Using the WebSphere Business Events REST API to define event processing logic

James M. Siddle
Software Engineer
IBM China

Stephen Godwin (stephen.godwin@uk.ibm.com)
Software Engineer
IBM China

08 September 2010

This article describes how to use the REST API in WebSphere® Business Events V7 to define event processing logic. It covers the REST resources, URIs, and representations needed to define event processing logic, and provides sample Java™ code.

REST requests and responses

REST provides simple, scalable APIs by reusing the existing capabilities of the HTTP protocol. REST requests are either PUT, POST, GET or DELETE requests--the standard HTTP protocol methods. You may be familiar with the GET and POST methods, which are used by web browsers to retrieve web pages and submit forms. DELETE and PUT are not commonly seen outside of REST. REST requests are sent to specific URIs, for example: `http://myhost/customer/id/12345`. The URI represents a resource, in this case, a specific customer.

Sending a GET request retrieves a representation of the resource. Typically XML and JSON formats are supported. You can specify the format you want in the request's `Accept` header. The GET request's body is normally left empty. In large systems where scalability is important, GET requests can be cached. Careful use of the caching-related headers allows excellent scaling of REST APIs.

POST and PUT requests are used to create and update resources. When creating a resource, you typically use POST if the server will generate the URI of the new resource and PUT if the request URI identifies the new resource. The request body usually contains a representation of the resource to be created. The response body provides information about the result; for example, a new resource identifier or validity information.

A DELETE request removes the specified resource. DELETE requests don't require a body, and information such as success or error data are expected in the response.

For all types of requests, REST uses an HTTP status code in the response to indicate success or failure. A code in the `2xx` class, such as 201, indicates success. A code in the `4xx` class, such as 400 indicates an error.

Introduction

WebSphere Business Events V7 (hereafter called Business Events) is a REST-based API for defining event processing logic. This article describes how to use the new interface to define and publish simple event processing logic through REST resources such as projects, assets, and the repository. A sample Java™ application is used to show example requests and responses in action, and the Business Events Tester (also new in V7) is used to test event processing logic defined using the REST API.

REST is the architectural style of the worldwide web, where resources (such as web pages) are accessed through a small set of well-defined operations. The specific representation of returned resources is determined through *content negotiation*, in which a client indicates its preferred formats and the server attempts to fulfill the client's preferences. REST has become popular for web services because it's easy to understand and use, and is well supported by existing tools and technologies. The typical mapping of REST concepts to the HTTP standard, which underpins the worldwide web, is described in REST requests and responses.

The Business Events REST API exposes resources to allow clients to define event processing logic using hosted projects. This article describes the use of key resources, including projects (which are containers for assets), assets (such as filters and interaction blocks), and the repository (the container for the Business Events runtime application). We'll describe URIs, HTTP operations, and formats for each resource, in an order that matches the typical flow of requests and responses in the REST API.

To demonstrate the REST API, a sample application is used, based on the Java programming language and the JSON4J library. To test event processing logic defined using the sample application, the Business Event Tester (BET) is used. The BET is a new feature of Business Events V7 that you can use to inject events, examine runtime filter evaluations, and see the resulting actions.

Set up your environment

Before going any further, let's set up your environment so you can use the sample application and Business Events Tester. If you just want to quickly get an understanding of the REST API, you may want to skip ahead to the Define event processing logic section of this article.

Download and compile the sample application

First, you'll need a copy of the Eclipse IDE installed. The sample application was written using Rational® Application Developer V7.5.4 (built on Eclipse), but is packaged as a standard Eclipse Java project which should work in any recent version of Eclipse with the Java Development Toolkit (JDT). It is known to work, for example, in Eclipse 3.2. Check the Resources section for more information on obtaining a copy of Eclipse.

Next, download and import the sample application into Eclipse. The download is included in the Downloads section of this article. To import the sample application, extract the archive contents directly into your Eclipse workspace, then import the project into your workspace by selecting

File => Import => Existing Projects into Workspace. Browse to the workspace directory, select **WBERestClient** from the list of projects to import, then click **Finish**.

The project should now be available in Eclipse, so the next step is to make sure it compiles. The Java code requires a compiler compliance level of 5.0. To ensure this is set correctly, select the **Window => Preferences**, then select **Java / Compiler** in the options tree, and choose **5.0** for **Compiler compliance level**.

To run the sample Java application, the code requires the Java library JSON4J.jar, which is installed in the director\lib folder in the Business Events installation folder. You are permitted to make and use a copy of the JSON4J library that is included with Business Events. Such use of that library is governed by the terms and conditions of the IBM International Program License Agreement and the applicable License Information document held by you (or your organisation) for the Business Events product.

Make sure a copy of JSON4J.jar is in the root folder of the WBERestClient project, and is available in the project build path. To see the project build path, right-click on the **WBERestClient** project, select **Build Path => Configure Build Path**, and then open the **Libraries** tab.

The WBERestClient should now compile correctly.

