BEA WebLogic Server

Using WebLogic Events (Deprecated)

WebLogic Server Version 6.0
Document Edition 6.0
December 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA WebLogic Server Administration Guide

Document Edition	Date	Software Version
6.0	December 2000	BEA WebLogic Server 6.0

Contents

1 Overview of WebLogic Events

WebLogic Event API provides a lightweight event management system using a publish/subscribe paradigm. For example, a WebLogic/JDBC client can submit (publish) events to a WebLogic Server. Other clients of the WebLogic Server can register interest in (subscribe) to those events. The WebLogic Server informs subscribers of new events when they occur.

A client can specify conditions, called an *evaluator*, that must be satisfied for an event to be delivered to them. Evaluators can prevent unnecessary network traffic. Evaluators are executed on the WebLogic Server.

The client also specifies what happens when the event occurs. The *Action* resulting from an event can be implemented either on the server or the client side. See Registering interest in an event later in this document.

As a service, WebLogic Events has access to all of WebLogic's other services, like JDBC, RMI, logging, instrumentation, Workspaces, etc. All of these services are integrated in WebLogic. Their APIs share many common aspects that make building a complex networked application easier; your application can use several services, all of which can share access to objects and client resources.

Several WebLogic Servers can operate together as a WebLogic Cluster to manage notifications and registrations, since any WebLogic Server can publish and subscribe to events on other servers simultaneously.

WebLogic Server implements JavaSoft's Java Messaging Service (JMS) specification. You can use WebLogic JMS in any application where you can use WebLogic Events. WebLogic JMS offers features not found in WebLogic Events, such as message persistence, point-to-point messaging, and guaranteed message delivery sequence. Since WebLogic JMS is an industry-standard interface, we recommend that you

implement new event-based applications using WebLogic JMS. You may still choose to use WebLogic Events in applications that do not require the more sophisticated features JMS offers. The WebLogic Events service is small and fast, but limited compared to JMS. Read more about WebLogic JMS in Using WebLogic JMS.

WebLogic Event architecture

The Topic Tree

The Topic Tree is the chief architectural feature of WebLogic Events. The Topic Tree lives on the WebLogic Server and is populated by all of the Event Topics that clients have subscribed to. It is the data structure used to remember and process WebLogic Events as they are subscribed to and published by WebLogic Clients.

Structure of the Topic Tree

The tree structure allows event types to be grouped into categories and further sub-categories, where each branch in the tree represents a sub-category of the event it branches from. In a well organized Topic Tree, as we move from the root towards the leaf nodes, the Event Topics become more specific.

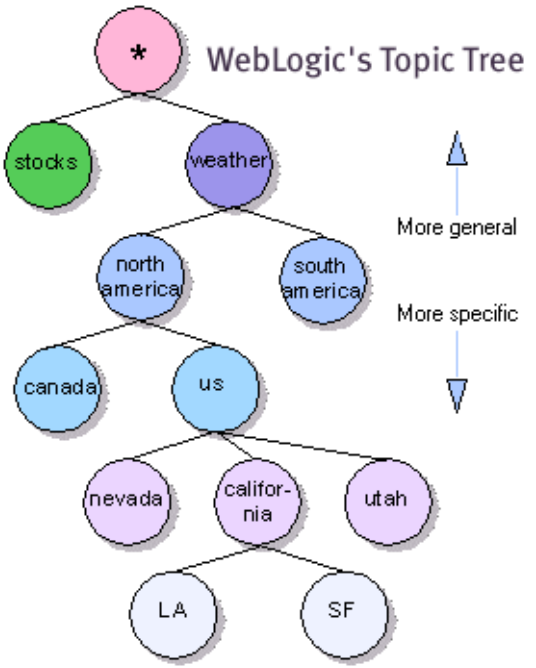
The notation used to describe events in the tree is similar to the dot-notation of domain addresses. Each word represents an event at a particular branch in the tree. For example *comms.devices.telephone.ring* or *comms.devices.telephone.page*. This allows clients to subscribe to a specific Event Topic, using the full event qualifier. This model also allows a client to subscribe to a general category of Event Topics, by only specifying interest to a branch level. E.g. *comms.devices.telephone* would listen for any events pertaining to a telephone.

However, the organization of the tree is the responsibility of the client applications that make up the WebLogic framework. It is your responsibility to program your system so that it organizes the events sensibly, to make the most of this structure.

An example of a structured tree

At the top (or root) of every Topic Tree is a wildcard topic that essentially denotes "every kind of event," which is notated with an asterisk (*). All other topics are considered to be more specific than the root topic. An application that registers interest in the root topic is able evaluate every event that occurs on the WebLogic Server, in the Topic Tree.

Figure 1-1 WebLogic's Topic Tree



In the example shown above, there are two major branches of topics that descend from the root, stocks and weather. We build a typical Topic Tree here for registrations of interest in weather in two California cities, Los Angeles and San Francisco. These topics would be notated as:

weather.northamerica.us.california.la

and

weather.northamerica.us.california.sf

Registering interest in an event

How the Topic Tree is populated

The Topic Tree is dynamically built inside the WebLogic Server as clients subscribe to Event Topics. If a client subscribes to an Event Topic that does not exist in the Topic Tree, a new node, and the new branches required to reach that node, are created automatically. The subscribing client will now receive notice of the new Event whenever it is published.

How a client registers interest in an Event Topic

A WebLogic Client must register interest in a topic with the WebLogic Server in order to evaluate, and act upon events when they are published. Any WebLogic Client application on the network can register interest in any number of Event Topics via the WebLogic EventRegistration services.

A registration is submitted to the WebLogic Server usually with the following pieces of information:

Which event to subscribe to, described by Registration Parameters

- How to Evaluate the event when it occurs, specified via an `EvaluateDef` object.
- What happens when the event evaluates true, specified via an `ActionDef` object.

This is described later in more detail with code examples. See Registering interest in an event.

How a client unregisters interest in an Event

A client application can unregister interest in one of two ways:

Use the count property to control when interest is unregistered. There are several ways that you can control the length of your event registration.

- Call the `EventRegistration.unregister()` method.

NOTE: For event registrations where both the `evaluate()` and the `action()` methods live in the WebLogic Server, it is the responsibility of the client to unregister interest. If the `action()` method lives in the client, then deregistration takes place automatically when the WebLogic client disconnects.

Processing an event

This section describes how event propagation works. Understanding this will help in understanding how to use WebLogic Events in your network applications.

How the Topic Tree is traversed

Any application can submit an event to the WebLogic Server. An event is submitted with a set of event parameters that qualify its scope. Once it is submitted, the WebLogic Server tries to find an exact match for the specific event in the Topic Tree. If found, the EventRegistration(s) for that EventTopic are processed (described next). If no exact match is found, or no clients have registered interest in that EventTopic, then the event is considered as *not delivered* at this point. Next, the Topic Tree is ascended to the next less-specific EventTopic, and the EventRegistrations there are processed, and so on, until the top of the topic tree is reached.

