TIBCO ActiveMatrix BusinessWorks™ Plug-in for Mobile Integration Expresso Server Guide

Software Release 6.2.0
December 2016

Two-Second Advantage®

TIBCO®

Important Information

Contents

Figures 4
TIBCO Documentation and Support Services 5
Introduction to Expresso 6
Typical Workflow in Expresso 7
Starting the Expresso Server 7
Compiling a Mobile Client Application 8
Provider 9
JSON Schema for Defining Providers 9
Implementation Guidelines for Defining Providers 11
Defining and Registering a Provider 11
Getting a Subscription Request from the Expresso Server 11
Services 12
GET /schemas/provider-schema 12
GET /schemas/subcription-schema 14
POST /system/providers 14
Response Status Codes 15
Configuration Files 16
expresso.properties 16
pojoprovider.properties 17
quartz.properties 18
Writing a Provider in Java 19
Implementing a Provider in Java 19
Implementing an Event for a Provider 22
Configuring an Event with the Provider 24
Running the Provider 25
Creating a Provider in TIBCO ActiveMatrix BusinessWorks 6.x 26
Workday Provider Sample 26
Hiring an Employee 26
Creating a Job Position (Optional) 28
Terminating an Employee 30
Running a TIBCO ActiveMatrix BusinessWorks 6.x Project 31
SalesForce Provider Sample 32
Creating a SalesForce Provider 33
Editing the Saved WSDL 35
Configuring the TIBCO ActiveMatrix BusinessWorks 6.x Project 36

Figures

Components of Expresso 6

TIBCO Documentation and Support Services

Documentation for this and other TIBCO products is available on the TIBCO Documentation site. This site is updated more frequently than any documentation that might be included with the product. To ensure that you are accessing the latest available help topics, visit:

https://docs.tibco.com

Product-Specific Documentation

The following documents for this product can be found in the TIBCO Documentation Library:

- TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration Installation
- TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration User's Guide
- TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration Developer's Guide
- TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration Expresso Server Guide
- TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration Release Notes

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to the following web address:

https://www.tibcommunity.com

Introduction to Expresso

While working with a mobile application (app), almost everything that happens in the app is based on *events* and *actions*. For example, clicking a button or icon on the mobile is a click event. Based on the *event*, you can trigger any *action* such as navigating to another page or invoking a JavaScript.

Based on the event-action model, Expresso enables you to automate your flows where received events can trigger pre-configured actions. Expresso exposes REST APIs to make this possible.

Expresso consists of the following components:

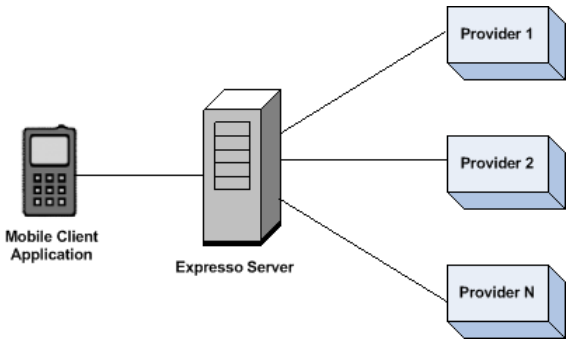
- The **mobile client application** provides the user interface to view and select events, pick the action that needs to be triggered for the selected event, map attributes, add filters, set schedules, change profile details, mark favorite pods, and brew an expression. After brewing the expression, users must run it to start receiving notifications and alerts based on the events created.

  In terms of APIs, the mobile client application consumes the APIs exposed by the Expresso server.
- The **Expresso server** provides the interface for making providers available to the mobile client application. The server mandates that a provider must describe its events and actions in a format understood by the server. In this case, the format is a JSON format conforming to the JSON provider definition schema published to the server.

  In terms of APIs, the server enables the providers to expose the APIs which can then be consumed by the mobile app users.
- **Providers** are third-party applications that provide events to the server. Provider applications need to register themselves with the Expresso server for being a part of the Expresso inventory.

  In terms of APIs, the providers expose APIs that can be communicated by the Expresso server to the mobile client application.

  Components of Expresso



To summarize, the Expresso Server exposes REST APIs to allow any type of providers to plug-in and expose events, actions, and pods to the mobile application. It also provides REST APIs to create your own mobile app.

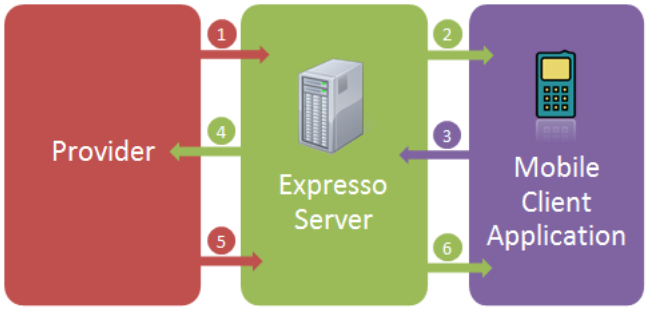
TIBCO ActiveMatrix BusinessWorks™ Plug-in for Mobile Integration 6.x enables you to develop providers for Expresso using the **ExpressoNotify** activity and an **Expresso Provider** shared resource. For more details on the activity and the shared resource, refer to *TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration User's Guide*. The ActiveMatrix BusinessWorks platform handles most of the activities necessary for an Expresso provider thereby making the developer experience simpler and richer.

Providers can also be developed outside of the TIBCO ActiveMatrix BusinessWorks Plug-in for Mobile Integration 6.x environment. For example, you can develop providers using Java.

Expresso ships with the following sample providers installed in `$TIBCO_HOME\expresso\providers`:

- TIBCO ActiveMatrix BusinessWorks 6.x - SalesForce and Workday
- POJO - JIRA and Twilio

Typical Workflow in Expresso



1. The provider registers with the Expresso server with details of events and actions. Multiple providers can register with the server.
2. The Expresso server shows all registered events to the mobile app user.
3. The mobile app user defines (includes mapping attributes and adding filters) and "brews" the expression. When an expression is brewed, the Expresso server stores a definition, creates an instance of the definition, and runs the instance.
4. The Expresso server subscribes for the event to the provider.
5. The provider listens for subscription requests and when an event comes in, the information is passed to the server.
6. The server matches the filter criteria and then executes the action defined by the mobile app user.

Starting the Expresso Server

Run `$TIBCO_HOME\expresso\1.0\bin\expresso.exe`.

Compiling a Mobile Client Application

A sample mobile client application is shipped with Expresso. This sample application can be used to connect to the Expresso server, create, and brew an expression.

Build and run the sample using Xcode. When running the iPhone Simulator, configure the Expresso Server URL on the Login screen.

For more information on compiling the mobile client application, refer to:

```
TIBCO_HOME\expresso\client\iOS_mobile_app\Expresso iOS 1.0 library Usage Guide.pdf
```

Provider

An Expresso provider is a RESTful service provider that provides business data, events, and services to the Expresso server relevant to a specific business domain.

An event pod is a provider resource that is stateful and asynchronous in nature and provides business events to the Expresso server through a subscription mechanism. It enables the Expresso server to subscribe to it by sending an HTTP POST request with Expresso Event Source Request object as its body. The Expresso Event Source Request object contains an Expresso *web hook URL* along with other data. The provider stores the Expresso *web hook URL* and posts business events on this URL whenever available. The Event source defines its output event structure in the form of a JSON schema. An Event Source enables terminating the subscription by sending an HTTP DELETE request with Expresso Event Source Request object as its body.

For example, a JIRA provider can have an event source that notifies Expresso whenever a defect is logged.

JSON Schema for Defining Providers

Providers must conform to the following JSON provider definition schema published the server. A provider must provide its description and register itself with Expresso by invoking the RESTful provider registration service of the Expresso server. The provider registration service adds the provider to the Expresso's provider inventory. The provider is then made available to the mobile app users.

```
{
  "type": "object",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Schema for Expresso provider definition",
  "title": "Provider",
  "required": [
    "events",
    "name",
    "description",
    "actions"
  ],
  "properties": {
    "events": {
      "type": "array",
      "items": {
        "type": "object",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Event definition",
        "title": "Event",
        "required": [
          "uniqueName",
          "eventSubscriptionUrl",
          "usage",
          "eventSchema",
          "description"
        ],
        "properties": {
          "uniqueName": {
            "type": "string",
            "description": "Unique name of the event"
          },
          "eventSubscriptionUrl": {
            "type": "string",
            "description": "The resource url for susbcription to the event "
          },
          "usage": {
            "type": "string",
            "description": "Sample usage of this event"
```

```
          },
          "eventSchema": {
            "type": "string",
            "description": "Schema of the event data"
          },
          "description": {
            "type": "string",
            "description": "Short description of the event"
          }
        }
      },
      "description": "Events supported by the provider"
    },
    "name": {
      "type": "string",
      "description": "Unique name for the provider"
    },
    "outputSchemaForEvent": {
      "type": "string"
    },
    "description": {
      "type": "string",
      "description": "Short description of the provider"
    },
    "userActivation": {
      "type": "string",
      "description": "Optional schema for activation of the provider"
    },
    "modifiedDate": {
      "type": "integer"
    },
    "actions": {
      "type": "array",
      "items": {
        "type": "object",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Definition of Action supported by the provider",
        "title": "Action",
        "required": [
          "actionEndpointUrl",
          "description",
          "actionSchema",
          "uniqueName"
        ],
        "properties": {
          "actionEndpointUrl": {
            "type": "string",
            "description": "The resource URL for invoking the action"
          },
          "description": {
            "type": "string",
            "description": "Short description of the action"
          },
          "actionSchema": {
            "type": "string",
            "description": "Input schema for the action"
          },
          "uniqueName": {
            "type": "string",
            "description": "Unique name of the action within the provider
definition"
          }
        }
      },
      "description": "Actions supported by the provider"
    },
    "createdDate": {
      "type": "integer"
    }
  }
}
```

Implementation Guidelines for Defining Providers

A provider must adhere to the following implementation guidelines:

- Have a simple and unique name in the context of the Expresso provider inventory.
- Provide services specific to a single business domain.
- Have simple, precise, and non-nested input and output schemas. It must not define more than 4-6 properties.