Using the sample application

The sample application demonstrates how to define event processing logic, and specific usage examples can be found throughout this article. However, those examples require the application to be running, so the next step is to run the application and issue a simple request.

To run the application, find the WBERestClient.java source file in the **Package Explorer** view of the Java perspective. Right-click on the file, then select **Run As => Java Application**. The application will start, using the Eclipse console view for input and output.

You should be prompted for the host and port of the Business Events system you want to access, along with security credentials. If you don't know what these are, contact your Business Events administrator. After providing valid host, port, and security credentials, you should see a list of options like this:

```
*****************************************************
* WebSphere Business Events - example REST API client *
*****************************************************

|--- Choose an option ---|
Target host/port: localhost:9080

Project:    (p1) Create/update project
            (p2) Read project information
            (p3) Upload project assets
            (p4) Read project asset summary
            (p5) Create a filter
            (p6) Create an interaction block
            (p7) Check project validity
            (p8) Download project to file
```

```
Repository: (r1) Publish project file
            (r2) Read repository asset summary
            (r3) Clear repository

General:    (b) Read WBE build level
            (t) Set target WBE system
            (q) Quit
```

You're now ready to try sending a request to Business Events using the REST API. We'll start with something simple: obtaining the Business Events build level. The build level describes the specific software build of the Business Events system, and can be obtained via the following URI:

```
http://{host}:{port}/wbe/rest/r1/build
```

where *{host}* and *{port}* are the hostname and HTTP port number for your copy of Business Events.

To read the build level, the application issues an HTTP GET request to the above URI. The Business Events REST API should respond with HTTP code 200, indicating success, along with the build information that the application should display, as follows:

```
Received HTTP response: 200 OK
REST API responded with:
   Build name:    IBM WebSphere Business Events 7.0.0.0
   Build version: 20091120_2024
   Build date:    November 20 2009
Hit enter to continue...
```

When you hit the Enter key, the sample application should return to the list of options.

Accessing REST resources through a web browser

You may also want to try entering the build resource URI into a browser, which uses HTTP GET requests to retrieve web pages. The response won't include HTML content, however, so your web browser is unlikely to know how to display it.

Firefox®, for example, prompts the user to choose an application to open the returned data with. In this case, you can select a text editor to view the data that is returned.

The sample project

The sample application is based on a Business Events project that was developed for this article. You'll find the project in demoProjects.xml, available in the imported Java project under the projects folder.

The project contains several pre-defined events, intermediate objects, actions, and filters for a simple retail scenario. These assets will be used when defining event processing logic. Note that the project does not contain interaction blocks; the article will describe how to define and run an interaction block using the REST API and the Business Events Tester.

The projects folder also contains a project file named invalidProject.xml that we'll use to demonstrate an error-handling feature of the REST API.

Define event processing logic

In this section, we'll walk through the steps to define the event processing logic.

Step 1. Create a hosted project

The first step when defining event processing logic is to create a hosted project. A hosted project is created by issuing a PUT request to a URI such as:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}
```

Again, *{host}* and *{port}* are the hostname and HTTP port number of the installed copy of Business Events. The remaining elements of the URI, separated by the `/` character, are described in Table 1.

Table 1. URI element description

Element	Description
`wbe`	Indicates that the URI is targeting WebSphere Business Events...
`rest`	…and is targeting a resource published via the REST API.
`r1`	Revision 1 of the REST API is required
`dirs`	A directory is being accessed
`{dirName}`	The name of the directory being accessed
`projects`	A project within the above directory is being accessed
`{projName}`	The name of the project being accessed

The PUT request either creates a new project, or updates an existing project with the given name and in the given directory, if one exists. Note that directories are created implicitly during project creation.

Whether creating or updating, the request must provide a representation of the new resource in the body of the HTTP request. This is a crucial aspect of REST--the passing of representations of resources. The sample application sends the following:

```
{
   "owner":"jeremyJones",
   "privacy":"public"
}
```

The resource representation is formatted using JavaScript Object Notation (JSON), which is a lightweight data format often used in web 2.0 applications. Most resources in the Business Events REST API are represented using JSON, though in some cases XML is also supported. In this article, all requests use JSON unless otherwise stated. The JSON data above has been reformatted for this article because the raw data is difficult to read. See Resources for information on how to reformat your own JSON data.

The `owner` field indicates who owns the project, while the `privacy` field indicates who else is allowed to access the project.

The PUT request must tell the REST API which data format the resource representation is encoded using, and can also request a particular format for any response data. HTTP headers are used to achieve this, as shown here:

```
Content-Type: application/json
Accept: application/json
```

To issue a project creation request from the sample application, select option **p1**.

The full HTTP request, including other standard HTTP headers (the description of which is beyond the scope of this article) looks like this:

```
PUT /wbe/rest/r1/dirs/demoDirectory/projects/demoProject HTTP/1.1
Authorization: Basic 98jd2j09kd09kkd3k09d3dj898jd==
Content-Type: application/json
Accept: application/json
User-Agent: Java/1.6.0
Host: localhost
Connection: keep-alive
Content-Length: 39

{"owner":"jeremyJones","privacy":"public"}
```

The first time the request is issued, the REST API responds with:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=UTF-8
Content-Language: en-GB
Content-Length: 66
Set-Cookie: <snip – cookie data omitted for clarity>
Date: Thu, 21 Jan 2010 13:36:36 GMT
Server: WebSphere Application Server/7.0
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Cache-Control: no-cache="set-cookie, set-cookie2"

{"name":"demoProject","id":"44dc32d6-a1ba-4fee-9ee4-48ef58005f19"}
```

The response code `201` indicates that a new project resource was created. The other standard HTTP headers provide further information about the result, including the type and length of the response body. The response body includes the project name and identifier, for reference.

