BEA WebLogic Portal™

Guide to
Events and Behavior Tracking

Version 4.02
Document Date: November 2001

Guide to Events and Behavior Tracking

Document Edition	Date	Software Version
4.02	November 2001	WebLogic Portal 4.0 WebLogic Personalization Server 4.0

Contents

1. Overview of Events and Behavior Tracking

2. Creating Custom Events

3. Persisting Behavioral Tracking Data

4. JSP Tag Library Reference for Events and Behavior Tracking

Index

About This Document

This document describes events and behavior tracking in BEA WebLogic Portal™ and BEA WebLogic Personalization Server™.

This document includes the following topics:

- Chapter 1, “Overview of Events and Behavior Tracking,” which describes the high-level architecture for events and behavior tracking. It also provides detailed information about each event type.
- Chapter 2, “Creating Custom Events,” describes how to create custom events, custom behavior tracking events, custom event listeners, and custom behavior tracking listeners.
- Chapter 3, “Persisting Behavioral Tracking Data,” which describes how to record behavior tracking data and the database structure for behavior tracking.

What You Need to Know

This document is intended for the following audiences:

- The Commerce Business Engineer (CBE) or JSP content developer, who uses JSP templates to specify which products and Web site content trigger events.
- The business analyst, who defines the company’s business protocols for its Web sites. This user may design scenario actions used in campaigns.
- The System Analyst or Database Administrator, who administers databases.
- The Java developer, who creates Java code for custom events.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.beasys.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Portal and the WebLogic Personalization Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Portal and WebLogic Personalization Server documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

The following WebLogic Portal and WebLogic Personalization Server documents contain information that is relevant to using events and behavior tracking.

- *Guide to Using the E-Business Control Center*.
- *Guide to Registering Customers and Managing Customer Services.*

Contact Us!

Your feedback on WebLogic Portal and WebLogic Personalization Server documentation is important to us. Send us e-mail at **docsupport@beasys.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Portal and WebLogic Personalization Server documentation.

In your e-mail message, please indicate that you are using the documentation for WebLogic Portal and WebLogic Personalization Server **Product Version:** release.

If you have any questions about this version of WebLogic Portal or WebLogic Personalization Server, or if you have problems installing and running WebLogic Portal or WebLogic Personalization Server, contact BEA Customer Support through BEA WebSUPPORT at **www.beasys.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

Convention	Item
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

1 Overview of Events and Behavior Tracking

To help personalize campaigns and to effectively analyze customer interactions with a Web site, you need a comprehensive event tracking and logging system. To fulfill this requirement, BEA WebLogic Portal and BEA WebLogic Personalization Server include an Event and Behavior Tracking system. Events identify how a customer is currently interacting with an e-commerce site and the Behavior Tracking system records the event information. With these systems you have the ability to specify, customize, and record selected information. Event data can be used by leading e-analytics and e-marketing systems to evaluate behavioral and transactional data from your online customers. With this analysis you can create and enhance personalization rules, customize product offers, and optimize interactive marketing campaigns. This topic introduces you to Events and Behavior Tracking and provides a general survey of the elements that make up this system.

This topic includes the following sections:

- What Are Events?
- Behavior Tracking
- Standard Events
- Event Generators
- Event Mechanism
- Event Sequence

What Are Events?

In general, an event is a notification that something has happened in a computer program. WebLogic Portal and WebLogic Personalization Server provide various points for generating events. Events provide a detailed and comprehensive view of the entire customer life cycle across your e-commerce site. These points can be tailored for your applications.

You can use events with campaigns to enhance promotion of products and services. Additionally, you can use events to gather intelligence to evaluate the effectiveness of a campaign. Underlying campaigns are scenarios. Scenarios are executed in the context of a campaign. Scenarios are a set of rules, called scenario actions, that allow you to personalize customer experiences on your e-commerce site. For example, if a customer clicks a *Subscribe Me* link on your Web site, you may want to send that customer an e-mail confirming the subscription. Using events and scenarios, you can choreograph the interactions between customers and your Web site.

With regard to tracking visitor behavior for analysis, the primary interest is in what the customer saw and what the customer did. Inherent in this investigation is information about when customers came to the site and when they left it, plus knowledge about which rules were fired during their visit.

Behavior Tracking

The Event service passes messages to Behavior Tracking. When Behavior Tracking is turned on, this data is recorded in a relational database. This information can then be used by data-mining systems to provide Web site customer information for e-marketing analysis. Behavior Tracking provides the following kinds of information:

- When did customers start, end, or login to their sessions?
- What content or products did customers see?
- What content or products did customers click on?
- What did customers put in their shopping cart?

- What did customers buy?
- What rules were triggered?

The information generated from these events allows various kinds of behavior analyses, such as the following:

- **Associations**: When one event can be correlated to another event.
- **Sequences**: When one event leads to another later event.
- **Classification**: The recognition of patterns and a resulting new organization of data.
- **Clustering**: Finding and visualizing groups of facts not previously known.
- **Forecasting**: Discovering patterns in the data that can lead to predictions about future customer behavior.

Standard Events

This section provides information about the standard events provided by BEA. Specifically, it contains a description of each kind of event, what generates the event, the class where event generation occurs, which product contains the event, and the elements of the event. Events elements comprise the data that is present within each event object.

Events are organized into categories. The following list presents each type of event category along with a brief description of what actions generates the event:

- **Session**: The start time, end time, and if executed, the login time of the customer's session.
- **Registration:** The customer registers on the e-commerce site.
- **Product**: The customer is presented with a product or clicks (selects) the presented product.
- **Content**: The customer is presented some content, such as an ad, or clicks (selects) the presented content.

- **Cart**: An item is added, removed, or updated to the customer's shopping cart. Also generated when an entire order is purchased.
- **Buy**: The customer completes the purchase of one or more items.
- **Rules**: The rules that are fired as a customer navigates a Web site.
- **Campaign**: The events generated within the context of a campaign.

Session Events

Session events fire at the start time, end time, and if executed, the login time of a customer's session.

SessionBeginEvent

Description	Occurs when a customer begins interacting with a Web site.
Generator	See "Servlet Lifecycle Events and Servlet Filter Events" on page 1-14.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

SessionEndEvent

Description	Occurs when a customer leaves a Web site, or when the customer's session has timed out.
Generator	See "Servlet Lifecycle Events and Servlet Filter Events" on page 1-14.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

SessionLoginEvent

Description	Occurs when a customer logs on a Web site.
Generator	`TrackingEventHelper.dispatchSessionLoginEvent()`, `P13NAuthFilter`, and/or Input Processor. See "Login and Creation Events" on page 1-16.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

Registration Event

Only one registration event exists. It is described in the following table.

UserRegistrationEvent

Description	Occurs when customer registers on a Web site.
Generator	`TrackingEventHelper.dispatchUserRegistrationEvent()` and/or Input processor.

Example Class	`examples.wlcs.sampleapp.customer.webflow.LoginCustomerIP` located in `PORTAL_HOME\applications\wlcsApp\wlcs\WEB-INF\src`

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

Product Events

These events occur when customer is presented with a product or clicks (selects) the presented product.

ClickProductEvent

Description	Occurs when a customer clicks a product link.
Generator	JSP Tag. Also see "Servlet Lifecycle Events and Servlet Filter Events" on page 1-14.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`
	`sku`
	`category-id`
	`application-name`

Products	WebLogic Portal only.

DisplayProductEvent

Description	Occurs when a product is displayed to the customer.
Generator	JSP Tag

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`
	`sku`
	`category-id`
	`application-name`

Products	WebLogic Portal only.

Content Events

These events occur when the customer is presented some content, such as an advertisement, or clicks the presented content.

ClickContentEvent

Description	Occurs when a customer clicks some Web site content, such as a link or banner.
Generator	JSP Tag. Also see “Servlet Lifecycle Events and Servlet Filter Events” on page 1-14.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

DisplayContentEvent

Description	Occurs when content is presented to a customer, usually any content from a content management system.
Generator	JSP Tag

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

Cart Events

These events indicate that one or more items are added or removed from a customer's shopping cart.

AddToCartEvent

Description	Occurs when an item is added to a customer's shopping cart.
Generator	Pipeline component. Located in `PORTAL_HOME\applications\wlcsApp-project\application-sync\pipelines`.
Example Class	`examples.wlcs.sampleapp.tracking.pipeline.AddToCartTrackerPC` located in `PORTAL_HOME\applications\wlcsApp\src`

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`sku`
	`quantity`
	`unit-list-price`
	`currency`
	`application-name`

Products	WebLogic Portal only.

RemoveFromCartEvent

Description	Occurs when an item is removed from a customer's shopping cart.
Generator	Pipeline component. Located in `PORTAL_HOME\applications\wlcsApp-project\application-sync\pipelines`
Example Class	`examples.wlcs.sampleapp.tracking.pipeline.RemoveFromCartTrackerPC` located in `PORTAL_HOME\applications\wlcsApp\src`

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`sku`
	`quantity`
	`unit-price`
	`currency`
	`application-name`

Products	WebLogic Portal only.

PurchaseCartEvent

Description	Occurs once for an entire order, unlike the `BuyEvent`, which occurs for each line item. This event is useful for campaigns. You can use it when writing scenario actions to know when your customer makes a purchase with specific characteristics, such as an order greater than $100 or the purchase of a particular product.
Generator	Pipeline component. Located in `PORTAL_HOME\applications\wlcsApp-project\application-sync\pipelines`.
Example Class	`examples.wlcs.sampleapp.tracking.pipeline.PurchaseTrackerPC` located in `PORTAL_HOME\applications\wlcsApp\src`

Elements	`session-id`
	`user-id`
	`event-date`
	`event-type`
	`total-price`
	`order-id`
	`currency`
	`application-name`

Products	WebLogic Portal only.

Buy Event

Only one buy event exists. It is described in the following table.

BuyEvent

Description	Occurs when a customer completes the purchase. A `BuyEvent` occurs for each line item. A purchase may consist of one or more line items. A line item may consist of one or more items. For example, although a particular line item may have quantity of four items, only one `BuyEvent` occurs.
Generator	Pipeline component
Example Class	`examples.wlcs.sampleapp.tracking.pipeline.PurchaseTrackerPC` located in `PORTAL_HOME\applications\wlcsApp\src`

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`sku`
	`quantity`
	`unit-price`
	`currency`
	`application-name`
	`order-line-id`

Products	WebLogic Portal only.

Rules Event

Only one rule event exists. It is described in the following table.

RuleEvent

Description	Indicates the rules that were fired as a customer navigates a Web site.
Generator	Fired internally from advislets

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`ruleset-name`
	`rule-name`

Products	Specific to WebLogic Personalization Server, available in WebLogic Portal.

Campaign Events

These events occur when a customer participates in a campaign.

CampaignUserActivityEvent

Description	Occurs when a customer participates in a campaign. Specifically, this event is fired whenever one or more scenario actions are true and the campaign service is activated. You can limit this event to a single occurrence for a particular scenario. This event is intended for use by analytic software.
Generator	Fired internally from the campaign service

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`campaign-id`
	`scenario-id`

Products	WebLogic Portal only.

DisplayCampaignEvent

Description	Occurs when campaign content, such as an ad, is presented to the customer. Specifically, this event is fired whenever a campaign placeholder displays an ad placed in the ad bucket by a campaign. You can use this event to trigger another campaign. Analytic software uses this event to determine if a customer saw an ad as a result of a campaign.
Generator	Fired internally from the campaign service

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`
	`campaign-id`
	`scenario-id`
	`application-name`
	`placeholder-id`

Products	WebLogic Portal only.

ClickCampaignEvent

Description	Occurs when a campaign item, such as an ad, is clicked on by the customer. Specifically, this event is fired whenever a customer clicks a campaign ad that was placed in the ad bucket by a campaign. You can use this event to trigger another campaign. Analytic software uses this event to determine if a customer clicked on an ad as a result of a campaign.
Generator	Fired internally from campaign service. Also see “Servlet Lifecycle Events and Servlet Filter Events” on page 1-14.

