Build an event-driven framework with Apache Geronimo and JMS

Using Java reflection to define event types

J. Jeffrey Hanson (jeff@jeffhanson.com), Chief Architect, eReinsure.com, Inc.

Summary: One of the most important aspects of an enterprise framework is its ability to respond to real-time signals and events in a timely manner. Discover the technologies and tools that the Apache Geronimo framework provides to enable applications and services to effectively respond to these signals and events and propagate them as messages to interested components residing across a platform's virtual layers. These technologies, including service-oriented architecture (SOA) and an effective event-driven interaction framework using Java™ reflection, can help reduce the complexities of designing an effective event-driven software system while adding flexibility.

Date: 27 Jun 2006
Level: Intermediate
Also available in: Japanese

Responding to real-time changes and events as they occur is an important requirement for an enterprise framework. This article introduces you to technologies and mechanisms embodied within the Apache Geronimo framework that enable applications and services to effectively respond to real-time stimuli, then send and receive events across an architecture's virtual layers.

Dynamic workflow and integration systems are running into roadblocks as developers attempt to design and build them using traditional sequential processing methods. Therefore, the need for more appropriate, event-aware techniques and tools is quickly becoming urgent. SOA and event-driven programming seek to solve this complex challenge.

An SOA presents a loosely coupled development model and runtime environment. It enables service providers and service consumers to work with dynamic component interactions to construct interaction models that exploit the flexibility and power of this model. An event-driven interaction model is better able to respond in a timely manner to dynamic events and signals than traditional synchronous mechanisms, in part because event-driven programming within an SOA takes on many of the same characteristics that distributed systems inherently need, including specialization, modularity, and adaptability.

Event-driven architecture

The Gartner Group introduced event-driven architecture (EDA) in 2003 as a methodology for designing and building systems, services, and applications in which events are routed between loosely coupled event receivers. An event-driven system consists of *event producers* and *event receivers*. Event producers publish events to *event channels* that distribute the events to subscribed event receivers. The event channel forwards events to receivers as producers publish them. If no receiver is available, an event channel can store an event, then forward it to the receiver when the receiver becomes available later. This process is referred to as *store and forward*.

EDA uses messaging concepts as the means of interaction between two or more entities. The interactions are initiated by firing a signal or message that corresponds to some business-domain event. All subscribers to each given event are notified as the event occurs. The subscribers can then act on the event.

EDA benefits from the following attributes:

- **Decoupled associations:** Event publishers and event subscribers don't need to know the existence of each other beforehand.
- **Many-to-many interactions:** One or more events can affect one or more subscribers.
- **Event-based control flow:** Application flow becomes more natural as the application responds to events as they happen.
- **Asynchronous messaging:** Business logic can occur concurrently with events.

By building applications and systems around an EDA, you can construct them in a manner that facilitates more responsiveness, because by design, event-driven systems are more adaptable to unpredictable and changing environments.

The benefits of event-driven design and development

Event-driven programming enjoys many benefits. For example, such programming:

- Has reduced complexity for developing and maintaining distributed systems.
- Offers easier and less-expensive assembly and configuration of applications and services.
- Promotes source code and component reuse, thereby reducing bugs and facilitating agile development and deployment.

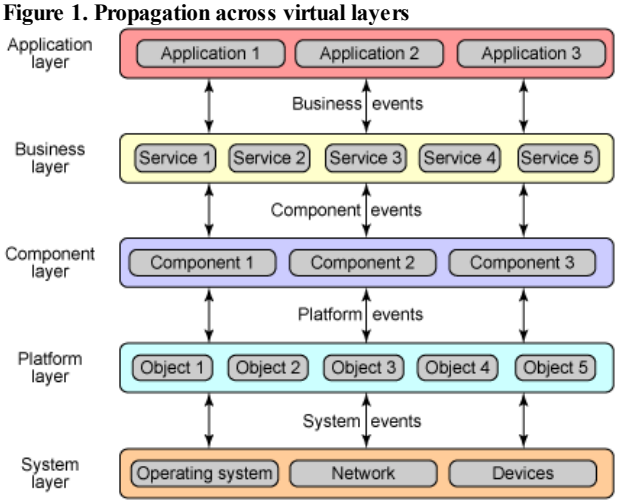
In the short term, event-driven design and development allow quicker and easier customization. In the long term, system status is more accurate.