How each EventRegistration is processed

Each client that has an interest in a particular EventTopic should have registered an EventRegistration with that topic. So, each EventTopic in the Topic Tree has a list of EventRegistrations, which describes how each client is interested in the EventTopic. When an EventTopic is matched to an Event, it processes each EventRegistration in the following way:

- If the EventTopic is an exact match to the Event, the Event is evaluated by the EventRegistration. (i.e. the following conditions are skipped)
- However, If this is a less specific Event Topic than the actual published event. i.e. it is an Event Topic that is higher up the Topic Tree, then the following considerations are made.
  - If the event was *not delivered* to a more specific EventTopic successfully, it is evaluated by the EventRegistration.

- If the event was delivered to a more specific Event Topic, then the `sink` flag of the EventRegistration is considered.

`sink`

A registration can be flagged as a *sink*, which means that it is guaranteed to receive a chance to evaluate *all* events in which it has registered an interest, as well as *all* events associated with more specific topics below its registration in the Topic Tree. (Registering for the root topic (*) with the sink flag set to true guarantees a chance to evaluate every event submitted to the WebLogic Server. If your evaluate method for such a registration does nothing but return true, you effectively act upon every event submitted to the WebLogic Server.)

If you set a registration's sink flag to `false` (the default), your client only receives notification when that exact event occurs and not when more specific events below that branch occur. However, there is an exception to this rule:

Even with the `sink` flag set to `false`, the EventRegistration evaluates a more specific event if that event was not delivered successfully to a more specific Event Topic. This would happen if no client had registered interest in the more specific Event Topic at that time. Because a topic does not exist in the tree until a client has registered interest in it, you should be careful when evaluating events. Do not assume that setting `sink` to `false` ensures your client only receives events exactly related to that topic.

Because events are evaluated this way, you can establish EventRegistrations to catch events that no clients have registered to receive.

How events are evaluated by an EventRegistration

When a client registers interest in an event via an EventRegistration, it must specify an Evaluate object, that is associated with the EventRegistration. Once an event reaches a matching registration, the WebLogic Server calls the Evaluate object's `evaluate()` method. The Evaluate class, which implements the interface `weblogic.event.evaluators.EvaluateDef`, is guaranteed to implement this method, and is usually a user written class, or one of the default weblogic evaluators. The Evaluate class must be installed on the server, and must lie in the server's `CLASSPATH`.

The `evaluate()` method is passed the parameters that accompany the event. The custom method may analyze the event parameters, and return either true or false. When true, the WebLogic Server, invokes the Action object's `action()` method, unless `phase` is set to false.

`phase`

When a client registers interest in an topic, it may set the `phase`, which negates the logic that triggers the Action. For example, if an application is interested in when the weather is sunny in San Francisco, the registration will be as follows:

- Topic is *weather.northamerica.california.sf*
- Evaluate parameters are `SKYINDICATOR="fogginess"`, `INDICATORLEVEL="over"`, and `INDICATORVALUE="40%"`
- Phase is false

Now that the logic is reversed, the client will be notified when it is *not* foggy in San Francisco. One would hope this means it is sunny!

How the action process works

If the evaluation process succeeds, the action class for that registration is called.

The Action class is a user-written class, which implements the interface `weblogic.event.actions.ActionDef`. Your action class can perform any action that can be written in Java. Examples of action classes are the `ActionEmail`, `ActionUDP`, and `ActionNull` included in the `weblogic.event.actions` package.

An Action class may notify the WebLogic client that issued the registration of interest that the evaluator returned true. See below for an example of client-side notification.

More about parameters

Parameters are used by several objects in the WebLogic Server, including:

- *Registration management.* Administrative parameters, used for registration management, set limits on how many times the evaluator will be called and other administrative details.
- *Registration of interest.* Registration parameters are a set of name=value pairs that together define the scope of the interest in an event. With these parameters, the WebLogic Server can filter events to further qualify whether or not an event

should be evaluated. A registration is always submitted to the WebLogic Server with a set of registration parameters (that is itself a subset of the registration's administrative parameters).

- *Event parameters*. Event parameters, like Registration parameters, are a set of name=value pairs that further qualify the event. An event is always submitted to the WebLogic Server with a set of event parameters.

A parameter is constructed as a ParamSet object, which may itself be an array of ParamSets. The value associated with each parameter of a ParamSet is a ParamValue object, which may itself be an array of ParamValues.

2 WebLogic Events Objects and Their Classes

The WebLogic Events API includes the following packages:

Package weblogic.event.actions
Package weblogic.event.common
Package weblogic.event.evaluators

There are five basic types of objects in WebLogic Events:

- Evaluate and Action objects. Constructed in the WebLogic Server at the time of registration; they store information about how events should be evaluated and acted upon. These objects are arguments for the registration.
- EventTopic objects. An EventTopic is a object that represents a node in the Topic Tree. It has methods for submitting events to the topic and registering an interest in the topic. It also allows the user to control the lifetime of the topic.
- EventRegistration objects. Store information about the registration; this includes the identity of the registeree, the time of registration, and class information about Evaluation and Action objects, and are submitted to the WebLogic Server at registration time.
- EventMessage objects. Encapsulate Events as they are filtered up through the Topic Tree and are evaluated and acted upon based upon current EventRegistration(s) at each EventTopic.

- Parameter objects. Store specific details about the scope of events, evaluators, and actions.

Evaluate and Action objects

The packages for `weblogic.event.evaluators.*` and `weblogic.event.actions.*` contain classes and interfaces that are used to:

- Write user-defined evaluator and action classes (implementing the EvaluateDef and ActionDef interfaces)
- Submit objects that instantiate these classes, along with registration parameters and other settings, to the WebLogic Server at registration time (the Evaluate and Action classes)

When you register interest in an event, you must also submitthe classnames of an evaluate object, and an action object as two of thearguments for the weblogic.common.EventServices.getEventRegistration()

method. The evaluate class that you write must implement the interface `EvaluateDef`The action class that you write must implement the interface `ActionDef`You can write a single class that implements both interfaces.

The constructors for these objects take the full package name of the user-written class and a set of parameters (a ParamSet) as arguments. The evaluate and action classes are instantiated inside the WebLogic Server at registration. Since the Java class loader does not permit the passing of arguments to the constructors of dynamically loaded classes, the constructors for these classes must be a default constructor—that is, one with no arguments. For this reason, the `registerInit()` method is used to supply the registration parameters to the newly constructed evaluate or action objects. This gives these objects an opportunity to inspect and act upon the evaluate parameters and action parameters that were submitted with the registration.

The EvaluateDef and ActionDef interfaces

`weblogic.event.evaluators.EvaluateDef`

`weblogic.event.actions.ActionDef`

Each of these packages has an interface: `EvaluateDef` and `ActionDef`. Other classes in these packages, like `EvaluateTrue` and `ActionEmail`, implement the EvaluateDef and ActionDef interfaces. You should inspect these as examples of how to write your own Evaluate and Action classes.