Elements	`event-date`
	`event-type`
	`session-id`
	`user-id`
	`document-type`
	`document-id`
	`campaign-id`
	`scenario-id`
	`application-name`
	`placeholder-id`

Products	WebLogic Portal only.

Servlet Lifecycle Events and Servlet Filter Events

The following events are generated using the Servlet 2.3 API:

- SessionBeginEvent
- SessionEndEvent

These events are defined as part of the Servlet 2.3 lifecycle events. They are listeners on the session `Created()` and session `Destroyed()` events, which are generated by the servlets defined in the `web.xml` file. This file is located at:

```
PORTAL_HOME\applications\wlcsApp\wlcs\WEB-INF
```

where `PORTAL_HOME` is the directory in which you installed BEA WebLogic Portal or BEA WebLogic Personalization Server.

The following events are generated by JSP tags and filtered by the Servlet 2.3 `<filter>` element:

- ClickContentEvent
- ClickProductEvent
- ClickCampaignEvent

For each Web page displayed, the Web Application servlet checks for the presence of a click event in the `HttpServletRequest`. Each page click is then filtered by Web Application servlet as defined by the Servlet 2.3 filter `<element>`. The click events are generated automatically when the `<filter>` element is called on each invocation of

the servlet. The `ClickThroughFilter` determines which type of event is generated by checking the event type in the `HttpServletRequest`. The valid types are defined at:

```
PORTAL_HOME\classes\clickthrough-event-types.properties
```

where `PORTAL_HOME` is the directory in which you installed BEA WebLogic Portal or BEA WebLogic Personalization Server.

Event Generators

The standard events supplied by BEA are generated at important points in an e-commerce site. The components that enable events include Java APIs, JSP tags, JSP scriptlets, Webflow input processors, Pipeline components, content selectors, and classification advislets. You can add or customize generators for each of the following events:

- `DisplayContentEvent`
- `DisplayProductEvent`
- `ClickContentEvent`
- `ClickProductEvent`

Note: `DisplayProductEvent` and `ClickContentEvent` are available in WebLogic Portal only.

Each of these events are generated by JSP tags. You can use the JSP tags that initiate these events to specify which products and what content generates these events. For example, in the wlcsApp E-Commerce Application, the JSP tag for the `DisplayProductEvent` is located in the `details.jsp`.

The tag shown in Listing 1-1 generates an event for any product displayed on a catalog detail page. If you want to generate an event for one particular product, you can write a scriptlet that keys off the SKU for that product.

Listing 1-1 JSP Tag

```
<%-- once the product is displayed, fire off a displayProductEvent --%>
<productTracking:displayProductEvent documentId="<%= item.getName() %>"
        documentType="<%= DisplayProductEvent.ITEM_BROWSE %>"
                        sku="<%= item.getKey().getIdentifier() %>" />
```

When you add a JSP tag for an event, you should include a reference to the tag library descriptor, as shown below:

```
<%@ taglib uri="productTracking.tld" prefix="productTracking" %>
```

Notes: For more information about JSP tags, see Chapter 4, "JSP Tag Library Reference for Events and Behavior Tracking."

The `details.jsp` is located at:

```
PORTAL_HOME\config\wlcsDomain\wlcsApp\wlcs\commerce\catalog\details.jsp
```

where `PORTAL_HOME` is the directory in which you installed WebLogic Portal.

Login and Creation Events

This section discusses different methods for generating login and user registration events.

You can generate the SessionLoginEvent in either of the following ways:

If you are manually using the `<um:login>` tag or `weblogic.servlet.security.ServletAuthentication` to handle login, use the `com.bea.p13n.tracking.TrackingEventHelper.dispatchSessionLoginEvent()` method.

If you are directly using `j_security_check FORM`-based login, register the `com.bea.p13n.servlets.P13NAuthFilter` as the `<auth-filter>` in your Web Application's `WEB-INF\weblogic.xml` file. You do not need to code a JSP or Webflow Processor.

Use the `com.bea.p13n.tracking.TrackingEventHelper.dispatchUserRegistrationEvent()` method to generate the UserRegistrationEvent. You should generate this event after the SessionLoginEvent (which should occur during user creation). You can use either an Input Processor or in a JSP.

If you are using the Portal Webflow framework, the SessionLoginEvent and the UserRegistrationEvent are generated automatically from the `com.bea.portal.appflow.processor.security.PostLoginProcessor` in the security webflow as needed.

Event Mechanism

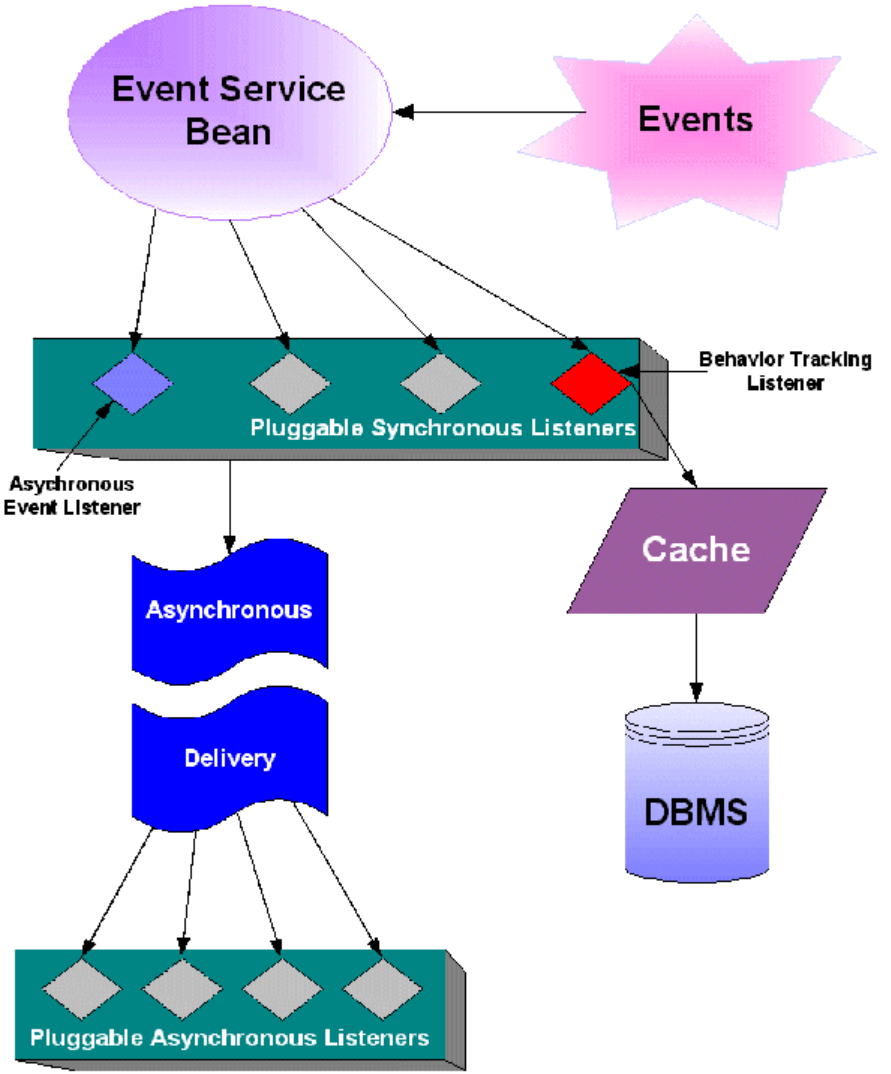
The Event service is an extensible, general purpose, event construction and propagation system. As shown in Figure 1-1, an event is generated by a trigger, such as a JSP tag, which creates the event object, locates the Event service bean, and passes the event object to the Event service. The Event service works with plug-in listeners that disseminate events to listeners interested in receiving the events. At creation time, each event listener returns the list of event types that it wants to receive. When the Event service receives an event, it checks the type of the event and sends the event to all listeners that are subscribed to receive that event's type.

The Event service has two sets of listeners: those that respond to events synchronously and those that respond to events asynchronously. The synchronous listeners use the thread of execution that created and transmitted the event to perform actions in response to that event. Behavior Tracking listeners use only the synchronous listeners. The asynchronous listeners receive the event from the thread where it was created and some time later, handles the event in a different thread of execution. The asynchronous service exists so that long-running event handlers can execute without delaying the application from a Web site visitor's perspective.

Whether a particular plug-in listener is installed on the synchronous or the asynchronous side of the Event service is based on the requirements of the application and is specified in the `application-config.xml` file.

Note: To edit the `application-config.xml` file, use the WebLogic Server Administration console. For more information, see "Installing a Listener Class in the Event Service" on page 2-8.

Figure 1-1 Event Mechanism



Event listeners implement the `com.bea.p13n.events.EventListener` interface. The interface defines signatures for two public methods:

- `public String[] getTypes()`

- `public void handleEvent( Event theEvent )`

The first method returns a list of event types that the listener is interested in receiving from the Event service. For example, if a listener is designed to receive events of type *Foo*, the listener returns *Foo* as an item in the array returned from invoking `getTypes()` on the listener. The second method is invoked when an event is passed to the listener. A listener has no knowledge of whether it is synchronous or asynchronous.

If you wish to create a listener interested in only campaign events, you would list the listener's fully-qualified classname in the `application-config.xml` file in either the `eventService.listeners` property or the `asynchronousHandler.listeners` property (for synchronous or asynchronous handling, respectively). The listener would implement the `EventListener` interface and return the following event types:

```
{"ClickCampaignEvent","DisplayCampaignEvent","CampaignUserActiv
ityEvent" }
```

when its `getTypes()` method is invoked.

Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

To edit the `application-config.xml` file, use the WebLogic Server Administration console. For more information, see "Installing a Listener Class in the Event Service" on page 2-8.

After the listener is installed, events of one of these three types arrive through the listener's `handleEvent( Event theEvent )` interface.

The Asynchronous Delivery graphic in Figure 1-1 indicates that the asynchronous event handler receives events transmitted asynchronously from the synchronous side of the Event service. It then dispatches events to the pluggable asynchronous listeners based on the event types each listener is subscribed to receive.

Event Sequence

Figure 1-2 and Figure 1-3 provide a sample of the firing of events. These figures are intended to give you a sense of the order in which events fire, not a comprehensive examination of event sequencing.

