Create Flex 4 and Java Web applications

Discover object remoting

Arron Ferguson

April 27, 2010

Rich Internet Applications (RIAs) often utilize different bundled technologies. Choosing the right grouping of technologies can expedite development times and provide users a complete and rich Internet experience. Discover how to use Java™ EE platform components on the server side, the Adobe® Flex™ platform on the client side, and the MySQL® database server for storage persistence.

Current Web technologies have an increased demand placed on them. They must be able to manage user accounts, upload content, and stream video. This demand requires RIA developers to seek technologies that streamline development workflow while at the same time providing commonly sought-after features. The challenge to developers is picking the right set of technologies to provide these services.

Frequently used acronyms

- AMF: Action Message Format
- API: Application programming interface
- CSS: Cascading stylesheet
- GUI: Graphical user interface
- HTTP: Hypertext Transfer Protocol
- JAR: Java archive
- POJO: Plain old Java object
- RIA: Rich Internet Application
- RPC: Remote procedure call
- SDK: Software development kit
- SQL: Structured Query Language
- UI: User interface
- WAR: Web archive
- XML: Extensible Markup Language

Adobe Flex is a client-side technology that provides developers with a rich set of API calls for creating GUIs, drawing graphics, playing and streaming media, and connecting to Web services. On the server side, Java technology provides abilities such as connecting to relational database management systems (RDBMSs), multi-threaded processing of service requests, and optimum scaling with increased demand. Using these two technologies together offers a powerful technology stack that satisfies the demand of RIA applications.
This article demonstrates how to write a simple, yet powerful RIA that utilizes Flex for the client, Java technology for the server, and MySQL for the back-end database.

The sample application

The sample application (available from the Download section below) provides a rich UI that supports creating, reading, updating, and deleting (CRUD) contact information through an Adobe Flash® (SWF) application. This three-tiered Web architecture is depicted in Figure 1, where the client is represented by the SWF file embedded within a Web page, the server application is run within a Java servlet container (in this case, Apache Tomcat), and the database is MySQL. Combined, these three tiers create a power-distributed application.

Figure 1. The Contacts application

For communication between the Flash application and the Java servlet container, the Adobe BlazeDS framework provides object remoting—a form of RPC that allows Adobe ActionScript™ objects to call Java objects and vice versa. Communication between the Java server application and the relational database is handled by the Hibernate Object Relational Mapping (ORM) framework. Hibernate allows Java objects to be transformed to SQL code and vice versa.

The application: server tier

The first step is to create a Java class that encompasses the information required to store contact information. The sample application contains a simple model with basic information. The attributes and the data types required for Contact objects are:

- String emailAddress
- String firstName
- long id
- String lastName
- String phoneNumber
- long serialVersionUID
Annotating the business object

The Java Contact class is considered a POJO (Plain old Java object) that acts as a business object, meaning that it represents business domain characteristics and behaviors. The data inside Contact objects needs to be persisted to the database. The solution is to use an ORM framework such as Hibernate, which performs much of the work in mapping objects to records within database tables and back again. If Java Persistence API (JPA) annotations are used, very little coding is required to fulfill ORM. Listing 1 demonstrates the annotated Java class Contact.

Listing 1. The Java Contact class

```java
package bcit.contacts;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name="contact")
@NamedQueries( {
    @NamedQuery(name = "contact.findAll", query = "from Contact"),
    @NamedQuery(name = "contact.getById", query = "select c from Contact c where c.id = :id")
} )
public class Contact {

    private static final long serialVersionUID = 123456789L;

    public Contact() {
        firstName = "N/A";
        lastName = "N/A";
        emailAddress = "N/A";
        phoneNumber = "N/A";
    }

    public Contact(String first, String last, String email, String number) {
        firstName = first;
        lastName = last;
        emailAddress = email;
    }

    public String getEmailAddress() {
        return emailAddress;
    }

    public String getFirstName() {
        return firstName;
    }

    public long getId() {
        return id;
    }

    public String getLastName() {
        return lastName;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setEmailAddress(String address) {
        emailAddress = address;
    }

    public void setFirstName(String first) {
        firstName = first;
    }

    public void setId(long newId) {
        id = newId;
    }

    public void setLastName(String last) {
        lastName = last;
    }

    public void setPhoneNumber(String number) {
        phoneNumber = number;
    }

    public String toString() {
        return "Contact: " + firstName + " " + lastName + " Email: " + emailAddress + " Phone: " + phoneNumber;
    }
}
```
phoneNumber = number;
}

