Implementing Event-Driven Architecture with Software AG Products

Version 9.12

October 2016

This document applies to Software AG Product Suite Version 9.12 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: SAG-EDA-912-20180503

Table of Contents

About this Guide

This document gives you an overview of Software AG Business Events, which is Software AG's infrastructure for managing simple event-based interactions and more complex event analysis for pattern matching in real-time.

Software AG Business Events offers the following key features and functionality:

- It is a solution for creating, processing, and monitoring events.
- It provides the infrastructure to rapidly build and adapt event-driven applications.
- It improves an organization's ability to comprehend the current state of the physical world and business environment and react rapidly to changes.

Deprecation of Software AG NERV

The Software AG NERV component and the low-level Java API to it is now deprecated. Note that despite this deprecation, Software AG products continue to communicate using events and you can still use the high-level webMethods Integration Server built-in services to send and receive events.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* .
UPPERCASE	Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+).
Italic	Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text.
`Monospace font`	Identifies text you must type or messages displayed by the system.

Convention	Description
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com. The site requires Empower credentials. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

1 What is Event Driven Architecture?

Concepts

Software AG Event-Driven Architecture (EDA) is a methodology that allows you to process the events that shape your everyday business environment. An *event* can be something as simple as an electrical component being switched on or off, or more complicated, such as a bid being made in an auction house for the painting of a great master. An event represents something that has happened, and it may or may not require some follow-up action to be taken. An event can also represent something that was expected to happen but has failed to happen.

We all experience an event driven world every day. We walk through an airport and hear announcements of planes arriving and departing (these are events). The announcements (events) are emitted even if nobody is listening. If, however, the announcement is for my plane, then I will accept it, and start taking an action. My action may be to run to the gate, while someone else's activity might be to walk and get a snack. This is the basis of EDA: events are emitted and listeners can either take action on them or ignore them. The action I take is totally self-contained and does not rely on the activity of another person.

The term *event-driven* indicates that when an event happens, it can have a significance which requires some follow-up action to be taken. An event can be noticed by several observers or listeners, and each observer can react to the event differently. For one observer, an event might represent some critical status which requires immediate action. For another observer the same event might not be relevant at all.

The significance of a single event is sometimes only visible when viewed in the context of other events that together form a pattern. For example, if cash is withdrawn at a cash machine in the city center, this is not unusual, but if cash is withdrawn at many different cash machines on the same day throughout the city using the same card, this might raise the suspicion that the card is stolen.

If we change the focus from everyday events that we observe in the world around us to events that can have an influence on the way a company does its business, we can see that events such as the following could trigger a component in a predefined workflow:

- A trade order has been issued.
- A reading of sensor data (e.g. GPS, temperature or RFID reader) has occurred.
- A business process has reached completion.
- A software component has started successfully.

The existence of an event can be the trigger for processes, such as the invocation of a service, the initiation of a business process, or the publication of relevant information. Software AG Event-Driven Architecture picks up on these ideas and provides a set of concepts for dealing with events at all stages throughout the processing chain.

Components of the EDA Environment

An EDA system typically contains the following components:

- An *event bus* that routes the events. The event bus is the central nervous system of an EDA system. The event bus supports multiple channels simultaneously, with each channel being used to transport logically related events.
- A *service bus* that connects applications to the event bus.
- One or more *event publishers* to create events and publish them in channels on the event bus.
- One or more *event subscribers* that read the events from the event bus and perform a preset action on the basis of the information contained in the event.
- A Complex Event Processing engine that executes queries that process incoming events, and publishes simple and complex events for post-processing actions. It must be able to handle a high throughput of events with extremely low latency and evaluate patterns in event data.
- Logging, monitoring, and performance tools for administration purposes.
- A dashboarding tool to create interactive, analytical, real-time dashboards.
- A Business Rules engine to capture, automate, and flexibly change business policies.
- Tools for governing event types and event channels.
- Tools for creating event-driven applications.
- A central store for persisting event data at rest.
- A Business Activity Monitoring solution to define and monitor events and event patterns that occur throughout an organization.
- An integrated data grid technology to support scale to enterprise-class event processing use cases for data in use.

2 How is EDA Implemented by Software AG?

Software AG EDA Components

The Software AG EDA portfolio consists of the following components:

EDA Component	Software AG Component	Additional Information
Event Bus	webMethods Universal Messaging or webMethods Broker	
Service Bus	Software AG NERV	NERV is Software AG's solution for event routing and transformation in the Software AG Common Platform.
	webMethods Integration Server	Integration Server provides built-in services for EDA.
Event Type Repository	Event Type Store (run-time component)	The Event Type Store is a run-time repository that contains schemas of the events on the event bus. The event types are required in order to interpret the payload of events on the event bus.
	CentraSite Registry Repository (design-time component)	CentraSite enables you to archive, categorize and govern event type definitions.
Event Type development tool	Software AG Designer, Events Development perspective	An Eclipse-based tool for creating and maintaining event types.
CEP Engine	Apama Correlator	

EDA Component	Software AG Component	Additional Information
CEP application design tool	Apama perspectives in Software AG Designer	Apama perspectives running in Software AG Designer are the main entry point for developing Apama applications.
Dashboarding tool	Software AG MashZone	
	Software AG MashZone	
Business Rules tool	webMethods Business Rules	
A tool for monitoring events and event patterns	webMethods Optimize Analytic Engine	The webMethods Business Activity Monitoring solution enables you to define and monitor events and event patterns throughout an organization.
Monitoring tool	Event Bus Console	A text-based output console for monitoring traffic on event streams.
Sample event publishing tool	Event Generator	A tool for creating sample events and publishing them to the Event Bus. You can download it from Software AG Tech Community.

NERV

Software AG NERV is an integration framework for event routing and transformation. It plays a pivotal role in ensuring the communication between event-enabled applications in the Software AG Common Platform.

NERV offers the following capabilities:

- Simple management through common integration patterns.
- Easy data exchange between Software AG components.
- Support of heterogeneous data landscape.

Event Types

An event type is a schema definition that describes how events in an event stream are structured. Event types are first-class objects that are declared at a high level in the environment and can be processed by webMethods and non-webMethods products.

Events in the same stream always have the same payload structure. The schema defines which data fields are present in each event, the data type of each field, and the order in which the fields appear. Each event stream has exactly one event type associated with it. One event type can be used as the schema for more than one event stream. All event publishers on a given stream must ensure that their published events comply with the stream's schema, and all subscribers must be aware of the schema that describes the events received. In this respect, the schema represents a contract between publishers and consumers of events of a specific type.

Event types are implemented as schemas that conform to the W3C XML Schema (XSD) specification. Within the Event Type Editor, they are displayed as a hierarchy of nodes representing the content of the event. The nodes can be field nodes, composite nodes, or references to structures in other schemas. Field nodes are leaves within the node hierarchy enabling users to specify typed text strings in the XML event. Composite nodes are containers for field nodes, composite nodes, and reference nodes. At the underlying XSD level:

- The root node is invisible and is represented as a top-level element declaration with the `substitutionGroup="eda:Payload"` attribute
- Composite nodes correspond to element declarations with a complex content model
- Field nodes are element declarations with a simple type
- References refer to top-level element or type definitions in other component schemas.