You will use ParamSet objects to set parameters for evaluation of and action on events. These parameters must be known to all the parties interested in them. There are no fixed relationships between the parameters for events, registrations, evaluators, and actions, but the developer may build in relationships, depending upon the application.

Here is a good example of how you might build relationships between parameters. In this case, the parameters of the `evaluate()` method must match the parameters of submitted events, and the parameters for the `action()` method must match the parameters of a registration. The topic of interest in weather is San Francisco, and parameters for the evaluation and the event must match in order for the `evaluate()` method to return true. Likewise, in this example, the action to take when the fogginess factor hits a certain low is to send email; consequently, the registration parameters must supply all the information necessary for the action class to send email. The ParamSets likely for this particular event, registration, evaluation, and action might be:

The EvaluateDef and ActionDef interfaces can be implemented by asingle class that contains both an evaluate() and an action() method. Using a single class hasthe advantage that both methods can have access to the same variables.

Methods you will implement

```
public boolean evaluate(EventMessageDef eventMsg)
                        throws ParamSetException;

public void action(EventMessageDef eventMsg);
```

Each of these methods is passed an Object that implements an `EventMessageDef` interface. Referencing the Object by the Interface abstracts us from the details of the implementation of this object. (The

underlying object may been a client or server side implementation). The `EventMessageDef` object contains information about the event and the event parameters. You can access these via the methods defined in the interface.

`registerInit()`

Because the Java class loader does not permit the passing of arguments to the constructors of dynamically loaded classes, the constructor for any user-written class that implements an interface must be a default constructor, that is, one that takes no arguments. For this reason, the registerInit() method is used to supply the registration parameters (a ParamSet object) to the the newly-constructed evaluate or action objects. This gives these objects an opportunity to inspect and act upon the registration parameters.

`isLongRunning()`

This method is deprecated in version 2.5. Users who implement the interfaces for evaluators and actions no longer need to specify this method. Evaluate and action methods now run by default in a separate thread that is selected from a pool of threads in the WebLogic Server, for faster, more efficient operation.

EventTopic objects

`weblogic.event.common.EventServicesDef`

`weblogic.event.common.EventTopicDef`

As of Release 3.0, WebLogic Events now supports EventTopics as first-class objects for use in applications that wish to send and receive event messages. This provides a simple approach to event-based programming. With an EventTopic object, a WebLogic client application can get a subtopic, send an EventMessage, or register an interest in an event.

You request an EventTopic from the EventServices factory, by calling the `EventServicesDef.getEventTopic()` method. You can create subtopics with the `EventTopicDef.getEventTopic()` methods. Here is an example:

```
EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER.CA.SF");
```

where *t3services* is a remote interface obtained from a JNDI lookup.

You can also control the length of the lifetime of an EventTopic, by setting it to `EventTopicDef.EPHEMERAL` or `EventTopicDef.DURABLE` in the `EventTopicDef.getEventTopic()` methods. By requesting the EventTopic "root" from the EventServices factory and creating subtopics that are `DURABLE`, you can exercise more control over the size and shape of the Topic Tree. Here is an example:

```
EventTopicDef topic = t3services.events().
                      getEventTopic("WEATHER.CA.SF",
                                    EventTopicDef.DURABLE);
```

You can use an EventTopic object to get or create subtopics in the Topic Tree. The subtopic can represent more than a single node in the Topic Tree. Just call `getEventTopic()` on the EventTopic itself, as shown here:

```
  EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER");
  EventTopicDef weatherCA   = topic.getEventTopic("CA");
  EventTopicDef weatherCASF = topic.getEventTopic("SF");
  EventTopicDef weatherNYNY = topic.getEventTopic("NY.NY");
```

Once you have an EventTopic, you can submit EventMessages or EventRegistrations to the topic. There is more detail on this in the Implementing with WebLogic Events section below. Here are two brief examples. The first registers interest in a weather event:

```
  EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER.CA.SF");
  Evaluate eval =
    new Evaluate("weblogic.event.evaluators.EvaluateTrue");
  Action action = new Action(this);
  EventRegistrationDef er = topic.register(eval, action);
```

The second example submits an EventMessage for the same topic in the Topic Tree:

```
  EventTopicDef topic =
    t3services.events().getEventTopic("WEATHER.CA.SF");
  ParamSet ps = new ParamSet();
  ps.setParam("TEMPERATURE", 23);
  topic.submit(ps);
```

You can also associate an access control list with an EventTopic and control which users can either submit or receive events. For more on ACLs, read Setting up ACLs for WebLogic Events in the WebLogic Realm.

EventRegistration objects

`weblogic.event.common.EventServicesDef`

`weblogic.event.common.EventRegistrationDef`

When a client registers interest in an event, it is notified when that event occurs. In order to be able to evaluate and act upon events, you must register an interest in an event.

You can use the `EventTopic.register()` method (with an Evaluate object and an Action object) to get an EventRegistration. This is the easiest way to register interest in an event.

You can obtain an interface to an EventRegistration object from the EventServices factory with the method `getEventRegistration()`. Then, register interest in the event as shown here:

```
EventRegistrationDef erDef=
  t3services.events().
    getEventRegistration(String topicName,
                         Evaluate evaluator,
                         Action action,
                         boolean sink,
                         boolean phase,
                         int count);
```

Where t3services is the remote services factory obtained from a

JNDI lookup, and the parameters above are:

`EventRegistrationDef` *erDef*

The method returns a `EventRegistrationDef` interface object. Again, this interface provides your client access to all of the methods in the real EventRegistration object, which may exist on the server.

`String` *topicName*

topicName specifies the EventTopic you are interested in as a dot-notation formatted string that can be parsed (for example, “weather.northamerica.us.california"). A topic can also be specified as an array of strings, where each element in the array corresponds to a subtopic (for example, “weather", “northamerica", "us", “california"). Each topic is added to the Topic Tree in the WebLogic Server dynamically as new registrations are received. Of course, it is necessary for applications that are registering interest in events to know the topics for which applications will submit events, and vice versa.

`Evaluate` *evaluator*

An `Evaluate` object, which is used to instantiate the user-written evaluate class for execution on the WebLogic Server. When constructing your Evaluate object, you specify the full package name of your EvaluateDef class, and a set of evaluation parameters, (ParamSet) that qualify the topic of interest.

`Action` *action*

An Action object, which is used to instantiate the user-written action class, to be invoked when the event is evaluated as true. You construct an Action object by specifying either:

- The full package name of your ActionDef class, which will be instantiated and executed on the server.
- A local instance of an ActionDef object itself, which will be invoked locally on the client.

You also specify a set of parameters (ParamSet) that qualify how the action should be carried out.