EDA and SOA together

Unlike sequential or procedural systems -- in which clients must poll for change requests -- an EDA allows systems and components to respond dynamically, in real time, as events occur. EDA complements SOA by introducing long-running processing capabilities. Therefore, businesses benefit because event consumers receive events as they happen, and loosely coupled services can be invoked to provide more timely and accurate data to customers.

Within an EDA, you can transmit events across all segments of an SOA, including physical tiers and an architecture's virtual layers, so that systems respond and act effectively. Figure 1 illustrates events propagated across layers of an architectural stack.

Figure 1. Propagation across virtual layers



As you can see, an event can occur as a result of any change in an application, business, component, platform, or system layer. From a technical standpoint, business events are considered higher level in nature than, say, system events or component events.

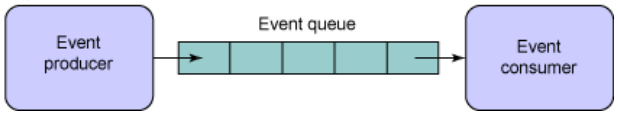
The cause of an event -- *event causality* -- is an important factor in understanding that event. Event causality can be broken down into *horizontal causality* and *vertical causality*. Horizontal causality occurs when the event publisher resides on the same layer in an architecture's virtual layers as the event receiver. Vertical causality occurs when the event publisher resides on a different layer than the event receiver.

EDA and event queues

Event-driven programming is structured around the concept of decoupled relationships between event producers and event consumers. In other words, an event consumer doesn't care where or why an event occurs; rather, it's concerned that it will be invoked when the event has occurred. Systems and applications that separate event producers from event consumers typically rely on an event dispatcher, or *channel*. This channel contains an event queue that acts as an intermediary between event producers and event handlers.

Figure 2 illustrates the relationships between producers, consumers, an event channel, and a topic, or *queue*.

Figure 2. Event channel



The role of the event queue is to store events received from producers and transmit these events to consumers when each consumer is available -- typically, in the order the events are received.

Event queues and topics

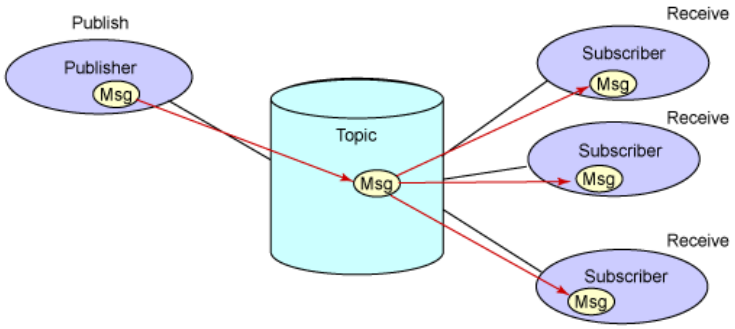
Most event-driven systems rely on prebuilt event queue technologies, such as a Message-Oriented Middleware (MOM) framework. MOM is a type of asynchronous messaging model framework based around a message queue.

The primary advantage of a MOM framework is its ability to store messages indefinitely and route them to consumers when the consumers are ready to receive them. MOM systems work within the following messaging models:

- **Point-to-point:** This model is based on message repositories known as *queues* in which messages can be sent from one or more producers to a single consumer.
- **Publish/subscribe:** This model is based on message repositories known as *topics* in which messages can be published from one or more producers to one or more subscribed consumers.

Figure 3 illustrates the interactions between one publisher, an event channel, a topic, and multiple consumers who are subscribed to a given message type.

Figure 3. Interactions between a publisher, subscribers, an event channel, and a topic



The Java Message Service (JMS) framework is a Java application program interface (API) abstraction on the MOM model.

Using JMS in EDA

Java technology provides the JMS as a common way for Java programs to create, send, receive, and read messages. It's a framework of interface and class abstractions of common concepts and semantics found in most messaging systems.