You can specify a cardinality for all visible nodes, whereas the hidden root node has a fixed cardinality of `1`, denoting that a valid XML document has exactly one root element.

The XSD, as generated by the Event Type Editor, is only a subset of the full XML Schema specification. However, you can use an almost arbitrary XSD as event schema, as long as you do the following:

- Add the following import statement:

```
<xsd:import namespace="http://namespaces.softwareag.com/EDA/Event"
schemaLocation="Event/Envelope.xsd"/>
```

> **Note:** Depending on the location of the event type schema within the Event Types directory, the `schemaLocation` attribute may contain additional leading `../` steps for moving up in the directory hierarchy.

- Add the `substitutionGroup="eda:Payload"` attribute to the declaration of the element to be the root of the event XML.

Here is a section of a sample event type schema:

```
<xsd:complexType name="PartInventoryLowType">
  <xsd:annotation>
    <xsd:documentation>Report inventory low for a part</xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
     <xsd:element name="Part">
       <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="ItemID" type="xsd:string" minOccurs="1"/>
          <xsd:element name="ItemName" type="xsd:string" minOccurs="0"/>
          <xsd:element name="Model" type="xsd:string" minOccurs="0" />
          <xsd:element name="Color" type="xsd:string" minOccurs="0" />
          <xsd:element name="Shape" type="xsd:string" minOccurs="0" />
        </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
     <xsd:element name="InventoryLevel" type="xsd:integer"/>
     <xsd:element name="DesiredInventoryLevel" type="xsd:integer"/>
  </xsd:sequence>
</xsd:complexType>
```

Here is a sample event instance:

```
<PartInventoryLow>
  <Part>
    <ItemID>ABC123</ItemID>
    <ItemName>Widget </ItemName>
    <Model>XYZ</Model>
    <Color>Silver</Color>
    <Shape>Oval</Shape>
  </Part>
  <InventoryLevel>58</InventoryLevel>
  <DesiredInventoryLevel>1000<DesiredInventoryLevel>
</PartInventoryLow>
```

Event Type Store

The Event Type Store provides a central location per installation where predefined and user-defined event types are stored. This shared location is used by all EDA participants within the respective installation to retrieve deployed custom event types at run time.

At design time, a local copy of the predefined event types of the Event Type Store is available for reference. By default, it is located in the *Software AG_directory*/common/PredefinedEventTypes directory. You can import this directory as an existing project in Software AG Designer to inspect the event types. User-defined event types can be created using the Event Type Editor and stored in the local copy.

Event types in the local copy must be deployed to the runtime store, so that EDA participants that process an event stream can retrieve the schema definition of the event.

For more information about deploying Event Types, see "Deploying EDA Assets" on page 42.

Event Type Governance

Use CentraSite to register, categorize, and govern event definitions. The Events Development perspective in Software AG Designer offers publish and unpublish functionality for the transfer of event types to and from CentraSite. You can also use CentraSite to inspect the dependencies between event type schemas and imported component schemas.

Event Bus

The event bus functionality for EDA components is implemented by webMethods Universal Messaging or webMethods Broker by using the JMS protocol.

webMethods Universal Messaging is a Message Oriented Middleware product that guarantees message delivery across public, private and wireless infrastructures.

webMethods Broker is a high-performance message server that provides organizations with the foundation for state-of-the-art integrated electronic business applications. Broker provides communication among distributed application components in the event-driven architecture.

Note: In addition to webMethods Universal Messaging and webMethods Broker, Software AG EDA supports the use of several third party JMS providers, such as WebSphere MQ, WebLogic, Sonic MQ, Oracle Streams Advanced Queuing (AQ) and JBoss Messaging.

Event Bus Channels and Topics

webMethods Universal Messaging and webMethods Broker are JMS (Java Message Service) providers that support point-to-point and publish and subscribe messaging. This enables JMS applications to profit from the reliability and performance of Universal Messaging or Broker. The message protocol used for webMethods Business Events is JMS.

For webMethods Business Events, EDA participants publish their events to JMS topics and subscribe to JMS topics. Software AG EDA uses the JNDI mechanism for directory and naming services.

The default JNDI destination and JMS topic name has the format:

```
Event::Namespace::EventTypeName
```

The namespace is the path within the Event Type Store where the event type is located. Backslash characters in the path are replaced by the channel name delimiter (by default, two colons "::").

For example:

- For the CableboxHealth event type, which is located in the WebM/Sample/CableboxMonitoring folder in the Event Type Store, the default channel name is:

```
Event::WebM::Sample::CableboxMonitoring::CableboxHealth
```

- For the noTns event type, which does not have a namespace and is located in the WebM/External folder in the Event Type Store, the default channel name is:

```
Event::WebM::External::noTns
```

Event Structure

Each event on the event bus is composed of the following parts:

- **Header**

  The header contains system-defined event attributes:

Event Attribute	Description
Start	Start date and time of the event.
End	Optional. End date and time of the event. The use of this field depends on how the event is being used. If the value is absent, the consumer application may set a default one, such as start time plus one millisecond.
Kind	Optional. Indicates whether the event is a new event (`Event`) or a heartbeat (`Heartbeat`). A heartbeat event indicates the temporal progress of the stream. If a value is not specified, the default is `Event`.
Type	The unique identifier of the event type. Event types use qualified names (QNames) as the mechanism for concisely identifying the particular type. The event type combines the URI and local name as a string. For example: `{http://namespaces.softwareag.com/EDA/WebM/Process/1.0}ProcessInstanceChange` is the event type identifier that reports changes to a process instance. **Note:** Event types without a namespace use only their local name as event identifier. For example, the noTns.xsd event type's identifier is `noTns`.
Version	Optional. The version of the event type with which the event instance is compatible. Users specify this value if they have

Event Attribute	Description
	chosen to support event type versioning. An event should not specify a version if the event type supports versioning.
CorrelationID	Optional. A unique identifier used to associate the event instance with other event instances.
EventID	Optional. A unique identifier of the event. EDA clients can distinguish between different event instances.
Priority	Optional. The priority of the event. Possible values are: ■ `Normal` (default value) ■ `High`
ProducerID	Optional. A unique identifier of the event producer.
UserID	Optional. A unique identifier of the user who emitted the event.
FormatVersion	Optional. The version of the event format. NERV creates automatically a value for this attribute. Check the value in the received event to see if the event body contains headers and payload. ■ If this attribute is not present in the event headers, the event body contains both headers and payload. ■ If this attribute is present in the event headers, and its value is `9.0`, the event body contains only payload.
CustomHeaders	Optional. A parent header element for any user-defined headers included as sub-elements.

All messages support the same set of header fields. Header fields contain predefined values that allow clients and providers to identify and route messages. Each of the fields supports its own set and get methods for managing data; some fields are set automatically by the send and publish methods, whereas others must be set by the client. The header contains the start and end timestamp of the event.