Further PUT requests to the project URI will prompt an `HTTP/1.1 204 No Content` response, which indicates that the request was successful but that no response data was sent. Putting a new project representation to the URI for an existing project causes the project to be updated.

Having created a hosted project, we can now read a representation of the project back from the REST API by issuing a GET request to same URI we used to create the project. The HTTP response body will contain a representation of the project that was created. Selecting option **p2** in the sample application will display specific fields from the returned response:

```
Reading project from: http://localhost:9080/wbe/rest/r1/dirs/demoDirectory/projects/
demoProject
Received HTTP response: 200 OK
REST API responded with:
   Name          :    demoProject
   Directory     :    demoDirectory
   Owner         :    jeremyJones
   Privacy       :    public
   Creation time :    2010-01-21T13:36:36Z
   Project ID    :    44dc32d6-a1ba-4fee-9ee4-48ef58005f19
```

In addition to the owner and privacy that we set during project creation, the project representation includes the name, directory, creation time, and project identifier. The additional fields provide a full picture of the project. You'll notice that we didn't send all of the above fields during creation--resource representations sent during creation or update requests do not necessarily have to provide values for optional fields.

See Resources to learn more about the Projects resource, along with all the other REST resources discussed in the following sections.

Step 2. Load assets into the hosted project

Having created a hosted project, we now need to populate it with assets to use as building blocks for the event processing logic. We'll upload a pre-created Business Events project file to achieve this. This project matches the recommended Business Events workflow, where building block assets such as events, actions, and intermediate objects are defined using Design Data then used by business users in the Design tool.

To uploaded the project file we'll issue a PUT request similar to Step 1 and we'll use the same URI:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}.
```

However rather than sending JSON, we'll set the content type as follows:

```
Content-Type: application/x-wbe+xml
```

This tells the REST API that the request body is formatted using the Business Events internal project file format, and contains a collection of assets rather than higher level project definition details as we saw previously. The content type `application/xml` is avoided by the REST API here because the Business Events file format should not be accessed directly by users; it's used for uploading and downloading complete project files that can be opened using the Event Flow and Design Data tools.

Select option **p3** in the sample application to upload a project. You'll be prompted to select a file from the local file system using a Java dialog. Select demoProject.xml under the **Projects** folder in your imported Eclipse project.

The sample application should display something like this:

```
Uploading project file to : http://localhost:9080/wbe/rest/r1/dirs/demoDirectory/
projects/demoProject
Received HTTP response: 204 No Content
Hit enter to continue...
```

The `204 No Content` response code indicates that the upload was successful, so we're now set to work with a collection of assets in the project.

Step 3. Read an asset list and summary

Before defining event processing logic, it's a good idea to check that the hosted project contains the assets you expect. The REST API supports extensive querying of assets through the **assets** resource, which is a sub-resource of both hosted projects and the runtime repository (which we'll come to later). That is, both hosted projects and the runtime repository contain assets, and you can issue requests directly related to those assets.

Using the sample application, select option **p4**. The application will issue a GET request to read information from the REST API, targeting a URI such as:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}/assets?
attribs=(name,guid,typeName)&sortBy=(typeName,name)
```

This URI is more complex than we've seen so far. Let's break it down to understand the specific request that is being made. After the HTTP scheme, host, and port, we see that a project is being targeted, just as with project creation and asset uploading. Then we see `/assets`, which indicates that the URI is targeting project assets. In this case no specific asset identifier is given, so the URI targets all assets in the project. The URI then has a query string, consisting of two distinct query parameters:

```
attribs=(name,guid,typeName)
sortBy=(typeName,name)
```

The first parameter tells the API which asset attributes should be returned for the query; in this case the name, globally unique identifier, and a human readable asset type are being requested. The second parameter tells the API how to sort the response data. The human readable type and asset name are used here, meaning that assets of the same type will appear together, in alphabetical order.