`boolean` *sink*

If `sink` is true, the registration will receive notification of every event in which it has registered interest as well as notification for every event below the registered topic in the Topic Tree. For example, setting *sink* to true for a registration for the topic *weather.northamerica.us.california* would assure that this registration gets a chance to evaluate events for topics *weather.northamerica.us.california*, as well as *weather.northamerica.us.california.la* and *weather.northamerica.us.california.sf*. The default value of sink is true.

When `sink` is `false`, the registration will still receive any event messages directed at a more specific topic if there are not successfully delivered.

`boolean` *phase*

If *phase* is set to `false` the logic of the evaluation is reversed. The default value is `true`. For example, if an evaluator for a weather topic returns true when a "fogginess" parameter is reported as over a certain value, we can set phase to `false`, and use the same evaluator to return true if the "fogginess" is *under* a certain value.

`int` *count*

count specifies the number of times a registration may evaluate an event. After the count has expired, the registration will automatically be canceled. If unset, the default is `EventRegistrationDef.UNCOUNTED`. Another option added in Release 3.0 is `EventRegistrationDef.ON_DISCONNECT`, which automatically cancels an event registration when its client disconnects.

Once you have successfully obtained an interface to the EventRegistrationDef object, you must register it with the WebLogic server using its `register()` method. This will return a unique identifying number at instantiation time, whether or not the `register()` method succeeds. If the `register()` method succeeds, the `EventRegistrationDef.isRegistered` variable is set to true.

The EventRegistration class has accessors (like `getEvaluator()`) to return the arguments supplied when an EventRegistration object was requested.

You can unregister by calling the `unregister()` method on the EventRegistrationDef object. If you do not have access to the EventRegistration object you can use the `unregister()` method of the `EventServicesDef` interface, accessible through:

```
t3client.event.services().unregister(int regID);
```

Where t3client is your T3Client object, and regId is the uniqueidentifier returned when the EventRegistrationDef object was registered.

After a registration succeeds, there are internal parameters that are available for the Action parameters and the Evaluate parameters. They include the following (depending on the package):

- `EVENT_SERVER_REGISTRATION_TIME`
- `EVENT_SERVER_REGISTRATION_THREAD`

- EVENT_CLIENT_REGISTER_TIME
- EVENT_CLIENT_REGISTER_THREAD
- EVENT_CLIENT_REGISTER_HOST

EventMessage objects

Events are submitted to the WebLogic Server as EventMessage objects. The easiest way to submit an EventMessage is to request an EventTopic from the EventServices factory with the `EventServicesDef.getEventTopic()` method. Then create a ParamSet, and submit the EventMessage by calling the `EventTopic.submit()` method, which takes a ParamSet as its argument.

You can also request an EventMessage object from the EventServices factory (rather than constructing the object), with the `EventServicesDef.getEventMessage()` method. EventMessages implement the interface `EventMessageDef`.

Although any application can submit events to the WebLogic Server, we restrict this discussion to Java applications that can use Java objects.

The `getEventMessage()` factory method takes two arguments: the topic, and a set of parameters (ParamSet) that qualify the event. To submit an event to the WebLogic Server, you request it from the EventServices factory and then call the `submit()` method on the object. Other methods in the class give you access to the event parameters and allow you to display details about the event. The EventMessage object is passed to the `evaluate()` method by the WebLogic Server, which makes the Event parameters accessible for comparison by the evaluator.

ParamSet and ParamValue objects

Events, registrations, evaluations, and actions all use parameters to qualify scope. Parameters are handled in the WebLogic Events by weblogic.common.ParamSet objects, which contain weblogic.common.ParamValues. WebLogic uses ParamSets and ParamValues to pass data between clients and servers.

A ParamSet parameter is a name=value pair, like `SKYINDICATOR="fogginess"`. The *name* of a parameter is its *keyname*, and all ParamSet contents are accessible by keyname. For each keyname in a ParamSet, you set a corresponding ParamValue. (Note that the variables in ParamTypes for mode, desc, type, and name are not used for events.)

Constructing a ParamSet of just name=value pairs is a simple operation, but powerful enough to allow you to set up complex relationships between ParamSets and ParamValues if necessary. For example, here is how you would create three name=value pairs to set evaluation criteria for an registration of interest in the weather in San Francisco:

```
ParamSet evalRegParams = new ParamSet();
evalRegParams.setParam("SKYINDICATOR",      "fogginess");
evalRegParams.setParam("INDICATORLEVEL",    "over");
evalRegParams.setParam("INDICATORVALUE",    "40");
```

These parameters are used as a constructor for the Evaluate class, which is itself used as an argument for the EventRegistration. We would also set similar parameters when we submitted an Event to the WebLogic Server about the state of weather in San Francisco, for example:

```
ParamSet eventParams = new ParamSet();
eventParams.setParam("SKYINDICATOR",        "fogginess");
eventParams.setParam("INDICATORLEVEL",      "equals");
eventParams.setParam("INDICATORVALUE",      "35");
```

The event parameters are used as an argument for a `getEventMessage()` method. When an event occurs, the event server passes the event to the Evaluate method, which makes the Event parameters available to the Evaluate class. You use the `weblogic.event.evaluators.EvaluateDef.registerInit()` method to recall the registration:

```
public void registerInit(ParamSet params) {
  weatherSymbol = params.getValue("SKYINDICATOR").asString();
  weatherLevel  = params.getValue("INDICATORVALUE").asInt();
}
```

And then we can compare the Event and Registration parameters like this:

```
public boolean evaluate(EventMessage ev) {
  ParamSet eventParams = ev.getParameters();
  if (eventParams.getValue("SKYINDICATOR").asString()
              .equalsIgnoreCase(weatherSymbol))
  {
```

```
    int eventLevel =
        eventParams.getValue("INDICATORVALUE").asInt();
    if (eventLevel == weatherLevel)
    return true;
  }
  return false;
}
```

With this simple illustration of how ParamSets are set and retrieved, you also have a basic outline of how the event registration, event submission, and evaluate processes inter-operate.

Using ParamSets Efficiently

There are some efficiency considerations when using ParamSets and the objects that they qualify. It is neither required nor desirable to create a new EventMessage and its associated ParamSet each time you want to submit an event for a particular topic.

This code snippet, which creates a new ParamSet and EventMessage for each submission, will generate 100 ParamSets, about 300 ParamValues (since two ParamValues are added automatically by `EventMessage.submit()`), 100 events, and will make 100 ParamValue lookups in the ParamSet:

```
for (int i = 0; i < 100; i++) {
  ps = new ParamSet();
  EventMessageDef em = t3.services.events()
                         .getEventMessage(topic, ps);
  ps.setParam("number", i);
  em.submit();
}
```

It is more efficient to create a ParamSet and an EventMessage as instance variables in the class, and then modify them and resubmit them as necessary. This example will generate 1 ParamSet, 3 ParamValues, 1 Event, and will make 100 ParamValue lookups in the ParamSet:

```
ps = new ParamSet();
EventMessageDef em = t3.services.events()
                       .getEventMessage(topic, ps);

for (int i = 0; i < 100; i++) {
  ps.setParam("number", i);
  String status = em.submit();
}
```

The most efficient approach is to make a reference to the underlying ParamValue and set it repeatedly. This example shows how you can use this approach to prevent multiple ParamValue lookups for the counter “number":

```
ps = new ParamSet();
ParamValue num = ps.getParam("number");
EventMessageDef em = t3.services.events()
                        .getEventMessage(topic, ps);

for (int i = 0; i < 100; i++) {
  num.set(i);
  String status = em.submit();
}
```

The last code snippet will generate 1 ParamSet, 3 ParamValues, 1 Event, and will make 1 ParamValue lookup in the ParamSet.