Through the JMS interfaces, message producers and consumers can send and receive messages in point-to-point or publish/subscribe models. The following list shows the primary components found in JMS:

- **ConnectionFactory:** An object used to create a JMS connection
- **Connection:** A connection to a JMS system
- **Destination:** An abstraction of either a message topic or a message queue
- **Session:** A context within which to send and receive messages
- **MessageProducer:** A component that a session creates and that's used for sending messages to a destination
- **MessageConsumer:** A component that a session creates and that's used for receiving messages from a destination

---

A simple event framework using Geronimo and JMS

Apache Geronimo is bundled with the Active MQ open source messaging provider. Active MQ supports JMS, thereby providing a means for applications built around the Geronimo framework to take advantage of the messaging capabilities of JMS.

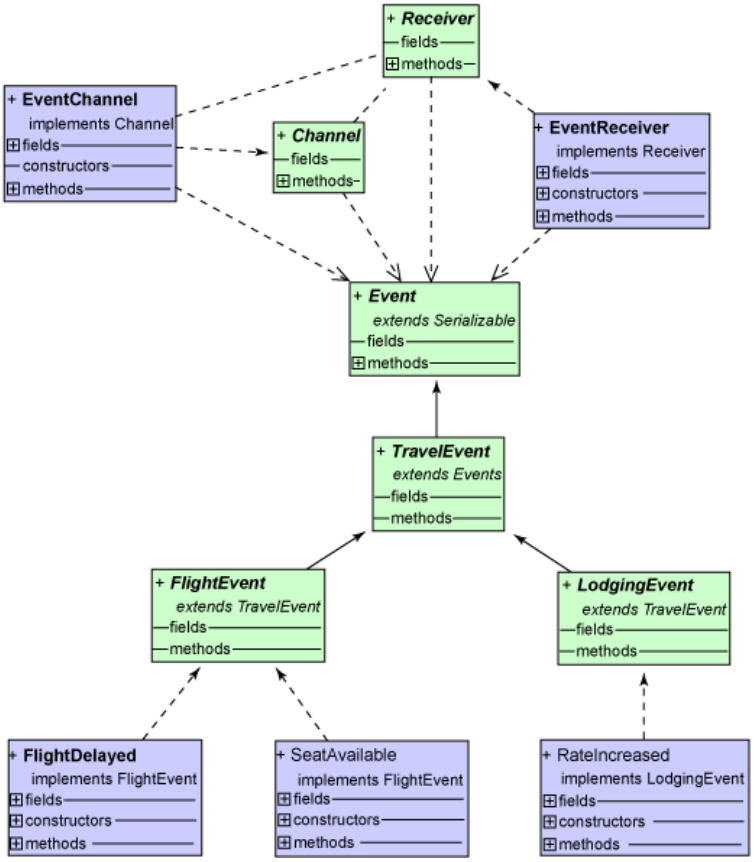
The following sections define a simple event framework built using Geronimo, Active MQ, and the concepts and semantics of JMS. The event framework defined in these sections consists of an event channel, event publishers, and event receivers. The event channel is responsible for registering and unregistering event receivers and for routing event messages from event publishers to event receivers in an asynchronous manner. The unique concept that this framework presents is the event channel's ability to filter and route messages to the appropriate receivers based on the type of Java class or interface that the event object implements.

Filter and route events using class/interface hierarchies

Typical messaging systems allow message subscribers to define the type of event they will receive based on a dot-delimited string, such as *travel.flights.seats* or *travel.lodging.rates*. The event framework presented in this article also allows subscribers to subscribe to specific types of events; however, the event types are defined by a hierarchy of Java classes and interfaces.

The event framework might represent a dot-delimited message type hierarchy, such as the class hierarchy shown in Figure 4.

Figure 4. Event application class relationships



Based on this diagram, an event receiver subscribed to the events represented by the `Event` interface would receive all events, while an event receiver subscribed to events represented by the `FlightEvent` interface would only receive events based either on that interface or on the `FlightDelayed` class or the `SeatAvailable` class. This design allows event receivers to subscribe to multiple event types at once. For example, an event receiver could subscribe to all events by invoking the `subscribe()` method of the event channel with a parameter of `Event.class`. If new event types are added, the event receiver automatically begins to receive them as they are published.

The event channel deals with event hierarchies by determining the most specific subtype of the event interface subscribed by an event receiver. For example, the `FlightDelayed` class, shown in Listing 1, implements the `FlightEvent` interface; therefore, the event channel would first look for subscribers of the `FlightDelayed` class, then the `FlightEvent` interface, and so on up the class/interface hierarchy.