The REST API responds with a JSON array, containing one object per asset. Each object contains the attribute fields we requested, for a particular asset, for example:

```
[
{
"typeName":"Action",
"guid":"5D1B5506_0AB4_4EB6_BACF_0E80DEDE0E6F",
"name":"GenerateLead"
},
{
"typeName":"Action",
"guid":"0FF2EDB7_E50C_4C60_AA49_F46216E117F1",
"name":"SendVouchers"
},
   …
]
```

The sample application will display the data that is returned (but in an easier to read form), will prompt you to select an asset to read more information for, and will then retrieve the asset you selected by issuing a GET request to a URI, such as:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}/assets/{guid}
```

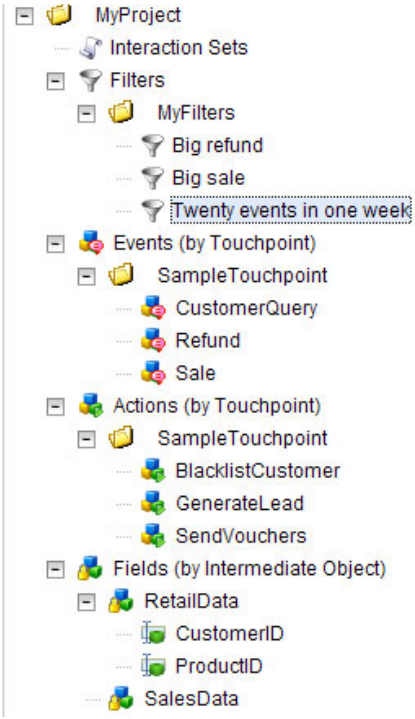
You'll notice that the URI again targets project assets, but this time terminates in a unique identifier for the asset, so this URI targets a specific asset. The following JSON fragment shows common attributes you can expect when retrieving an asset, in this case for the `GenerateLead` action that we saw above. Note that you may see the fields in a different order, and each asset will contain fields specific to that asset type.

```
{
   "action":{
      "name":"GenerateLead",
      "id":"5D1B5506_0AB4_4EB6_BACF_0E80DEDE0E6F",
      "modifytime":"2010-01-08T16:08:25Z",
      "createtime":"2010-01-08T15:51:03Z",
      "touchpoint":"SampleTouchpoint",
      "engine":"Director",
      "state":"MOD_PROJECT_ONLY",
      "enabled":"true",
      <snip>
   }
}
```

After receiving the asset, the sample application parses the JSON data, and then displays the common attributes for the asset.

The Business Events Design tool, which is a good example of what's possible through the REST API, uses the asset list query to retrieve a list of assets and associated data. Figure 1 shows the asset list displayed for the sample Business Events project:

Figure 1. Design tool asset list



The Design tool also reads assets using this mechanism. The following figure shows how the Design tool displays a filter asset:

Figure 2. Design tool filter asset

Step 4. Create a filter

We're now ready to define event processing logic, starting by creating a filter definition. A filter is a boolean expression that is used to work out whether actions should be triggered when events are received by Business Events.

A filter is created by issuing a POST request to the assets resource, sending the filter representation in the request body. The URI for the assets resource looks like this:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}/assets
```

You'll notice that this is the same URI that we used for reading a list of assets, minus the query parameters.

The filter definition is sent in the request body, in JSON. It should contain standard asset attributes, as seen above for the `GenerateLead` action, though some attributes (such as the creation time and

GUID) will be calculated by the REST API, so don't be surprised if the values change if you provide them all. A filter definition is quite complex, so we won't go into detail here. However you may want to examine the `ResourceBuilder` class in the sample application for an example of how to build a filter. For more information on REST resource schemas, check out Resources.

Option **p5** in the sample application builds a new filter that checks for a certain number of events within a given timeframe. You'll be prompted to enter a number of days, a number of events, and a name for the filter. The application will then create a filter that evaluates to true if more than that number of events has been received within the number of days you specified. Later steps in this article are based on a specific example, so you may want to enter 4 days, 4 events, and the name `Five events in four days` at this point.

Make sure you've issued a request to create a filter. You should see something like this:

```
Received HTTP response: 201 Created
REST API responded with:
   Asset Identifier         : 22B982EB4A4F500C1011DF94D260E85A
   Valid                    : true
   Validity string          : This asset is valid.
   Project validity changed : false
Hit enter to continue...
```

The response to the HTTP request contains information describing the result, which the sample application parses and displays. The asset identifier is the GUID of the new filter, and can now be used to access the asset; the other fields indicate that the asset is valid, and that the validity of the project overall hasn't changed.

It's a good idea to check the response data when creating your own assets; but don't be too concerned if assets appear as invalid. The important thing is to check the overall project validity before trying to publish it to the repository, which we'll do later.

Step 5. Create an interaction block

Next, we'll create an interaction block using the filter we just created. We'll build the definition up in several stages, because interaction blocks refer to events, actions, filters, and intermediate object fields, depending on the required logic. Later steps in this article are based on a specific interaction block example, so to aid understanding you may want to build the example using the `CustomerQuery` and `GenerateLead` event and action, and using the `Five events in four days` filter defined in the previous step.

The sample application builds an interaction block that triggers an action for an event, assuming a particular filter evaluates to true. The application queries the hosted project to determine events, actions, and filters that can be used in the interaction block, and then issues a POST to create the new asset, as we saw with the filter definition.