Events and ParamSets are *serially reusable*, but they are not *threadsafe*, that is, you can reuse them but not concurrently in multiple threads. To make the same code snippet multi-thread safe, for example, we would wrap the Event submission in a synchronized block:

```
ps = new ParamSet();
ParamValue num = ps.getParam("number");
EventMessageDef em = t3.services.events()
                        .getEventMessage(topic, ps);

for (int i = 0; i < 100; i++) {
  synchronized (em) {
    num.set(i);
    em.submit();
  }
}
```

Note that although you must create the new ParamSet before you request the new EventMessage (since the ParamSet object is used in the `getEventMessage()` method), it is *not* necessary to call the `ParamSet.setValue()` method until the instant before the `Event.submit()` method is called (or `register()` method in the case of Evaluate and Action constructors). It is only when `submit()` or `register()` is called that the ParamSet is actually examined.

Implementing with WebLogic Events

There are two primary implementations of WebLogic Events: building WebLogic Events applications that can register interest in events, which involves writing `evaluate()` and `action()` methods and building ParamSets; and building event generation into other applications. In these examples, we illustrate this process with four classes:

1. A class to evaluate events
2. A class to act upon appropriate events
3. A class that will register interest in an event
4. A class that will send events to the WebLogic Server

In the example code below, the application allows you to register interest from the command line in a stock and set the price at which you want to buy; then you can send a series of events to the event server that puts stock up for bid. When a bid that matches your offer to buy is evaluated in the WebLogic Server, the action—to send you email notification—is invoked.

Note that you can implement the EvaluateDef and ActionDef interfaces with a single class that has both `evaluate()` and `action()` methods.

Following the stock example is an example that illustrates client-side notification. Client-side notification allows the Action method to be executed on the T3Client rather than in the WebLogic Server.

Writing the Evaluate class

The example application evaluates an event—someone submitting an intent to sell certain stocks at a particular price—against a registration of interest in buying certain stocks at a particular price. The Evaluate class that we write implements the interface `weblogic.event.evaluators.EvaluateDef`.

Step 1. Importing packages

We import the following packages for all WebLogic Events classes:

- `weblogic.common.*;` for access to ParamSets, ParamValues
- `weblogic.event.common.*;` for access to common WebLogic Events objects

For the Evaluate class, we also import `weblogic.event.evaluators.EvaluateDef`, which is the interface this class implements.

In this class, we also create a class variable "services" that defines the WebLogic Server services that the application will use to access the EventServices object factory. The `setServices()` method is called when the evaluator is executed at runtime.

Step 2. The registerInit() method

Since dynamically loaded classes—both the Evaluate and Action classes are loaded dynamically into the WebLogic Server at registration time — cannot pass arguments in a constructor, the `registerInit()` method is used to pass registration parameters to the newly-constructed Evaluate object. The WebLogic Server passes the Evaluate class the ParamSet *params* that was created for the Evaluate class during the registration process.

In this case, we are interested in the "SYMBOL" and the "TRIGGERVALUE" parameters that accompany the registration of interest. We will compare those parameters to the parameters of the submitted event in the `evaluate()` method.

```
public void registerInit(ParamSet params)
     throws ParamSetException
{
  regSymbol       = params.getValue("SYMBOL").asString();
  regTriggerValue = params.getValue("TRIGGERVALUE").asInt();
  System.out.println("Symbol/Trigger Value = " +
     regSymbol + "/" +
     regTriggerValue);
}
```

We print a line to stdout to confirm the registration parameters that we found.

Step 3. The evaluate() method

The `evaluate()` method, put simply, compares the parameters set by the registration of interest in an event to the parameters of the event itself. If it returns true, the WebLogic Server invokes the `action()` method to take action on the event.

In this example, we compare the stock SYMBOL of interest with the stock SYMBOL that is submitted as an event. If the SYMBOL of the event is the one this registration is interested in, we go on to check the BID submitted by the event and see if it matches the TRIGGERVALUE that was registered as interesting.

```
  public boolean evaluate(EventMessageDef ev)
       throws ParamSetException
  {
    // Get the event parameters
    ParamSet eventParams = ev.getParameters();

    // Compare the value of the event "SYMBOL" parameter
    // to the value set for "SYMBOL" at registration time
    if (eventParams.getValue("SYMBOL").asString()
                       .equalsIgnoreCase(regSymbol)) {

      int eventValue = eventParams.getValue("BID").asInt();

      // Then determine whether the event value equals
      // the trigger value set at registration time
      if (eventValue == regTriggerValue)
return true;
    }
    return false;
  }
```

That completes the Evaluate class. The full code example follows.

Code for the EvaluateStocks (evaluate) class

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.common.*;
import weblogic.event.evaluators.EvaluateDef;

public class EvaluateStocks implements EvaluateDef {
```

```
  String  regSymbol;
  int     regTriggerValue;
  private boolean verbose = false;

  T3ServicesDef services=null;

  // Saves the services object
  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  // Gets the registration parameters we will use
  // to evaluate events
  public void registerInit(ParamSet params)
       throws ParamSetException
  {
    regSymbol       = params.getValue("SYMBOL").asString();
    regTriggerValue = params.getValue("TRIGGERVALUE").asInt();
    System.out.println("Symbol/Trigger Value = " +
       regSymbol + "/" +
       regTriggerValue);
  }

  public boolean evaluate(EventMessageDef ev)
       throws ParamSetException
  {
    // Get the event parameters
    ParamSet eventParams = ev.getParameters();

    // Compare the value of the event "SYMBOL" parameter
    // to the value set for "SYMBOL" at registration time
    if (eventParams.getValue("SYMBOL").asString()
                      .equalsIgnoreCase(regSymbol)) {

      int eventValue = eventParams.getValue("BID").asInt();

      // Then determine whether the event value equals
      // the trigger value set at registration time
      if (eventValue == regTriggerValue)
return true;
    }
    return false;
  }
}
```

Writing the Action class

The action we take when our `evaluate()` method returns true is to send email to an address that we provided during the registration of interest in the event. The Action class implements the interface `weblogic.event.actions.ActionDef`.

Step 1. Importing packages

In addition to `weblogic.common.*` and `weblogic.event.common.*`, we import the interface that we implement: `weblogic.event.actions.ActionDef`.

In this class, we also create a class variable "services" that defines the WebLogic Server services that the application will use to access the EventServices object factory. The `setServices()` method is called when the action is executed.

Step 2. The registerInit() method

Like the Evaluate class, the Action class cannot be constructed with arguments to its constructor since it is loaded dynamically into the WebLogic Server. Consequently, the `registerInit()` method is used to pass Action registration parameters to the newly-constructed Action object. The WebLogic Server passes the registration ParamSet *params* to the Action class with this method, where we have access to the parameters interesting for the `action()` method that we will write in the next step.

In this example, we are interested in information about how to send email to the person who registered interest in an event. We retrieve just the parameters that we need to send email in the `action()` method, the addressee and the SMTP hostname. Both of these parameters were required for registration of interest.