Listing 1. The FlightDelayed class

```
class TravelEvent extends Event {}
      class FlightEvent extends TravelEvent {}
      class LodgingEvent extends TravelEvent {}

      public class FlightDelayed
        implements FlightEvent
      {
        private String message = "";

        public FlightDelayed()
        {
        }

        public FlightDelayed(String message)
        {
          this.message = message;
        }

        public String getMessage()
        {
          return message;
        }

        public void setMessage(String message)
        {
          this.message = message;
        }
      }
```

Event channels

An *event channel* is a component that event publishers use to publish events and that event receivers use to subscribe and receive events. The interface for a simple event channel is shown in Listing 2.

Listing 2. Interface for a simple event channel

```
public interface Channel
      {
        void start();

        void stop();

        void publish(final Event event);

        void subscribe(final Receiver receiver,
                       final Class eventClass);

        void unsubscribe(final Receiver receiver,
                         final Class eventClass);
      }
```

Java reflection

The Java Runtime Environment (JRE) maintains information about each class or interface in the `java.lang.Class` class. An instance of a `Class` object presents detailed method signatures, field types, constructor signatures, modifiers, and constants.

The reflection API Javadoc (see Resources at the end of this article for a link) defines how a developer can programmatically discover information at run time about classes, interfaces, arrays, and the like.

Notice that the `subscribe()` and `unsubscribe()` methods require a parameter of type `Class`, which the event channel uses to determine the type of event to which the receiver will be subscribed or unsubscribed.

To avoid event receivers polling to determine when an event occurs, event receivers are invoked through the `receive()` method of the `Receiver` interface. Java reflection is used whenever an event is published to the event channel to determine which subscribers are to receive the event. The `receive()` method is then invoked on these objects. Listing 3 shows a simple event receiver.

Listing 3. Implementation for a simple event receiver

```
public class EventReceiver
        implements Receiver
      {
        private static final transient Log log =
          LogFactory.getLog(EventReceiver.class);

        private String id = "";

        public EventReceiver()
        {
        }

        public EventReceiver(String id)
        {
          this.id = id;
        }

        public void setId(String id)
        {
          this.id = id;
        }

        public String getId()
        {
          return id;
        }

        public void receive(final Event event)
        {
          log.info("EventReceiver [" + id
                   + "] received event [" + event.getMessage() + "]");
        }
      }
```

Listing 4 shows an excerpt from the event channel.