Each of the queries is made to the assets resource, similar to the asset list query performed at Step 3. In this case, however, the queries are modified with a filter query parameter. Not to be confused with Business Events filter definitions, filter query parameters are used to limit the set of results returned by a query for a list of items.

Select option **p6** in the sample application, and you should see something similar to the following:

```
Reading event list from project: http://localhost:9080/wbe/rest/r1/dirs/demoDirectory/
projects/demoProject/assets?filter=(type=10)
Received HTTP response: 200 OK
Select an event for your interaction block:
   Idx Name
   0   CustomerQuery
   1   Refund
   2   Sale
Choose an asset:
```

Here, the sample application has obtained a list of events from the hosted project, and is prompting a selection. The key thing to note is the filter query parameter `filter=(type=10)`, which restricts the list of assets in the result set to be events only.

Select an event by typing in a number, and the sample application will prompt you to select a filter:

```
Reading filters for event: http://localhost:9080/wbe/rest/r1/dirs/demoDirectory/
projects/demoProject/assets?filter=
(filtersForEvent='6E5C139A_26B2_475D_82F3_CBB19356AB9D')
Received HTTP response: 200 OK
Select a filter to check in your interaction block:
   Idx Name
   0   Twenty events in one week
   1   Five events in four days
Choose an asset:
```

Again the key thing to note is the filter query parameter:
`filter=(filtersForEvent='6E5C139A_26B2_475D_82F3_CBB19356AB9D')`

Here, the sample application is restricting the results to filter assets that can be used with the given event; the unique ID `6E5C139A_26B2_475D_82F3_CBB19356AB9D` refers to the event `CustomerQuery`. This step prevents invalid filters from being evaluated when the selected event is received. If you selected the `Refund` or `Sale` events, more filters will be available because those events construct intermediate object fields that the filters require; as such the `Big Sale` and `Big Refund` filters can be used when those events are received.

Select your filter, and the sample application will move on to action selection. You should see similar output as for the previous step, but again the filter query parameter will differ:
`filter=(actionsForEvent='6E5C139A_26B2_475D_82F3_CBB19356AB9D')`

Similar to Business Event filters, the set of actions that can be generated as a result of a particular event is limited; by using the `actionsForEvent` filter query parameter, a list of viable actions is returned.

Select an action, and then enter a name for your interaction block. The sample application will now create a new interaction block definition based on your inputs. As for filter definitions, interaction block definitions are quite complex, so you should refer to the `ResourceBuilder` class and Resources section to see how to build one.

One additional thing to note though is that the sample application creates an interaction block with the `RetailData.CustomerID` context relationship. See the Context relationships for more information.

Context relationships

A context relationship defines the scope of event processing logic, and influences the events that are visible to filters during evaluation, among other things. The sample application creates an interaction block with the `RetailData.CustomerID` context relationship, which is defined in the sample project uploaded during Step 2. This means that the evaluation of the filter we created in Step 5 only counts events previously received for the customer to which the current event relates.

As a more concrete example, consider an interaction block that triggers the `GenerateLead` event when three `CustomerQuery` events are received within two days. By using the `RetailData.CustomerID` context relationship, the system will generate leads for a particular customer based on that customer's activity rather than on every customer's activity.

Step 6. Check project validity

We've now successfully defined a filter and an interaction block in the hosted project, so we're almost ready to publish and try out the event processing logic. First, though, we need to validate the project to avoid discovering problems during publication.

Validation is performed using the assets resource, similar to the way we did the list query in Step 2. Select option **p7** in the sample application to query the validity of the assets in the project. Note that the query parameters are different from those used in Step 2:

```
attribs=(name,valid,typeName,validationString)&sortBy=(valid,typeName)
```

Here, the application is asking for the asset name and human readable type, along with a Boolean indicator of validity and a string describing the validity. The query also requests that the assets are sorted by their validity, then by type. The sample application displays the information it receives, except for the Boolean validity indicator which was included for sorting purposes. You should see several lines like this:

```
Filter               Five events in four days       This asset is valid.
Interaction-Block    Generate lead from recent quer This asset is valid.
```

The third column contains the validation string, which should indicate that all of the assets are valid in this case. Later on, we'll see an example of the kind of error message you might expect to see when checking the validity of your own project.

Step 7. Publish the project

The final step is to publish the event processing logic to the Business Events repository. The assets deployed to the repository are collectively referred to as the runtime application.

To publish the assets from the hosted project we've been working in, you need to download the project in a form suitable for publication. It shouldn't come as a surprise that you request the download by issuing a GET to the project URI:

```
http://{host}:{port}/wbe/rest/r1/dirs/{dirName}/projects/{projName}
```

The request **must** include an `Accept` header as follows:
`Accept: application/x-wbe+xml`

This tells the REST API that the project should be returned in the Business Events internal file format. A successful response will have HTTP response code 200, with a content-type of `application/x-wbe+xml`. The response body will contain the project contents.