- **Filterable Properties (optional)**

  webMethods Events support JMS message selector properties, also known as filterable properties. If you mark a field node in the event type as filterable, its value is added to the filterable properties. For example, for the BoothDemo event shown below, if the *Producer* and the *Presenter* fields are marked as filterable, the following key-value pairs are added:

```
PulseCommon$Producer=”Event Generator”
Presenter=”dada”
```

At run time, when events are transmitted over the event bus, event consumers subscribing to the event bus to receive events may apply a message selection filter, so that only events that match certain selection criteria are consumed. These criteria can be, for example, whether the event type is a normal event or a heartbeat, or whether the value of an element from the body of the event exceeds a certain value.

Event header elements are always added to the filterable properties with an additional prefix `$Event$` in the key. If a node in the event schema is marked as filterable, the element is added to the filterable properties when the event is published. This allows event receivers to use filterable properties based on element values.

- **Body**

  The body contains the payload of the event. The body contains the data fields of the event, as specified in the event's schema.

Here is a sample event:

```
<evt:Event xmlns:evt="http://namespaces.softwareag.com/EDA/Event>
   <evt:Header>
     <evt:Type>{http://namespaces.softwareag.com/EDA/WebM/Sample/Pulse}Pulse
     </evt:Type>
     <evt:Start>2012-05-20T16:53:46.918-06:00</evt:Start>
     <evt:End>2012-05-20T16:53:47.918-06:00</evt:End>
     <evt:Kind>Event</evt:Kind>
     <evt:EventId>0f375801-dbd4-4a46-9f70-7015deca6c80</evt:EventID>
   </evt:Header>
   <evt:Body>
     <p1:BoothDemo
      xmlns:p1="http://namespaces.softwareag.com/EDA/WebM/Sample/Pulse">
     <p1:PulseCommon>
     <p1:Producer>Event Generator</p1:Producer>
     <p1:Subject>Pulse Test Event</p1:Subject>
     <p1:Coordinates>
       <p1:Longitude>87.44988659217529</p1:Longitude>
       <p1:Latitude>83.11056319477842</p1:Latitude>
     </p1:Coordinates>
     </p1:PulseCommon>
     <p1:Presenter>dada</p1:Presenter>
     <p1:DemoTopic>Demo2</p1:DemoTopic>
     <p1:Date>2012-04-05T17:09:33.112+03:00</p1:Date>
     </p1:BoothDemo>
   </evt:Body>
</evt:Event>
```

Heartbeats

A heartbeat is a special kind of event without a payload. It indicates that the event bus channel on which it is being sent is active but that no payload events are currently being sent on the same channel.

The header of a heartbeat event specifies event type corresponding to the channel the heartbeat is being sent on, the start date and time of the heartbeat, and the *Kind* header field is set to `Heartbeat`.

An example of the use of heartbeats is for CEP applications, in which heartbeats can be used within a non-event detection query to determine whether the timespan in which a certain pattern did not occur has expired.

Some EDA participants may not support heartbeats. Event receivers can suppress receiving heartbeat events by using message selection filtering on the value of the *Kind* attribute. The following message selector can be used for this purpose:

```
$Event$Kind <>'Heartbeat' or $Event$Kind is null
```

Here is a sample heartbeat event:

```
<evt:Event xmlns:evt="http://namespaces.softwareag.com/EDA/Event">
  <evt:Header>
    <evt:Type>{http://namespaces.softwareag.com/EDA/WebM/Sample/
       InventoryMgmt/1.0}PartInventoryLow</evt:Type>
    <evt:Start>2010-05-20T16:53:46.918-06:00</evt:Start>
    <evt:Kind>Heartbeat</evt:Kind>
      <evt:EventId>0f375801-dbd4-4a46-9f70-7015deca6c80</evt:EventId>
  </evt:Header>
</evt:Event>
```

3 How EDA Components Connect to the Event Bus

Mechanisms for Connecting to the Event Bus

Software AG Event-Driven Architecture (EDA) supports the following ways of connecting to the Event bus:

- **NERV**

  NERV is a solution that enables Software AG products to communicate using events. It uses the Apache Camel integration framework for event routing, filtering, and variation. NERV uses a Camel component configured for webMethods Universal Messaging or Broker as a transport layer and JMS topics as destination endpoints.

  For more information about using NERV, see "Connecting to the Event Bus Using NERV " on page 22.

- **Integration Server**

  The Integration Server interacts with many Software AG products, and provides pre-configured public services for use in the EDA environment. It supports JMS connections to webMethods Universal Messaging and Broker, and it can act as an event publisher or subscriber. As a publisher, Integration Server can convert IS document types into events and publish them to the event bus. As a subscriber it can transform received events into IS document types.

  In addition, the Integration Server:

  - Receives events from the event bus using JMS triggers.
  - Includes built-in services for EDA to send, subscribe to, and unsubscribe from EDA events via NERV.

  The Integration Server offers a variety of bus connectivity and data transformation features, and it contains functionality that enables you to transform non-Software AG EDA event data into Software AG EDA event data. If a third party product generates events that do not conform to the webMethods events schema, they can be converted to the webMethods event schema by using the document transformation capabilities of Integration Server. Also, Integration Server supports sending non-Software AG EDA events to the event bus.

  For more information about using the EDA-related Integration Server built-in services, see the PDF publication *webMethods Integration Server Built-In Services Reference*.

Connecting to the Event Bus Using NERV

NERV is included by default in all Software AG Common Platform profiles. This documentation assumes you are familiar with and have a working knowledge of OSGi implementation and architecture.

To interact with NERV, you should:

- Verify your transport layer configuration, and modify it if necessary, as described in "Configuring the Transport Layer for NERV " on page 26.
- Emit events.
- Create a custom NERV emit logic (optional), as described in "Creating Custom NERV Emit Bundles" on page 33.
- Subscribe to an event type from your own application, as described in "Creating Custom NERV Consume Bundles" on page 35.

Connecting to the Event Bus Using the EDA-Related Integration Server Built-In Service

Integration Server enables you to transform non-Software AG EDA event data into Software AG EDA event data using the `pub.event.nerv:send` built-in service. Integration Server constructs an EDA event using the parameters defined in the service and sends the event to the event bus using NERV. You can use the pub.event.nerv:subscribe built-in service to subscribe to events that NERV receives, and unsubscribe using pub.event.nerv:unsubscribe.

Important: The procedure below explains how to use the pub.event.nerv:send service to send events. It assumes that you are familiar with working with built-in services and flow services in Software AG Designer. For more information about IS built-in services, see the PDF publication *webMethods Integration Server Built-In Services Reference*. For more information about working with flow services, see the PDF publication *webMethods Service Development Help*.

To send EDA events using the `pub.event.nerv:send` IS built-in service

1. In the Service Development perspective in Designer, create a new document type from an existing event type, for example the PartInventoryLow event.
   a. Use the PartInventoryLow event name as a name for the new document type and click **Next**.
   b. Select XML Schema as source type and click **Next**.
   c. Browse to the PartInventoryLow event type in the Event Type Store and click **Next**.

      By default, the PartInventoryLow event type is located in the *Software AG_directory*\common\EventTypeStore\WebM\Sample\InventoryMngt\1.0 directory.
   d. Select the PartInventoryLow element as the payload root node and click **Next**.
   e. On the next page of the wizard you can configure the namespace prefixes to be used for representing namespaces found in the schema. Leave the entries unmodified, and click **Finish**.