Listing 4. Implementation for the event channel

```
public class EventChannel
        implements Channel
```

```
{
  private static final String TOPIC_NAME =
    "java:comp/env/EventTopic";

  private static final String MQ_URL = "tcp://localhost:61616";

  private HashMap subscribers = new HashMap();
  private TopicConnectionFactory factory = null;
  private Topic eventTopic = null;
  private TopicConnection topicConn = null;
  private TopicSession topicSess = null;
  private TopicSubscriber topicSubscriber = null;
  private TopicPublisher topicPublisher = null;
  private EventConsumer eventConsumer = null;

  private void handleEvent(Event event)
  {
    final Set received = new HashSet();

    for (Class eventClass = event.getClass();
         Event.class.isAssignableFrom(eventClass);
         eventClass = eventClass.getSuperclass())
    {
      ArrayList receiverList = new ArrayList();
      getReceiversForEvent(getEventLeafInterface(eventClass),
                           receiverList);
      Receiver[] receivers = new Receiver[receiverList.size()];
      receiverList.toArray(receivers);
      for (int i = 0; i < receivers.length; i++)
      {
        invokeOnce(received, receivers[i], event);
      }
    }
  }

  private void invokeOnce(Set received,
                          Receiver receiver,
                          Event event)
  {
    received.add(receiver);
    receiver.receive(event);
  }

  private Class getEventLeafInterface(Class cls)
  {
    Class retVal = null;

    if (Event.class.isAssignableFrom(cls))
    {
      retVal = cls;
      if (cls.isInterface())
      {
        return retVal;
      }
    }

    Class[] interfaces = cls.getInterfaces();
    if (interfaces != null)
    {
      for (int i = 0; i < interfaces.length; i++)
      {
        if (Event.class.isAssignableFrom(interfaces[i]))
        {
          retVal = interfaces[i];
          break;
        }
        retVal = getEventLeafInterface(interfaces[i]);
      }
    }

    return retVal;
  }

  public void start()
  {
    try
    {
      factory = new ActiveMQConnectionFactory(MQ_URL);
      topicConn = factory.createTopicConnection();
      topicSess =
        topicConn.createTopicSession(false,
                                     Session.AUTO_ACKNOWLEDGE);
      eventTopic = topicSess.createTopic(TOPIC_NAME);
      topicSubscriber = topicSess.createSubscriber(eventTopic);
      topicPublisher = topicSess.createPublisher(eventTopic);

      eventConsumer = new EventConsumer(this);
      Thread consumerThread = new Thread(eventConsumer);
      consumerThread.setDaemon(false);
      consumerThread.start();
```

```
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }

    public void stop()
    {
      // close topic connections, sessions, consumers, etc.
    }

    public void publish(final Event event)
    {
      try
      {
        ObjectMessage eventMessage = topicSess.createObjectMessage();
        eventMessage.setObject(event);

        topicPublisher.publish(eventMessage);
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }

    public void subscribe(final Receiver receiver,
                          final Class eventClass)
    {
      ArrayList receiverList = null;

      Class leafCls = getEventLeafInterface(eventClass);

      if (subscribers.get(leafCls) == null)
      {
        receiverList = new ArrayList();
        subscribers.put(leafCls, receiverList);
      }
      else
      {
        receiverList = (ArrayList) subscribers.get(leafCls);
      }

      if (receiverList.indexOf(receiver) < 0)
      {
        receiverList.add(receiver);
      }
    }

    public void unsubscribe(final Receiver receiver,
                            final Class eventClass)
    {
      Class leafCls = getEventLeafInterface(eventClass);
      if (subscribers.get(leafCls) != null)
      {
        ArrayList receiverList = (ArrayList) subscribers.get(leafCls);
        receiverList.remove(receiverList);
      }
    }
  }
```

Note: The complete source code for the `EventChannel` class is available for download from the Downloads section near the end of this article.

Listing 5 shows an excerpt from the event consumer implementation.

Listing 5. Implementation for the event consumer

```
class EventConsumer
       implements Runnable, ExceptionListener
     {
       private boolean running = false;
       private boolean stopped = true;
       private EventChannel eventChannel = null;

       private EventConsumer(EventChannel eventChannel)
       {
         this.eventChannel = eventChannel;
       }

       public void run()
       {
         log.info("Event Consumer started");

         // Create a Topic Connection, Session, and a MessageConsumer for the Topic
         // loop until stopped and distribute events to the event channel
```

```
        // using the handleEvent method
        eventChannel.handleEvent(event);

        stopped = true;

        log.info("Event Consumer stopped");
      }

      public void shutdown()
      {
        running = false;

        while (stopped == false)
        {
          Thread.yield();
        }
      }
    }
```

Note: The complete source code for the `EventConsumer` class is available for download from the Downloads section at the end of this article.

Deploy and run the event framework within Geronimo

The event framework uses a Web application deployed within Geronimo to test each event type. In addition to the event framework, the Web application consists of one HTML form that allows event message input and one servlet that receives HTTP requests and dispatches the content to the event channel.

The HTML form (shown in Figure 5) simply allows three types of event messages to be sent to the dispatching servlet.

Figure 5. Start screen for the Web application

The event-dispatching servlet instantiates the event channel object and three sample event receivers. The event receivers are then subscribed to given events, which the servlet publishes to the event channel object. Listing 6 shows the servlet.