Defining and Registering a Provider

Procedure

1. Get the provider definition schema from the Expresso server (using GET /schemas/provider-schema).
2. Define the provider (event pods and actions) and register the provider by passing the description to the Expresso server (using POST /system/providers).

   The Expresso server adds the provider to the Expresso inventory.

Getting a Subscription Request from the Expresso Server

To get the schema for a subscription request body sent to the provider by the Expresso server, use POST /system/providers.

Services

The following services are available:

Service	Description
GET <baseurl>/schemas/provider-schema	Gets the provider definition schema from the Expresso server.
GET <baseurl>/schemas/subcription-schema	Gets the subcription request schema sent to the provider by the Expresso server.
POST <baseurl>/system/providers	Creates and registers providers.

GET /schemas/provider-schema

Use this service to get the provider definition schema from the Expresso server.

Response Body

```
{
  "type": "object",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Schema for Expresso provider definition",
  "title": "Provider",
  "required": [
    "events",
    "name",
    "description",
    "actions"
  ],
  "properties": {
    "events": {
      "type": "array",
      "items": {
        "type": "object",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Event definition",
        "title": "Event",
        "required": [
          "uniqueName",
          "eventSubscriptionUrl",
          "usage",
          "eventSchema",
          "description"
        ],
        "properties": {
          "uniqueName": {
            "type": "string",
            "description": "Unique name of the event"
          },
          "eventSubscriptionUrl": {
            "type": "string",
            "description": "The resource url for susbcription to the event "
          },
          "usage": {
            "type": "string",
            "description": "Sample usage of this event"
          },
```

```
          "eventSchema": {
            "type": "string",
            "description": "Schema of the event data"
          },
          "description": {
            "type": "string",
            "description": "Short description of the event"
          }
        }
      },
      "description": "Events supported by the provider"
    },
    "name": {
      "type": "string",
      "description": "Unique name for the provider"
    },
    "outputSchemaForEvent": {
      "type": "string"
    },
    "description": {
      "type": "string",
      "description": "Short description of the provider"
    },
    "userActivation": {
      "type": "string",
      "description": "Optional schema for activation of the provider"
    },
    "modifiedDate": {
      "type": "integer"
    },
    "actions": {
      "type": "array",
      "items": {
        "type": "object",
        "$schema": "http://json-schema.org/draft-04/schema#",
        "description": "Definition of Action supported by the provider",
        "title": "Action",
        "required": [
          "actionEndpointUrl",
          "description",
          "actionSchema",
          "uniqueName"
        ],
        "properties": {
          "actionEndpointUrl": {
            "type": "string",
            "description": "The resource URL for invoking the action"
          },
          "description": {
            "type": "string",
            "description": "Short description of the action"
          },
          "actionSchema": {
            "type": "string",
            "description": "Input schema for the action"
          },
          "uniqueName": {
            "type": "string",
            "description": "Unique name of the action within the provider
definition"
          }
        }
      },
      "description": "Actions supported by the provider"
    },
    "createdDate": {
      "type": "integer"
    }
  }
}
```

GET /schemas/subcription-schema

Use this service to get the subcription request schema sent to the provider by the Expresso server.

Response Body

```
{
  "type": "object",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "Event Subcriber definition",
  "title": "Subscriber",
  "required": [
    "expressoSubscriberId",
    "expressoCallbackUrl"
  ],
  "properties": {
    "expressoSubscriberId": {
      "type": "string",
      "description": "Subcriber Id"
    },
    "expressoCallbackUrl": {
      "type": "string",
      "description": "Webhook callback url"
    }
  }
}
```

POST /system/providers

Use this service to create and register providers. The following parameters need to be provided.

Parameters

- Name
- Actions
  - uniqueName - The name of the Expresso action. The name is displayed as **Domain Name** on the mobile application UI.
  - Description - Description of the action. The description is displayed as **Domain Description** on the mobile application UI.
  - actionSchema
  - actionEndpointUrl
- Events
  - uniqueName - The name of the Expresso event. This name is displayed as the name of the pod on the mobile application UI. The event name must be unique in the provider.
  - Description - Description of the event. The description is displayed as the description of the pod on the mobile application UI.
  - eventSchema
  - description
- description

Response Status Codes

HTTP Status Code	Description
200	Operation successful.
204	No content.
401	Authentication failed.
403	Forbidden.
404	Requested resource not found.

Configuration Files

Expresso provides the following configuration files:

- `expresso.properties`: configure various properties related to the Expresso server.
- `pojoprovider.properties`: configure details of a POJO provider.
- `quartz.properties`: configure Quartz to use the appropriate persistent job store and also add the required dependencies.

expresso.properties

The `expresso.properties` file enables you to set various properties related to the Expresso server:

Property	Description
`local.host`	Host name or IP address of the Expresso server. Default host is localhost, 0.0.0.0.
`local.port`	Local port of the Expresso server. Default value is 36136.
`public.host`	Public host name or IP address of the Expresso server.
`public.port`	Public port of the Expresso server.

Sample File

```
#########################
# Expresso server config #
# Usage: java ExpressoServer -Dexpresso.config.file=<path>/expresso.properties
#########################

#public.host = expresso.com
#public.port = 36136
#local.host = 0.0.0.0
local.port = 36136
socket.connectTimeout = 20000
socket.readTimeout = 20000
provider.port = 40000

#######################################################
# Twilio Provider config to be configured by the user#
#######################################################

#expresso.providers.twilio.account_token = <value>
#expresso.providers.twilio.account_sid = <value>
#expresso.providers.twilio.from_number = <value>

#######################################################
# Email Provider config to be configured by the user #
#######################################################

mail.sender.email = <value>
mail.sender.password = <value>
mail.subject.default = Notificaton from Expresso
mail.smtp.host = smtp.gmail.com
mail.smtp.port = 465
```

```
# to enable authentication
mail.smtp.auth = true

# to use SSL
mail.smtp.socketFactory.port = 465
mail.smtp.socketFactory.class = javax.net.ssl.SSLSocketFactory

# to use TLS
#mail.smtp.starttls.enable = true

########################################
# iOS push notification provider config #
########################################

expresso.providers.pushnotifications.apns.host = gateway.push.apple.com
expresso.providers.pushnotifications.apns.port = 2195
expresso.providers.pushnotifications.apns.threadPoolSize = 4
expresso.providers.pushnotifications.apns.ssl.pkcs.keystore.file.path =
%TIBCO_EXPRESSO_HOME_ESC%/config/expresso_apns_prod.p12
expresso.providers.pushnotifications.apns.ssl.pkcs.keystore.password = tibco
expresso.providers.pushnotifications.apns.ssl.jks.truststore.file.path =
%TIBCO_EXPRESSO_HOME_ESC%/config/apns_prod_truststore.jks
expresso.providers.pushnotifications.apns.ssl.jks.truststore.password = tibco

############################################################
# Max. number of user messages to be packaged in one chunk #
############################################################
expresso.user.max_no_of_messages = 50
```

pojoprovider.properties

The `pojoprovider.properties` file enables you to set various properties related to POJO providers.

Property	Description
`local.host`	Local host name or IP address of the provider server. Default host is 0.0.0.0.
`local.port`	Local port on which the provider server wants to listen to. Default value is 40000.
`public.host`	Public host name or IP address of the provider server.
`public.port`	Public port of the provider server.
`expresso.server.baseUrl`	URL that points to the Expresso server to send registration requests.
`expresso.pojo.provider.samples.path`	List of all the jars which provide POJO provider implementations.

Sample File

```
#########################
# POJO Provider config #
# Usage: java ProviderServer -Dpojoprovider.config.file=C:/tibco/expresso/providers/
pojo/config/pojoprovider.properties
#########################

# POJO Provider server port (public and/or local) to listen to:
```

```
# 1. Subscriptions from Expresso server
#    2. Messages received for different POJO implementation events

#public.host = provider.com
#public.port = 80

#local.host = localhost
#local.port = 40000

# URL to point to Expresso server to send registration requests
expresso.server.baseUrl=http://localhost:36136

# Allows users to list all the jars which provide POJO provider implementations.
expresso.pojo.provider.samples.path=C:/tibco/expresso/providers/pojo/lib/
sampleProviders.jar

##########################
# Twilio Provider config #
##########################

expresso.providers.twilio.account_sid = <value>
expresso.providers.twilio.account_token = <value>
expresso.providers.twilio.incoming_sid_number = <value>

##########################
# Timer Provider config  #
##########################
expresso.providers.timer.interval_ms = 60000
expresso.providers.timer.message = Hello Expresso
```

quartz.properties

Expresso embedds a Quartz scheduler for scheduling Expresso executions.

The out-of-box configuration uses an in-memory job store called `RAMJobStore`. All jobs and triggers are stored in RAM and therefore do not persist between program executions; this has the advantage of not requiring an external database. To persist jobs across program executions, you can configure Quartz to use the appropriate persistent job store and also add the required dependencies. You can configure Quartz in the `quartz.properties` configuration file located in the `EXPRESSO_HOME/config` directory.

For additional information, refer to the Quartz documentation available at: http://quartz-scheduler.org/documentation/quartz-2.x/configuration/.

Sample File

```
# Default Properties file for use by StdSchedulerFactory
# to create a Quartz Scheduler Instance, if a different
# properties file is not explicitly specified.
#

org.quartz.scheduler.instanceName: DefaultQuartzScheduler
org.quartz.scheduler.rmi.export: false
org.quartz.scheduler.rmi.proxy: false
org.quartz.scheduler.wrapJobExecutionInUserTransaction: false

org.quartz.threadPool.class: org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount: 10
org.quartz.threadPool.threadPriority: 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread: true

org.quartz.jobStore.misfireThreshold: 60000

org.quartz.jobStore.class: org.quartz.simpl.RAMJobStore
```

Writing a Provider in Java

The pojoProvider server facilitates writing providers in Java. It exposes abstract classes through the pojoProvider-api JAR that needs to be extended to implement providers and their events. The provider has to only implement logic to process data coming from external services and subsequently generate appropriate event data. The registration of providers with the Expresso server, and the maintenance of subscriptions and unsubscriptions for events is handled by the pojoProvider server internally. The pojoProvider server communicates both with the Expresso server as well as external services that want to send data to concerned providers.