2. Create a new empty flow service.
3. In the Input/Output tab of the Flow service editor, in the Input Parameters panel, insert a document reference to the new document type you created in step 1.
4. In the Tree tab of the Flow service editor, insert an INVOKE `pub.event.nerv:send` step.
5. In the Pipeline view, link the document reference from the Pipeline Input area to the event/body node of the `pub.event.nerv:send` service in the Service Input area.
6. In the Service Input area, set the value of the *Kind* variable to `Event`.
7. In the Service Input area, set the value of the *Type* variable to the full event type name, in this example `{http://namespaces.softwareag.com/EDA/WebM/Sample/InventoryMgmt/1.0}PartInventoryLow`.
8. In the Service Input area, set the value of the *documentTypeName* variable to refer to the document type you created in step 1.

   This is required in order to assert that the namespace declarations are added to the XML document emitted as an EDA event.
9. Right click and select **Run As > Run Flow Service** to send events using the flow service. You can use the following input:

```
<?xml version="1.0" encoding="UTF-8"?>
<IDataXMLCoder version="1.0">
<record javaclass="com.wm.data.ISMemDataImpl">
<record name="PartInventoryLow" javaclass="com.wm.data.ISMemDataImpl">
<record name="ns:PartInventoryLow" javaclass="com.wm.data.ISMemDataImpl">
<record name="ns:Part" javaclass="com.wm.data.ISMemDataImpl">
  <value name="ns:ItemID">154</value>
  <value name="ns:ItemName">Car</value>
  <value name="ns:Model">Audi A4</value>
  <value name="ns:Color">Grey</value>
  <value name="ns:Shape">Standard</value>
</record>
  <value name="ns:InventoryLevel">1</value>
  <value name="ns:DesiredInventoryLevel">2</value>
</record>
</record>
</record>
</IDataXMLCoder>
```

Note: The *name* attribute of the second `<record>` element must match the name of the document reference configured as the input of the flow service.

4 Configuring NERV

Configuring the Transport Layer for NERV

When you install NERV, the default transport layer for NERV is defined as `nsp://<host_name>:9000`.

You can configure the NERV transport layer in two ways:

- By modifying the **Default JMS Provider** configuration property in Command Central.

  When NERV starts, it instantiates the transport layer, as defined by the value of the **Default JMS Provider** property. If no such property exists, or if it has an empty value or contains an invalid configuration, NERV displays a warning message. Every time your EDA applications try to emit events using the default emit logic or to subscribe to event types, NERV produces a warning message in the log file of the corresponding Common Platform profile and any existing subscriptions to default endpoints are deleted. You must provide a valid property for the NERV transport layer and subscribe again to the endpoints.

  For more information about modifying the NERV transport layer configuration property, see "Modifying the Transport Layer for NERV " on page 26.

- By deploying a user-defined NERV component bundle, which defines a Camel JmsComponent, called `nervDefaultJMS`. When activated in the Common Platform, this bundle takes precedence over the default behavior.

  If a user-defined NERV component bundle is activated in the Common Platform, it takes precedence over the default transport layer defined by the value of the **Default JMS Provider** configuration property in Command Central. This means that all modifications to the **Default JMS Provider** configuration property will be ignored. If the user-defined component bundle is stopped or uninstalled, NERV re-instantiates the default component configured using the **Default JMS Provider** configuration property in Command Central.

  For more information about creating custom NERV component bundles, see "Creating Custom NERV Component Bundles" on page 30.

Modifying the Transport Layer for NERV

Using the Software AG Command Central user interface, you can modify the initial transport layer definition and specify a different transport layer to be used by NERV for each profile in a given installation. The default NERV behavior supports the use of webMethods Universal Messaging or webMethods Broker as JMS providers.

Note: Before you define a custom transport layer, you must make sure that it has been properly configured and is running in the Common Platform.

Before You Begin

Before you modify the NERV configuration using the Command Central user interface, make sure that NERV nodes have previously been added to the Command Central landscape. Note that:

- The NERV component ID has the following format:

  ```
  OSGI-profile_name-NERV
  ```

  where *profile_name* is the name of the Common Platform profile in which a NERV instance is running. For example, in the case of a NERV instance installed and running in the IS_default profile, the NERV component ID in the Command Central landscape is:

  ```
  OSGI-IS_default-NERV
  ```

- A single NERV configuration called COMMON-SYSPROPS-com.softwareag.eda.nerv.properties exists per every webMethods product suite installation profile.

For more information about using the Command Central user interface, see *Software AG Command Central Help*.

Modifying the Transport Layer Configuration

To modify the default transport layer definition used by NERV

1. In Command Central, navigate to **Environments > Instances > All > <*profile_name*> >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.
2. Change the default transport layer used by NERV by modifying the value for the **Default JMS Provider** property.

   Note: This property supports only the use of webMethods Universal Messaging or webMethods Broker as JMS providers.

3. Save your changes.

NERV detects that the configuration has been updated, and starts to use the new settings automatically. If you have modified the transport layer definition, all emit and consume routes that are created using the NERV API are automatically restarted and begin to use the new transport layer definition. However, any routes that are defined inside custom emit or consume bundles need to be updated manually.

Modifying NERV Error Handling

NERV provides some error handling properties to ensure the successful delivery of events. You can modify these properties using the Command Central user interface.

To modify NERV error handling properties

1. In Command Central, navigate to **Environments > Instances > ALL > <*profile_name*> >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.
2. Modify the values for the following properties:

Property	Description
Redelivery Attempts	The maximum number of attempts that NERV makes to redeliver events, in case the initial delivery attempt fails. The default value is `100`. Set to `-1` for infinite redelivery attempts. **Note:** When you set up the guaranteed delivery level to `MAXIMUM_STRONG` or `MAXIMUM_EVENTUAL`, the value for this property is disregarded. Instead, NERV attempts to redeliver events infinitely.
Redelivery Delay	The interval (in milliseconds) at which NERV makes the redelivery attempts. The default value is `3000`.
Dead Letter Channel	The location where NERV sends any undelivered events after the maximum number of redelivery attempts has been reached. The default value is `@url:sag.install.area/common/nerv/dlc`. **Note:** When NERV runs in the Common Platform, the `@url:sag.install.area` token is dynamically resolved to point to your Software AG installation directory.

3. Save your changes.

Guaranteed Delivery of Events with Ehcache

When a NERV emit application sends events, the events first go through Ehcache, where they can be persisted temporarily. After that, the events are sent to an internal channel, where they are processed and sent asynchronously to the Event Bus. To ensure that no events are lost before reaching the Event Bus, NERV enables you to use Ehcache by setting a level of guaranteed delivery for your events. After an event is successfully delivered to the event bus, it is deleted from the cache.

Note: The Guaranteed delivery feature in NERV ensures that all events are successfully sent to the event bus. However, in order to guarantee the successful delivery of all events to the subscribers, it is recommended that you

use durable subscriptions. Otherwise, events might be lost on the consumer side.