Listing 6. Implementation of the dispatch servlet

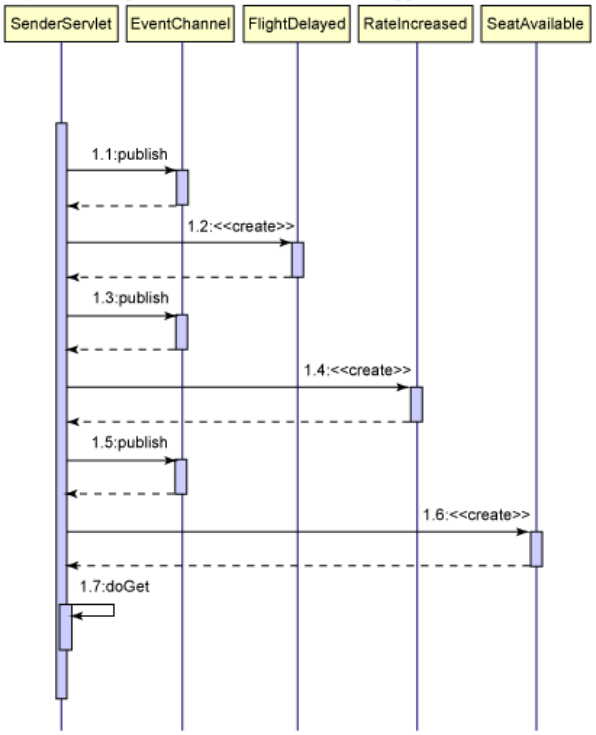
```
public class SenderServlet extends HttpServlet
    {
      private EventChannel eventChannel = null;
      private EventReceiver allTravelEventReceiver = null;
      private EventReceiver flightEventReceiver = null;
      private EventReceiver lodgingEventReceiver = null;

      public void init()
        throws ServletException
      {
        super.init();

        eventChannel = new EventChannel();
        eventChannel.start();

        // create event receivers
        allTravelEventReceiver =
          new EventReceiver("allTravelEventReceiver");
        flightEventReceiver =
          new EventReceiver("flightEventReceiver");
        lodgingEventReceiver =
          new EventReceiver("lodgingEventReceiver");

        // subscribe to all Travel events
        eventChannel.subscribe(allTravelEventReceiver,
                               TravelEvent.class);

        // subscribe to Flight events
        eventChannel.subscribe(flightEventReceiver,
                               FlightEvent.class);

        // subscribe to Lodging events
        eventChannel.subscribe(lodgingEventReceiver,
                               LodgingEvent.class);
      }

      public void destroy()
      {
```

```
    super.destroy();

      // unsubscribe all event receivers and stop the event channel
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException
  {
    // respond with input form
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException
  {
    String txtMsg = req.getParameter("txtMsg");
    if (txtMsg != null && txtMsg.length() > 0)
    {
      String flightDelayed = req.getParameter("FlightDelayed");
      String rateIncreased = req.getParameter("RateIncreased");
      String seatAvailable = req.getParameter("SeatAvailable");

      if (flightDelayed != null)
      {
        // send a Flight event
        eventChannel.publish(new FlightDelayed(txtMsg));
      }
      else if (rateIncreased != null)
      {
        // send a Lodging event
        eventChannel.publish(new RateIncreased(txtMsg));
      }
      else if (seatAvailable != null)
      {
        // send a Flight event
        eventChannel.publish(new SeatAvailable(txtMsg));
      }
    }

    doGet(req, res);
  }
}
```

Note: The complete source code for the `SenderServlet` class is available for download from the Downloads section at the end of this article.

The application flow for the event framework as invoked by the event dispatching servlet is shown in Figure 6.

Figure 6. Sequences for the event Web application



Deploy the application

The classes for the event framework and the Web application are packaged in a .war file and placed within the *GERONIMO_HOME*/deploy directory. With the .war file created and copied to the deploy directory, Geronimo automatically deploys it on startup. Applications placed in the deploy directory are hot-loaded, enabling Geronimo to reload the application at run time when changes are made. This makes it convenient for debugging the applications.

Run the application

You start the Geronimo application server using the startup script (startup.bat or startup.sh) found in the *GERONIMO_HOME*/bin directory. When you invoke the Geronimo startup script, the Geronimo console window appears. With the event framework's Web application deployed, Geronimo's console window on startup contains lines similar to those shown in Listing 7, confirming that the Web application has started successfully.