The pojoProvider server saves the event-specific subscriber data in the .expressoprovider folder created in your home directory. This is useful when you restart the pojoProvider server. It helps resume functioning from the last-left state. That is, if the pojoProvider server is restarted even when there are existing subscriptions, the server can resume functioning when it is started again.

A provider can be written in Java using the pojoProvider-api.jar shipped along with Expresso. Writing a provider in Java consists of two steps:

1. Adding configuration data as per user requirements for the provider that is to be written in the pojoprovider.properties file located in:

   %TIBCO_EXPRESSO_PROVIDERS_HOME_ESC%/pojo/config/pojoprovider.properties

2. Implementing the POJO provider in Java with the help of two abstract classes found in the pojoProvider-api.jar:

   - AbstractExpressoProvider: a class needs to extend this abstract class to create a Java provider.
   - AbstractExpressoEvent: a class needs to extend this abstract class to create an event for a provider.

Implementing a Provider in Java

For ease of explanation, the process of implementing a provider in Java is explained using an example of a "Timer" provider that generates events at regular intervals and publishes a message to the Expresso server. The time interval and message are set in the properties file and can be changed by the user.

1. Specifying data in the properties file. The configuration data that is required by the user can be specified in the pojoprovider.properties file. For this example, two properties are added:

   - expresso.providers.timer.interval.millsecs = 3000
   - expresso.providers.timer.message = Hello Expresso

2. Implementing the POJO provider in Java. A provider can register itself with the Expresso server stating its name, description, and a list of events that it has to offer. The library offers two abstract classes that a user can extend to realise their provider and its events.

   The project also needs the commons-configuration-1.10.jar and jetty-7 libraries in the build path.

Procedure

1. Write a provider class that extends the AbstractExpressoProvider class.

2. Define the class by mentioning its name and description by overriding the `getName()` and `getDescription()` methods.
3. Return a list of events that the provider has by overriding the `getProviderEvents()` method.
4. If required, write initialization code, such as loading data from the `pojoprovider.properties` file, in the overridden `init()` method.

Writing a Class that Extends the AbstractExpressoProvider Class

The following example illustrates the methods that the extending class must implement. The `TimerProvider.java` class extends the `AbstractExpressoProvider` class.

```
import java.util.HashMap;
import java.util.Iterator;
import com.tibco.expresso.providers.api.AbstractExpressoEvent;
import com.tibco.expresso.providers.api.AbstractExpressoProvider;

public class TimerProvider extends AbstractExpressoProvider{

//These are user defined fields, specific to the provider

private static final String TIMER_PROPS = "expresso.providers.timer";
public static HashMap<String, String> timerData = new HashMap<String,
String>();
public static final String TIMER_INTERVAL =
"expresso.providers.timer.interval_ms";
public static final String TIMER_MESSAGE = "expresso.providers.timer.message"

@Override
public String getName()
{
    // TODO Auto-generated method stub
    // name of the provider should be returned here

    return "Timer";
}
@Override
public String getDescription()
{
    // TODO Auto-generated method stub
    // description of the provider should be returned here.

    return "Provides Timer events";
}

// The getProviderEvents() method will be explained and overridden
//in later parts once we create events using the AbstractExpressoEvent
//class as it returns a map of events that the provider has to offer.

@Override
public HashMap<String, AbstractExpressoEvent> getProviderEvents()
{
    return null;
}

//This method can be used to initialize and load data specific to the
//provider on its start up. All the properties mentioned in the
//pojoprovider.properties file are available to the “config”
//object of every provider and can be retrieved as shown below.
//For this provider, we are saving them in timerData HashMap to
//use later on.

@Override
public void init()
{

//Set the provider colour if required.
//This colour will reflect in all the event pods specific to this
//provider in the mobile app. The default value of colour is taken as
//white if not set by the provider.

  setProviderColour(Colour.Grey)

       Iterator<String> timerKeys = config.getKeys(TIMER_PROPS);
       while(timerKeys.hasNext())
       {
            String key = timerKeys.next();
            timerData.put(key, config.getString(key));
          }
```

```
    }
}
```

Implementing an Event for a Provider

Procedure

1. Write an event class that extends the `AbstractExpressoEvent` class.
2. Define an event data class whose object represents and returns event data generated by the event back to the Expresso server.
3. Override the `subscribeToEvent()` and `unSubscribeFromEvent()` to handle subscriptions for the event. The `subscribeToEvent()` method is called with a `providerWebhookUrl` parameter which is generated specific to every event by the provider server. This URI listens to a POST call from external services and subsequently routes it to the event through the `onExternalEvent()` method.
4. If the provider requires communication with external services, handle it in the overridden `onExternalEvent()` method.
5. Notify the subscriber using the `notifySubscriber()` method either in the `onExternalEvent()` method itself, or elsewhere as per specific provider requirement.

Writing a Class that Extends the AbstractExpressoEvent Class

This sample illustrates the methods that the extending class must implement. Here, `TimerEvent.java` extends the `AbstractExpressoEvent` class.

```
public class TimerEvent extends AbstractExpressoEvent
{

// provider specific fields

    private boolean isRunning = false;
    private Timer timer;

/ EventData class: Every event needs to publish data back to the Expresso
//server. On specific intervals, the timer event will notify the Expresso
//along with an event data object of type TimerEventData. This event data
//class is event specific. This class is set while creating an object in
//the provider class. pojoProvider internally converts the data to JSON
//format as required by Expresso while publish events using an object of
//the event data class.

public static class TimerEventData {

        private String message;

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

        public String getMessage() {
            return message;
        }

    }

// subscribeToEvent is called by the pojoProvider on two occasions. When the
//first subscription to an event is received, and if pojoProvider is
//restarted and existing subscriptions from Expresso were present on it. The
//pojoProvider sends a providerWebhookUrl to the provider. This is meant to
//be used by the provider if it listens to another service. For timer, it
//isn't required, but it is used in the TwilioProvider as the url to listen
//for messages coming from Twilio.
//For example: http://(somehost).com:80/events/Twilio/SMSReceived. Such url
//is then registered with Twilio for listening to webhook callbacks from
//Twilio. For timer, the events are generated locally and thus the
//providerWebhookUrl isn't required. On receving a call, this method should
//implement logic to initiate the publishing of events. Here, the
//timer.start() call does the same.

@Override
public boolean subscribeToEvent(URI providerWebhookUrl)
{
    timer = new Timer();
    isRunning = true;
    timer.start();
    return true;
}

//Once the subscription is received, in case the provider depends on external
//service to notify it with data, the onExternalEvent method is called along
//with the request object and a response object. The provider should extract
//data from the request object and set the response object according to what
//the external service desires. Please refer to the TwilioReceiveSMS class
//in the samples to see this usage. For timer, as no external service is
//communicating with it, This method is left blank. On receiving of external
//event, or on fulfilment of the providers own conditions, the provider has
//to call the notifySubscriber() method passing the event data object
//to it. In this case, we are doing so after regular intervals using the
//Timer class.

@Override
public void onExternalEvent(HttpServletRequest request,HttpServletResponse
```

```
response)
{
        // TODO Auto-generated method stub
}

class Timer extends Thread {

        private int interval;
        private final TimerEventData eventData;

Timer()
{
/** Use data from properties file to decide time interval between
  * events and the message to send in event data. Default is interval
  * is 30000 ms and default message is "Hello Expresso"
*/
    intrvl = TimerProvider.timerData.get(TIMER_INTERVAL);
    if (intrvl == null)
           intrvl = "30000";
    interval = Integer.parseInt(intrvl);
    msg = TimerProvider.timerData.get(TIMER_MESSAGE);
    if (msg == null)
           msg = "Hello Expresso";
    /*
     * Create object of class that represents the event data.
    */
    eventData = new TimerEventData();
    eventData.setMessage(msg);

}


        @Override
        public void run() {
            while (isRunning) {
                try {
                    Thread.sleep(interval);
                    /*
                     * Notify subscriber
                     */
                    notifySubscriber(eventData);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } // loop ends
        } // run() ends
    } // class Timer ends

//unSubscribeFromEvent is called when the last subscriber unsubscribes from
//the event. In this method the provider is supposed to handle logic to
//stop sending events. In the TimerProvider we set the flag to false.

@Override
public boolean unSubscribeFromEvent()
{
    isRunning = false;
    return true;
}


}
```

Configuring an Event with the Provider

Modify the `getProviderEvents()` method in the `TimerProvider` class to return a map containing event names and its corresponding objects.

The `pojoProvider` server internally converts the data to the JSON format as required by the Expresso server during provider registration.

```
public HashMap<String, AbstractExpressoEvent> getProviderEvents()
{

HashMap<String, AbstractExpressoEvent> events = new HashMap<String,
AbstractExpressoEvent>();

//Event 1:
/Create an object of the Event created above.

TimerEvent timerEvent = new TimerEvent();

//Set the name, description and usage of that event.

timerEvent.setEventDetails("TimerEvent","Generates events after specific time
intervals","Timer event received with message :{message}");

//Set the class whose object will be returned as event data.

timerEvent.setEventSchemaClass(TimerEventData.class);

//Set the name of the provider to which this class belongs, this is used while
//notifying an event Expresso

timerEvent.setProviderName("Timer");

//Put the object in a map with the event name as key, make sure that the key
//matches exactly to the event name as it will be used by the provider
//server to lookup later.

events.put("TimerEvent", timerEvent);

//More events can be added in a similar way to the map.

return events;

}
```

Running the Provider

Procedure

1. Export the provider as a JAR file.
2. Append the path of the provider JAR to the `expresso.pojo.provider.samples.path` property in the `pojoprovider.properties` file.
3. Run `pojoProvider.exe`.
   The providers are automatically loaded and registered with the Expresso server.