By default, NERV reads the Ehcache persistence configuration settings from the *Software AG_directory*\common\conf\nerv\cache directory. It contains preconfigured MAXIMUM_STRONG.xml and MAXIMUM_EVENTUAL.xml files with settings used by Ehcache to guarantee the delivery of events. For more information about the available settings and their values, see Ehcache product documentation for 2.8 at http://ehcache.org/documentation.

Note: Persisting events with Ehcache in a disk store has a substantial impact on the maximum achievable event throughput.

You can modify the Ehcache configuration settings location that NERV uses by modifying the value of the **Cache Configuration Location** configuration property for your NERV installation node using the Command Central user interface. For more information about the **Cache Configuration Location** property, see " NERV Configuration Properties" on page 61.

Setting up NERV Guaranteed Delivery Level with Ehcache

To persist events with Ehcache, you must set a level of guaranteed delivery in the NERV configuration file.

To set up a guaranteed delivery level with Ehcache

1. In Command Central, navigate to **Environments > Instances > ALL > <*profile_name*> >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.
2. Modify the value of the **Guaranteed Delivery Level** property.

Set to...	To...
`NONE`	Not persist any events.
`MAXIMUM_STRONG`	(Default) Persist events synchronously whenever a cache update occurs.
`MAXIMUM_EVENTUAL`	Persist events asynchronously whenever a cache update occurs.

3. Save your changes.
4. Restart the JVM instance where NERV is running for the changes to take effect.

When you set up the guaranteed delivery level to `MAXIMUM_STRONG` or `MAXIMUM_EVENTUAL`, the value you have defined for the **Redelivery Attempts** property is disregarded. Instead, NERV attempts to redeliver events infinitely.

Important: When you set up the guaranteed delivery level to `MAXIMUM_STRONG` or `MAXIMUM_EVENTUAL` and then start NERV, it creates a JMS connection which performs synchronous emits to Universal Messaging. However, in case the JMS connection is created before NERV initializes, you must add an additional JVM property. To do this, in the *Software AG_directory*/profiles/*profile_name* /configuration directory, open the wrapper.conf file, add the following JVM property: `wrapper.java.additional.<new custom integer>=-Dnirvana.syncSendPersistent=true`, then save your changes and restart the Common Platform profile.

Locating your Disk Store Location with Ehcache

Each NERV node running on your system creates a different store location for persisted events. The store location for your events is set within the Ehcache configuration XML files using the *diskStore* property. This property has a *path* attribute with a default value of `${nerv.cache.disk.store.dir}/cachedata/nerv.caches`.

```
<diskStore path="${nerv.cache.disk.store.dir}/cachedata/nerv.caches"/>
```

If the *diskStore* property is not already configured as a JVM system property (*nerv.cache.disk.store.dir*), then at runtime it is set by NERV.

- For products running in the Software AG Common Platform, NERV sets the value to match the value of the `osgi.install.area` property. For example, when NERV runs in the SPM profile, the store location is set to:

  ```
  <diskStore path="C:/SoftwareAG/profiles/SPM/cachedata/nerv.caches"/>
  ```

- For products running outside the Software AG Common Platform, NERV sets the value to `./cachedata/nerv.caches`. For example:

  ```
  <diskStore path="C:/SoftwareAG/cachedata/nerv.caches"/>
  ```

For more information about the *diskStore* property, see Ehcache product documentation for 2.8 at http://ehcache.org/documentation.

Creating Custom NERV Component Bundles

You can create and initialize a custom NERV component bundle that will override the default NERV component defined using the **Default JMS Provider** configuration property in Command Central.

For more information about switching between the default and the custom NERV component definitions, see "Switching Between Default and Custom NERV Logic" on page 33.

To create a custom NERV component bundle

1. In Designer, create a plug-in project.

2. Remove all automatically generated files, except for the MANIFEST.MF file.
3. Add a *MyBlueprint* .xml file and a *MyNERVComponent* .xml file in the OSGI-INF/blueprint directory.

   The resulting structure is as follows:

```
<project_root>\META-INF\MANIFEST.MF
<project_root>\OSGI-INF\blueprint\
   MyBlueprint.xml
<project_root>\OSGI-INF\blueprint\
   MyNERVComponent.xml
```

   Note: You can also use the NERV Default Component Example wizard in Designer to create the project structure automatically.

4. Edit the *MyBlueprint* .xml file in the OSGI-INF/blueprint directory and reference the nervDefaultJMS component.

   The result should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">
    <service id="nervDefaultJMSService"
      interface="org.apache.camel.Component"
      ref="nervDefaultJMS" depends-on="nervDefaultJMS">
      <service-properties>
        <entry key="componentId" value="nervDefaultJMS"/>
      </service-properties>
    </service>
</blueprint>
```

5. Edit the *MyNERVComponent* .xml file in the OSGI-INF/blueprint directory containing the beans necessary for creating your Camel component.

   The result should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">
 <bean id="nervDefaultJMS"
  class="org.apache.camel.component.jms.JmsComponent">
   <property name="connectionFactory"
             ref="defaultCachedConnectionFactory" />
   <property name="destinationResolver"
             ref="defaultDestinationResolver" />
 </bean>
 <bean id="defaultCachedConnectionFactory"
   class="org.springframework.jms.connection.CachingConnectionFactory">
   <property name="targetConnectionFactory"
             ref="defaultConnectionFactory" />
   <property name="sessionCacheSize" value="2000" />
   <property name="cacheProducers" value="false" />
 </bean>
 <bean id="defaultConnectionFactory"
```

```
  class="org.springframework.jndi.JndiObjectFactoryBean"
   depends-on="defaultDestinationResolver">
   <property name="jndiTemplate" ref="jndiTemplate" />
   <property name="jndiName" ref="eventFactory" />
 </bean>
 <bean id="eventFactory" class="java.lang.String">
   <constructor-arg value="EventFactory" />
 </bean>
 <bean id="defaultDestinationResolver"
  class="com.softwareag.eda.jndi.NervResolver"
   depends-on="jndiTemplate" init-method="init">
   <property name="jndiTemplate" ref="jndiTemplate" />
   <property name="classLoader">
    <null />
   </property>
 </bean>
 <bean id="jndiTemplate"
  class="org.springframework.jndi.JndiTemplate">
   <property name="environment">
    <map>
     <entry key="java.naming.factory.initial"
      value="com.pcbsys.nirvana.nSpace.NirvanaContextFactory" />
     <entry key="java.naming.provider.url" value=
      "nsp://localhost:9000" />
     <entry key="com.webmethods.jms.naming.clientgroup"
            value="admin" />
     <entry key="connectionFactory" value-ref="eventFactory" />
    </map>
   </property>
 </bean>
</beans>
```

6. Edit the MANIFEST.MF file in the META-INF directory. You must also add some import-package clauses to get the bundle to work properly in the Common Platform.