Figure 1-2 Event Sequence Sample—Part 1

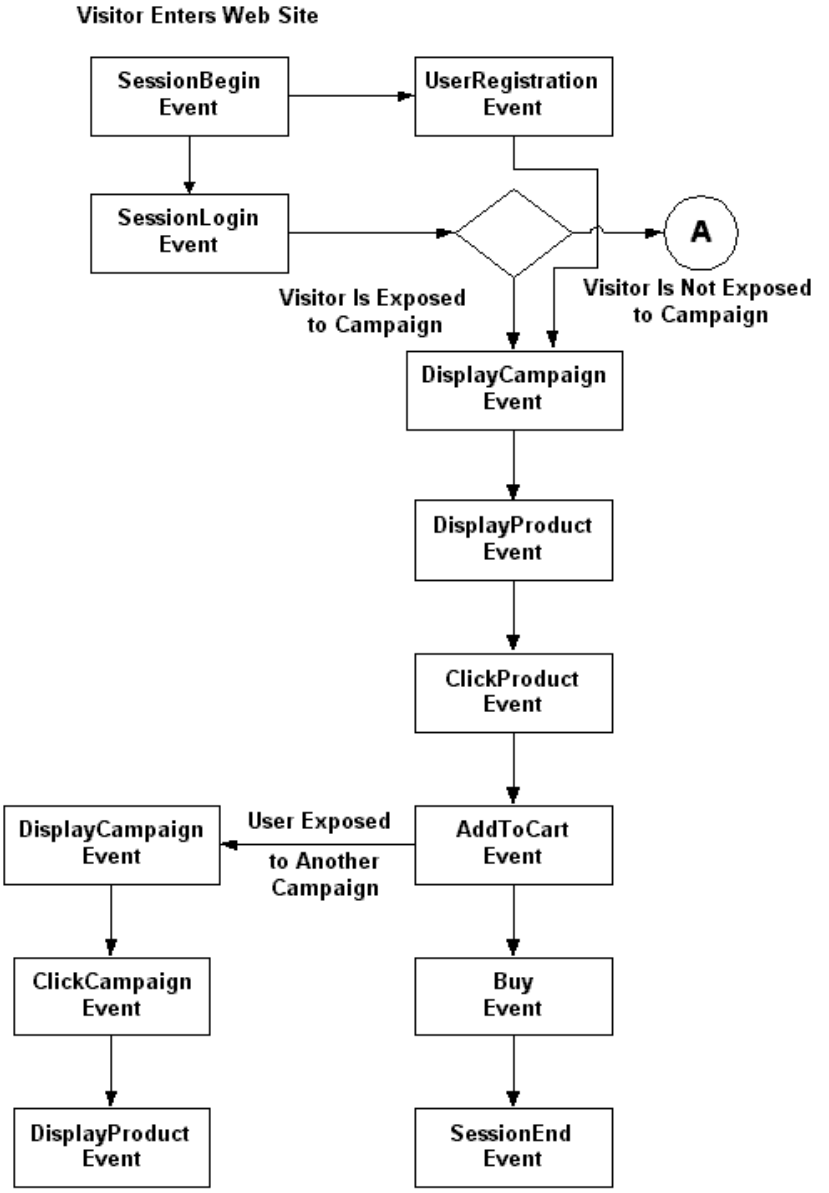
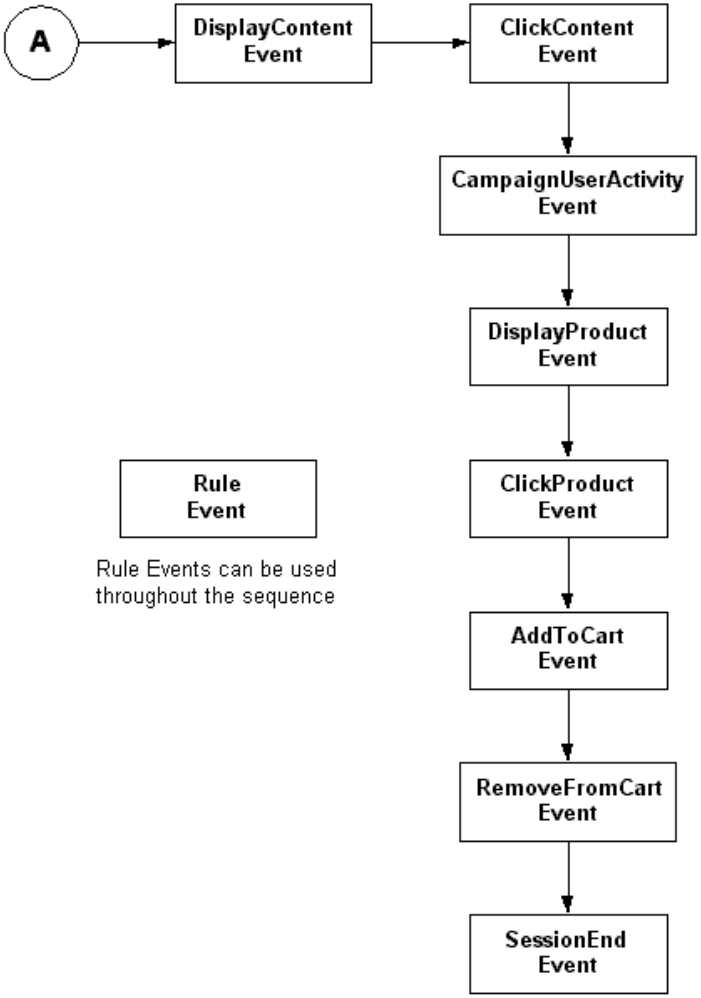


Figure 1-3 Event Seqeuence Sample—Part 2



2 Creating Custom Events

This topic provides the information necessary to write a custom event. You can create a custom event for anything you wish to track. For example, you could create an event that would tell you which pages are displayed for each customer. You could then use the information to determine how many pages are viewed on average per session and which pages are the most popular. Additionally, marketing professionals could use this event when developing scenario actions that are based on the display of particular pages. To demonstrate how to write a custom event, a simple example is provided. Each section references and expands the example.

This topic includes the following sections:

- Overview of Creating a Custom Event
- Writing a Custom Event Class
- Writing a Custom Event Listener
- Writing a Behavior Tracking Event Class
- Debugging the Event Service
- To register a custom event, use the Event Editor in the E-Business Control Center. Registering an event is actually creating a property set for the event. A step-by-step procedure is available in both the E-Business Control Center online help and “Creating and Managing Property Sets” in the Guide to Building Personalized Applications.

Overview of Creating a Custom Event

The creation of a custom event is a multiple-step process. The following list provides an overview of the process and references the information not covered in this topic:

- Write the code that defines the event and event listener.
- Install the event using the WebLogic Server Administration console. For more information, see "Installing a Listener Class in the Event Service" on page 2-8.
- Write the code to generate the event with a JSP tag or an API call.
- Register the event. For more information, see "Registering a Custom Event" on page 2-28.
- To record the event data to the `EVENT` table, create an entry for the event in the `EVENT_TYPE` table. For more information, see Chapter 3, "Persisting Behavioral Tracking Data."

Writing a Custom Event Class

To create a custom event, you first write an event object. This object encapsulates all the necessary information for correctly interpreting and handling the event when it arrives at a listener. All custom events must subclass the `com.bea.p13n.events.Event` class. This base class handles setting and retrieving an event's timestamp and type and provided access to the custom event's attributes. Two `Event` class methods set and retrieve attributes:

```
setAttribute( String theKey, Serializable theValue )
getAttribute( String theKey )
```

These methods can be called from the custom event's constructor to set attributes specific to the new event. Keep in mind that all objects set as values in the `Event` object must be Java serializable. The `getTimeStamp()` method returns the date of the event's creation in milliseconds. The type of an event is accessed using the `Event` class's `getType()` method. The timestamp and type of an `Event` object instance can be set only at creation time in the `Event` constructor.

To illustrate the process of creating a custom event, a simple example is presented here, called `TestEvent`. The example is a basic demonstration of how to create an event subclass. An actual custom event would probably be more elaborate.

A custom event must first have a type. This type should be passed to the superclass constructor (for example, in the `Event` class); this type is returned at `getType()` invocations on custom-event object instances. For example:

```
/** Event Type */
public static final String TYPE = "TestEvent";
```

To properly initialize the `Event` base class of the custom event object, the value `TYPE` is passed to the event constructor. The type of all events must be a simple Java string object.

After defining the type, you must define the keys that access the attributes stored in the custom event. These attributes can be given values in the constructor. For example, the `TestEvent` class has two properties, `userPropertyOne` and `userPropertyTwo`; the type of the value associated with `userPropertyOne` is a `String` and `userPropertyTwo` is a `Double`. The keys are defined as follows:

```
/**
 * Event attribute key name for the first user defined property
 * Attribute value is a String
 */
public static final String USER_PROPERTY_ONE_KEY =
    "userPropertyOne";

/**
 * Event attribute key name for the second user defined property
 * Attribute value is a Double
 */
public static final String USER_PROPERTY_TWO_KEY =
      "userPropertyTwo";
```

Finally, a constructor brings the event type and the process of setting attributes together to create an event object. The constructor looks like:

```
/**
 * Create a new TestEvent
 *
 *
 * @param userPropertyOne some user defined property typed as
 * a String
 * @param userPropertyTwo some user defined property typed as
 * a Double
 */
 public TestEvent( String userPropertyOneValue,
```

```
                    Double userPropertyTwoValue )
{
  /* calls the Event class constructor with this event's type */
  super( TYPE );

  if( userPropertyOneValue != null )
      setAttribute( USER_PROPERTY_ONE_KEY,
                    userPropertyOneValue );

  if( userPropertyTwoValue != null )
      setAttribute( USER_PROPERTY_TWO_KEY,
                    userPropertyTwoValue );
}
```

Putting all the parts together, the entire custom event class is shown in Listing 2-1.

Listing 2-1 TestEvent Class

```
/* Start TestEvent class */

public class TestEvent
  extends com.bea.p13n.events.Event
{
  /** Event Type */
  public static final String TYPE = "TestEvent";

  /**
   * Event attribute key name for the first user defined property
   * Attribute value is a String
   */
   public static final String USER_PROPERTY_ONE_KEY = "userPropertyOne";

  /**
   * Event attribute key name for the second user defined property
   * Attribute value is a Double
   */
   public static final String USER_PROPERTY_TWO_KEY = "userPropertyTwo";

  /**
   * Crate a new TestEvent
   *
   *
   * @param userPropertyOne some user defined property typed as a String
   * @param userPropertyTwo some user defined property typed as a Double
   */
   public TestEvent( String userPropertyOneValue,
                     Double userPropertyTwoValue )
```

```
   {
     /* calls the Event class constructor with this event's type */
     super( TYPE );

     if( userPropertyOneValue != null )
         setAttribute( USER_PROPERTY_ONE_KEY, userPropertyOneValue );

     if( userPropertyTwoValue != null )
         setAttribute( USER_PROPERTY_TWO_KEY, userPropertyTwoValue );
   }
}
/* End TestEvent class */
```

The example in Listing 2-1 shows you how to use the fundamental aspects of the `Event` base class and the event service. An actual custom event constructor would probably be more complex. For example, it might check for default values or disallow null attributes. Additionally, the custom-event object might have more methods or member data.

Writing a Custom Event Listener

In order to listen for an event, you must define an event listener. All event listeners must implement the `com.bea.p13n.events.EventListener` interface and have a no arguments (default) constructor. This interface specifies two methods that are fundamental to transmitting events of a given type to interested listeners:

```
public String[] getTypes()

public void handleEvent( Event ev )
```

The first method returns the types, in a string array, that the listener is interested in receiving. The event service dispatches events of a given type to listeners that return the event's type in the types array. When the event service has determined that a given listener has registered to receive the type of the current event, an event of that type is dispatched to the listener using the `handleEvent( Event ev )` call.

When writing a custom event listener, both methods must be implemented from the `EventListener` interface. Continuing with the `TestEvent` example, the `TestEventListener` listens for instances of `TestEvent` that are sent through the event service. This can be specified as follows:

```
/** The types this listener is interested in */
private String[] eventTypes = {"TestEvent"};

/**
  The method invoked by the event service to determine the
  types to propagate to this listener.
 */
public String[] getTypes()
{
  return eventTypes;
}
```

To handle the event, the `handleEvent( Event evt )` method is implemented as follows:

```
/**
 * Handle events that are sent from the event service
 */
public void handleEvent( Event ev )
{
  System.out.println("TestListener::handleEvent " +
                     " -> received an event" +
                     " of type: " + ev.getType() );

  /* Do the work here */

  return;
}
```

Putting all of these pieces together with a constructor, Listing 2-2 shows a simple event listener that registers to receive `TestEvent` objects.

Listing 2-2 Event Listener

```
import com.bea.p13n.events.EventListener;
import com.bea.p13n.events.Event;

/**
 * TestListener to demonstrate the ease with which listeners can be plugged
 * into the behavior tracking system.
 *
```

```
     * This class should be added to the property eventService.listeners
     * in order to receive events.  The fully qualified classname must be added
     * to this property; don't forget to add the ",\" at the end of the previous
     * line or the properties parser will not find the new classname.
     *
     * The types of events that are heard are listed in the eventTypes
     * String array.  Add and remove strings of that type as necessary.
     *
     * @author Copyright (c) 2001 by BEA Systems, Inc. All Rights Reserved.
     */
    public class TestListener
        implements EventListener
    {

    private String[] eventTypes = {"TestEvent"};

    public TestListener()
    {
    }

    public String[] getTypes()
    {
        return eventTypes;
    }

    public void handleEvent( Event ev )
    {
         System.out.println("TestListener::handleEvent -> received an event" +
                            " of type: " + ev.getType() );

         return;
    }
}
```

As with writing a simple event, writing a simple `EventListener` is also straightforward. Any event listener's internals should be generic; the same `TestEventListener` instance may not handle all `TestEvent` objects. Therefore `TestEventListener` should be entirely stateless and should operate on data that is contained in the event object or stored externally (that is, in a database).