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "id", nullable = false, updatable=false)
private long id;

@Column(name = "lastName", nullable = false, unique = false)
private String lastName;

@Column(name = "firstName", nullable = false, unique = false)
private String firstName;

@Column(name = "emailAddress", nullable = false, unique = false)
private String emailAddress;

@Column(name = "phoneNumber", nullable = false, unique = false)
private String phoneNumber;

public void setPhoneNumber(String number) { phoneNumber = number; }
public String getPhoneNumber() { return phoneNumber; }
public String getEmailAddress() { return emailAddress; }
public void setEmailAddress(String address) { emailAddress = address; }
public String getFirstName() { return firstName; }
public void setFirstName(String first) { firstName = first; }
public String getLastName() { return lastName; }
public void setLastName(String last) { lastName = last; }
public long getId() { return id; }
public void setId(long newId) { id = newId; }

@Override
public String toString() {
    return id + " " + firstName + " " + lastName + " " + emailAddress + " " + phoneNumber;
}

The class is simple, but there is a lot going on with annotations:

- **@Column**: Labels the property as a column within the database, with choices including the name of the column, whether it's unique, and whether a column is nullable
- **@Entity**: Declares the class as an entity bean, meaning that it is a POJO slated for persistence
- **@GeneratedValue**: Specifies the strategy for generating primary keys; choices are AUTO, IDENTITY, SEQUENCE, and TABLE
- **@Id**: States that the property is the unique identifier (that is, primary key) for each Java object
- **@NamedQuery**: Lists a group of named queries
- **@NamedQuery**: Declares a predefined query as a string literal that can later be referenced for execution
- **@Table**: Designates the Java class as a table within the database
Each time an in-memory Java object is required to be persisted, Hibernate transforms the state information of any Java objects into an SQL update. Likewise, SQL statements with result sets are used to populate Java objects. The result is that all objects can be saved as records within the database, and all records can be retrieved and transformed back into Java objects.

The annotations inform Hibernate what within a class should be considered persistent. But they are only part of the picture.

**The business service: database connectivity**

A service class is required to perform the calls to Hibernate in order to execute ORM. **Listing 2** displays the `ContactsService` class, which acts as the application service.

**Listing 2. The ContactsService class**

```java
public class ContactsService {
    private static Logger logger = Logger.getLogger(ContactsService.class);
    private static final String PERSISTENCE_UNIT = "contacts";
    private static EntityManagerFactory emf = null;
    static {
        logger.info("LOADING CONTACTSSERVICE CLASS.");
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
    }
    public ContactsService() {
        super();
    }
    public void addContact(Contact c) {
        if(c == null) {
            return;
        }
        EntityManager em = emf.createEntityManager();
        logger.info("PERSISTENCE ENTITYMANAGER ACQUIRED.");
        logger.info("ABOUT TO ADD CONTACT: fName: " + c.getFirstName() + ", lName: " + c.getLastName() + ", email:" + c.getEmailAddress() + ", phone: " + c.getPhoneNumber());
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            em.merge(c);
            tx.commit();
        } catch (Exception e) {
            logger.error("CONTACT APP PERSISTING ERROR: " + e.getMessage());
            tx.rollback();
        } finally {
            logger.info("CONTACT APP CLOSING ENTITY MANAGER.");
            em.close();
        }
    }
    public void editContact(Contact c) {
        logger.info("CONTACT TO UPDATE: " + c);
        addContact(c);
    }
}
```
public void deleteContact(Long id) {
    logger.info("ABOUT TO DELETE CONTACT");
    EntityManager em = emf.createEntityManager();
    logger.info("PERSISTENCE ENTITYMANAGER ACQUIRED.");

    Query contactByIdQuery = em.createNamedQuery("contact.getById");
    contactByIdQuery.setParameter("id", id);
    Contact c = (Contact) contactByIdQuery.getSingleResult();
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        em.remove(c);
        tx.commit();
    } catch (Exception e) {
        logger.error("CONTACT APP PERSISTING ERROR: " + e.getMessage());
        tx.rollback();
    } finally {
        logger.info("CONTACT APP CLOSING ENTITY MANAGER.");
        em.close();
    }
}

public List<Contact> getContacts() {
    logger.info("ABOUT TO RETRIEVE CONTACTS");
    EntityManager em = emf.createEntityManager();
    logger.info("PERSISTENCE ENTITYMANAGER ACQUIRED.");

    Query findAllContactsQuery =
        em.createNamedQuery("contact.findAll");
    List<Contact> contacts = findAllContactsQuery.getResultList();
    if (contacts != null) {
        logger.debug("CONTACT APP RETRIEVED: " + contacts.size() + " CONTACT(S)");
    }
    return contacts;
}
}

Each method gains a reference to an EntityManager, which represents the in-memory cache. Caching is a powerful feature that ensures efficiency, because sending and receiving data from a database can be an expensive operation. You have only to ensure that every cache created is committed to the database or rolled back if not wanted.