```
public void registerInit(ParamSet params) {
  smtphost = params.getValue("SMTPhost").toString();
  to       = params.getValue("Addressee").toString();
}
```

Step 3. The action() method

In this example class, the action we take if our evaluator returns true is to notify the person who registered interest in buying stock that the registered stock is being offered at the price of interest. We have access to the event parameters, which we can include in the email message. In this example we also print a line to stout in the WebLogic Server that the action is taking place, and we include the addressee and the quoted price of interest.

We use the `sendMail()` method, which takes 5 arguments: an SMTP hostname, the email address of the sender, the email address of the message recipient, a subject, and the body of the message. We call the `dump()` method on the event itself to produce a display of the interesting event for inclusion in the email.

```
public void action(EventMessageDef ev) {
  try {
    ParamSet eventParams = ev.getParameters();
    int eventValue = eventParams.getValue("BID").asInt();

    System.out.println("*** Mailing stock event to " + to +
                       " at price: " + eventValue);
    Utilities.sendMail(smtphost,
                       "events@weblogic.com",
                       to,
                       "Stock Event triggered!",
                       ev.dump());
  }
  catch (ParamSetException e) {
    System.out.println("No BID price in ParamSet");
  }
  catch (java.io.IOException ioe) {
    System.out.println("Failed to connect: [" + ioe + "]");
  }
}
```

Finally, we check for ParamSetExceptions if our `try` block fails. We also catch IO exceptions, in case there is a problem with sending the email.

This completes the Action class. The full code example follows.

Code for the MailStockInfo (action) class

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.actions.ActionDef;
import weblogic.event.common.*;

public class MailStockInfo implements ActionDef {

  String smtphost = "";
  String to       = "";

  T3ServicesDef services = null;

  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  public void registerInit(ParamSet params) {
    smtphost = params.getValue("SMTPhost").toString();
    to       = params.getValue("Addressee").toString();
  }

  public void action(EventMessageDef ev) {
    try {
      ParamSet eventParams = ev.getParameters();
      int eventValue = eventParams.getValue("BID").asInt();

      System.out.println("*** Mailing stock event to " + to +
                         " at price: " + eventValue);
      Utilities.sendMail(smtphost,
                         "errors@weblogic.com",
                         to,
                         "Stock Event triggered!",
                         ev.dump());
    }
    catch (ParamSetException e) {
      System.out.println("No BID price in ParamSet");
    }
    catch (java.io.IOException ioe) {
      System.out.println("Failed to connect: [" + ioe + "]");
    }
  }
}
```

Registering interest in an event

The class we write for registration of interest takes arguments from the command line that it uses to build a set of registration parameters. Then we construct an EventRegistration using these parameters, as well as the Evaluate and Action objects that instantiate the Evaluate and Action classes that we have just finished. Finally we submit the registration.

Step 1. Importing packages

In addition to the packages `weblogic.common.*` and `weblogic.event.common.*` that are imported for all WebLogic Events applications, we also import the following packages for the register class:

- `weblogic.event.actions.*` for the Action object used as a constructor for this registration
- `weblogic.event.evaluators.*` for the Evaluate object used as a constructor for this registration

Step 2. Checking the command-line arguments

We pass this registration to the WebLogic Server via a single command-line, and we retrieve the arguments for later use. The first step is to check that we have the correct number of arguments, and, if not, to print out usage information.

```
if (argv.length !=5> {
  System.out.println("Usage: "
           + "java tutorial.event.stocks.Register "
           + "WebLogicURL STOCKSYMBOL PRICE SMTPHOST EMAIL");
  System.out.println("Example: "
           + "java tutorial.event.stocks.Register "
           + "t3://localhost:7001 SUNW 75 "
           + "smtp.foo.com demos@foo.com");
  return;
}
```

Step 3. Processing the command-line arguments

We use the first command-line argument (the URL of the WebLogic Server) to create a T3Client and connect.

```
T3Client t3 = null;
try {
  t3 = new T3Client(argv[0]);
  t3.connect();
```

We use the second and third command-line arguments to build a ParamSet object that we will use to supply registration parameters to the Evaluate class. These parameters will be compared against similar parameters of events that are submitted to the WebLogic Server.

```
ParamSet evRegParams = new ParamSet();
evRegParams.setParam("SYMBOL",           argv[1]);
evRegParams.setParam("TRIGGERVALUE",     argv[2]);
```

Finally, we use the last two command-line arguments to build a second ParamSet object that we will use to supply registration parameters to the Action class, in this case, information for sending email.

```
ParamSet acRegParams = new ParamSet();
acRegParams.setParam("SMTPhost",  argv[3]);
acRegParams.setParam("Addressee", argv[4]);
```

Step 4. Obtaining the EventServices factory

All even registration is achieved via the `EventServicesDef` interface, otherwise known as the WebLogic EventServices factory. You obtain a remote interface to the EventServices factory, via the `T3ServicesDef` interface, otherwise known as the WebLogic T3Services factory. You look up the T3Services factory in the WebLogic JNDI tree using the following code:

```
  T3ServicesDef t3services;
  Hashtable env = new Hashtable();
  env.put(Context.PROVIDER_URL, weblogic_url);
  env.put(Context.INITIAL_CONTEXT_FACTORY,
          weblogic.jndi.WLInitialContextFactory.class.getName());
  Context ctx = new InitialContext(env);
  t3services = (T3ServicesDef)
ctx.lookup("weblogic.common.T3Services");
```

```
ctx.close();
```

Where `weblogic_url` is the URL of your WebLogic Server. You access the EventServices factory via the `T3Services` interface:

```
EventServicesDef eventServices = t3services.event();
```

You application uses the `EventServicesDef` API to access the event functionality on the WebLogic Server.