Creating a Provider in TIBCO ActiveMatrix BusinessWorks 6.x

Expresso ships with two TIBCO ActiveMatrix BusinessWorks samples installed in `$TIBCO_HOME\expresso\providers`:

- SalesForce
- Workday

Workday Provider Sample

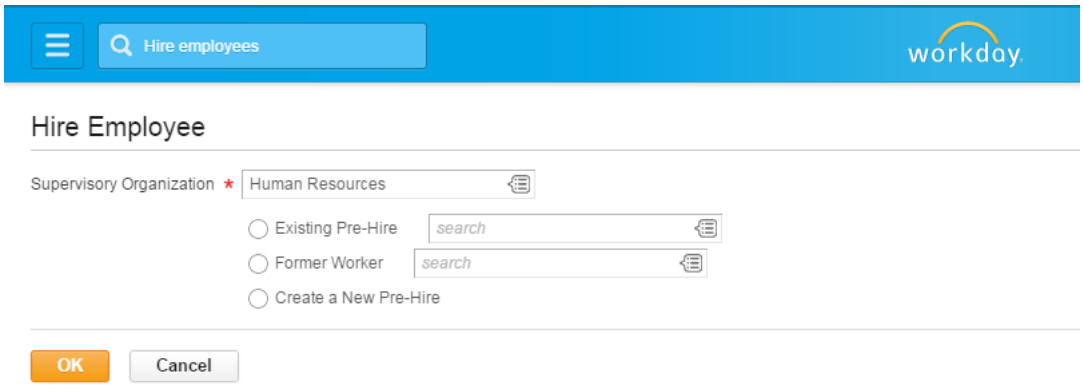
Workday provider in TIBCO ActiveMatrix BusinessWorks 6.x has two events - **Hire Employee** and **Terminate Employee**. Whenever a new employee is hired or terminated in the Workday application, the provider notifies the Expresso server about the events along with associated data. This section describes the following:

- Hiring an employee
- Creating a job position (optional)
- Terminating an employee
- Running a TIBCO ActiveMatrix BusinessWorks 6.x project

Hiring an Employee

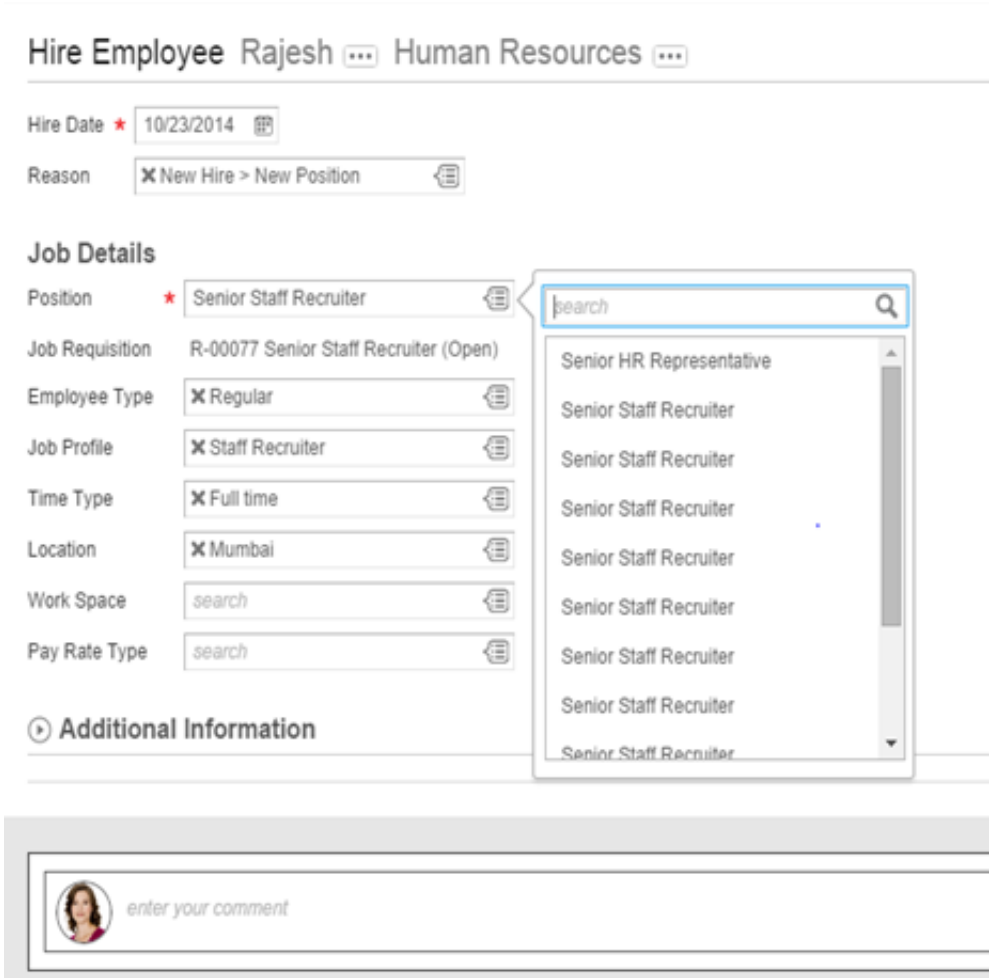
Procedure

1. Login to the Workday instance.
2. Click on the search field and search for the **Hire Employee** task.
3. To hire an employee against a previously created job position in the Human Resources department, select **Human Resources** as **Supervisory Organization**, select **Create a New Pre-hire**, and click **OK**.

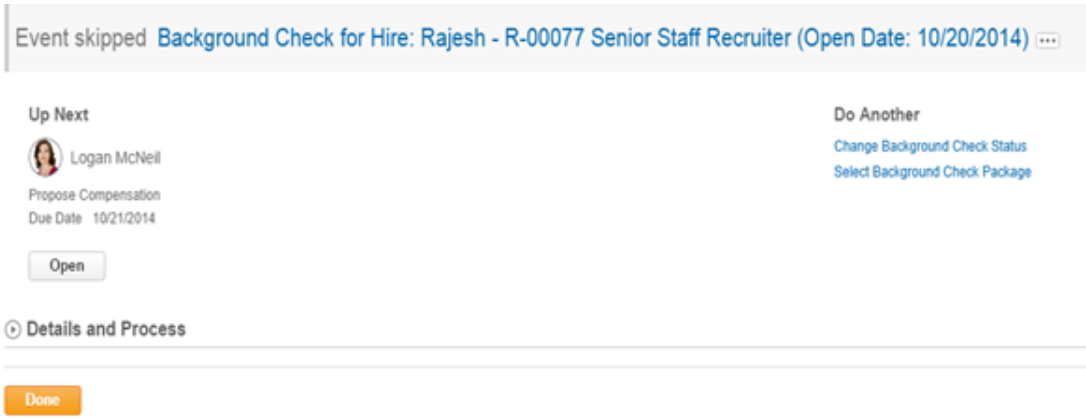


4. On the next screen, specify details such as country, name, and contact number. Mandatory fields are marked with an asterisk (*).
5. Click **OK**.
6. On the next screen, specify **Hire Date**. Select **Position** from available open job positions. If no position is available, you can create a position. Specify details such as the following and click **Submit**.

- Employee Type: Regular
- Job Profile: Staff Recruiter
- Time Type: Full Time
- Location : Mumbai

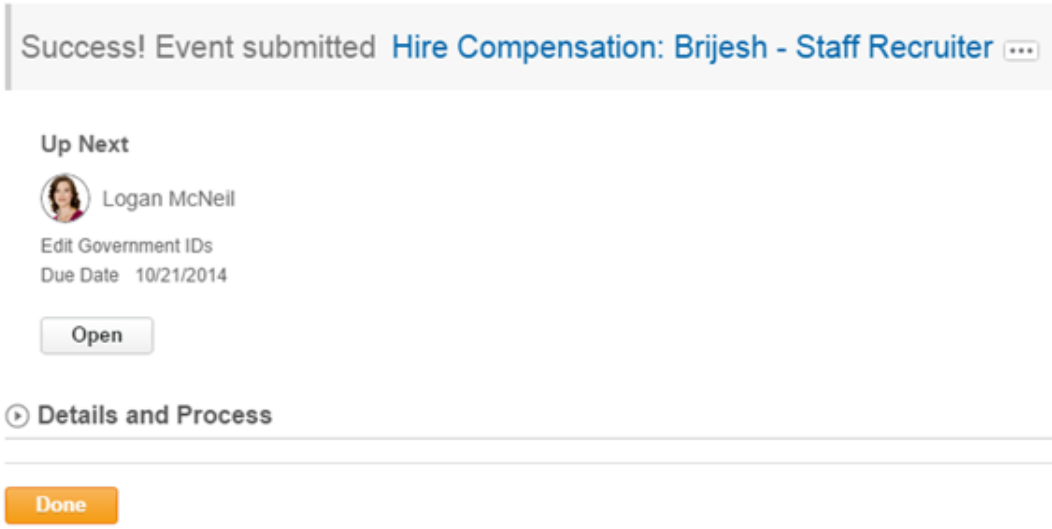


7. On the next screen, click **Open** to assign compensation for the hired employee.



8. Click **Submit** to assign default salary. Click on the salary to assign a new salary instead of default salary.

   The employee is successfully hired.