In JPA, the term given to the cache is persistence context, and it is represented by the EntityManager class. Each persistence context manages a set of entities, which are Java objects that have been annotated with the @Entity annotation. The EntityManagerFactory class represents a persistence unit, which is responsible for configuring connections to a data store (for example, a relational database), managing entity types (that is, all the classes within a given context that you need to map to a data store), and finally providing instances of a persistence context (that is, an EntityManager).

Try a no-charge version of DB2® Express 9 database server
DB2 Express-C is designed to be up and running in minutes, is easy to use and embed, includes self-management features, and embodies all of the core capabilities of DB2 for
Although the process of creating a persistence context is inexpensive time-wise, the process of creating a persistence unit is costly. Setting up connectivity to a data store, finding all classes annotated as entities, and configuring the persistence logic to bind these classes to entities in the data store is not a quick operation. Therefore, you should create an `EntityManagerFactory` instance once at application start-up. As for persistence contexts, take care to ensure that an `EntityManager` is destroyed before another one is created. Another important rule to follow is the `entitymanager-per-request` pattern. This pattern groups database calls (for example, requests and updates) so that they can all be sent at once. Doing so ensures full advantage of JPA's caching mechanism.

The next requirement is the client side.

**The application: client tier**

The Flex framework allows you to create applications that users can play in Adobe Flash Player. Flex consists of:

- A declarative XML UI language known as **MXML**
- The ActionScript programming language
- Run time libraries for creating UIs, Web connectivity, and many other features
- Developer tools for compiling applications into SWF files

The client application referenced in this article uses Flex version 4. Before approaching the client-side application, it is important to understand how Flex applications are created and how they exist as executables within Flash Player.

First, you can create applications using a combination of MXML markup and ActionScript code. A common workflow is to create much of the GUI (presentation) using the MXML format, and then use ActionScript code for event handling and business logic. Because both MXML and ActionScript are text-based, a standard text editor and the Flex SDK are all you need to create Flash applications.

Second, once you've written a Flex application, you compile the code using the MXML compiler. The MXML compiler creates SWF files that can then be run inside a Web browser (via the Flash Player browser plug-in).

Finally, Flash applications run within the ActionScript Virtual Machine 2 (AVM2), which uses the timeline paradigm. This paradigm breaks execution up into frames—much like a movie. You specify the number of frames per second in a Flash application at compile time. Additionally, Flash Player breaks execution up into the following ordered tasks:

- Flash Player events such as the timer and mouse events
- User code
• Pre-rendering logic, where Flash Player attempts to determine whether the GUI needs to be updated because of data value changes
• User code that was bound to the data value changes
• Flash Player rendering

If there are few frames per second to render, then much of the user code can be executed. If, however, the frame rate is high (for example, 60 frames per second), then Flash Player may not be able to execute much of the user code, because the user code may take more time than what can be granted. It's important to keep this in mind when writing for Flash Player.

**MXML**

MXML is a powerful declarative XML format that helps:

• Minimize the amount of code required to build a GUI because of the declarative nature of the XML format
• Reduce the complexity of GUI code by allowing for a clear separation of presentation logic and interaction logic
• Promote use of design patterns when approaching software development

Listing 3 displays the MXML Application class.