   Below you can find an example of the edited MANIFEST.MF file:

```
Manifest-Version: 1.0
Bundle-Name: component.nervDefaultJMS
Bundle-Vendor: Software AG
Bundle-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Description: Software AG NERV Components Default Bundle
Import-Package: com.pcbsys.nirvana.nSpace;version=0,
 com.softwareag.eda.jndi;version=0,
 com.webmethods.jms.impl;version=0,
 com.webmethods.jms.naming;version=0,
 org.apache.camel;version=0,
 org.apache.camel.component.jms;version=0,
 org.springframework.jms.connection;version=0,
 org.springframework.jms.support.destination;version=0,
 org.springframework.jndi;version=0
 Bundle-DocURL: http://www.softwareag.com
 Bundle-SymbolicName: component.nervDefaultJMS
 Implementation-Version: 1.0
```

7. Build your project using webMethods Asset Build Environment, and deploy it using webMethods Deployer.

   For more information, see the PDF publication *webMethods Deployer User's Guide*.

Using the Default NERV Emit Logic

As installed, NERV provides default emit logic which you can use to emit events of a specified event type. The `com.softwareag.eda.nerv.core` bundle contains the *EventEmitter* service interface that you can reference when you create your application.

The default NERV emit logic uses the transport layer specified by the **Default JMS Provider** configuration property in Command Central, or by the nervDefaultJMS component defined in a deployed NERV component bundle. You can modify the default transport layer definition, as described in "Modifying the Transport Layer for NERV " on page 26, or by deploying a custom NERV component bundle which contains a nervDefaultJMS component definition.

The default NERV emit logic uses the error handling mechanism defined via specific configuration properties.

The name of the JMS topic where NERV publishes the events by default is derived from the Event Type of each event. For example, events of type

```
{http://namespaces.softwareag.com/EDA/WebM/Sample/CableboxMonitoring}
CableboxHealth
```

are sent to a JMS topic named

```
Event::WebM::Sample::CableboxMonitoring::CableboxHealth
```

You can also define your customized event routing logic by creating a NERV emit bundle for each event type you emit. This will override the default NERV emit logic. For more information about creating custom NERV emit bundles, see "Creating Custom NERV Emit Bundles" on page 33.

Switching Between Default and Custom NERV Logic

You can switch from the default NERV component definition and emit logic to your custom NERV component and emit bundles by deploying and activating a user-defined bundle within the Common Platform.

To switch back to the default NERV logic, you must stop the custom bundle within the Common Platform. This automatically activates the default logic again.

Creating Custom NERV Emit Bundles

You can create and initialize a custom NERV emit bundle that will override the default emit logic described in Using the Default NERV Emit Logic.

To create a custom NERV emit bundle

1. In Designer, create a plug-in project.

2. Remove all automatically generated files, except for the MANIFEST.MF file.
3. Add a *MyBlueprint* .xml and a *MyNERVEmit* .xml file in the OSGI-INF/blueprint directory.

   The resulting structure is as follows:

```
<project_root>\META-INF\MANIFEST.MF
<project_root>\OSGI-INF\blueprint\
   MyBlueprint.xml
<project_root>\OSGI-INF\blueprint\
   MyNERVEmit.xml
```

   Note: You can also use the NERV Emit Example wizard in Designer to create the project structure automatically.

4. Edit the *MyBlueprint* .xml file in the OSGI-INF/blueprint directory to reference the nervDefaultJMS component and the default error handler preconfigured and exposed by NERV.

   The result should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">
    <reference id="nervDefaultJMS" interface="org.apache.camel.Component"
     filter="(componentId=nervDefaultJMS)"/>
    <reference id="nervDefaultErrorHandler"
     interface="org.apache.camel.builder.ErrorHandlerBuilder"
     filter="(errorHandlerId=nervDefaultErrorHandler)"/>
</blueprint>
```

5. Edit the *MyNERVEmit* .xml file in the OSGI-INF/blueprint directory containing your custom Camel context and a route for your event type.

   By convention, NERV derives the name of the in-memory channel used as a starting point for event routing from the fully qualified event type name by substituting “{” and “}” with “_”. For example, for events of type:

```
{http://namespaces.softwareag.com/EDA/WebM/Sample
 /CableboxMonitoring}CableboxHealth
```

   the endpoint URI of the in-memory channel is:

```
vm://_http://namespaces.softwareag.com/EDA/WebM/Sample/
CableboxMonitoring_CableboxHealth?size=1000&amp;blockWhenFull=true
```

   The in-memory channel is created as an internal queue with a default size of 1000 maximum number of messages, and the calling thread is set to block and wait if the channel is full.

   Below you can find an example containing the default routing configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://camel.apache.org/schema/spring
```

```
  http://camel.apache.org/schema/spring/camel-spring.xsd">
 <camelContext id="emitContext" autoStartup="true"
  xmlns="http://camel.apache.org/schema/spring">
  <route errorHandlerRef="nervDefaultErrorHandler">
   <from uri="vm://_http://namespaces.softwareag.com/EDA/WebM/Sample/
    CableboxMonitoring_CableboxHealth?size=1000&amp;blockWhenFull=true" />
   <!-- Please edit the following URI in order to change the default
         routing logic -->
    <to uri="nervDefaultJMS:topic:Event::WebM::Sample::CableboxMonitoring
       ::CableboxHealth" />
  </route>
 </camelContext>
</beans>
```

6. Edit the MANIFEST.MF file in the META-INF directory. You must also add some import-package clauses to get the bundle to work properly in the Common Platform.

   Below you can find an example of the edited MANIFEST.MF file:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: emit.NERVEmitExample
Bundle-SymbolicName: emit.NERVEmitExample
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: com.softwareag.eda.nerv,
 com.softwareag.eda.store.api,
 org.apache.camel,
 org.apache.camel.builder,
org.osgi.framework
```

7. Build your project using webMethods Asset Build Environment, and deploy it using webMethods Deployer.

   For more information, see the PDF publication *webMethods Deployer User's Guide*.

> **Note:** When you deploy your custom NERV emit bundle, it overrides the default NERV emit logic. For more information about how to switch back to the default emit logic, see "Switching Between Default and Custom NERV Logic" on page 33.

Creating Custom NERV Consume Bundles

To create a custom NERV consume bundle

1. In Designer, create a plug-in project.
2. Remove all automatically generated files, except for the MANIFEST.MF file.
3. Add a *MyBlueprint* .xml and a *MyNERVConsume* .xml file in the OSGI-INF/blueprint directory.
4. Create your application-specific Camel processor implementation. In this example, it is the com.softwareag.eda.consumer.ConsoleConsumer class.

   The resulting structure is as follows:

```
C:\<project_name>\cableboxhealth_transformer_bean\src\com\softwareag\eda\
```

```
    consumer\ConsoleConsumer.java
C:\<project_name>\cableboxhealth_transformer_bean\
    META-INF\MANIFEST.MF
C:\<project_name>\cableboxhealth_transformer_bean\
    OSGI-INF\blueprint\MyBlueprint.xml
C:\<project_name>\cableboxhealth_transformer_bean\
    OSGI-INF\blueprint\
    MyNERVConsume.xml
```

5. Create an implementation of the org.apache.camel.Processor interface, which consumes the received events.

   In the example below, the implementation prints out the content of the event body into the System Output stream.