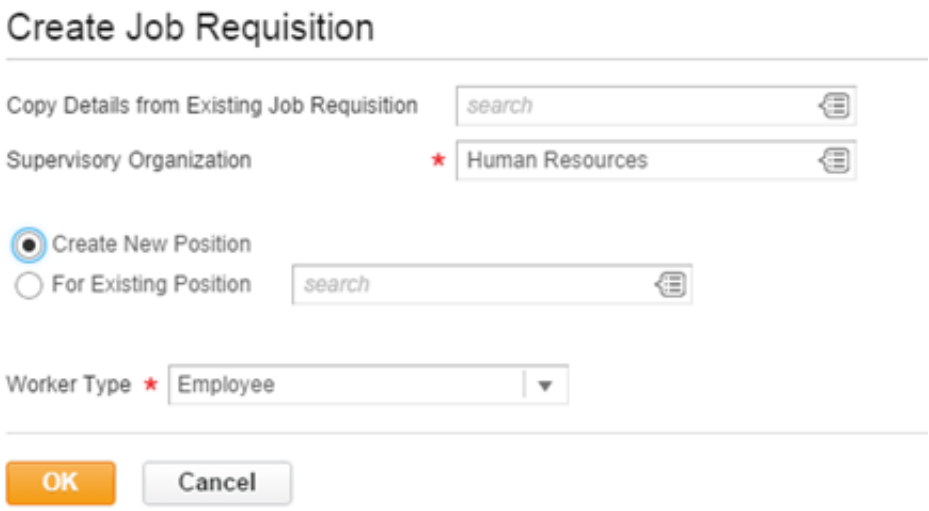
9. Click **Done**.
   Workday sends a notification about the event to the **Notification URL** specified in the **Edit Subscription** screen.

Creating a Job Position (Optional)

To hire an employee, it is mandatory to have a position within the organization. To create a position in Workday, create a job requisition.

Procedure

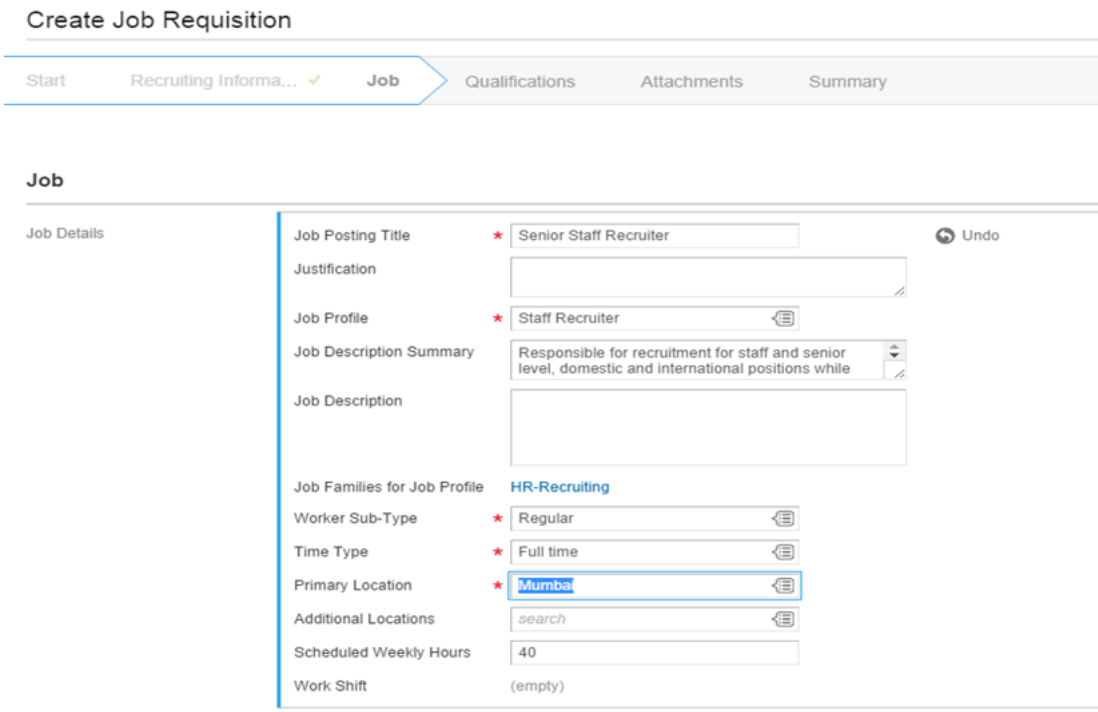
1. Click on the search field and search for **Create Job Requisition**.
2. Select **Human Resources** as **Supervisory Organization** and create a new position as shown in the following image.



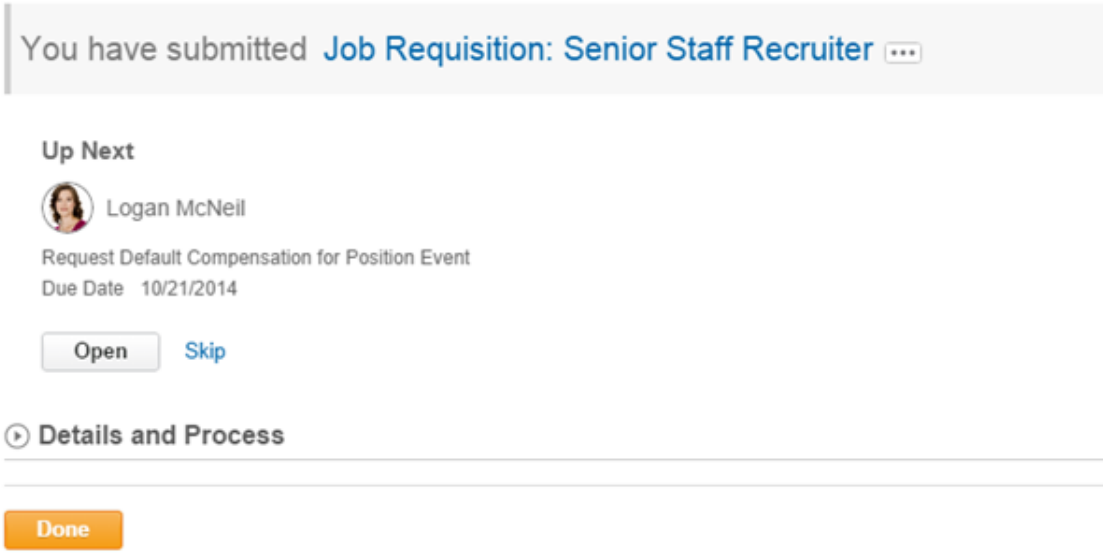
3. On the **Recruiting Information** screen, specify the following mandatory fields and click **Next**.
   - **Number of Openings**
   - **Reason**: Create Job Requisition > New Position > Requesting Additional Staff

- Other mandatory fields marked with an asterisk (*).

4. Specify mandatory fields such as **Job Posting Title**, **Job Profile**, **Worker Subtype**, **Time Type**, **Location**, and click **Next**.

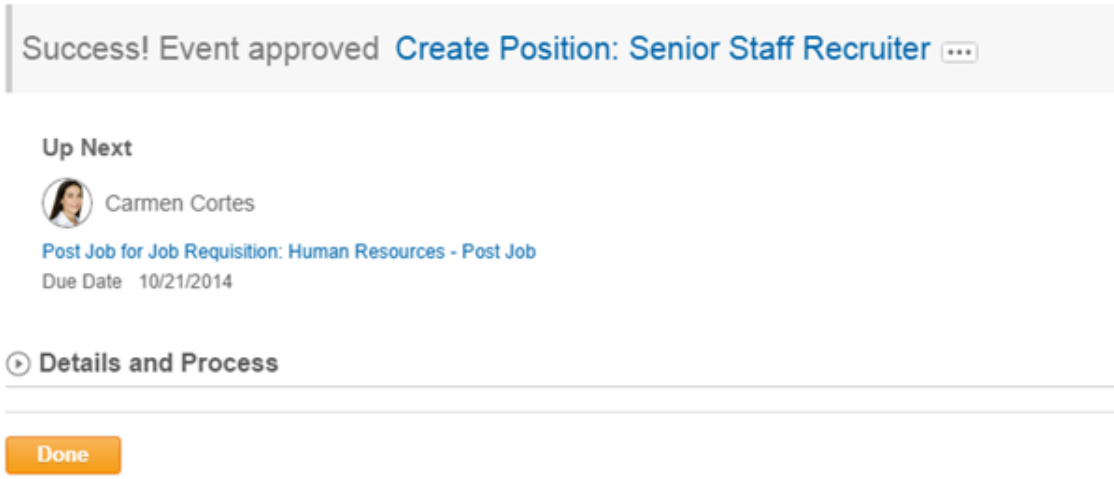


5. (Optional) Specify optional details such as **Education**. You can also attach files on the **Attachment** screen if required. Click **Next**.
6. On the Summary screen, click **Submit**.
7. Click **Open** to assign a default compensation for the newly created job position.



8. Assign the salary and click **Approve**.
9. For change organization assignment, click **Skip.**
10. Click **Review**.

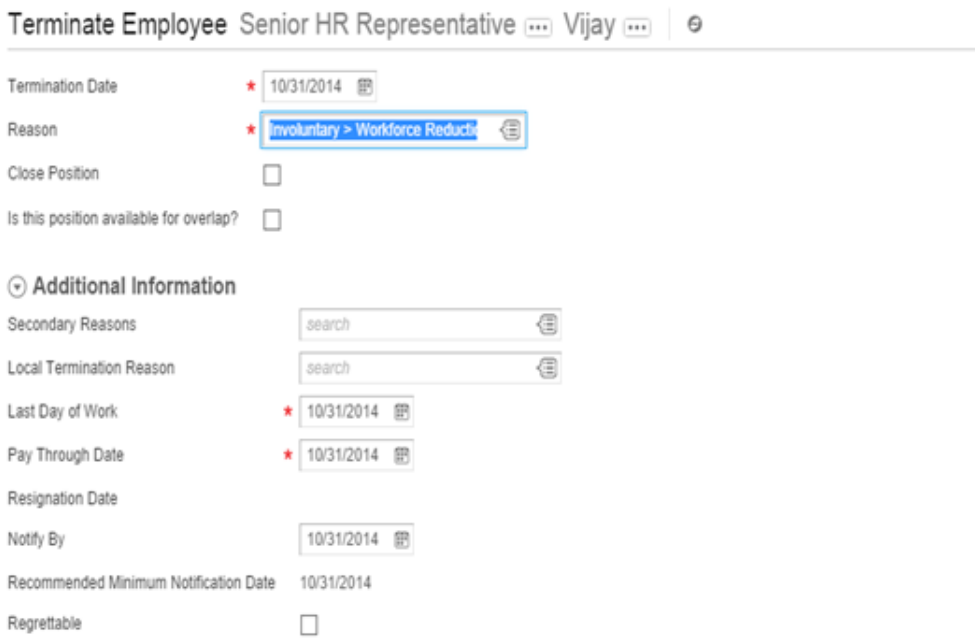
11. Click **Done**.
    The position is created.