Note: Multiple instances of any listener may execute concurrently.

Installing a Listener Class in the Event Service

To add or remove listeners to the event service, use the WebLogic Server Administration Console. To enable Behavior Tracking, you must add Behavior Tracking as a listener.

Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

To add a synchronous or asynchronous listener, take the following steps:

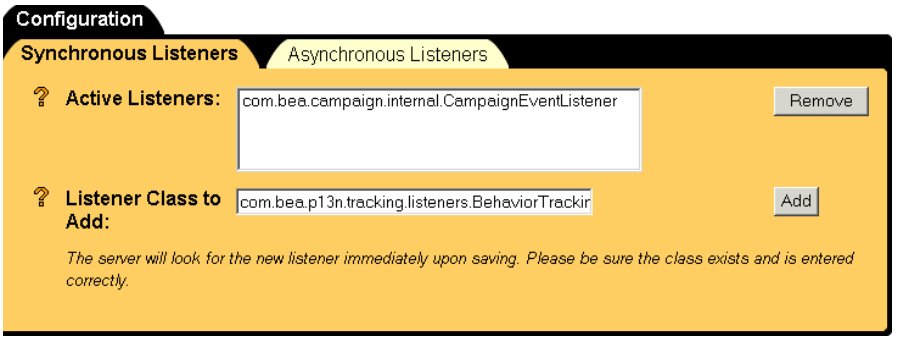
Note: Behavior Tracking listeners can only be implemented as synchronous listeners.

1. In the WebLogic Server Console, navigate to Synchronous or Asynchronous Listeners tab in the node tree for wlcsDomain as follows:

   `http://hostname:port/console → wlcsDomain → Deployments → wlcsApp → Service Configurations → Event Service → Configuration Tab → Synchronous Listeners or Asynchronous Listeners`

2. Add the synchronous or asynchronous listener to the corresponding fields, as shown in Figure 2-2.

Figure 2-1 WebLogic Server Administration Console—Event Service



Writing a Behavior Tracking Event Class

A Behavior Tracking event is a special type of event that tracks a customer's interactions with an e-commerce site. E-analysis systems use the data gathered from Behavior Tracking events to evaluate customer behavior. The evaluation is primarily used for campaign development and optimizing customer experience on a Web site.

A Behavior Tracking event and its listeners are created in much the same way as the `TestEvent` class and `TestEventListener` examples. A simple example is also presented here. The example tracking event is called `TestTrackingEvent`. All Behavior Tracking events persisted (recorded) to a database for use with BEA Behavior Tracking are handled by the `com.bea.p13n.tracking.listeners.BehaviorTrackingListener`. The `BehaviorTrackingListener` extends the `com.bea.p13n.events.EventListener` class.

The `BehaviorTrackingListener` receives and persists Behavior Tracking events from the event service when it is plugged into one of the listener's properties in the `application-config.xml` file.

Notes: For scalability reasons, you should plug the `BehaviorTrackingListener` into the `eventService.listeners` property.

This listener receives events from the event service and adds them to a buffer that is intermittently persisted to the Event tables in the database. The frequency of the sweeping of events from the buffer is controlled by the following properties in the `application-config.xml` file:

- `MaxBufferSize` – Sets the maximum size of the event buffer. Setting this to 0 means all events are persisted as they are received.
- `SweepInterval` – Sets the interval, in seconds, at which to check the buffers to see whether events in the buffer must be persisted. Events are persisted when either the maximum buffer size (`MaxBufferSize`) is reached or the maximum time to wait in the buffer (`SweepMaxTime`) has been exceeded.
- `SweepMaxTime` – Set the time, in seconds, to wait before forcing a flush to the database. This is the longest amount of time that an event can exist in any cache.

You should tune these properties to optimize performance. A buffer sweep should be performed often enough that writing to the database is not too time consuming but not so frequent that the operation is wasteful.

Configuring Events Buffer Sweeping

Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

To configure the sweeping of the events buffer, take the following steps:

1. In the WebLogic Server Console, navigate to Behavior Tracking in the node tree for wlcsDomain as follows:

   `http://hostname:port/console` → `wlcsDomain` → `Deployments` → `wlcsApp` → `Service Configurations` → `Behavior Tracking`

2. Enter the new buffer values in the appropriate fields, as shown in Figure 2-2.

Figure 2-2 WebLogic Server Administration Console—Behavior Tracking

Facilitating OffLine Processing

For facilitating offline processing of customer interactions with a Web site, Behavior Tracking events are designed to be persisted to a table in the database, called the `EVENT` table. Part of the process of recording data from Behavior Tracking events is creating an XML representation of the data, which is stored in the `xml_definition` column of the `EVENT` table. You can persist events in an alternate location and table structure as requirements dictate. This discussion assumes that you are planning to use the BEA Behavior Tracking event persistence mechanism. Therefore, to persist events in the provided `EVENT` table, your custom event must conform to the descriptions in this section so that it is created and persisted properly.

To formally specify the data comprising a Behavior Tracking event, you need to develop an XML-XSD schema for the new event. While XSDs are not used internally to verify the creation of XML, the XML that is created represents the event's data in the database. If the event class is properly developed and used, it will conform to the XML-XSD schema. With an XSD document, development of the constructor and attribute keys for a Behavior Tracking event follows easily.

To correctly turn a Behavior Tracking event into an XML representation, the Behavior Tracking event must have several pieces of member data that fully describe an XML instance document for the schema associated with the event type. This data describes the namespace and XSD file associated with the event. For example, Listing 2-3 and Listing 2-4 show the association between the following files:

`com.bea.campaign.tracking.events.ClickCampaignEvent` and

`/lib/schema/ClickCampaignEvent.xsd` in `PORTAL_HOME\lib\campaign\ejb\campaign.jar`.

For more examples, look at the existing XSD files.

Listing 2-3 ClickCampaignEvent.java

```
/**
   Event for tracking click of campaign
 */
public class ClickCampaignEvent
    extends ClickEvent
{
    /** The event type */
    public static final String TYPE = "ClickCampaignEvent";
```

```
/**
   The XML namespace for this event
*/
private static final String XML_NAMESPACE =
   "http://www.bea.com/servers/commerce/xsd/tracking/clickcampaign";

/**
   The XSD file containing the schema for this event
*/
private static final String XSD_FILE = "ClickCampaignEvent.xsd";

/**
 * Event attribute key name for the campaign id
 * Attribute value is a String
 */
public static final String CAMPAIGN_ID = "campaign-id";

/**
 * Event attribute key name for the scenario id
 * Attribute value is a String
 */
public static final String SCENARIO_ID = "scenario-id";

/**
 * Event attribute key name for storefront (aka application)
 * Attribute value is a String
 */
public static final String APPLICATION_NAME = "application-name";

/**
 * Event attribute key name for item category id
 * Attribute value is a String
 */
 public static final String PLACEHOLDER_ID = "placeholder-id";

/**
   Suggestions for entry into the documentType data passed to the constructor
   Attribute value is a String
*/
public static final String BANNER_AD_PROMOTION = "bannerAdPromotion";

/**
   These are the keys and their order for elements that
   will be present in the XML representing this object
*/
private static final String localSchemaKeys[] =
{
    SESSION_ID, USER_ID, DOCUMENT_TYPE, DOCUMENT_ID,
```

```
    CAMPAIGN_ID, SCENARIO_ID, APPLICATION_NAME, PLACEHOLDER_ID
};

/**
 * Create a new ClickCampaignEvent.
 *
 * @param theSessionId from HttpSession.getId()
 * @param theUserId from HttpServletRequest.getRemoteUser() or
 * equivalent (null if unknown)
 * @param theRequest the http servlet request object
 * @param aDocumentType Document Type for the clicked content (optionally
 * null)
 * @param aDocumentId Document ID for the clicked content (optionally null)
 * @param aCampaignId campaign id for the campaign from which the item was
 * clicked
 * @param aScenarioId scenario id for the scenario (within the campaign)
 * for which the item was clicked
 * @param aApplicationName application name (aka storefront) (optionally
 * null)
 * @param aPlaceholderId a placeholder id
 */
public ClickCampaignEvent( String theSessionId,
                           String theUserId,
                           HttpServletRequest theRequest,
                           String aDocumentType,
                           String aDocumentId,
                           String aCampaignId,
                           String aScenarioId,
                           String aApplicationName,
                           String aPlaceholderId )
{
    super( TYPE,
           theSessionId,
           theUserId,
           XML_NAMESPACE,
           XSD_FILE,
           localSchemaKeys,
           theRequest,
           aDocumentType,
           aDocumentId);

    if( aCampaignId != null ) setAttribute( CAMPAIGN_ID, aCampaignId );
    if( aScenarioId != null ) setAttribute( SCENARIO_ID, aScenarioId );
    if( aApplicationName != null ) setAttribute( APPLICATION_NAME,
        aApplicationName );
    if( aPlaceholderId != null ) setAttribute( PLACEHOLDER_ID,
        aPlaceholderId );
}
```

```
}
```

Notice the cross-reference between `ClickCampaignEvent` and the XSD schema.

Listing 2-4 Corresponding XSD Schema

```
<xsd:schema

targetNamespace="http://www.bea.com/servers/commerce/xsd/tracking
/clickcampaign"

xmlns="http://www.bea.com/servers/commerce/xsd/tracking/clickcamp
aign"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2000/10/XMLSchema
    http://www.w3.org/2000/10/XMLSchema.xsd"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="2.1">
        .
        .
        .
```

The source code for your Behavior Tracking event should also list the keys and their order for creating an XML instance document from an event object. For an example, see Listing 2-3. The structure of an XSD document and details on XML namespaces can be found at http://www.w3.org/XML/Schema. Several XSD schemas for BEA Behavior Tracking events can be found in `/lib/schema` at the following location:

```
PORTAL_HOME\lib\p13n\ejb\events.jar
```

where `PORTAL_HOME` is the directory in which you installed BEA WebLogic Portal or BEA WebLogic Personalization Server.

The namespace and schema are specified as:

```
/**
  The XML namespace for this event
 */
private static final String XML_NAMESPACE=
    "http://<your URI>/testtracking";
```

```
/**
  The XSD file containing the schema for this event
 */
private static final String XSD_FILE="TestTrackingEvent.xsd";
```

Note: These values are used when creating an instance document to populate the fields.

The `schemaKeys` are a list of strings which are keys to the event class's `getAttribute` and `setAttribute` methods. These keys are used to extract the data that populate elements in the XML instance document which represent the Behavior Tracking event. The keys should be listed in an array that consists of string-typed objects. Their order specifies the order in which they appear in the XML instance document. In the XSD files that the Behavior Tracking system generates, the order of the elements is important; an XML file will not validate with an XSD file if elements are out of order. Elements can be omitted by using the XML `numOccurs` keyword and setting the value to zero. For examples of how this is done, see the XSD schemas for BEA Behavior Tracking events in `/lib/schema`, at the following location:

```
PORTAL_HOME\lib\p13n\ejb\events.jar
```

An example array for the Behavior Tracking version of the `TestEvent` described above might appear as:

```
/**
 These are the keys and their order for elements that
 will be present in the XML representing this object.
 */
private static final String localSchemaKeys[] =
{
    SESSION_ID, USER_ID, USER_PROPERTY_ONE_KEY,
        USER_PROPERTY_TWO_KEY
};
```

The `SESSION_ID` and the `USER_ID` are data elements in the `localSchemaKeys` array that are useful in implementing a tracking event. The `SESSION_ID` is the WebLogic Server session ID that is created for every session object. (For more information, see the WebLogic Server 6.0 Documentation Center.) The `USER_ID` field (which may be null) is the username of the Web site customer associated with the session from which the event was generated. For some events, a user may not be associated with an event; as previously mentioned, the `numOccurs` for the `USER_ID` field in an XSD file should be zero. To persist events in the EVENT table, the `SESSION_ID` must be non-null.