Select option **p8** in the sample application to download the hosted project. You'll be prompted to enter a filename, such as MyProject.xml, and the project will be saved. The sample application will display the full path to the project. Business Events project files have the .xml suffix as standard, and it's a good idea to follow the standard naming practice here, because this means your file will be visible to dialogs displayed when opening or reading Business Events project files.

Next, you may want to clear the repository. Take care when issuing the following request; your repository may contain event processing logic for several users or may contain live production assets, so clear the repository with caution.

Option **r3** issues an HTTP DELETE request to the repository URI:
`http://{host}:{port}/wbe/rest/r1/runtime/repository`

A successful operation results in a `204 No Content`, indicating that the operation was a success, and that no additional information is provided.

If you choose to clear the repository, you may want to select option **r2** at this point to view a list of repository assets. We'll examine this further later, but if you cleared the repository you should see an empty list.

To publish the project you downloaded earlier, issue a PUT request to the repository URI, with a content type of `application/x-wbe+xml` and a request body containing the project contents. Next, select option **r1** in the sample application, choose your project file, and you should see something like this:

```
Publishing project to repository at: http://localhost:9080/wbe/rest/r1/runtime/repository
Received HTTP response: 204 No Content
```

Again, the `204 No Content` response indicates success, which means the event processing logic has been published to the repository and is now active as part of the runtime application! You should now be able to send events to Business Events, and observe the behaviour that we defined.

To check that the project has been published, select option **r2** in the sample application. The application should display a list of assets from your project, meaning the assets have been successfully published. If any assets were already present in the repository when you published the project, they will also be listed.

The sample application obtains this list using a query to the assets resource, but this time, in the context of the repository:

`http://{host}:{port}/wbe/rest/r1/runtime/repository/assets?...etc`

The assets resource can be used for both hosted projects and the repository, so asset queries formulated against a hosted project can also be expected to work against the repository.

Deploying invalid projects

Before trying out the event processing logic, let's take a moment to see what happens if you deploy an invalid project.

Select option **r1** in the sample application, and then select **invalidProject.xml** from the **Projects** folder in your imported Eclipse project. This particular project contains different assets than the project we've been working with so far, including some invalid assets.

The application attempts to publish the project file using a PUT request, but receives the following HTTP error:
`400 Bad Request`

This is because we tried to deploy a project that contains invalid assets--specifically, two building block assets (an event object and an action) are missing, so the assets that refer to them are invalid. The HTTP response body contains details about the errors, encoded using XML as a result of content negotiation. You'll notice that there's a general error message:

```
BEER6717E: It was not possible to deploy the given project to the
WBE runtime repository because it contains incompletely defined assets.
```

This is a standard error message returned when attempting to publish an invalid project. You should also see two error messages that relate to specific assets, for example the asset with GUID `7A8D885C_6B60_447C_A3EB_4058603A07CB` has the error: `BEER1217E: Event object Change Of Home Address does not exist.`

The REST API returns up to ten validation errors to help the publishing user understand why a publication request failed. Although there may be more than ten errors in the project, the limit of ten is imposed to prevent unnecessarily overloading the client application.

The validation mechanism used to check projects during publication is the same mechanism used to validate hosted projects, which we used in Step 6. So error messages returned when validating your hosted projects will also be returned during publication, assuming you haven't fixed the problems in the meantime.

Test the event processing logic

To test the event processing logic, you can use the Business Events Tester (BET), a widget that can be loaded into the Business Space application included with Business Events. The following sections describe how to use the BET to test the filter and interaction block published to the repository by the sample application. For simplicity, these instructions assume that an interaction

block triggered by `CustomerQuery` is deployed, where the `GenerateLead` action is created based on a filter that checks whether more than four events have been received within four days.

Open the Business Events Tester

First, open the Business Space in a web browser. In a Windows® installation of Business Events, you can find a shortcut to Business Space in the Business Events program group. Or, you can paste the following URI into a web browser (substituting the host and port as needed): `http://localhost:9080/mum/bootstrap/login.jsp`

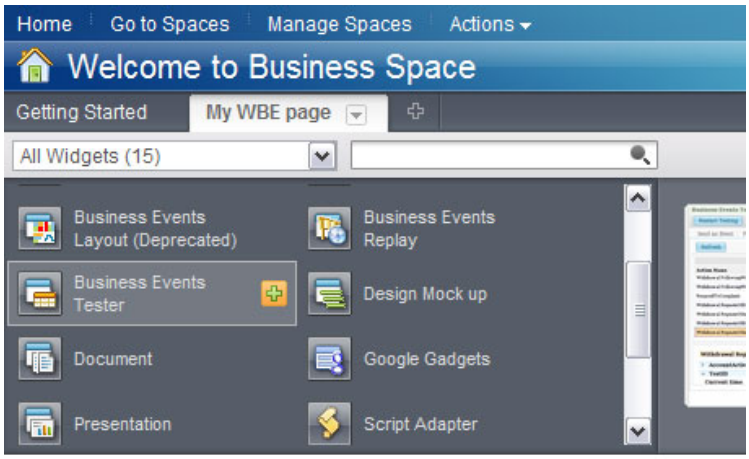
If security is enabled, you'll be prompted to log in with a valid username and password. The default Business Events installation enables and configures security, so use the credentials you provided to the installer or contact your Business Events administrator to obtain a valid username and password.