Terminating an Employee

Procedure

1. In Workday, click on the search field and search for **Terminate Employee**.
2. Enter the name of the employee to be terminated and click **OK.**
3. On the next screen, specify the mandatory details and click **Submit**.



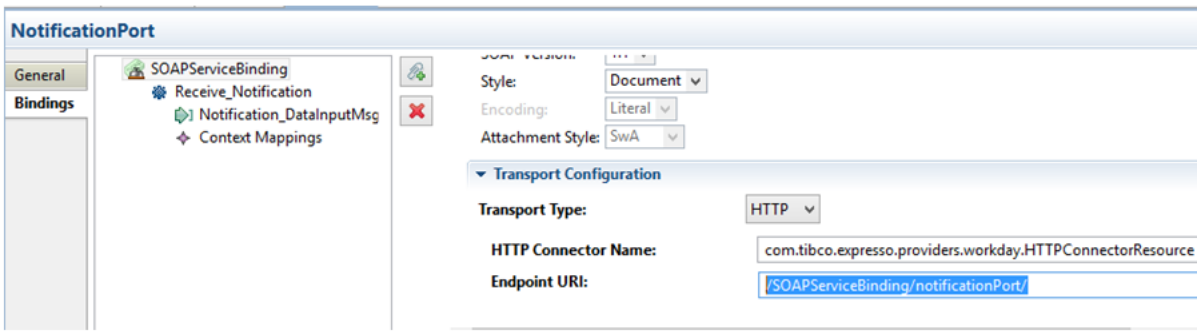
The employee is terminated and workday sends a notification which triggers the `WorkdayEvent` service. The `NotifyTerminateEmployee` activity sends a notification to the Expresso server with data such as employee name, employee ID, date when the employee is terminated, and the department.

Running a TIBCO ActiveMatrix BusinessWorks 6.x Project

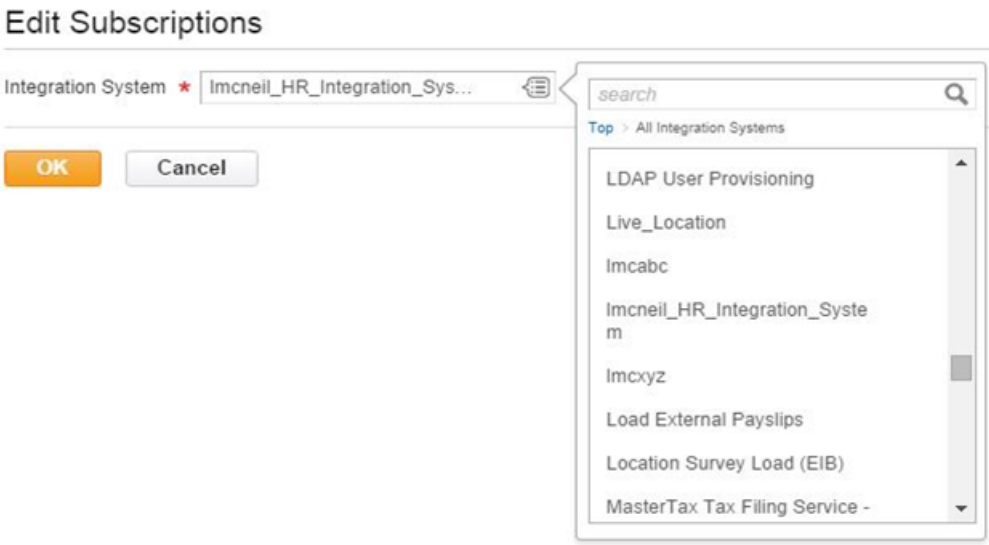
The `WorkdayEventProcess` service uses `Notification.wsdl` and implements the `Receive_Notification` operation. It has two Expresso Notify activities, `NotifyHireEmployee` and `NotifyEmployeeFired` which send a received notification to Expresso.

Procedure

1. Configure the shared resource:
   - ExpressoServer Resource: Specify the Expresso server host and port.
   - WorkdayHTTPConnector: Specify values for `PROVIDER.HOST` and `PROVIDER.PORT`.
   - HTTPConnector: Specify values for `WORKDAY.NOTIFICATION.PORT`. (**Note:** This port is used by the web service to listen for events from Workday.)
2. Configure the notification URL:
   a) Note the SOAP Endpoint URI of the `WorkDayEvent` service.

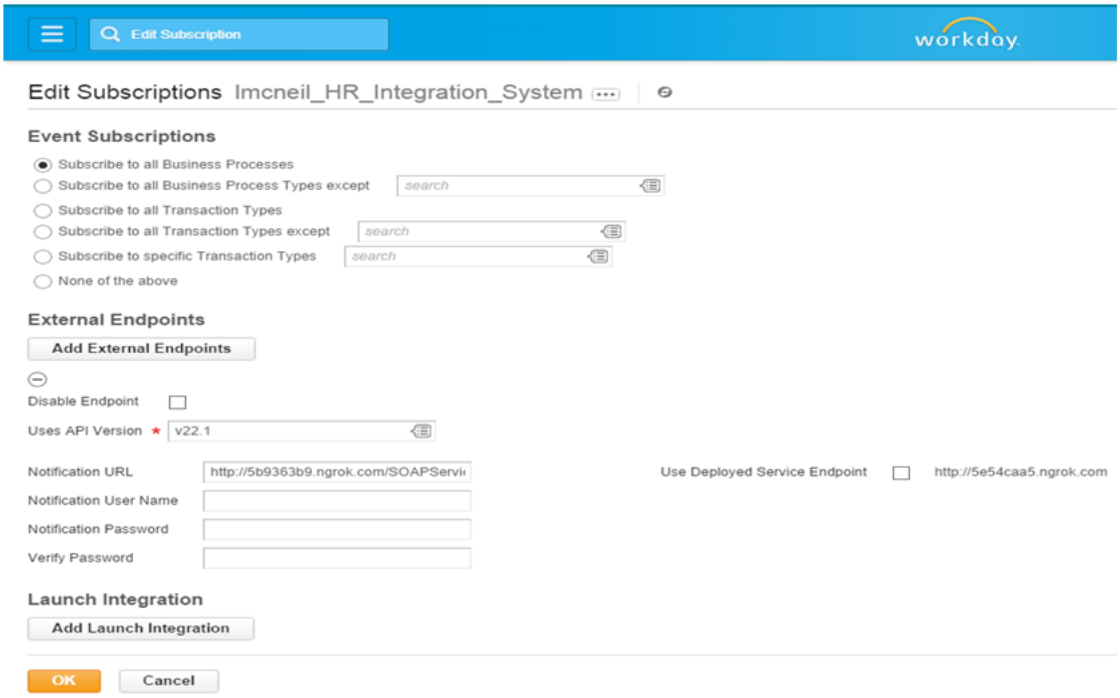


   b) In Workday, search for **Edit Subscriptions**.
   c) On the **Edit Subscriptions** screen, under **Integration System**, choose the integration system and click **OK**.

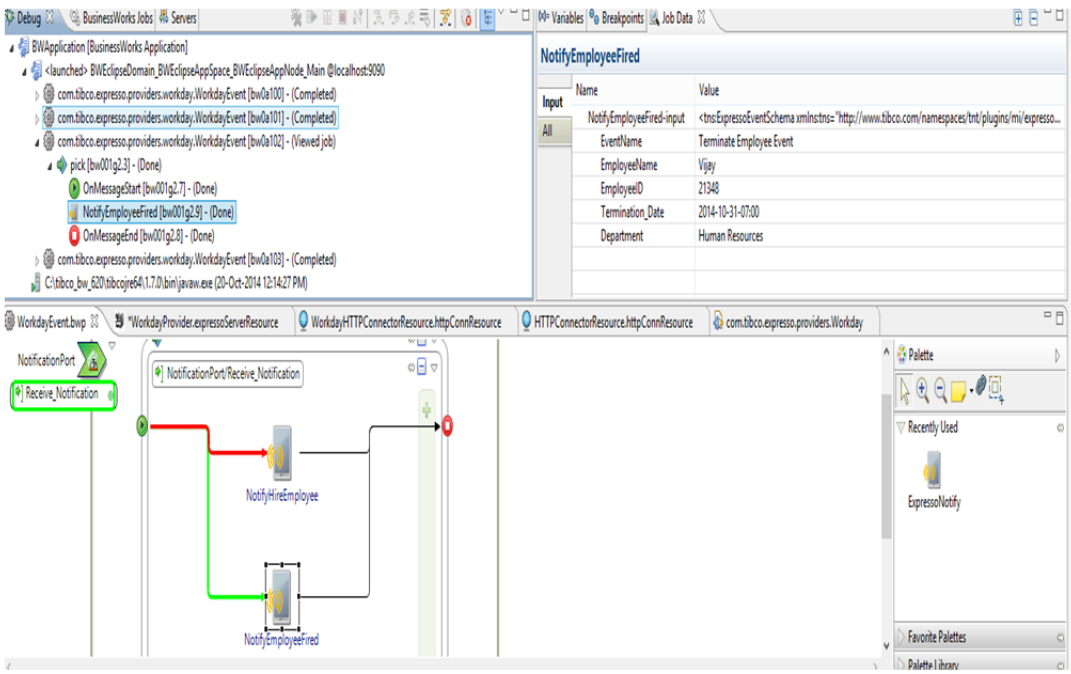


   d) Select **v22.1** as the **Uses API Version**. Specify the **Notification URL** and click **OK**.

> Configure the **Notification URL** in the workday instance with the URI where the Workday BusinessWorks application is hosted, followed by the SOAP endpoint URI.



3. Run the project and execute the **Hire Employee** or **Terminate Employee** business process in Workday.
   The following figure shows the service for the **Terminate Employee** Event.