All Behavior Tracking events must extend the `com.bea.p13n.tracking.events.TrackingEvent` class. This class defines three keys that are useful for setting attributes for all tracking events, as follows:

- `TrackingEvent.SESSION_ID`
- `TrackingEvent.USER_ID`
- `TrackingEvent.REQUEST`.

These keys are used in `setAttribute` calls made in the `TrackingEvent` constructor when setting the `SESSION_ID`, `USER_ID`, and `REQUEST` (an `HttPServletRequest` object), respectively. They should also be used to retrieve values associated with each key when invoking `Event.getAttribute` (`String Key`) on event objects that extend `TrackingEvent`.

TrackingEvent Base Class Constructor

The `TrackingEvent` base class has a constructor that is more complicated than the `Event` class's constructor. The `Event` constructor is invoked by the `super( String eventType )` call in the `TrackingEvent` constructor. The `TrackingEvent` constructors are shown in Listing 2-5 and Listing 2-6.

Listing 2-5 Tracking Event Constructor—Example 1

```
/**
 * Create a new TrackingEvent.
 *
 * @param theEventType the event's type
 * @param theSessionId from HttpSession.getId()
 * @param theUserId from HttpServletRequest.getRemoteUser() or equivalent
 * (null if unknown)
 * @param theXMLNamespace the namespace for an XML representation of this event
 * type
 * @param theXSDFile the file that contains the schema which specifies and
 * enforces typing on the data in the XML file
 * @param theSchemaKeys the list of keys (in their order in the XSD schema)
 * representing the data to be persisted in this event's XML
     */
public TrackingEvent( String theEventType,
                      String theSessionId,
                      String theUserId,
```

```
                String theXMLNamespace,
                String theXSDFile,
                String[] theSchemaKeys )
```

The `TrackingEvent` constructor shown in Listing 2-6 takes an `HttpServletRequest` object.

Listing 2-6 Tracking Event Constructor—Example 2

```
/**
 * Create a new TrackingEvent.
 *
 * @param theEventType the event's type
 * @param theSessionId from HttpSession.getId()
 * @param theUserId from HttpServletRequest.getRemoteUser() or equivalent
 * (null if unknown)
 * @param theXMLNamespace the namespace for an XML representation of this event
 * type
 * @param theXSDFile the file that contains the schema which specifies and
 * enforces typing on the data in the XML file
 * @param theSchemaKeys the list of keys (in their order in the XSD schema)
 * representing the data to be persisted in this event's XML
 * @param theRequest the http servlet request object
 */
public TrackingEvent( String theEventType,
                      String theSessionId,
                      String theUserId,
                      String theXMLNamespace,
                      String theXSDFile,
                      String[] theSchemaKeys,
                      HttpServletRequest theRequest )
```

In the first constructor, shown in Listing 2-5, the only data that is optional (that is, that can be null) is `theUerId`; all other data is required so that the tracking event is correctly persisted to the `EVENT` table. In the second constructor, shown in Listing 2-6, the `HttpServletRequest` object can be passed in from generating locations where the `HttpServletRequest` object is available. This object provides the data needed to fire rules against event instances.

Note: In order to fire rules on a custom Behavior Tracking event, the `HttpServletRequest` and the `USER_ID` must be non-null. Generally, a non-null `USER_ID` means that a customer is logged into a Web site. Rules cannot be fired on an event with a null-user.

The `TestTrackingEvent` constructor is shown in Listing 2-7.

Listing 2-7 TestTrackingEvent Constructor

```
/**
 * Create a new TestTrackingEvent
 *
 * @param theSessionId from HttpSession.getId()
 * @param theUserId from HttpServletRequest.getRemoteUser() or equivalent
 * (null if unknown)
 * @param userPropertyOne some user defined property typed as a String
 * @param userPropertyTwo another user defined property typed as a Double
 */
public TestTrackingEvent( String theSessionId,
                          String theUserId,
                          String userPropertyOneValue,
                          Double userPropertyTwoValue )
{
    super( TYPE, theSessionId, theUserId, XML_NAMESPACE, XSD_FILE,
           localSchemaKeys );

    if( userPropertyOneValue != null )
        setAttribute( USER_PROPERTY_ONE_KEY, userPropertyOneValue );

    if( userPropertyTwoValue != null )
        setAttribute( USER_PROPERTY_TWO_KEY, userPropertyTwoValue );

}
```

This constructor calls the `TrackingEvent` constructor to populate the required values and then sets the attributes necessary for this particular Behavior Tracking event type.

The entire `TestTrackingEvent` is shown in Listing 2-8.

Listing 2-8 TestTracking Event

```
import com.bea.p13n.tracking.events.TrackingEvent;

/**
 * Test, user-defined behavior tracking event.
 *
 * This event can be persisted to the database.
 *
*/
public class TestTrackingEvent
    extends TrackingEvent
{

    /** Event type */
    public static final String TYPE = "TestTrackingEvent";

    /**
      The XML namespace for this event
     */
    private static final String XML_NAMESPACE="http://<your URI>/testtracking";

    /**
      The XSD file containing the schema for this event
     */
    private static final String XSD_FILE="TestTrackingEvent.xsd";

    /**
     * Event attribute key name for the first user defined property
     * Attribute value is a String
     */
    public static final String USER_PROPERTY_ONE_KEY = "userPropertyOne";

    /**
     * Event attribute key name for the second user defined property
     * Attribute value is a Double
     */
    public static final String USER_PROPERTY_TWO_KEY = "userPropertyTwo";

    /**
      These are the keys and their order for elements that
      will be present in the XML representing ths object.
     */
    private static final String localSchemaKeys[] =
    {
        SESSION_ID, USER_ID, USER_PROPERTY_ONE_KEY, USER_PROPERTY_TWO_KEY
    };
```

```
    /**
     * Create a new TestTrackingEvent
     *
     * @param theSessionId from HttpSession.getId()
     * @param theUserId from HttpServletRequest.getRemoteUser() or equivalent
     * (null if unknown)
     * @param userPropertyOne some user defined property typed as a String
     * @param userPropertyTwo another user defined property typed as a Double
     */
    public TestTrackingEvent( String theSessionId,
                              String theUserId,
                              String userPropertyOneValue,
                              Double userPropertyTwoValue )
    {
        super( TYPE, theSessionId, theUserId, XML_NAMESPACE, XSD_FILE,
               localSchemaKeys );

        if( userPropertyOneValue != null )
            setAttribute( USER_PROPERTY_ONE_KEY, userPropertyOneValue );

       if( userPropertyTwoValue != null )
           setAttribute( USER_PROPERTY_TWO_KEY, userPropertyTwoValue );
    }
}
```

The `TestTrackingEvent`, shown in Listing 2-8, correctly sets its own attributes and sets the attributes in its instantiation of `TrackingEvent`. This enables correct population of the XML instance document at the time of its creation. Recall that the XML instance document represents the `TestTrackingEvent` in the database's `EVENT` table.

If you want the custom Behavior Tracking event type to be persisted in the database, the event must be added to the `behaviorTracking.persistToDatabase` property in the `application-config.xml` file. If you are not persisting the event, you do not need to add the event type to this property.

Installing Behavior Tracking Events

Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

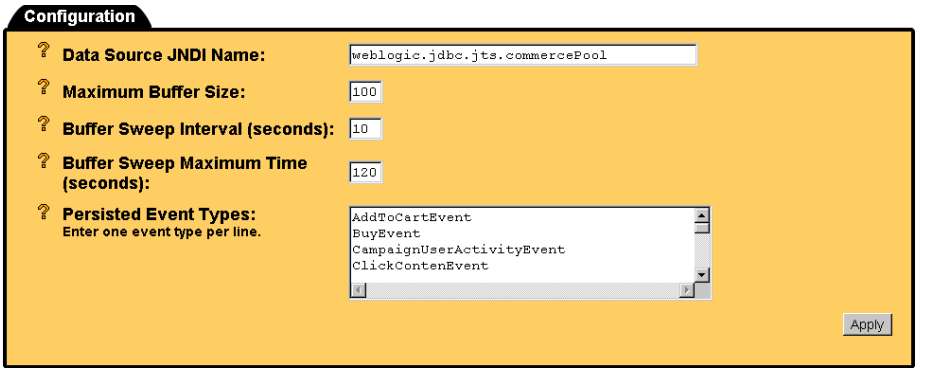
To install a Behavior Tracking Event listener, take the following steps:

1. In the WebLogic Server Console, navigate to Behavior Tracking in the node tree for wlcsDomain as follows:

   `http://hostname:port/console → wlcsDomain → Deployments → wlcsApp → Service Configurations → Behavior Tracking`

2. Enter the name of the event in the Persisted Event Types field, as shown in Figure 2-3.

Figure 2-3 WebLogic Server Administration Console—Behavior Tracking



XML Creation of Behavior Tracking Events

When persisting Behavior Tracking events to the `EVENT` table, the bulk of the data must be converted to XML. The XML document should conform to an XML XSD schema that you create which specifies the order of the XML elements in the XML instance document. Additionally, the schema must include the types of elements and their cardinalities. The process of creating XML from an event object is handled by a helper class that utilizes variables and constants in a Behavior Tracking event's class file. All schema documents use the namespace: "http://www.w3.org/2000/10/XMLSchema" and all instances of Behavior Tracking schemas use the namespace: "http://www.w3.org/2000/10/XMLSchema-instance". The XML created in Listing 2-9 will conform to the XSD schema.

Listing 2-9 XSD Document Example

```
<schema targetNamespace="http://www.bea.com/servers/wlcs3.5/xsd/tracking/buy"
    xmlns:bt="http://www.bea.com/servers/wlcs3.5/xsd/tracking/buy"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="BuyEvent">
            <complexType>
                <sequence>
                <element ref="bt:event_date"/>
                <element ref="bt:event_type"/>
                <element ref="bt:session_id"/>
                <element ref="bt:user_id" minOccurs="0"/>
                <element ref="bt:sku"/>
                <element ref="bt:quantity"/>
                <element ref="bt:unit_price"/>
                <element ref="bt:currency" minOccurs="0"/>
                <element ref="bt:application_name" minOccurs="0"/>
                <element ref="bt:order_line_id"/>
            </sequence>
        </complexType>
    </element>
    <element name="event_date" type="timeInstant"/>
    <element name="event_type" type="string"/>
    <element name="session_id" type="string"/>
    <element name="user_id" type="string"/>
    <element name="sku" type="string"/>
    <element name="quantity" type="double"/>
    <element name="unit_price" type="double"/>
    <element name="currency" type="string"/>
    <element name="application_name" type="string"/>
    <element name="order_line_id" type="long"/>
</schema>
```

Creation of an event's representation in XML takes place generically relative to the event's type. Consequently, to create an accurate XML instance document, each event must specify the namespace, event type, elements, and order of its elements. Using the `TestTrackingEvent` example, the XML representing an instance of the `TestTrackingEvent` is constructed as follows:

Note: Assume that `testTrackingEvent` is a well-formed instance of a `TestTrackingEvent`.

1. Get the event's type with the `testTrackingEvent.getType()` call.

2. Get the event's namespace with the `((TrackingEvent)testTrackingEvent).getXMLNamespace()`call.
3. Get the event's XSD filename with the `((TrackingEvent)testTrackingEvent).getXSDFile()` call.