```
package com.softwareag.eda.consume;
import org.apache.camel.*;
public class ConsoleConsumer implements Processor {
  @Override
  public void process(Exchange exchange) throws Exception {
    Message message = exchange.getIn();
    String body = message.getBody(String.class);
    System.out.println(body);
  }
}
```

6. Edit the *MyBlueprint* .xml file in the OSGI-INF/blueprint directory and reference the nervDefaultJMS component.

   The result should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
      http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">
  <reference id="nervDefaultJMS" interface="org.apache.camel.Component"
      filter="(componentId=nervDefaultJMS)"/>
</blueprint>
```

7. Edit the *MyNERVConsume* .xml file in the OSGI-INF/blueprint directory and reference the nervDefaultJMS component. You must declare your processor class as a bean and include it in a Camel route.

   The result should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd">
  <camelContext id="consumeContext" autoStartup="true"
     xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="nervDefaultJMS:topic:Event::WebM::
        Sample::CableboxMonitoring::CableboxHealth" />
      <to uri="consoleConsumer" />
   </route>
  </camelContext>
  <bean id="consoleConsumer"
    class="com.softwareag.eda.consumer.ConsoleConsumer" />
```

```
</beans>
```

8. Edit the MANIFEST.MF file in the META-INF directory. You must also add some import-package clauses to get the bundle to work properly in the Common Platform.

   Below you can find an example of the edited MANIFEST.MF file:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: consume.cableboxhealth
Bundle-SymbolicName: consume.cableboxhealth
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: com.softwareag.eda.nerv,
 com.softwareag.eda.store.api,
 org.apache.camel,
 org.osgi.framework
```

9. Build your project using webMethods Asset Build Environment, and deploy it using webMethods Deployer.

   For more information, see the PDF publication *webMethods Deployer User's Guide*.

Creating Custom NERV Java Archive Bundles

You can create your custom applications which use NERV's capabilities to send and consume events. These applications can contain custom component configuration definitions, as well as emit or consume configurations. They must be built as Java Archive projects using Designer, and then deployed and activated within the Common Platform.

Important: You must ensure that the contents of your Java Archive projects are correctly defined and built. The procedure below only focuses on how to create valid NERV Java Archive bundles.

To create a NERV Java Archive bundle

1. In Designer, create a valid plug-in Java project.
2. Build your plug-in Java project and make sure that the resulting .jar file is located directly under the project directory.
3. Optionally, add other dependent .jar files under your Java project directory.

   The resulting structure is as follows:

```
<project_root>\<plugin_jar_file>
<project_root>\<additional_plugin_jar_file_1>
...
<project_root>\<additional_plugin_jar_file_n>
```

4. Build your project using webMethods Asset Build Environment, and deploy it using webMethods Deployer.

   For more information, see the PDF publication *webMethods Deployer User's Guide*.

Configuring Event Redelivery and Routing

You can use Java system properties to customize how NERV handles the redelivery of events and how NERV handles the routing of events when the queue is full. You add Java system properties to the *Software AG_directory*\profiles\<profile_name>\configuration\custom_wrapper.conf file in the following format:

```
wrapper.java.additional.n=-Dproperty_name=value
```

where *n* is a unique sequence number.

Use the following Java system properties to customize the redelivery and the routing of events:

- **com.softwareag.eda.nerv.in.memory.failOnRedelivery** - Boolean. Defines whether NERV throws an exception or adds an event to the local queue when the event is sent after a preceding event failed to reach the destination service. Valid values are:
  - `true` - The sender receives the exception for the last redelivery attempt.
  - `false` - Default. The event that has not reached the destination service is added to the local event queue.
- **com.softwareag.eda.nerv.in.memory.blockWhenFull** - Boolean. Defines whether NERV blocks the sending of new events when the local event queue size is equal to the configured queue capacity. Valid values are:
  - `true` - Default. The sender thread is blocked until NERV delivers some of the queued events to the destination services.
  - `false` - The sender receives an exception.

5 Using NERV Outside the Software AG Common Platform

By default, NERV is installed as part of the Software AG Common Platform. However, standalone applications can also use the NERV functionality outside the Common Platform.

To use the NERV functionality with applications that do not run on the Common Platform, you must:

- In the classpath of your applications, include the nerv-classpath-basic.jar file available in the *Software AG_directory*/common/lib directory.
- Provide a folder where NERV configuration bundles are previously deployed.

You can access the complete NERV functionality by using the *NERVSingleton* class. For more information about these methods, see the NERV Java docs.

If your application uses the default *nervDefaultJMS* component, you can benefit from the default NERV emit logic, as described in "Using the Default NERV Emit Logic" on page 33. If you want to define custom NERV emit logic that will override the default logic, see "Creating Custom NERV Emit Bundles" on page 33.

To ensure the successful delivery of events, NERV provides an error handling mechanism. For more information about configuring NERV error handling options, see "Modifying the Transport Layer Configuration" on page 27.

6 Deploying and Testing EDA Solutions

Deploying EDA Assets

Deployment is the process of moving EDA assets from the design environment into the run-time or production environment.

EDA assets can be deployed to one or more target runtimes using the webMethods Deployer's repository-based deployment. To use this deployment method, you must have the Asset Build Environment (ABE) installed.

The EDA assets you create prior to deployment must have a specific structure in order to be deployable using webMethods Deployer.

- **Event Types Deployment composites**

  Event Types Deployment composites are valid Event Types projects - a parent project directory with an Event Types subdirectory containing event type schemata. The event type schemata are considered individual assets and are packed by the Asset Build Environment into zip archives. Multiple event type schemata can be packed in a single zip file.

  Note: Event type schemata with namespaces that do not start with the `http://namespaces.softwareag.com/EDA` string are deployed to the WebM/External directory of the Event Type Store.

- **NERV Deployment composites**

  NERV Deployment composites are directories containing one or more NERV bundles (.jar files). The Asset Build Environment packs each NERV bundle into a single .zip archive.

  Three types of NERV configuration bundles exist:

  - **NERV Component bundles**

    The NERV component configuration bundles are valid OSGi bundles that declare a single Camel component. The bundle directory must have the following structure:

    - A *MyNERVComponent* .xml file and a *MyBlueprint* .xml file under the OSGI-INF\blueprint directory.
    - A MANIFEST.MF file under the META-INF directory.
    - A component declared as an OSGi service in a blueprint configuration file using a specific syntax, for example:

```
<service id="nervDefaultJMSService"
   interface="org.apache.camel.Component"
   ref="nervDefaultJMS" depends-on="nervDefaultJMS"> <service-
properties> <entry key="componentId" value="nervDefaultJMS"/>
</service-properties> </service>
```

  - **NERV Emit bundles**

The NERV emit bundles are valid OSGi bundles that declare Camel endpoints. The bundle directory must have the following structure:

- An *MyNERVEmit* .xml file and a *MyBlueprint* .xml file under the OSGI-INF \blueprint directory.
- A MANIFEST.MF file under the META-INF directory.

When the Asset Build Environment creates ACDL files, the NERV emit route bundles are parsed and all ComponentIDs are extracted and declared as dependencies. The dependencies are declared in the blueprint.xml file as follows:

```
<reference id="nervDefaultJMS"
   interface="org.apache.camel.Component"
   filter="(componentId=nervDefaultJMS)"/>
```

- **NERV Consume bundles**

  The NERV consume bundles are valid OSGi bundles that declare Camel endpoints. The bundle directory must have the following structure:

  - A *MyNERVConsume* .xml file and a *MyBlueprint* .xml file under the OSGI-INF \blueprint directory.
  - A MANIFEST.MF file under the META-INF directory.