Listing 7. Successful startup of the Web application

```
00:12:33,921 INFO  [EventChannel] Starting EventChannel...
00:12:33,937 INFO  [EventChannel] Creating topic connection...
00:12:35,062 INFO  [EventChannel] EventChannel started
00:12:35,062 INFO  [EventChannel] Event Consumer started
00:12:35,093 INFO  [SenderServlet] AllTravelEventReceiver
  [com.jeffhanson.eda.EventReceiver@f84033]
00:12:35,093 INFO  [SenderServlet] FlightEventReceiver
  [com.jeffhanson.eda.EventReceiver@3ee73b]
00:12:35,093 INFO  [SenderServlet] LodgingEventReceiver
  [com.jeffhanson.eda.EventReceiver@16127f4]
```

When an event is sent to the servlet, the servlet publishes it to the event channel. If a `Flight-Delayed` message containing the text `Flight 2365 to Detroit will be delayed 15 minutes` is sent to the servlet, the Geronimo console window shows something similar to Listing 8.

Listing 8. Successful event publishing

```
00:12:53,718 INFO  [SenderServlet] >>>>>
00:12:53,718 INFO  [SenderServlet] >>>>>
00:12:53,734 INFO  [EventChannel] Publishing event
  [com.jeffhanson.eda.events.business.FlightDelayed@863854]
00:12:53,859 INFO  [EventReceiver] EventReceiver [flightEventReceiver]
  received event [Flight 2365 to Detroit will be delayed 15 minutes]
00:12:53,859 INFO  [EventReceiver] EventReceiver [allTravelEventReceiver]
  received event [Flight 2365 to Detroit will be delayed 15 minutes]
```

Summary

Designing an effective event-driven software system that can respond to real-time changes and events in a timely manner is a complex task. An SOA, together with an effective event-driven interaction framework using Java reflection, can reduce complexities and add flexibility. The Geronimo platform provides APIs and tools, including a JMS provider, that you can use to build a powerful event-driven interaction framework.

Moving from linear enterprise programming to a service-oriented design can only produce benefits that apply to the SOA model. Refactoring systems to harvest business services should result in a modular framework of services and components. You can reuse these components to fit many different applications if the interaction model of the service infrastructure is agile and extensible. SOA, EDA, and Apache Geronimo provide the foundation for a powerful and effective software infrastructure.

Downloads

Description	Name	Size	Download method
Java source files	GeronimoEDASrc.zip	12KB	HTTP
WAR file	EventSender.zip	13KB	HTTP

Information about download methods

Resources

Learn

- Get articles that provide resources for event-driven programming and complex event architectures.
- Access the reflection API Javadoc.
- Check out articles and feeds found at SOA Refactoring for information about EDA, MOM, and SOA.
- Get JMS resources at the Sun Development Network.
- Visit Apache Active MQ for information on Active MQ and JMS.
- Take the tutorial at The Reflection API to learn more about Java reflection.
- Check out the Apache Geronimo home page for Geronimo resources.

- Check out the developerWorks Apache Geronimo project area for articles, tutorials, and other resources to help you get started developing with Geronimo today.
- Find helpful resources for beginners and experienced users at the Get started now with Apache Geronimo section of developerWorks.
- Check out the IBM Support for Apache Geronimo offering, which lets you develop Geronimo applications backed by world-class IBM support.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Browse all the Apache articles and free Apache tutorials available in the developerWorks Open source zone.
- Browse for books on these and other technical topics at the Safari bookstore.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Download Apache Geronimo, Version 1.0.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.
- Download your free copy of IBM WebSphere® Application Server Community Edition V1.0 -- a lightweight J2EE application server built on Apache Geronimo open source technology that is designed to help you accelerate your development and deployment efforts.

Discuss

- Participate in the discussion forum.
- Stay up to date on Geronimo developments at the Apache Geronimo blog.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

Jeff Hanson has more than 20 years of experience in the software industry, including work as senior engineer for the Microsoft® Windows® port of the OpenDoc project and lead architect for the Route 66 framework at Novell. Jeff is currently the chief architect for eReinsure.com, Inc. and builds Web service frameworks and platforms for J2EE-based reinsurance systems. Jeff is the author of numerous articles and books, including .NET versus J2EE Web Services: A Comparison of Approaches, *Pro JMX: Java Management Extensions*, and *Web Services Business Strategies and Architectures*

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

Average rating (14 votes)

1 star 1 star
2 stars 2 stars
3 stars 3 stars
4 stars 4 stars
5 stars 5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

Notify me when a comment is added1000 characters left

Post

Be the first to add a comment