Using the schema keys from the `TestTrackingEvent` class, values are inserted into the XML document. Schema key/attribute value pairs correspond to XML elements in this way:

```
<schema Key>value</schema Key>
```

The helper class that creates XML for Behavior Tracking assumes that the elements inserted into an XML instance document are not deeply nested. Additionally, the `toString()` method is used to create a representation of the value object that is retrieved through the `Event` classes's `getAttribute( String Key )` call. The contents of the string returned by invoking `toString()` on the value object must match the type specified in the event's schema document. The `TestTrackingEvent` retrieves values using the following keys in the order specified in the `schemaKeys` array:

- `SESSION_ID`
- `USER_ID`
- `USER_PROPERTY_ONE_KEY`
- `USER_PROPERTY_TWO_KEY`

The values for these keys are retrieved using the `testTrackingEvent.getAttribute( <schema Key> )` call. The order in which the XML formatted key/value pairs are inserted into the instance document is specified by the constant `schemaKeys` array, which is defined and populated in the `TestTrackingEvent` class.

The steps assembled to create an XML instance document for the `TestTrackingEvent` are presented in Listing 2-10.

Listing 2-10 XML Instance Document Example

```
<TestTrackingEvent
    xmlns="http://<your URI>/testtracking"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://<your URI>/testtracking
TestTrackingEvent.xsd"
```

```
    >
<event_date>XML time instant formatted event date</event_date>
<event_type>TestTrackingEvent</event_type>
     <session_id>theSessionIdValue</session_id>
     <user_id>theUserIdValue</user_id>
     <userPropertyOne>userPropertyOneValue</userPropertyOne>
     <userPropertyTwo>userPropertyTwoValue</userPropertyTwo>
</TestTrackingEvent>
```

The XML creation is performed automatically when events arrive at the `com.bea.p13n.tracking.listeners.BehaviorTrackingListener`, which enables Behavior Tracking in WebLogic Portal or WebLogic Personalization Server. The Behavior Tracking listener is installed by adding it to the `<EventService Listeners="...">` property in the `application-config.xml` file. For information on how to install a Behavior Tracking listener, see"Installing Behavior Tracking Events" on page 2-20.

You must be careful when defining the namespaces, XSD documents, and schema keys variables in custom Behavior Tracking event classes, especially if they will be persisted to the `EVENT` table. The method for creating and storing XML presented in this discussion exactly follows the variables and constants specified in the event class. You are free to develop other ways of creating and storing XML; this section is directed only at the process of persisting XML Behavior Tracking representations in the BEA `EVENT` table.

Note: The Event's date is retrieved using the `Event` class's `getTimeStamp()` call, which returns a Java primitive `long` typed value. That `long` must be converted into the type specified for the `event_date` element in the XSD schema document. The type in this case is time instant. Event date and event type the first two elements in all XML instance documents created through the `BehaviorTrackingListener`.

Custom Behavior Tracking Event Listeners

To create a custom Behavior Tracking listener, in addition to or instead of the default `BehaviorTrackingListener`, follow the example presented in "Writing a Custom Event Listener" on page 2-5. Add the new event types to the custom listener's `eventTypes` array (for example, `TestTrackingEvent`). A given listener can listen

for any number of event types that may or may not be Behavior Tracking events. The custom Behavior Tracking listener can be installed on either the synchronous or asynchronous side of the event service, whichever is appropriate.

Writing Custom Event Generators

Once events are created, you must set up a mechanism for generating events in the application. Events may be generated from pipeline components, input processors, JSP scriptlets, or JSP tags. Some Behavior Tracking events are generated from within WebLogic Portal or WebLogic Personalization Server software.

After determining the mechanism for generating events, tracking events can be sent to the event system using the `com.bea.p13n.tracking.TrackingEventHelper` class. This class defines helper methods that pass events to the event service. Listing 2-11 shows an example of passing the `TestTrackingEvent`.

Listing 2-11 Dispatching an Event

```
/*
 * Create the event
 */
Event theEvent = new TestTrackingEvent( "<some session id>",
                                        "<some user id> ",
                                        new String("userPropertyOneValue"),
                                        new Double( 3.14 ) );

/*
 * Dispatch the event
 */
TrackingEventHelper.dispatchEvent( theEvent );
```

To dispatch a `TestEvent` to the event service, the event service name can be looked up in the JNDI, and an instance of the `EventService` bean can be obtained by invoking the `create()` method on an `EventServiceHome` instance. The JNDI name of the `EventServiceHome` interface is the classname of the `EventServiceHome` class (`com.bea.p13n.events.EventServiceHome`). Listing 2-12 shows an example.

Listing 2-12 JNDI Example

```
import com.bea.p13n.util.helper.JNDIHelper;
import com.bea.p13n.events.Event;
import com.bea.p13n.events.EventServiceHome;
import com.bea.p13n.events.EventService;

import javax.ejb.CreateException;
import javax.rmi.PortableRemoteObject;

/* code here */

    public void demonstrateEventDispatch()
    {
        Event event = <some event instance>;

        try
        {
            EventServiceHome home = (EventServiceHome)
                JNDIHelper.lookup( "java:comp/env/ejb/EventService" ),
                    EventServiceHome.class );
            EventService eventService = home.create();
            eventService.dispatchEvent( event );
        }
        catch( Exception e )
        {
           /*
            Do exception handling here
           */
        }
    }
/* more code here */
```

Debugging the Event Service

To debug the event service, create a `debug.properties` file in the following directory:

`%PORTAL_HOME%\debug.properties` (Windows)

`$PORTAL_HOME/debug.properties` (UNIX)

The contents of this file are shown in Listing 2-13.

Listing 2-13 Debugging the Event Service

```
usePackageNames: on
com.bea.p13n.cache: on

# Turns on debug for all classes under events
com.bea.p13n.events: on
# com.bea.p13n.events.internal.EventServiceBean: on

# Turns on debug for all classes under
# com.bea.p13n.tracking: on
com.bea.p13n.tracking.internal persistence: on

# Selectively turn on classes
com.bea.p13n.mbeans.BehaviorTrackingListerner: on
com.bea.p13n.tracking.listeners.BehaviorTrackingListerner: on
com.bea.p13n.tracking.SessionEventListerner: on
```

Registering a Custom Event

When you create a custom event, you must register the event. Registering a custom event lets the E-Business Control Center know that the custom event exists. Registering permits campaign developers using the E-Business Control Center to create scenario actions that refer to the event. Registering also identifies the event's properties.

Caution: Whenever you change the event code, you must update the event registration. Conversely, whenever you change the event registration, you must also update the event code. A possible ramification of event modification is that the scenario actions that refer to the event's properties may need to be modified.

Note: You cannot change any of the standard events supplied with WebLogic Portal or WebLogic Personalization Server.

To register a custom event, use the Event Editor in the E-Business Control Center. Registering an event is actually creating a property set for the event. A step-by-step procedure is available in both the E-Business Control Center online help and "Creating and Managing Property Sets" in the *Guide to Building Personalized Applications.*

3 Persisting Behavioral Tracking Data

To record how online customers are interacting with your e-commerce site, you can record event information to a database. These kinds of events are called Behavior Tracking events. E-analytics and e-marketing systems can then analyze these events offline to evaluate customer behavior and transactional data. You can use the knowledge gained from analysis to create and optimize personalization rules, set up product offers, and develop interactive marketing campaigns. This section describes the requirements and database schema needed to log event data for analytical use.

This topic includes the following sections:

- Activating Behavior Tracking
- Data Storage
- Constraints and Indexes
- Scripts

Activating Behavior Tracking

Before Behavior Tracking events can be recorded to a database, you must enable the Behavior Tracking listener. This is accomplished by adding a class to the `application-config.xml` file.

Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

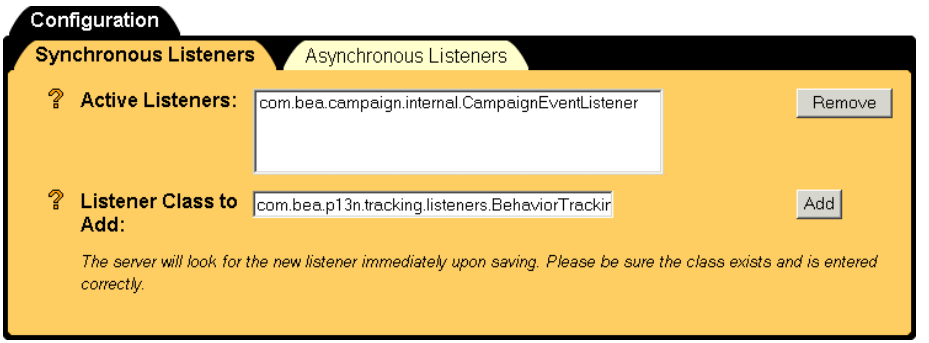
To add the Behavior Tracking listener, take the following steps:

1. In the WebLogic Server Console, navigate to Synchronous or Asynchronous Listeners tab in the node tree for wlcsDomain as follows:

   `http://hostname:port/console` → `wlcsDomain` → `Deployments` → `wlcsApp` → `Service Configurations` → `Event Service` → `Configuration Tab` → `Synchronous Listeners`

2. Add the Behavior Tracking listener (`com.bea.p13n.tracking.listeners.BehaviorTrackingListener`) to the Listen Class to Add field, and then click the Add button. See Figure 3-1.

Figure 3-1 WebLogic Server Administration Console—Event Service



Note: You must configure your database before activating Behavior Tracking. For information on how to do this, see “Production Environment Scenario” on page 3-15.

Event Properties

This section describes Behavior Tracking properties more fully and details the mechanism that persists Behavior Tracking event data to the database. Each Behavior Tracking event property described here can be configured in the WebLogic Server Administration Console. “Configuring the Behavior Tracking Service in WebLogic Server” on page 3-3 details how to set these properties.

As previously mentioned, Behavior Tracking events are placed in a buffer and then intermittently persisted to the Event tables in the database where they can be analyzed offline. An asynchronous service is used so that long-running event handlers can execute without delaying the application from a Web site visitor’s perspective.

The buffered Behavior Tracking events are swept into the database using a pool of data connections. The default Data Source is `weblogic.jdbc.jts.commercePool`. You can use a different Data Source. To do this, create and configure the new Data Source (see “Configuring a Data Source” on page 3-4) and substitute the name of the default Data Source with the name of the new Data Source in the WebLogic Server Administration Console.

The particular events that are persisted to the database are specified in the `behaviorTracking.persistToDatabase` property. You can view and alter the list of the persisted events in the WebLogic Server Administration Console. The types in this list must match the type specified in the event; for example, the `SessionBeginEvent` has as its type the string “`SessionBeginEvent`”.

The frequency of the sweeping of events from the buffer is controlled by the following properties in the `application-config.xml` file:

- `MaxBufferSize`
- `SweepInterval`
- `SweepMaxTime`

You should tune these properties to optimize performance. A buffer sweep should be performed often enough that writing to the database is not too time consuming but not so frequent that the operation is wasteful.

Configuring the Behavior Tracking Service in WebLogic Server

To configure the various Behavior Tracking properties, take these steps:

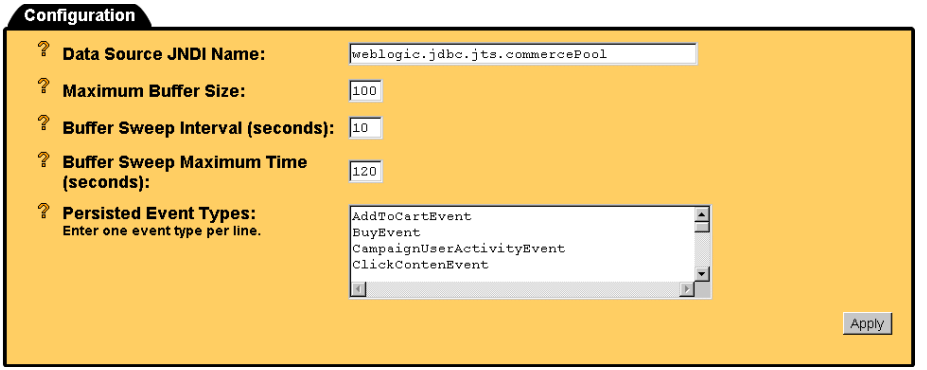
Warning: For proper operation, the WebLogic Server requires that changes to the `application-config.xml` file be made using the WebLogic Server Administration Console.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

1. In the WebLogic Server Console, navigate to the Behavior Tracking Service (shown in Figure 3-1) in the node tree for wlcsDomain, as follows:

   `http://hostname:port/console → wlcsDomain → Deployments → wlcsApp → Service Configurations → Behavior Tracking`

Figure 3-2 WebLogic Server Administration Console—Behavior Tracking Service



2. To change the Data Source, enter the fully-qualified name of the Data Source in the Data Source JNDI Name field.
3. To change the sweeping of events from the buffer, enter the new buffer values in the appropriate fields.
4. To specify whether a particular event is persisted, add or remove the event from the Persisted Event Types list box.