**Listing 3. The ContactsApp class**

```xml
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:contact="bcit.contacts.*" creationComplete="initPage();" layout="vertical" frameRate="30" pageTitle="Contacts Example" horizontalAlign="center" verticalAlign="middle" backgroundColor="#A9C0E7">

<mx:Style>
  .mainBoxStyle {
    borderStyle: solid;
    paddingTop: 5px;
    paddingBottom: 5px;
    paddingLeft: 5px;
    paddingRight: 5px;
  }

  .textMessages {
    fontWeight: bold;
  }
</mx:Style>

<mx:RemoteObject id="remotingService" showBusyCursor="false" destination="contacts" fault="handleFault(event);" result="handleResult(event);"/>

<mx:Script>
<![CDATA[

import mx.rpc.events.FaultEvent;
import mx.rpc.events.ResultEvent;
import mx.collections:ArrayCollection;
import bcit.contacts.dto.Contact;

[Bindable]
private var contacts:ArrayCollection = new ArrayCollection();
]]>
</mx:Script>
```
// For more on the Bindable metadata tag, see the devguide_flex3.pdf
// document, page 1249 (1257 in PDF page numbering)

[Bindable]
private var message:String = "Status: Ready";

private var contact:Contact;

public function setControlBarValid(valid:Boolean):void {
    if(valid) {
        // if the selected item is -1, then no item is selected but at
        // the same time the fields are valid which means the user chose
        // to add a contact, not update one
        if(contactsDataGrid.selectedIndex == -1) {
            createButton.enabled = valid;
        } else {
            editButton.enabled = valid;
        }
    } else {
        // else nothing is valid
        createButton.enabled = false;
        editButton.enabled = false;
    }
}

private function initPage():void {
    editContactForm.setApp(this);
    contact = new Contact();
    getAllContacts();
    resetPage();
}

private function createContact():void {
    contact = editContactForm.getContact();
    remotingService.addContact(contact);
    message = "Status: Contact Added";
    getAllContacts();
}

private function editContact():void {
    var id:Number = contact.id;
    contact = editContactForm.getContact();
    contact.id = id;
    remotingService.editContact(contact);
    message = "Status: Contact Edited";
    getAllContacts();
}

private function deleteContact():void {
    if(contactsDataGrid.selectedItem != null) {
        var c:Contact = contactsDataGrid.selectedItem as Contact;
        // no sense in sending the whole contact - just send the id
        // to cut down on bandwidth
        remotingService.deleteContact(c.id);
        message = "Status: Contact Deleted";
    }
    getAllContacts();
}

private function getAllContacts():void {
    loadButton.enabled = false;
    remotingService.getContacts();
    loadButton.enabled = true;
    resetPage();
}

private function populateFormWithContact():void {
contact = contactsDataGrid.selectedItem as Contact;
editContactForm.setContact(contact);
editButton.enabled = true;
deleteButton.enabled = true;
}

private function resetPage():void {
    editContactForm.clearForm();
    contact = new Contact();
    createButton.enabled = false;
    editButton.enabled = false;
    deleteButton.enabled = false;
    contactsDataGrid.selectedIndex = -1;
}

private function handleFault(e:FaultEvent):void {
    message = "Status: Error"
    + "\nFault code: " + e.fault.faultCode
    + "\nFault detail: " + e.fault.faultDetail
    + "\nFault string: " + e.fault.faultString;
}

private function handleResult(e:ResultEvent):void {
    // can get the results by accessing e.result property
    // mx.controls.Alert.show(e.toString());
    contacts = e.result as ArrayCollection;
    var number:int = contacts.length;
    //if(number == 1) {
    //    message = "Status: Retrieved 1 contact";
    //} else {
    //    message = "Status: Retrieved " + contacts.length + " contacts";
    //}
}

]]>
</mx:Script>

<mx:VBox styleName="mainBoxStyle">
    <mx:Text id="titleText" text="Single click to select a contact"/>

    <contact:ContactsDataGrid id="contactsDataGrid" dataProvider="{contacts}"
        itemClick="populateFormWithContact();"
        doubleClick="populateFormWithContact();"/>

    <contact:EditContactForm id="editContactForm"/>

    <mx:ControlBar horizontalAlign="center">
        <mx:Button label="List" id="loadButton" click="getAllContacts()"
            toolTip="Retrieve contacts from the server"/>
        <mx:Button label="Add" id="createButton" click="createContact()"
            toolTip="Create a new contact"/>
        <mx:Button label="Update" id="editButton" click="editContact()"
            toolTip="Edit a selected contact"/>
        <mx:Button label="Delete" id="deleteButton" click="deleteContact()"
            toolTip="Delete a selected contact"/>
        <mx:Button label="Clear Form" id="clearButton" click="resetPage()"
            toolTip="Clear the form"/>
    </mx:ControlBar>

    <mx:TextArea text="{message}" styleName="textMessages" wordWrap="true"
        verticalScrollPolicy="auto" horizontalScrollPolicy="off" editable="false"
        width="100%"/>