Step 5. Creating and submitting the registration

`weblogic.event.common.EventTopicDef`

`weblogic.event.common.EventRegistrationDef`

`weblogic.event.actions.ActionDef`

`weblogic.event.evaluators.EvaluateDef`

To register, you first get an EventTopic (the one in which you wish to register interest) from the EventServices factory as shown here:

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
```

Then use the EventTopic to register, by calling `EventTopicDef.register()`. It takes at least two arguments (see below for more arguments for the `register()` method):

- An Evaluate object
- An Action object

The Evaluate and Action objects you pass to the `register()` method must each be constructed with two arguments, the names of the classes we wrote above, and the ParamSets that we constructed with the command-line arguments in this class.

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
Evaluate eval =
  new Evaluate("tutorial.event.stocks.EvaluateStocks",
               evRegParams);
Action action =
  new Action("tutorial.event.stocks.MailStockInfo",
```

```
      acRegParams);
  EventRegistrationDef er = topic.register(eval, action);
```

Note that you can also construct a new Action object with an Object as an argument—not the name of a class. This allows a client-side program to pass in a local copy of an Action, which means that when the Evaluate method returns true, the Action class will be executed on the client, which allows for client-side notification, or callbacks. Here is an example, although it doesn't belong with the class we're using in this explanation:

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
Evaluate eval =
  new Evaluate("tutorial.event.stocks.EvaluateStocks",
               evRegParams);
Action action = new Action(this);
EventRegistrationDef er = topic.register(eval, action);
```

You cannot use an Object as an argument for the Evaluate constructor; the Evaluate object is always executed on the Server.

In addition to the Evaluate and Action objects that are required for each registration, you can supply other arguments to the `register()` method, including:

- Boolean to indicate whether the topic is a sink (default is false)
- Boolean to indicate the topic's phase, that is, whether Evaluate methods that return "true" or "false" should be evaluated (default is true)
- Constant to indicate the count, which is the maximum number of events this registration should receive before automagically unregistering itself (default is `EventRegistrationDef.UNCOUNTED`). New in Release 3.0 is the option `EventRegistrationDef.ON_DISCONNECT` indicates that the registration should be cancelled when the client with a registered interest is disconnected. (If you are using a client-side object for the Action, this happens automatically; this applies to an event registration for which the Action object is located on the WebLogic Server, and the client is responsible for unregistering when its interest is completed.)

Here is an example that illustrates setting the sink, phase, and count for an event registration:

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
Evaluate eval =
  new Evaluate("tutorial.event.stocks.EvaluateStocks",
               evRegParams);
```

```
  Action action =
    new Action("tutorial.event.stocks.MailStockInfo",
       acRegParams);
  EventRegistrationDef er =
        topic.register(eval, action, true, false,
                       EventRegistrationDef.ON_DISCONNECT);
```

After we submit this registration to the WebLogic Server, we disconnect in a `finally` block.

```
      int regid = er.getID();
      System.out.println("Registration ID is " + regid);
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
```

That completes the register class. The full code example follows.

Code for the Register class

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.actions.*;
import weblogic.event.common.*;
import weblogic.event.evaluators.*;

public class Register {

  public static void main(String argv[]) throws Exception {

    // Get 5 command-line arguments that will be used for
    // setting registration parameters
    if (argv.length != 5)
    {
      System.out.println("Usage: "
                + "java tutorial.event.stocks.Register "
                + "WebLogicURL STOCKSYMBOL PRICE SMTPHOST EMAIL");
      System.out.println("Example: "
                + "java tutorial.event.stocks.Register "
                + "t3://localhost:7001 SUNW 75 smtp.best.com "
                + "demos@foo.com");
      return;
    }
```

```
    // Connect to the WebLogic Server using the URL supplied as the
first
    // command-line argument
    T3Client t3 = null;
    try {
      t3 = new T3Client(argv[0]);
      t3.connect();

     // Create a ParamSet to be used by the Evaluate method as each
     // Event is received to decide whether the Action method should
      // be called. We take the second and third command-line
      // arguments as values.
      ParamSet evRegParams = new ParamSet();
      evRegParams.setParam("SYMBOL",           argv[1]);
      evRegParams.setParam("TRIGGERVALUE",     argv[2]);

     // Create another ParamSet to be used by the Action method to
      // specify where to send the mail. We take the last two
      // command-line arguments as values.
      ParamSet acRegParams = new ParamSet();
      acRegParams.setParam("SMTPhost",  argv[3]);
      acRegParams.setParam("Addressee", argv[4]);

    // Create an EventTopicDef for the topic "STOCKS", and register
    // an interest in it with the EvaluateStocks evaluate class and
      // the ActionEmail action class.
      EventTopicDef topic =
        t3.services.events().getEventTopic("STOCKS");
      Evaluate eval =
          new Evaluate("tutorial.event.stocks.EvaluateStocks",
                       evRegParams);
      Action action =
          new Action("tutorial.event.stocks.MailStockInfo",
                     acRegParams);

      // Submit the EventRegistration to the WebLogic Server
      EventRegistrationDef er = topic.register(eval, action);
      int regid = er.getID();
      System.out.println("Registration ID is " + regid);
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
}
```

Sending events to the WebLogic Server

After we register our interest in an event, we need one more class to submit events to the WebLogic Server for evaluation. This example shows a simple class that—like the register class—takes a series of command-line arguments and uses them to set parameters for submitting events to the WebLogic Server.

Step 1. Importing packages

In this class, we import the packages `weblogic.common.*` and `weblogic.event.common.*`.

Step 2. Checking the command-line arguments

In this example, we ask the user to supply parameters that qualify the event. Here we check the number of command-line arguments and supply a usage example if the numbers do not match up.

```
if (argv.length != 4) {
  System.out.println("Usage: "
            + "java tutorial.event.stocks.SendEvents "
    + "WebLogicURL STOCKSSYMBOL STARTPRICE ENDPRICE");
  System.out.println("Example: "
            + "java tutorial.event.stocks.SendEvents "
    + "t3://localhost:7001 SUNW 75 95");
  return;
  }
```

Step 3. Processing the command-line arguments

We use the first argument supplied by the user, the URL of the WebLogic Server, to create a T3Client.

```
  T3Client t3 = null;
  try {
```

```
      t3 = new T3Client(argv[0]);
      t3.connect();
```

We use the other command-line arguments as values for the ParamSet that we will use in the constructor for the EventMessage. We supply the lower and upper bounds of a range of prices at which the stock symbol for this event is selling, and each integer within that range is then submitted as a separate event to the WebLogic Server. Instead of requesting a new EventMessage and constructing a new ParamSet for each event, we reuse the same objects and reset the parameter for each submission inside a loop. For more information on increasing the efficiency of your WebLogic Events code, check above.

```
EventTopicDef topic =
  t3.services.events().getEventTopic("STOCKS");
ParamSet eventParameters = new ParamSet();
eventParameters.setParam("SYMBOL", argv[1]);
int open  = Integer.parseInt(argv[2]);
int close = Integer.parseInt(argv[3]);
```

Step 4. Submitting events

We submit the a series of events with a range of prices inside a loop that does nothing except iterate through the range of prices, reset a parameter, and then submit the event to the EventTopic.

```
      for (int bid = open; bid < close; bid++) {
        eventParameters.setParam("BID", bid);
        System.out.println("Injecting price event with BID = " +
bid);
        String status = topic.submit(eventParameters);
      }
    }
```

Finally, we disconnect from the WebLogic Server.

