Reverse Ajax, Part 5: Event-driven web development

Mathieu Carbou

September 20, 2011

This series explores how to develop event-driven web applications using Reverse Ajax techniques. Part 1 introduced different ways to implement a Reverse Ajax communication. Part 2 explained how to implement Reverse Ajax using WebSockets and discussed the limitations of web servers using Comet and WebSockets. Part 3 explored the difficulties of implementing your own Comet or WebSockets communication system, and it introduced Socket.IO. Part 4 covered Atmosphere and CometD. In this final part of the series, learn about event-driven development. Build an example event-driven web application using the accompanying source code.

View more content in this series

Introduction

This series shows how to develop event-driven web applications using Reverse Ajax techniques. Part 1 introduced Reverse Ajax, polling, streaming, Comet, and long polling. Part 2 explained how to implement Reverse Ajax using WebSockets and discussed the limitations of web servers using Comet and WebSockets. Part 3 showed that implementing your own Comet or WebSockets communication system can be difficult if you need to support several servers, or need to provide an independent web application for users to deploy on their own server. Part 3 also discussed Socket.IO. Part 4 covered Atmosphere and CometD—the best-known open source Reverse Ajax libraries for Java technology servers.

Thus far you've learned about creating components that communicate through events. In this final part of this series, apply the principles of event-driven development and build a sample event-driven web application.

You can download the source code used in this article.

Prerequisites

Ideally, to get the most out of this article, you should know JavaScript and Java, and you should have some web development experience. To run the sample in the article, you will also need the latest version of Maven and the JDK (see Resources).
Terms
You might be familiar with event-driven architecture (EDA), EventBus systems, messaging systems, complex event processing (CEP), and channels. These terms and concepts have existed for many years. You'll likely hear more about them as the technology matures. This section provides brief explanations of these concepts.

Event
An occurrence of something that happens in a system. It usually has attributes, such as an occurrence date (timestamp), a source or location (the component we clicked on), and some data describing the event. An event can optionally have other attributes, depending on the system.

Event-processing architecture (EDA)
Also called event-based programming, this is an architectural design where your application consists of components that communicate and execute by sending and receiving events. A Java Swing graphic user interface (GUI) is an example of an EDA. Each Swing component can listen to events, react to them, send other events, and so on. EDA is composed of several pieces: event producers, event consumers, events, and processing software.

• Event producer - The component issuing the event. In the example in this article, a form submit button is an event producer.
• Event consumer - A component that listens for specific events. For example, in the example form submission case, the browser listens for clicks on the form submit buttons to send the form data to the server.
• Event-processing software - The system core where event producers publish events and event consumers register themselves to receive events. Depending on the software, the processing can be simple (just forward received events to consumer), or complex (CEP). With CEP, the software supports a variety of processing means, such as event aggregation, filtering, and transform. Esper is an example of such software. Event-processing software not only represents a standalone running application, it can also be a library integrated in your application.

Messaging systems
A type of event-driven application where event producers publish messages to channels and event consumers subscribe to the channels. Event producers and consumers have no link to each other and are completely independent. In this type of event-driven application, you often use the term message instead of event.

Channels
A way of categorizing events in a messaging system. They represent the destination where the event producers want the event to be sent. For example, in a chat room application, a channel would be /chatapplication/chatrooms/asdrt678. This channel would identify a specific chat room where event producers can send message, and to which the graphic component would subscribe in order to display newly arrived messages.
Some messaging systems provide two types of channels: queues and topics.

• Queues - When a message is delivered to a queue, only one event consumer will take it and process it. Other consumers won't see it. Queues can be made persistent to guarantee delivery. The best example of a queue would be a mailing request. A web
application publishes a message to the queue /myapp/mail/user-registration when a user registers. There could be several mailing applications subscribed to this queue. If there are none, the message will not be lost.