SalesForce Provider Sample

The SalesForce provider in TIBCO ActiveMatrix BusinessWorks 6.x demonstrates how to configure the notification message that is sent when a Lead is converted in Salesforce. This section describes:

- Creating a SalesForce provider

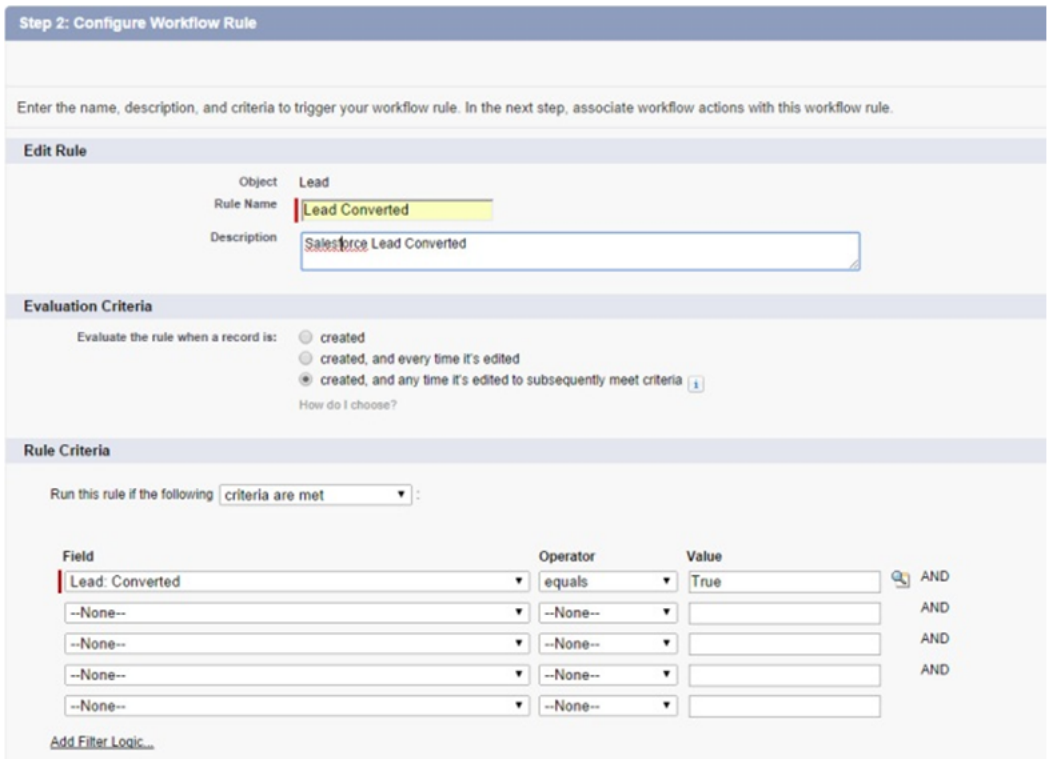
- Editing the saved WSDL
- Configuring the TIBCO ActiveMatrix BusinessWorks 6.x project

Creating a SalesForce Provider

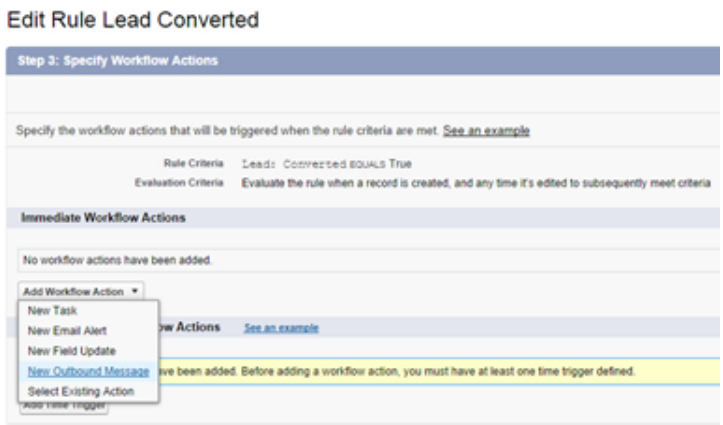
To create a SalesForce provider, you must open an account with https://developer.salesforce.com and configure outbound messages from it.

Procedure

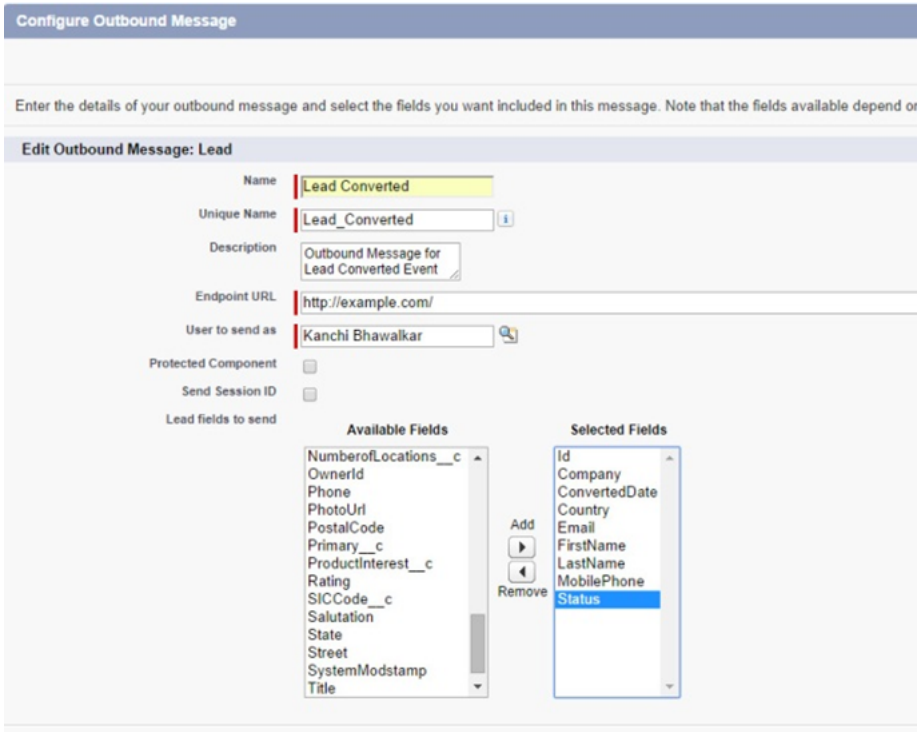
1. Open an account with https://developer.salesforce.com. After signing up, verify the account and log in to the homepage of the developer account.
2. On the home page, click **Create > Workflow & Approvals > Workflow Rules**. The All Workflow Rules screen appears. Click **New Rule** to create a new workflow rule.
3. Select **Lead** as the Object. Configure the rule fields as follows:



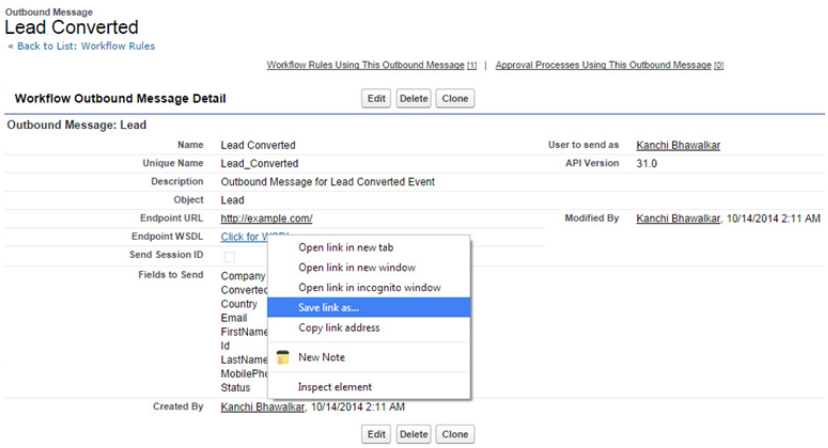
4. After saving it, add **New Outbound Message** as a workflow action.



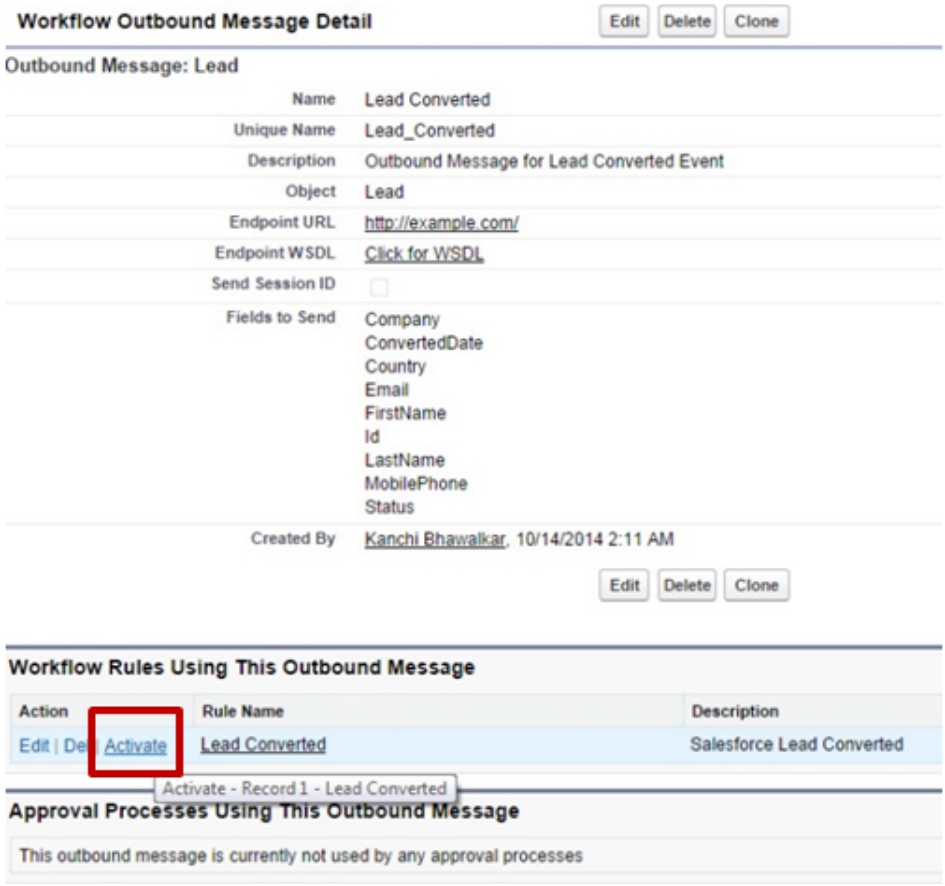
5. Configure an outbound message for the workflow rule. Configure the endpoint URL to a dummy value; you need to change it later. In **Lead Fields**, select fields from the lead object that you want to receive when a lead is converted in Salesforce. Salesforce sends a SOAP message along with these fields.