Next, add a page or open an existing page. You can add a page by clicking on the plus (+) button on the toolbar, as shown in Figure 3.

Figure 3. Adding a page to Business Space

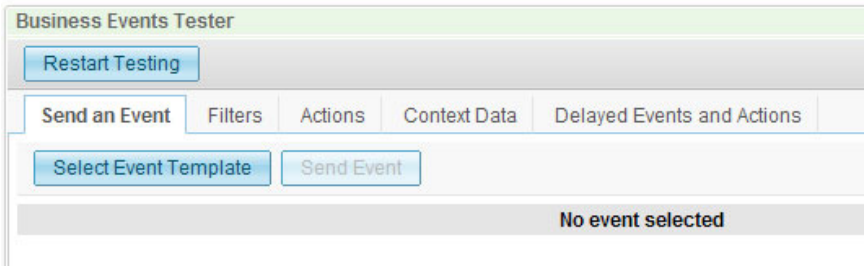
Enter a name for the page, and provide an optional description, then click **OK** to create the page. Next, add the Business Events Tester to your page, as shown in Figure 4.

Figure 4. Adding the Business Events Tester widget



The Business Events Tester should appear in your page, as shown in Figure 5.

Figure 5. The Business Events Tester



Click **Finish Editing** in the top right to close the widget selector.

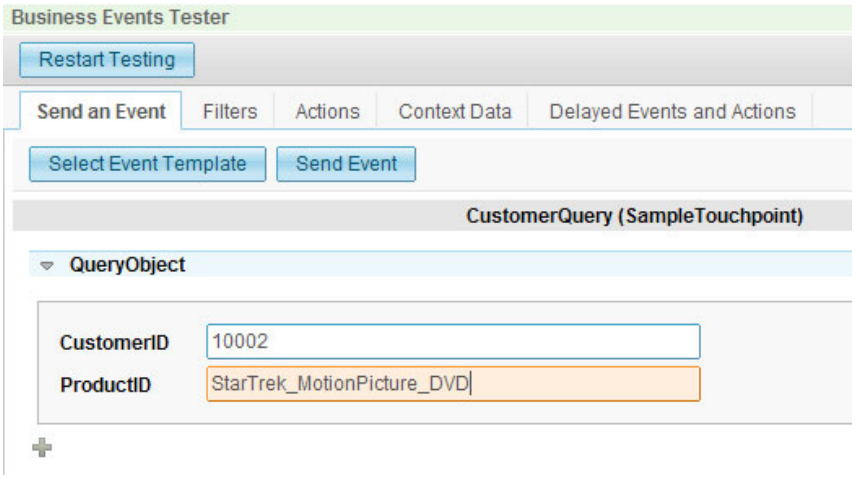
Inject events

You can now inject events into Business Events. First, select the event you chose for your interaction block by clicking **Select Event Template**, opening the touchpoint called **SampleTouchpoint**, then choosing your event.

The BET will be populated with a template for the event, which can be filled in with values and then injected into Business Events. Whichever event you selected, make sure you provide a value for the **CustomerID** field, as this field is used as the context for the event processing logic created by the sample application. If your interaction block uses the predefined `Big Sale` filter, provide a `Sale Amount` greater than 1000. Similarly if your interaction block uses `Big Refund`, provide a `Refund Amount` less than -1000 (for example, -1200). This ensures that actions will be triggered when Business Events receives the events.

Figure 6 shows a completed template for the `CustomerQuery` event:

Figure 6. An event template



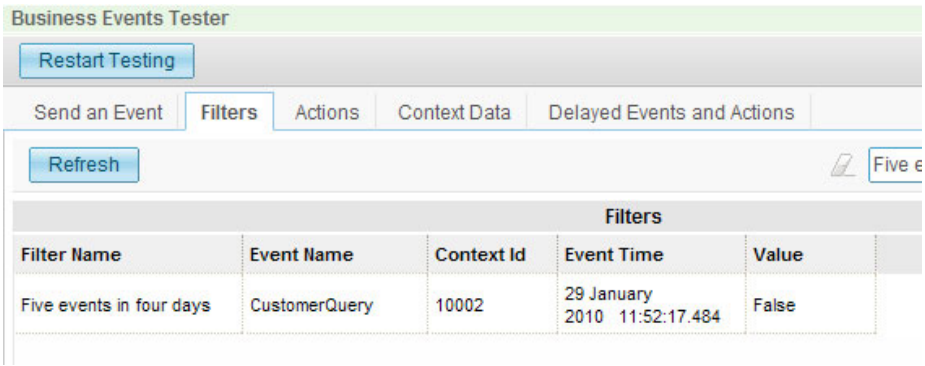
Provide values for the event, and then click **Send Event**.