- **Topics** - When a message is delivered to a topic, every subscriber receives it. A topic is usually not persistent. An example of a topic for monitoring software would be /event/system/cpu/usage, where a producer regularly sends the CPU usage. On the other side, there can be zero or more subscribers, depending on their interest.

**Publish/subscribe**

Event-driven solutions implement the publish/subscribe pattern. Event producers publish events in the processing software, and event consumers subscribe to receive them. The way the event consumers subscribe depends on the software. In messaging applications, they subscribe to channels (also optionally applying filtering rules on the event type, for example). With a CEP such as Esper, the subscription can be done through an SQL-like request to define which events you are interested in.

**Why use event-driven solutions?**

In a traditional communication scheme, if system A requires information from system B, a request will be sent to B. System B will process the request, and system A will hold for the response. When the processing is finished, the response will be sent back to system A. In this **synchronous** communication mode, the resource consumption is not efficient because of the processing time lost while waiting for the response.

In an **asynchronous** mode, system A would subscribe to information it wanted from system B. Then system A would optionally send a notification to system B, returning immediately, and system A would be available to process anything else. This step is optional. Usually, in event-driven applications you don't have to ask the other system to send events because you don't know what they are. When system B publishes the response, system A immediately receives it.

One advantage of event-driven architecture is that it allows better scalability. **Scalability** is the capability of a system to adapt to a change in demand, volume, or intensity while still meeting its objectives. By removing pause times, event-driven architectures usually perform better and have higher processing rates.

Another advantage is in the development and maintenance of the application. With event-driven solutions, each component of your application can be completely isolated and decoupled.

Event-driven solutions allow for the best reaction time because of the lowered latency of the communication.

**Applying event-driven solutions to the web**

Web frameworks used to rely on the traditional request-response pattern, which caused page refreshes. With the arrival of Ajax, Reverse Ajax, and powerful frameworks, such as CometD and Atmosphere, it's now easy to apply the concepts of event-driven architectures to the web to get the benefits of decoupling, scalability, and reactivity.
On the client side
Event-driven architecture can be applied on the client side for GUI development. Instead of creating a traditional web page, you can have a single web page acting as a container. Each component (each part of the page) can be isolated. You can have a Java Swing GUI on the web, like a Google page containing gadgets (see Resources for a link).

You'll need an event bus. For example, you can develop a JavaScript event bus that allows each page component to subscribe and publish to channels. Events can also be synchronized to trigger actions after two or more events are received. The event bus can be used for local events within a page, but you can also have plugins to support remote events by using CometD or Socket.IO.

On the server side
On the server side, you will need to set up a Reverse Ajax framework supporting event-driven architecture. Of those examined in previous parts of this series, only CometD has an event-driven approach. For other frameworks, you will need to add custom support, which is not trivial. You can also add a third-party messaging system such as JMS (for example, Apache ActiveMQ), or a CEP like Esper. A simpler solution is Redis, which supports basic publish/subscribe.

This series is about event-driven web and Reverse Ajax, so we'll concentrate on the client part and will not set up a complex messaging system.

Event-driven web example
The example you'll create with this article is a chat room web application that has a user panel containing the list of connected users. Your user name will be in bold, the active users (those still active after 20 seconds) will be in green, and those inactive after 20 seconds will be in orange. If a user connects or disconnects, the list is refreshed.

For security purposes, a session timeout of two minutes is configured in the web.xml file. After two minutes of inactivity, a pop-up will appear and you'll be redirected to the login page.

You are redirected to the login page as soon as you have no more session or if you have not connected yet. The login page asks for a user name and checks to see if it is available to log you in to the chat room.

Once logged in, you can send a message in the chat room to all users. A console is also present and logs all events received.

