the client-side version of the `fill()` method to be invoked when the user clicks on the button. The `ArrayCollection` called products is passed to the `fill()` method so it can be used as a data transfer buffer. This method call is forwarded to the server, where it is executed by the Java version of the `fill()` method, defined in `ProductAssembler`. When the server returns the data to the Flex client, the `DataGrid` is automatically updated by the `ArrayCollection` because of data binding, denoted by curly braces:

```
<mx:DataGrid dataProvider="{products}" editable="true"
width="100%" height="100%">
```

You can see this in action by placing breakpoints on `update()` method in `ProductService.java`.

Licensing

LCDS is available under several licenses, including a free license. The software provided is the same for all licenses.
The LCDS Trial Developer License is a non-expiring trial for development on one or more CPUs. This version lets developers create RIAs with rich data services capabilities. If your application requires more than one CPU with two cores for production deployment, or if you want licenses for Q/A and staging, contact your Adobe sales representative or channel partner to purchase the appropriate runtime license.
The LCDS Single-CPU License lets you run an application in a commercial, production environment on a single machine with one CPU with up to two cores. This version is ideal for use in small to medium-scale production applications and proof-of-concept projects.

Q/A and staging server licenses are available separately and are not free.

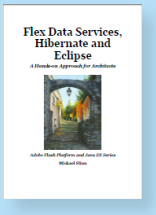
ABOUT THE AUTHORS

Mike Slinn is a software contractor specializing in Adobe Flex and Java. With over three decades of hands-on experience, Mike focuses on value creation, including technology, methodology and business drivers. He has provided litigation support for contractual and patent disputes and US Federal court has recognized him as a software expert. A graduate of Carleton University in Ottawa, Canada, Mike received a B. Eng. in Electronics in 1979 and became P. Eng. in B.C. in 1983. Mike has lived in Silicon Valley since 1996, and now resides in Half Moon Bay, CA with his wife and their ever-demanding parrot.

Resources

Flex Data Services Channel Designer
Flex Builder Plugin
LCDS Home Page
LCDS Download
LCDS Developer Guide
LCDS ASDoc
LCDS Javadoc
Adobe® Flash® Platform

RECOMMENDED BOOK



Flex Data Services, Hibernate and Eclipse is intended for experienced architects and programmers who would like to design and implement non-trivial applications using the Adobe Flash Platform using a Java EE back end. This book explains how to design, build and test the Flex Data Services and the server-side Java stack of an application built with the Adobe Flash Platform.

BUY NOW
http://www.slinnbooks.com/books/serverSide/index.shtml

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: 978-1-934238-88-2
ISBN-10: 1-934238-88-0
50795
9 781934 238882
$7.95