Configuring a Data Source

This section provides a brief description about configuring a new Data Source for a connection pool used for persisting events.

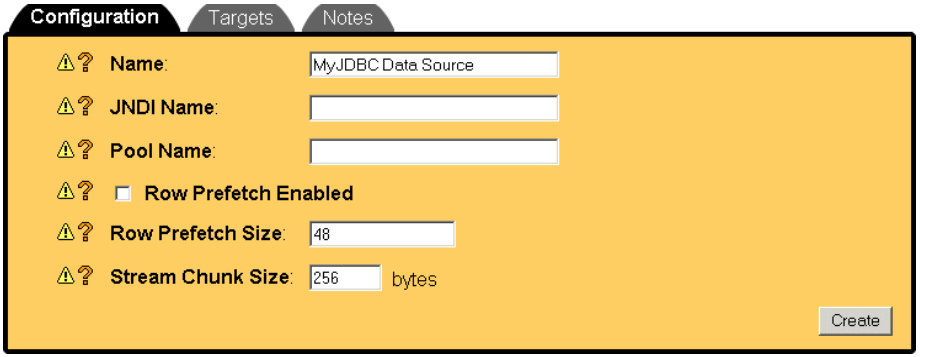
To configure a new Data Source, take the following steps.

Note: For more information on using the WebLogic Server Administration Console, see the WebLogic Server Documentation Center.

1. In the WebLogic Server Console, navigate to the Behavior Tracking Service (shown in Figure 3-1) in the node tree for wlcsDomain, as follows:

   `http://hostname:port/console → wlcsDomain → Services → JDBC → Data Sources → Behavior Tracking`

Figure 3-3 WebLogic Server Administration Console—JDBC Data Sources



2. In the right pane, click Configure a new JDBC Data Source.
3. Enter the appropriate values for the new Data Source in the appropriate tabs and fields.

Data Storage

This section provides an overview of relational databases and the database schemas and tables that are required for recording Behavior Tracking events.

Relational Databases

Relational databases have both logical and physical structures. Logically you may define one or more databases. Each database may contain one or more tables and indexes, and each table may have multiple columns and rows. The logical structure of databases is quite similar between vendors. However, the physical structure of a database is very vendor-specific. Essentially, the physical structure defines areas on disk drives where the data is stored. Each database environment uses its own terminology and implementation for storing data at the operating system level. For example, Oracle uses the term *tablespace* and the Microsoft SQL Server uses the term *filegroup*.

When a database structure is defined by a database administrator, attention must be paid to the location of specific tables. Some tables are static in that they do not change much; some tables are dynamic in that many rows are being added and deleted; and some tables are read frequently and some rarely. Depending on their behavior, tables should be placed on different physical locations. Some of the most highly-used tables in WebLogic Portal and WebLogic Personalization Server are used for Behavior Tracking. The activity of a single customer moving around your site may generate multiple table entries. Therefore, it is recommended that you place these tables on the fastest drives in the computer. Experienced database administrators are aware of many techniques for monitoring and configuring a database installation for optimal performance. If you do not have a database administrator working with your installation and you have a lot of activity on your site, you should bring in a well-qualified database administer for regular maintenance of your system.

Database Directory Paths

The default database directory paths are:

- `%WL_PORTAL_HOME%\db\<db vendor>\<db version>\...` (Windows)
- `$WL_PORTAL_HOME/db/<db vendor>/<db version>/...` (UNIX)

where `WL_PORTAL_HOME` is the directory in which you installed WebLogic Portal or WebLogic Personalization Server.

For example, if you are using Oracle 8.1.7 on UNIX, the location would be `$WL_PORTAL_HOME/db/oracle/817/....`

BEA provides scripts to help set up the database schema needed for recording Behavior Tracking events, as well as the schema needed for recording data associated with WebLogic Portal and WebLogic Personalization Server. This data includes information from orders, catalogs, products, portals, and portlets.

For Oracle databases, the tablespaces created for WebLogic Portal and WebLogic Personalization Server data are the `WLCS_DATA` and `WLCS_INDEX`.

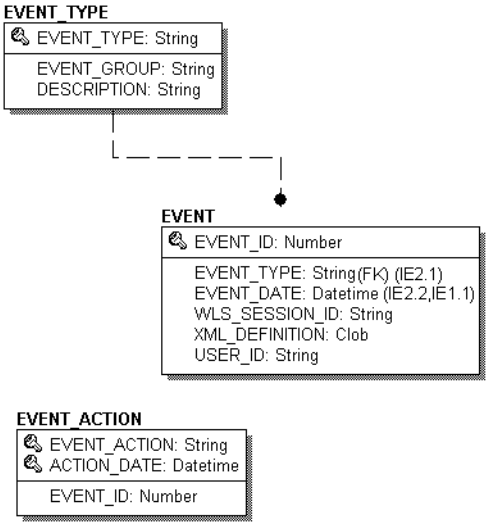
Note: `WLCS_DATA` and `WLCS_INDEX` are tablespace names created by BEA scripts. If you use a particular naming convention, you can rename them.

Behavior tracking uses a tablespace called `WLCS_EVENT_DATA`. This tablespace stores all Behavior Tracking tables, indexes, and constraints. Because of the potential for high volumes of data, this tablespace should be monitored closely.

Behavior Tracking Database Schema

Three tables are provided for the Behavior Tracking data. The `EVENT` table stores all event data. The `EVENT_ACTION` table logs actions used by third-party vendors against the recorded event data, and the `EVENT_TYPE` table references event types and categories in the `EVENT` table. Figure 3-4 shows a logical entity-relation diagram for the Behavior Tracking Database.

Figure 3-4 Entity-Relation Diagram for the Behavior Tracking Database



The EVENT Database Table

Table 3-1 describes the metadata for the EVENT table. This table stores all Behavior Tracking event data. It is an extremely active table.

See the section “Constraints and Indexes” on page 3-14 for information about the constraint defined for this table.

The Primary Key is `EVENT_ID`.

Table 3-1 The EVENT Table Metadata

Column Name	Data Type	Null Value	Description and Recommendations
`EVENT_ID`	`NUMBER`	`NOT NULL`	A unique, system-generated number used as the record ID. This field is the table’s primary key.
`EVENT_TYPE`	`VARCHAR(30)`	`NOT NULL`	A string identifier that shows which event was fired.
`EVENT_DATE`	`DATE`	`NOT NULL`	The date and time of the event.
`WLS_SESSION_ID`	`VARCHAR(254)`	`NOT NULL`	A unique, WebLogic Server-generated number assigned to the session.
`XML_DEFINITION`	`CLOB`	`NULL`	An XML document that contains pertinent event information. It is stored as a CLOB (Character Large Object).
`USER_ID`	`VARCHAR(50)`	`NULL`	The user ID associated with the session and event. If the user has not logged in this column will be null.

As shown in Table 3-1, the EVENT table has six columns; each column corresponds to a specific event element. Five of the EVENT table’s columns contain data common to every event type. The `XML_DEFINITION` column contains all information from these five columns plus event data that is unique to each event type. An XML document is created specifically for each event type. The data elements corresponding to each event type are captured in the `XML_DEFINITION` column of the EVENT table. These elements are listed in Table 3-2.

Table 3-2 XML_DEFINITION Data Elements

Event	Data Element
AddToCartEvent	event_date event_type session_id user_id sku quantity unit_list_price currency
BuyEvent	event_date event_type session_id user_id sku quantity unit_price currency application_name
CampaignUserActivityEvent	event_date event_type session_id user_id campaign_id
ClickCampaignEvent	event_date event_type session_id user_id document_type document_id campaign_id scenario_id application_name
ClickContentEvent	event_date event_type session_id user_id document_type

Table 3-2 XML_DEFINITION Data Elements (Continued)

Event	Data Element
ClickProductEvent	event_date event_type session_id user_id document_type document_id sku category_id application_name
DisplayCampaignEvent	event_date event_type session_id user_id document_type document_id campaign_id scenario_id application_name
DisplayContentEvent	event_date event_type session_id user_id document_type document_id
DisplayProductEvent	event_date event_type session_id user_id document_type document_id sku category_id application_name

Table 3-2 XML_DEFINITION Data Elements (Continued)

Event	Data Element
PurchaseCartEvent	session_id user_id event_date event_type total_price order_id currency application_name
RemoveFromCartEvent	event_date event_type session_id user_id sku quantity unit_price currency application_name
RuleEvent	event_date event_type session_id user_id ruleset_name rule_name
SessionBeginEvent	event_date event_type session_id user_id
SessionEndEvent	event_date event_type session_id user_id
SessionLoginEvent	event_date event_type session_id user_id

Table 3-2 XML_DEFINITION Data Elements (Continued)

Event	Data Element
UserRegistrationEvent	event_date event_type session_id user_id

The EVENT_ACTION Database Table

Table 3-3 describes the metadata for the EVENT_ACTION table. This table logs actions used by third-party vendors against the recorded event data. It is a fairly static.

The Primary Key is comprised of EVENT_ACTION and ACTION_DATE.

Table 3-3 EVENT_ACTION Table Metadata

Column Name	Data Type	Null Value	Description and Recommendations
EVENT_ACTION	VARCHAR(30)	NOT NULL	The event action taken such as BEGIN EXPORT or END EXPORT. This field is one of the table's primary keys.
ACTION_DATE	DATE	NOT NULL	The date and time of the event. This field is one of the table's primary keys.
EVENT_ID	NUMBER	NULL	The ID of the event that corresponds with the event action taken.

The EVENT_TYPE Database Table

Table 3-4 describes the metadata for the EVENT_TYPE table. This table references event types and categories in the EVENT table. This table is static.

See the section "Constraints and Indexes" on page 3-14 for information about the constraint defined for this table.

The Primary Key is EVENT_TYPE.

Table 3-4 EVENT_TYPE Table Metadata

Column Name	Data Type	Null Value	Description and Recommendations
EVENT_TYPE	VARCHAR(30)	NOT NULL	A unique, system-generated number used as the record ID. This field is the table's primary key.
EVENT_GROUP	VARCHAR(10)	NOT NULL	The event category group associated with the event type.
DESCRIPTION	VARCHAR(50)	NULL	A description of the EVENT_TYPE.

Note: To record custom events, you must create an entry in this table. If a custom event does not have a record in this table, you cannot persist it to the EVENT table.

Constraints and Indexes

There is a single foreign key constraint between the EVENT_TYPE columns in the EVENT and EVENT_TYPE tables. As previously mentioned, if a custom event does not have a record in the EVENT_TYPE table, it cannot be persisted to the EVENT table.

Other than Primary Keys on each of the tables, there are only two indexes on the EVENT table. One index is on the EVENT.EVENT_DATE column and the other index is comprised of the EVENT.EVENT_TYPE and EVENT.EVENT_DATE columns.

Scripts

BEA provides scripts to create the Behavior Tracking database schema and tables for Oracle databases. This section provides information about the structures used in both a development and a production environment.