```
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
```

This completes the class for submitting events to the WebLogic Server. The full code example follows.

Code for the SendEvents class

```
package tutorial.event.stocks;

import weblogic.common.*;
import weblogic.event.common.*;

public class SendEvents {

  public static void main(String argv[]) throws Exception {

    // Check the number of command-line arguments
    if (argv.length != 4) {
      System.out.println("Usage: "
               + "java tutorial.event.stocks.SendEvents "
               + "WebLogicURL STOCKSSYMBOL STARTPRICE ENDPRICE");
      System.out.println("Example: "
               + "java tutorial.event.stocks.SendEvents "
               + "t3://localhost:7001 SUNW 75 95");
      return;
    }

    // Connect to the WebLogic Server with the URL supplied as the
    // first command-line argument.
    T3Client t3 = null;
    try {
      t3 = new T3Client(argv[0]);
      t3.connect();

     // Bid up the stock to the point where it will make our Evaluate
     // method return true and call our Action method.  Note that in
      // order to change the event parameters, we do not need to
      // create a new event nor create a new ParamSet; just set the
      // values and submit the event. Also note that we use the
      // same topic "STOCKS" and the same parameter name "SYMBOL"
      // when we submit the event as when we registered an interest
      // in this event.
      ParamSet eventParameters = new ParamSet();
      EventTopicDef topic =
        t3.services.events().getEventTopic("STOCKS");

     // Use the second command-line arg for the value of the "STOCKS"
      // parameter.
      eventParameters.setParam("SYMBOL", argv[1]);

      // Use the last two command-line args for the begin and end
      // prices for the event.
```

```
      int open  = Integer.parseInt(argv[2]);
      int close = Integer.parseInt(argv[3]);

      for (int bid = open; bid < close; bid++) {
        eventParameters.setParam("BID", bid);
        System.out.println("Injecting price event with BID = " +
bid);
        String status = topic.submit(eventParameters);
      }
    }
    finally {
      try {t3.disconnect();} catch (Exception e) {;}
    }
  }
}
```

Here is a copy of the email message received when this example was run:

```
Topic: STOCKS
Registration:
Topic : STOCKS
ID      :11
Flags   :+Sink+Phase:true
Evaluate:tutorial.event.stocks.EvaluateStocks
Evaluate Params:
EVENT_CLIENT_REGISTER_TIME = Tue Sep 03 20:09:07  1996
SYMBOL = SUNW
TRIGGERVALUE = 75
EVENT_CLIENT_REGISTER_HOST = bigbox/107.4.192.255
EVENT_CLIENT_REGISTER_THREAD = main
EVENT_SERVER_REGISTRATION_THREAD = ExecuteThread
EVENT_SERVER_REGISTRATION_TIME = Tue Sep 03 20:09:10  1996

Action  :tutorial.event.stocks.MailStockInfo
Action Params:
EVENT_CLIENT_REGISTER_TIME = Tue Sep 03 20:09:07  1996
SMTPhost = smtp.myhost.com
Addressee = abc@myhost.com
EVENT_CLIENT_REGISTER_HOST = bigbox/107.4.192.255
EVENT_CLIENT_REGISTER_THREAD = main
EVENT_SERVER_REGISTRATION_THREAD = ExecuteThread
EVENT_SERVER_REGISTRATION_TIME = Tue Sep 03 20:09:10  1996

Count   :UNCOUNTED
EventMessage Parameters:
SYMBOL = SUNW
BID = 75
EVENT_SERVER_SUBMIT_THREAD = ExecuteThread
```

```
EVENT_SERVER_SUBMIT_TIME = Tue Sep 03 20:09:28  1996
-----------------------------------------
```

Using client-side notification

You may want the Action to be executed in the client rather than in the WebLogic Server. Client-side notification allows a T3Client, when registering interest in an event, to specify an Action object with its registration that is run in the local JVM. Rather than constructing the Action object with a String that is the full package name of a class on the WebLogic Server, the Action object is constructed with a reference to a (local) object that implements `weblogic.event.actions.ActionDef`.

Here is an example of a T3Client's registration of interest in an event that illustrates how the Action object is constructed for client-side notification. The Action object for this registration is a reference to an object "clientSideNotify" (which implements `weblogic.event.actions.ActionDef`) that is instantiated in the client and whose `action()` method is called each time the `evaluate()` method (always executed in the WebLogic Server) of the Evaluate class succeeds.

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();

Action action  = new Action(new clientSideNotify());
Evaluate eval  =
     new Evaluate("weblogic.event.evaluators.EvaluateTrue");

try {
  EventTopicDef topic =
    t3.services.events().getEventTopic("STOCKS");
  EventRegistrationDef er =
     topic.register(eval, action,
                    true,  // sink
                    true,  // phase
                    EventRegistrationDef.UNCOUNTED);

  int localregID = er.getID();
}
```

Note that you do not have to specify a different object; you can specify "this" as the object to receive notification.

There is a simple example of client-side notification in `tutorial/event/clientside/client1.java`.

Setting up ACLs for WebLogic Events in the WebLogic Realm

WebLogic controls access to internal resources like events through ACLs set up in the WebLogic Realm Entries for ACLs in the WebLogic Realm are listed as properties in the `weblogic.properties` file.

You can set the Permissions "submit" and "receive" for events by entering a property in the properties file. The receive permission has a dual purpose, since the ACL also controls registration and filters events from subordinate topics.

The ACL name "weblogic.event" controls access to all event services. Setting the Permissions "submit" and "receive" for the ACL name "weblogic.event" to "everyone" allows anyone to submit and receive events, unless a more specific Permission has been set.

Note that if you create an ACL for a particular object that has multiple permissions (in this case "submit" and "receive"), you must create an ACL for *each* permission. Even a more general ACL will not supply the permissions.

For example, if you create a general ACL to set the permissions for event receipt for the high-level topic "weather.northamerica" that allows everyone to receive events for that topic, and then you create an ACL that permits only joe and bill to *submit* events for the topic "weather.northamerica.us", *no one* will be able to *receive* events for that topic unless you create an ACL for it, in spite of the ACL that gives everyone permission to receive event notification for a more general topic. If you create an ACL for permissions on *any action* for the topic "weather.northamerica.us," you must specify users *every permission* for that topic.

If this ACL is *not* set, everyone is allowed to submit and received events.

Example:

```
weblogic.allow.receive.weblogic.event.weather.us=everyoneweblogic
.allow.submit.weblogic.event.weather.us=weatherWireweblogic.allow
.receive.weblogic.event.weather.us.ca.sf=billc,sam,donweblogic.al
low.submit.weblogic.event.weather.us.ca.sf=weatherWire
```

Note that both “submit” Permissions are required in this scenario. Because a specific Permission has been set to allow event notification for the subtopic “weather.us.ca.sf” to only 3 users, a specific Permission for “submit” for that topic must also be set, or no one will be able to submit events for the subtopic.