6. Save the outbound message, and open it to view the following tab. Salesforce provides a WSDL which confines to the SOAP outbound message that was configured. Click **Save link as** to save this WSDL file. Use the WSDL file to configure a SOAP service in TIBCO ActiveMatrix BusinessWorks 6.x.



7. After saving the WSDL, make sure that you activate the rule using the outbound message.



8. Click on the **Leads** tab in Salesforce. Click **New** to create a new lead.
9. Configure a new lead and save it. This lead is used later to trigger the Salesforce event.

Editing the Saved WSDL

Procedure

1. Remove the `binding` and `service` tags to make the saved WSDL abstract.
2. Delete the part highlighted in the following figure:

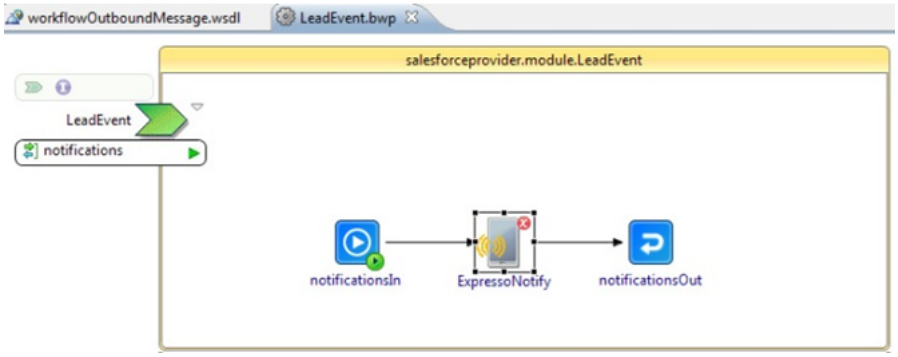
```
workflowOutboundMessage.wsdl
106      </message>
107      <message name="notificationsResponse">
108          <part element="tns:notificationsResponse" name="response"/>
109      </message>
110
111      <!-- PortType -->
112      <portType name="NotificationPort">
113          <operation name="notifications">
114              <documentation>Process a number of notifications.</documentation>
115              <input  message="tns:notificationsRequest"/>
116              <output message="tns:notificationsResponse"/>
117          </operation>
118      </portType>
119
120      <!-- Binding
121          You need to write a service that implements this binding to receive the notifications
122       -->
123      <binding name="NotificationBinding" type="tns:NotificationPort">
124          <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
125
126          <operation name="notifications">
127              <soap:operation soapAction=""/>
128              <input>
129                  <soap:body use="literal"/>
130              </input>
131              <output>
132                  <soap:body use="literal"/>
133              </output>
134          </operation>
135      </binding>
136
137      <!-- Service Endpoint -->
138      <service name="NotificationService">
139          <documentation>Notification Service Implementation</documentation>
140          <port binding="tns:NotificationBinding" name="Notification">
141              <soap:address location="http://example.com/"/>
142          </port>
143      </service>
144
145  </definitions>
```

3. Save the WSDL.
   You can now create a web service in TIBCO ActiveMatrix BusinessWorks 6.x that implements the WSDL.

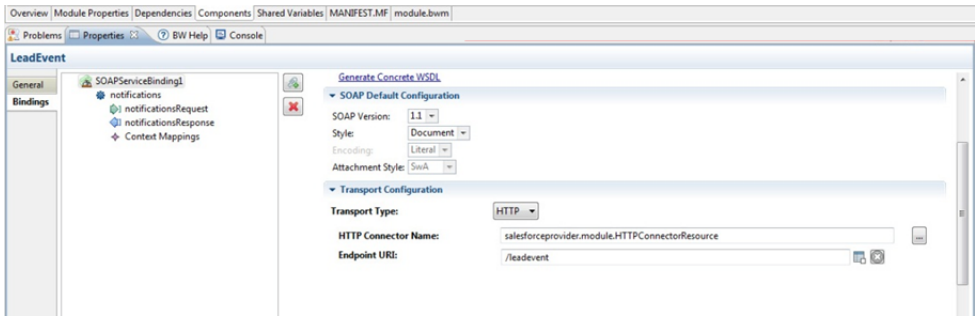
Configuring the TIBCO ActiveMatrix BusinessWorks 6.x Project

Procedure

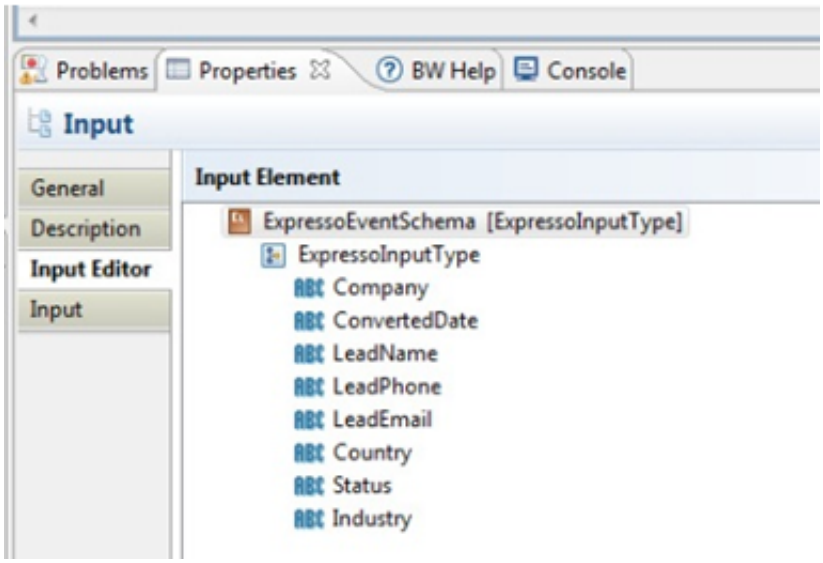
1. Using the abstract WSDL created in Editing the Saved WSDL, create a TIBCO ActiveMatrix BusinessWorks 6.x service. Implement the **notifications** operation for **NotificationPort** portType. Add the **ExpressoNotify** activity to the process.

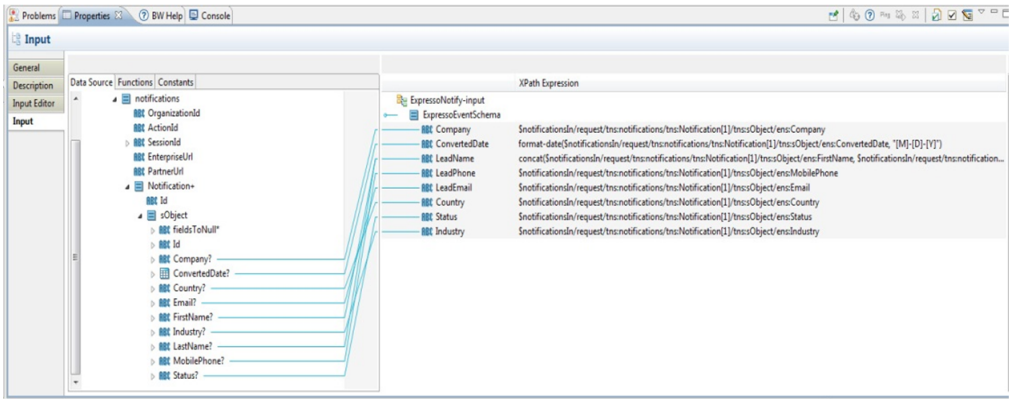


2. Configure the SOAP service binding for the service. Note the endpoint URL configured here. Also note the port that is configured for the HTTPConnectorResource. In this case, it is 8080.



3. Configure the schema for the **ExpressoNotify** activity using the **Input Editor** tab as showin in the following figure. This is your event schema; map it according to the incoming SOAP message.





4. Create an Expresso Provider shared resource and configure it as follows:

   Name: SalesForceEventProvider. This is displayed as the domain name of the provider.

   Expresso Server Cient Config: the server configuration details.

   Expresso Event Configuration > Event Name: this name is directly seen as the pod name.

   Expresso Event Configuration > Implementing Process: select the process containing the **ExpressoNotify** activity that was configured.

5. Similarly, create a workflow rule in Salesforce to send an outbound message when changes are made to a campaign. The name of the campaign is `Test Campaign`. The criteria for the workflow rule is "`Campaign: Campaign NameEQUALSTest Campaign`".

The fields to be included in the outbound message are: ActualCost, BudgetedCost, ExpectedResponse, Id, Name, NumberOfContacts, NumberOfLeads, NumberOfResponses, NumberSent, and Status.

Before implementing the service in TIBCO ActiveMatrix BusinessWorks 6.x, make sure you change the targetnamespace of the second WSDL, as it is the same as the first one and may create errors on startup.

6. Implement the service and add the **ExpressoNotify** activity as shown for the first service. Configure it as a second event with the SalesforceProvider shared resource. Note the endpoint URI for the binding.
7. Start the application. The SalesforceProvider is registered with the Expresso server with the two events and their schemas.
8. Configure the outbound message in the SalesForce cloud instance with the URI where the SalesForce BusinessWorks application is hosted, appending the URI from the SOAP endpoint"`/leadevent`" . Refer to Creating a SalesForce Provider.
9. Trigger the service by converting a lead or changing any field of the `Test Campaign`.