The web application is event-based. With the information above, you can easily define several events:

- A user is connected
- A user is disconnected
- The session has expired
- A chat message has been received
- The security filter blocked a request if you are not logged in
• A user becomes inactive
• A user becomes active
• All other events relative to UI coordination

Some events are only local to the web application. They are identified by a local bus, as shown in Listing 1:

**Listing 1. Bus setup**

```javascript
bus = {
  local: new EventBus({
    name: 'EventBus Local'
  }),
  remote: EventBus.cometd({
    name: 'EventBus Remote',
    logLevel: 'warn',
    url: document.location.href.substring(0, document.location.href.length - document.location.pathname.length) + '/async',
    onConnect: function() {
      bus.local.topic('/event/bus/remote/connected').publish();
    },
    onDisconnect: function() {
      bus.local.topic('/event/bus/remote/disconnected').publish();
    }
  })
};
```

Other events are remote, meaning they need a Reverse Ajax system, such as CometD, to be published among all clients. Figure 1 shows the example application.

**Figure 1. Example application**
You can download the example application. Many classes are plumbing classes for security, or for session and user management. This article shows the most important parts of the code, but it is suggested that you download and run the example application to take a deeper look at how it works.

The web application is composed of different components: a chat room, the user list, and the console. Each is quite independent and could be removed without impacting the others.

To set up the event-driven system, locally and remotely, the example uses Ovea’s EventBus system. It provides a local event bus, a bridge for CometD to have remote events, and a way to coordinate events (to trigger actions after several events have completed). You can substitute a different system, of course, if you choose. The example setup is in JavaScript, as shown in Listing 1.

Once the buses are in place, the application and components are event-based. In the example, the IDLE detection system is set up, as shown in Listing 2:

### Listing 2. IDLE detection system

```javascript
bus.local.topic('/event/dom/loaded').subscribe(function() {
  $.idleTimer(20000);
  $(document).bind('idle.idleTimer', function() {
    bus.local.topic('/event/idle').publish('inactive');
  });
  $(document).bind('active.idleTimer', function() {
    bus.local.topic('/event/idle').publish('active');
  });
});
```

With the code in Listing 2, the IDLE system sends events when it detects an activity. This code could be used in any application requiring an IDLE system. In this case, you need to translate it in remote events for the user activity. It is also done in JavaScript, as shown in Listing 3:

### Listing 3. User activity management

```javascript
bus.local.topic('/event/idle').subscribe(function(status) {
  bus.remote.topic('/event/user/status/changed').publish({
    status: status == 'active' ? 'online' : 'away'
  });
});

bus.remote.topic('/event/user/status/changed').subscribe(function(evt) {
  if(evt.user != me.name) {
    $('#users li').filter(function() {
      return evt.user == $(this).data('user').name;
    }).removeClass('online').removeClass('away').addClass(evt.status);
  }
});
```

The first subscription receives events from the IDLE system and sends the user state to the server. The other subscription received user state events from the server. Thus, as soon as a user state changes, the user color in the user list changes to green or orange.
When a user connects or disconnects, an event is sent, as shown in Listing 4:

**Listing 4. User list management**

```javascript
bus.remote.topic('/event/user/connected').subscribe(function(user) {
  $('#users ul').append(row(user));
});

bus.remote.topic('/event/user/disconnected').subscribe(function(evt) {
  $('#users li').filter(function() {
    return evt.user == $(this).data('user').name;
  }).remove();
});
```

The application code is simple, decoupled, and isolated. By reusing a lot of Ovea's technologies you can quickly create event-driven web applications. However, it isn't necessary since another system can be used in its place. This example only took one day of development, and half of it is plumbing code, including:

- Maven, for building the project
- The security features (login, logout, session timeout)
- RES services using Jersey

**Conclusion**

This series showed how to build responsive and scalable applications that provide a good user experience. Event-based web applications are a fairly new concept. Some web frameworks use them internally, but this article showed that you don't need a web framework to build such an application. Using good, specialized libraries is more than adequate in enabling a separation of responsibility between the Java developer and the web designer. Hopefully you have a deeper understanding of how to make event-driven development part of your regular routine. It's highly addictive; once you try it, you won't want to go back to a traditional framework.
Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>reverse_ajaxpt5_source.zip</td>
<td>925KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2011

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)