Development Environment Scenario

In a development environment, you may not want or need separate databases or tablespaces for recording Behavior Tracking events from the databases or tablespaces used for WebLogic Portal and WebLogic Personalization Server. Accordingly, you can include the Behavior Tracking database objects along side the database objects of these products. The easiest way to accomplish this is to execute the `create_all` script found in the `event` directory of your database installation.

Log into Oracle using SQL*Plus and execute the `create_all.sql` script in this location:

```
%WL_PORTAL_HOME%/db/oracle/817/event/create_all.sql
```

where `WL_PORTAL_HOME` is the directory in which you installed WebLogic Portal or WebLogic Personalization Server.

The `create_all` scripts in the `event` subdirectory executes the following scripts:

- **`drop_event.sql`**: Drops all the Behavior Tracking database objects.
- **`create_event.sql`**: Creates all the Behavior Tracking database objects.
- **`insert_event_type.sql`**: Populates the `EVENT_TYPE` table with base data.

Production Environment Scenario

This scenario is intended for use in an Oracle production environment where multiple tablespaces and their corresponding elements, such as tables and indexes, can reside in separate tablespaces and potentially on a different database server than WebLogic Portal or WebLogic Personalization Server database objects.

Before enabling the Behavior Tracking events, complete the following steps:

1. Identify the server and database used for recording Behavior Tracking events.
2. In the `WL_PORTAL_HOME/db/oracle/817/event` directory where `WL_PORTAL_HOME` is the directory in which you installed the WebLogic Portal or WebLogic Personalization Server:
   a. Edit the `create_event_tablespaces.sql` script to properly define the tablespace path and data filenames.
   b. Execute the `create_event_tablespaces.sql` to create the tablespaces.
   c. Edit the `create_event_users.sql` to ensure the correct user account will be created when this script is executed (the account name by default is `WLCS_EVENT`).
   d. Execute the `create_event_users.sql`.
3. Using SQL*Plus, connect as the user defined in `create_event_users.sql` and execute the script `create_all.sql`. This script will call `drop_event.sql`, `create_event.sql`, and `insert_event_type.sql`.
4. Change your Data Source information to point to this host, database instance, and user account. For more information, see “Event Properties” on page 3-3.

Description of Each Script

The Oracle scripts are described in the following list:

- `WL_PORTAL_HOME/db/oracle/817/event/create_all.sql`

  Executes the following scripts: `drop_event.sql`, `create_event.sql`, and `insert_event_type.sql`.

- `WL_PORTAL_HOME/db/oracle/817/event/create_event.sql`

  Creates the tables, indexes, and constraints associated with Behavior Tracking events.

- `WL_PORTAL_HOME/db/oracle/817/event/create_event_tablespaces.sql`

  Creates tablespaces for storage of Behavior Tracking events information.

- `WL_PORTAL_HOME/db/oracle/817/event/create_event_users.sql`

Creates the WLCS_EVENT database user and grants the appropriate privileges for working with the Behavior Tracking event tables.

- WL_PORTAL_HOME/db/oracle/817/event/drop_event.sql

  Drops the Behavior Tracking event tables.

- WL_PORTAL_HOME/db/oracle/817/event/insert_event_type.sql

  Populates the EVENT_TYPE table with base data.

4 JSP Tag Library Reference for Events and Behavior Tracking

This tag library contains several tag extensions used in the BEA WebLogic Portal and BEA WebLogic Personalization Server. Tags in this library are specifically used in the Events and Behavior Tracking component of the server.

The Events and Behavior Tracking tags allow you specify user behavior that you are interested in monitoring as users navigate across your site pages. These tags cause events to be generated which may be subsequently analyzed by third-party analytical tools.

The Events and Behavior Tracking tags are divided into two general areas: content tracking and product tracking. Content and product tracking tags can be used in any personalization or commerce application.

This topic includes the following sections:

- Content
  <tr:clickContentEvent>
  <tr:displayContentEvent>
- Product
  <trp:clickProductEvent>
  <trp:displayProductEvent>

Note: The `<tr:>` prefix means "track."
The `<trp:>` prefix means "track-product."

Content

Use the following code to import the content events tag library:

```
<%@ taglib uri="tracking.tld" prefix="tr" %>
```

Note: In the following tables, the Required column specifies if the attribute is required (yes) or optional (no). In the R/C column, C means that the attribute is a Compile time expression, and R means that the attribute can be either a Request time expression or a Compile time expression.

<tr:clickContentEvent>

The <tr:clickContentEvent> tag (Table 4-1) is used to generate a behavior event when a user has clicked (through) on an ad impression. This tag will return a URL query string containing event parameters. It is then used when forming the complete URL that hyperlinks the content.

Use the following code to import the content events tag library:

```
<%@ taglib uri="tracking.tld" prefix="tr" %>
```

Table 4-1 <tr:clickContentEvent>

Tag Attribute	Req'd	Type	Description	R/C
documentId	No	String	ID of the item that is displayed, if applicable (that is, an image URL or banner ad ID).	R
documentType	No	String	Type or category of the item that is displayed (if applicable).	R
id	No	String	Page variable which will hold the output of this tag.	C
userId	No	String	Name of the user that content was retrieved for. If the optional value is not provided, it will be set to the value of the request.getRemoteUser().	R

Example

The example below demonstrates a clickthrough example going to the Webflow servlet. This link will cause a clickthrough content event to be generated and also display the indicated content. The example shows how to generate a click content event after the user clicks a product description link. The default Webflow servlet's <filter> tag, specified in the application's web.xml file, generates a call to the ClickThroughEventFilter.doFilter() method. This method checks for ClickThroughConstants.EVENT_TYPE in the HttpRequest, and then fires the click event if it is present.

The ClickThroughConstants.EVENT_TYPE is generated by adding the <tr:clickContentEvent> tag in the JSP, as shown below:

```
<tr:clickContentEvent documentId="<%= documentId %>"
  documentType="<%= documentType %>"
  userId="<%= userId %>"
  id="outputFromTag"
/>
```

The following associates the desired content with a link that references the output from the above tag.

```
<A HREF="<webflow:createWebflowURL event="link.clickContent"
namespace="trackingWebApp_main" extraParams="<%= outputFromTag %>"
/>">Click Here to generate the clickContentEvent.</A>
```

Note: To redirect the user to another site, use `redirect="true"` in the `createWebflowURL` tag.

<tr:displayContentEvent>

The <tr:displayContentEvent> tag (Table 4-2) is used to generate a behavior event when a user has received (viewed) an ad impression, (typically a gif image).

Use the following code to import the content events tag library:

```
<%@ taglib uri="tracking.tld" prefix="tr" %>
```

Table 4-2 <tr:displayContentEvent>

Tag Attribute	Req'd	Type	Description	R/C
documentId	No	String	ID of the item that is displayed, if applicable (that is, an image URL or banner ad ID).	R
documentType	No	String	Type or category of the item that is displayed (if applicable).	R

Example

The example below shows a code snippet of processing that would follow a <cm:select> call. For each document returned but not displayed in this example, the <tr:displayContentEvent> tag generates an event and passes the document's ID and type.

```
<%@ taglib uri="tracking.tld" prefix="tr" %>
.
.
.
<es:forEachInArray id="nextRow" array="<%=headlines%>"
  type="com.bea.p13n.content.Content">
  <es:notNull item="<%=nextRow%>">
    <tr:displayContentEvent
      documentId="<%=nextRow.getIdentifier()%>"
      documentType="<%=headingProp%>"/>
  </es:notNull>
</es:forEachInArray>
```

Product

Use the following code to import the product events tag library:
```
<%@ taglib uri="productTracking.tld" prefix="trp" %>
```

Note: In the following tables, the Required column specifies if the attribute is required (yes) or optional (no). In the R/C column, C means that the attribute is a Compile time expression, and R means that the attribute can be either a Request time expression or a Compile time expression.

<trp:clickProductEvent>

The `<trp:clickProductEvent>` tag (Table 4-3) is used to generate a behavior event when a user has clicked (through) on a product impression. This tag will return a URL query string containing event parameters. It is then used when forming the complete URL that hyperlinks the content.

At least one of `sku`, `categoryId`, or `documentId` is required.

Use the following code to import the product events tag library:
```
<%@ taglib uri="productTracking.tld" prefix="trp" %>
```

Table 4-3 <trp:clickProductEvent>

Tag Attribute	Req'd	Type	Description	R/C
`applicationName`	No	String	The webApp or application name, if applicable. Can be used to separate data when multiple storefronts are hosted on the same server (or persisted to the same database).	R
`categoryId`	No	String or Category object	Category of the product associated with the content displayed, if applicable.	R
`documentId`	Yes	String	Name of the item that is displayed, if applicable (that is, an image URL or banner ad ID).	R

Table 4-3 <trp:clickProductEvent> (Continued)

Tag Attribute	Req'd	Type	Description	R/C
documentType	No	String	Type or category of the item that is displayed (if applicable).	R
sku	No	String or ProductItem object	ID of the product associated with the content item that is displayed, if applicable.	R
userId	No	String	Name of the user that content was retrieved for. If the optional value is not provided, it will be set to the value of the request.getRemoteUser().	R

Example

The example below demonstrates a clickthrough example going to the Webflow servlet. This link will cause a clickthrough content event to be generated and also display the indicated content. This example shows how to generate a ClickProductEvent having a document ID using the product name (productItem.getName()) and SKU of the product's identifier.

```
<%@ taglib uri="productTracking.tld" prefix="trp" %>
.
.
.
<%
detailsUrl = WebflowJSPHelper.createWebflowURL(pageContext,
"itemsummary.jsp", "link(" + detailsLink + ")",
"&" + HttpRequestConstants.CATALOG_ITEM_SKU + "=" +
productItem.getKey().getIdentifier() + "&" +
HttpRequestConstants.CATALOG_CATEGORY_ID + "=" +
category.getKey().getIdentifier() + "&" +
HttpRequestConstants.DOCUMENT_TYPE + "=" + detailsLink, true);
%>
<trp:clickProductEvent
   id="url"
   documentId="<%= productItem.getName() %>"
   sku="<%= productItem.getKey().getIdentifier() %>" />
<%
```

```
detailsUrl = detailsUrl + "&" + url;
%>
<a href="<%= detailsUrl %>">
```

<trp:displayProductEvent>

The `<trp:displayProductEvent>` tag (Table 4-4) is used to generate a behavior event when a user has received (viewed) a product impression, (typically a gif image).

At least one of `sku`, `categoryId`, or `documentId` is required.

Use the following code to import the product events tag library:

```
<%@ taglib uri="productTracking.tld" prefix="trp" %>
```

Table 4-4 <trp:displayProductEvent>

Tag Attribute	Req'd	Type	Description	R/C
`applicationName`	No	String	The webApp or application name, if applicable. Can be used to separate data when multiple storefronts are hosted on the same server (or persisted to the same database).	R
`categoryId`	No	String or Category object	Category of the product associated with the content displayed, if applicable.	R
`documentId`	No	String	Name of the item that is displayed, if applicable (that is, an image URL or banner ad ID).	R

`documentType`	No	String	Type or category of the item that is displayed (if applicable).
			`Suggestions:`
			`DisplayProductEvent.CATEGORY_BROWSE`
			`DisplayProductEvent.ITEM_BROWSE`
			`DisplayProductEvent.CATEGORY_VIEW`
			`DisplayProductEvent.BANNER_AD_PROMOTION`
				R

`sku`	No	String or ProductItem object	ID of the product associated with the content item that is displayed, if applicable.	R

Example

The example below shows a code snippet of processing that would follow the retrieval of a catalog item. The `<tr:displayProductEvent>` tag generates an event and passes the document's ID, type and SKU number of the product item.

```
<%@ taglib uri="productTracking.tld" prefix="trp" %>
.
.
.
<trp:displayProductEvent
    documentId="<%= item.getName() %>"
    documentType="<%= DisplayProductEvent.ITEM_BROWSE %>"
    sku="<%= item.getKey().getIdentifier() %>" />
```

Index

F

H

J

L