</mx:VBox>
</mx:Application>
Here are a few more points pertaining to Listing 3:

- The root element of an MXML document is a subclass of the Application class.
- The mx:Style element allows for CSS properties to define local styling to UI components. Styling can be done using local style definitions (as in Listing 3), references to external style sheets, inlining styles within the components, and using the setStyle method in ActionScript.
- The RemoteObject class represents an HTTP service object that performs remoting operations with a server.
- The mx:Script element includes ActionScript code blocks in a CDATA section.
- There is one layout (that is, the VBox class).
- Each time a UI component is declared in the application (for example, a TextArea), an instance variable is generated that can be referenced later within the application using the component's id attribute.
- Data binding is performed using braces (for example, the TextArea element's text attribute is bound to the ActionScript message instance variable).

**ActionScript**

While MXML defines the GUI, ActionScript offers the behavior for handling events, binding data (through the [Bindable] metadata tag), and the ability to call a remote service. In Listing 3, the methods createContact, editContact, deleteContact, and getAllContacts all call remote methods on the server side. When a remote method is called, ActionScript is offered the opportunity to handle the result and any fault by declaring callback functions. In Listing 3, the handleResult function receives the result as an object and casts it to an ArrayCollection. BlazeDS converted the List into an ArrayCollection on the server side.

Listing 4 presents the ActionScript class Contact, which you create to represent contact objects on the Flash side.

### Listing 4. The ActionScript Contact class

```
package bcit.contacts.dto {

[RemoteClass(alias="bcit.contacts.Contact")]
public class Contact {

    public function Contact() { id = -1; }

    public var id:Number;
    public var lastName:String;
    public var firstName:String;
    public var emailAddress:String;
    public var phoneNumber:String;

    public function toString():String {
        return id + ", " + firstName + " " + lastName + " " + emailAddress + " " + phoneNumber;
    }
}
}
```

These ActionScript objects are sent to the server side, where BlazeDS performs its magic and converts the ActionScript objects into Java objects. The ActionScript Contact class is considered a Data Transfer Object (DTO).
Configuring the application

The application also relies on configuration files that state setting specifics for the server. The two main areas of configuration within this application are Hibernate and BlazeDS.

Configuring Hibernate

You can configure Hibernate by using the standard JPA configuration file, persistence.xml, which is shown in Listing 5.

Listing 5. A subset of the persistence.xml configuration file

```xml
< persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  < persistence-unit name="contacts" transaction-type="RESOURCE_LOCAL">
    < properties>
      < property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
      < property name="hibernate.default_schema" value="contacts" />
      < property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
      < property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/contacts" />
      < property name="hibernate.archive.autodetection" value="class, hbm" />
      < property name="hibernate.connection.username" value="root" />
      < property name="hibernate.connection.password" value="root" />
    </ properties>
  </ persistence-unit>
</ persistence>
```

The persistence.xml file must go into the Web application’s WEB-INF/classes/META-INF folder for Hibernate to read it. With that in place, Hibernate requires the following information:

- Database dialect (that is, which database it is talking to, because many databases have slightly different SQL dialects)
- Table space via the default schema
- The database driver used to connect to the database
- The database URL
- What auto-detection should detect (for example, annotated classes, Hibernate mapping XML files, and so on)
- User name and password

Other information can help the performance of Hibernate but is not required.

Configuring BlazeDS

BlazeDS has four configuration files:

- `messaging-config.xml`: Defines publish-subscribe messaging information
- `proxy-config.xml`: Offers proxy service information for HTTP and Web services
• **remoting-config.xml**: Defines information for remoting services such as the one in the article application
• **services-config.xml**: The top-level configuration file that references the other configuration files and also provides security constraints, channels, and logging

Listing 6 demonstrates the services-config.xml file. Note that for the article application, only the remoting-config.xml file is relevant because the application is only using the BlazeDS remoting service.