Check filter evaluation and actions

Click on the **Filters** tab. The **Filters** tab shows information about filter evaluations that have been performed by Business Events as a result of events being received. This is useful in understanding how Business Events is handling events, and why actions may or may not have been triggered.

Figure 7 shows the Filters tab after the `CustomerQuery` event has been sent. Here, you can see that the filter `Five events in four days` has been evaluated as a result of the event, for the context identifier 10002, on the 29th of January. You can also see that the filter evaluated to false, which is consistent with the filter definition, which, in this particular example, requires more than four events to have been received within a timeframe of four days.

Figure 7. Filter evaluation: first event



You may see different values, depending on the interaction block that you defined.

Next, click the **Actions** tab. Based on the example scenario describe above, no actions should be displayed. To trigger an action, you need to send five more `CustomerQuery` events. Five more are needed (rather than four) because the current event is not counted by the `Five events in four days` filter.

Return to the **Send an Event** tab, click **Send Event** five more times, then return to the **Filters** tab to see how the filter evaluated for each event.

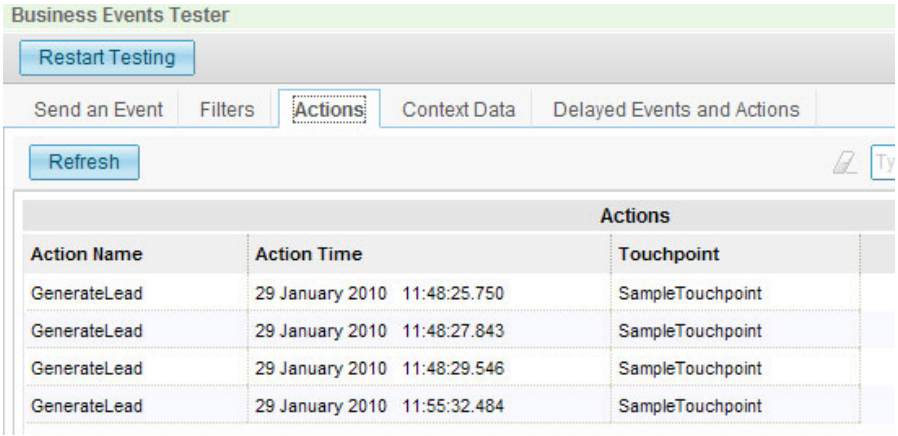
Figure 8. Filter evaluations: more events

Five events in four days	CustomerQuery	10002	29 January 2010 11:52:17.484	False
Five events in four days	CustomerQuery	10002	29 January 2010 11:54:51.406	False
Five events in four days	CustomerQuery	10002	29 January 2010 11:55:00.515	False
Five events in four days	CustomerQuery	10002	29 January 2010 11:55:08.359	False
Five events in four days	CustomerQuery	10002	29 January 2010 11:55:20.171	False
Five events in four days	CustomerQuery	10002	29 January 2010 11:55:32.453	True

The key thing to notice is that the filter evaluated to true for the sixth event, because more than four `CustomerQuery` events had been received in the context 10002 within the last four days.

Finally, return to the **Actions** tab to see the results. You should see the action listed, along with the touchpoint and action time. Figure 9 shows a number of actions generated by Business Events. The final action corresponds to the filter evaluation performed at 11:55:32 in Figure 8.

Figure 9. Actions



Congratulations! You've successfully defined, published, and tested event processing logic using the Business Events REST API and Business Event Tester!

Conclusion

In this article, you learned how to define and publish event processing logic using the new REST API in WebSphere Business Events V7. You learned about major REST resources, operations, and data formats supported by the REST API. You also tried out a sample Java application that demonstrates how event processing logic is programmatically defined using the API. Finally, you tested new event processing logic using the Business Events Tester, by injecting events, checking filter evaluations, and observing actions created by Business Events.

Downloads

Description	Name	Size
Sample application	WBERestClient.zip	39KB

Resources

- To obtain a copy of the Eclipse IDE, and for a wealth of other Eclipse information, visit http://www.eclipse.org/.
- There are several good resources for learning about JSON on the web. A good starting point is http://www.json.org/. A useful tool for working with JSON is the JSON Formatter and Validator, available from curiousconcept.com.
- The WebSphere Business Events Information Center topic WebSphere Business Events REST example has extensive documentation for the Business Events REST API, including a how-to guide, resource overview, resource reference, and troubleshooting guide.
- A useful resource for trying out REST requests and examining responses is the WebSphere Business Events Tool for ReST API exploration, available as an IBM SupportPac.
- Check out the developerWorks BPM zone to get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.

About the authors

James M. Siddle

James Siddle works as a Software Engineer on WebSphere Business Events, and led the development of the REST API for Version 7.0. He has been with IBM for three years, and previously worked for a major telecommunications provider.

Stephen Godwin

Stephen Godwin works as a Software Engineer at IBM Hursley Park in the United Kingdom. For over ten years at IBM, he has worked on WebSphere MQ, WebSphere Message Broker and WebSphere Application Server. He is currently working on Cast Iron Cloud2.