### Listing 6. A subset of the services-config.xml configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <service-include file-path="remoting-config.xml"/>
    <service-include file-path="messaging-config.xml"/>
    <service-include file-path="proxy-config.xml"/>
    <default-channels>
      <channel ref="contacts-amf"/>
    </default-channels>
  </services>

  <channels>
    <channel-definition id="contacts-amf" class="mx.messaging.channels.AMFChannel">
      <endpoint url="http://localhost:8080/contacts/messagebroker/amf" class="flex.messaging.endpoints.AMFEndpoint"/>
      <properties>
        <polling-enabled>false</polling-enabled>
      </properties>
    </channel-definition>
  </channels>

  <logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Error">
      <properties>
        <prefix>[BlazeDS] </prefix>
        <includeDate>false</includeDate>
        <includeTime>false</includeTime>
        <includeLevel>false</includeLevel>
        <includeCategory>false</includeCategory>
      </properties>
      <filters>
        <pattern>Endpoint.*</pattern>
        <pattern>Service.*</pattern>
        <pattern>Configuration</pattern>
      </filters>
    </target>
  </logging>
</services-config>
```

The services-config.xml configuration file references the other configuration files (which have to exist), configures BlazeDS logging, and sets up any channels. A **channel** is an abstraction for the protocol that is used for the client to communicate with the server. The article application uses the standard AMF protocol with no polling. **Polling** means that the client continually communicates with the server to ensure that the connection is still established—something not needed in this application.
The channel end point specifies the server URL. This end point is required for compiling the project; the client Flash application uses it as a hard-coded value so that it knows the server to connect to. You can actually define the end point URL right in the MXML or ActionScript code, instead.

Finally, the remoting-config.xml configuration file (shown in Listing 7) specifies the adapter class required to handle the remoting operations as well as the actual classes that respond to the remoting calls. (In this case, the bcit.contacts.ContactsService class was given as the responder to remote requests.)

Listing 7. A subset of the remoting-config.xml configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service id="remoting-service"
  class="flex.messaging.services.RemotingService">
  <adapters>
    <adapter-definition id="java-object" default="true"
      class="flex.messaging.services.remoting.adapters.JavaAdapter"/>
  </adapters>
  <default-channels>
    <channel ref="contacts-amf"/>
  </default-channels>
  <destination id="contacts">
    <properties>
      <source>bcit.contacts.ContactsService</source>
    </properties>
  </destination>
</service>
```

Conclusion

This article showed you how to write a Java server-side Web application that runs within Tomcat and answers requests for contact information. You also learned how to write a Flex application using both MXML and ActionScript to create a client-side Flash application. MySQL acted as the data store, and Hibernate—an ORM framework—was used to transform the Java objects into SQL statements that could query and update the MySQL database. Finally, the BlazeDS framework allowed the Flash application to make remote procedure calls and perform remoting on the Java server-side Web application.
Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JEE-BlazeDS-Flex-contacts.zip</td>
<td>7MB</td>
</tr>
</tbody>
</table>

Notes

1. This zip file contains all of the source code (Java, ActionScript 3, MXML) for this project, the Ant build file to generate the WAR, configuration files, and third-party libraries (in the form of JAR files) that this article references and uses.
Related topics

- **JPA Concepts**: Learn more at the Apache Software Foundation.
- **Managing Entities**: Learn how to work with entities. Take this Java EE tutorial.
- **Hibernate EntityManager User Guide**: Find the information you need to work with the Hibernate EntityManager.
- **Updated "Elastic Racetrack" for Flash 9 and AVM2** (Sean Christmann): Learn more about this frame execution model within Flash Player.
- **Creating a declarative XML UI language** (Arron Ferguson, developerWorks, September 2009): Learn more about declarative XML UIs.
- **Create a Flex component** (Sandeep Malik, developerWorks, July 2009): Learn how to create new Flex functionality from scratch.
- **Flex 4 features for creating Software as a Service** (Dan Orlando, developerWorks, July 2009): Learn how to use Flex 4 to enhance RIA user experience.
- **Evaluate IBM products in the way that suits you best**.
- **MySQL**: An open source RDBMS required for use with the example project in this article.
- **Apache Ant**: A Java-based build tool for building the example project.
- **JDK**: The Java SDK (JDK) version 6, required for compiling Java source code within the example project.
- **Adobe Flex SDK**: The Flex 4 SDK for compiling MXML and ActionScript source code within the example project.
- **Apache Tomcat**: The Apache Software Foundation servlet container that provides a Java HTTP Web server environment for running the example project.
- **Adobe BlazeDS**: The Adobe framework for connecting Flex technology to Java Platform, Enterprise Edition (Java EE). **This software is only for reference, as it is already included in the project download for this article.**
- **Red Hat Middleware Hibernate**: Red Hat ORM framework for Java EE container middleware. **This software is only for reference, as it is already included in the project download for this article.**

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)