  When the Asset Build Environment creates ACDL files, the NERV consume bundles are parsed and all ComponentIDs are extracted and declared as dependencies. The dependencies are declared in the *MyBlueprint* .xml file as follows:

```
<reference id="nervDefaultJMS"
   interface="org.apache.camel.Component"
   filter="(componentId=nervDefaultJMS)"/>
```

- **NERV Java Archive bundles**

  The NERV Java archive bundles are valid OSGi bundles that contain any executable Java code as one or several .jar files. The bundle directory must have the following structure:

  - One or several .jar files must be present under the project directory.

  When the Asset Build Environment creates ACDL files, only the .jar files are taken into consideration.

When you enable the creation of EDA composites and run the ABE build script, the script searches the specified source directories and creates a composite for each project directory that contains EDA assets.

For more information about installing the Asset Build Environment feature, see *Installing Software AG Products*. For more information about building composites for repository-based deployment, see *webMethods Deployer User's Guide*.

Specifics of Deploying NERV Bundles to the Software AG Common Platform

When you use webMethods Deployer to deploy NERV bundles to the Software AG Common Platform, the bundles are deployed to the *Software AG_directory*/common/nerv/bundles directory, from which they are loaded by all profiles in your Software AG installation.

If you want to deploy and activate NERV bundles only to a specific Common Platform profile, you must modify the default deployment location and make the Common Platform profile aware of the new location where your NERV bundles will be deployed.

To deploy and activate NERV bundles only for a specific Software AG Common Platformprofile

1. In Command Central, navigate to **Environments > Instances > ALL > SPM >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.
2. Note the initial value for the **Configuration Bundles Location** property. You will need this value for a later step in the procedure.
3. To specify a user-defined location for deployment of NERV bundles, modify the value for the **Configuration Bundles Location** property to point to a user-defined location. For example:

   ```
   C:/SoftwareAG/MyNERVLocation/bundles
   ```
4. Save the modifications and restart the SPM profile for your changes to take effect.
5. Deploy your NERV bundles using webMethods Deployer.

   This will deploy the NERV bundles to the user-defined location.
6. In Command Central, navigate to **Environments > Instances > ALL > SPM >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.
7. Set the value for the **Configuration Bundles Location** property to the initial value, as noted at the beginning of the procedure, then save the changes and restart the SPM profile.
8. To make a specific Common Platform profile aware of the user-defined locations where bundles reside, in Command Central, navigate to **Environments > Instances > ALL > <*profile_name*> >NERV> Configuration** tab, click the COMMON-SYSPROPS-com.softwareag.eda.nerv.properties file, and then click **Edit**.

   For example, for the IS_default profile, the location would be **Environments > Instances > ALL > IS_default >NERV> Configuration** tab.
9. Modify the value for the **Configuration Bundles Location** property to point to the default deployment location, as well as the user-defined deployment location. For example:

   ```
   C\:/SoftwareAG/common/nerv/bundles,
   C\:/SoftwareAG/MyNERVLocation/bundles
   ```

Note: The delimiter must be a comma.

10. Save the modifications and restart the specific Common Platform profile for the changes to take effect.

    Now the respective Common Platform profile loads all bundles from the default location, as well as the NERV bundles from the user-defined location.

Example of a Deployment Project Structure

You can use webMethods Asset Build Environment to build deployable composites from EDA event types and configuration projects. Here is an example of an EDA source repository directory and the deployable assets which are produced by the Asset Build Environment build script. In the example below the *build.source.dir* property is set to `/source` as a prerequisite.

- For Event Types with the following source repository structure:

```
/source/MyNewEvents/Event Types/MyCompany/Account.xsd
/source/MyNewEvents/Event Types/MyCompany/Receipt.xsd
```

  the Asset Build Environment build script creates the *MyNewEvents.zip* deployable composite, which contains the two Account and Receipt event types.

- For NERV configuration components with the following source repository structure:

```
/source/NERVComposite/component.myComponent/
   META-INF/MANIFEST.MF
/source/NERVComposite/component.myComponent/
   OSGI-INF/blueprint/blueprint.xml
/source/NERVComposite/component.myComponent/
   OSGI-INF/blueprint/component.xml
/source/NERVComposite/component.nervDefaultJMS/
   META-INF/MANIFEST.MF
/source/NERVComposite/component.nervDefaultJMS/
   OSGI-INF/blueprint/blueprint.xml
/source/NERVComposite/component.nervDefaultJMS/
   OSGI-INF/blueprint/component.xml
/source/NERVComposite/emit.BoothDemo/META-INF/MANIFEST.MF
/source/NERVComposite/emit.BoothDemo/OSGI-INF/blueprint/blueprint.xml
/source/NERVComposite/emit.BoothDemo/OSGI-INF/blueprint/emit.xml
```

  the Asset Build Environment build script creates the *NERVComposite.zip* deployable composite, which contains the two configuration bundles (*component.myComponent.jar* and *component.nervDefaultJMS.jar*), and the emit configuration bundle *emit.BoothDemo.jar*.

- For NERV Java archive bundles with the following source repository structure:

```
/source/MyNewJavaArchiveBundleProject/
    src/mynewjavaarchivebundleproject/Activator.java
/source/MyNewJavaArchiveBundleProject/META-INF/MANIFEST.MF
/source/MyNewJavaArchiveBundleProject/build.properties
/source/MyNewJavaArchiveBundleProject/build.xml
/source/MyNewJavaArchiveBundleProject/additional.jar
```

the Asset Build Environment build script creates the *MyNewJavaArchiveBundleProject.zip* deployable composite, which contains the two Java archives (*MyNewJavaArchiveBundleProject.jar* and *additional.jar*.

7 Visualizing Events

Visualizing Events with Software AGDashboarding Products

You can use one of Software AG's dashboarding products to visualize the event streams flowing through your system.

- **Software AG MashZone**

  Software AG MashZone delivers real-time operational insight dashboards direct from live information sources. You can combine data from any original source - data warehouses, news feeds, social media, Business Intelligence (BI) systems, streaming big data and even Microsoft® Excel® spreadsheets - to create real-time mashups for right-time decision-making.

  MashZone combines data from any source for data visualizations in real-time. Accessing the original data lets business users respond to changing conditions as they happen. One of MashZone's strengths is the ability to accept unstructured data, like a social media feed, and keep it updated in real-time.

- **Software AG MashZone**

  Software AG MashZone is a component of the IBO portfolio of products. MashZone provides a drag-and-drop interface that you use to create analytical dashboards for combining product-generated and third party data sources in real time. MashZone dashboards allow you to implement real-time monitoring and analysis of process performance.

  The data feeds are combined in one simple code-free step using graphical visualization so that you can create a dashboard that shows you all relevant information for your decision-making process.

  A MashZone application can combine product-generated data (from Software AG Process Performance Manager, spreadsheet data, ERP systems, CRM systems, or data warehouse systems, for example) with web data (such as Google Maps, statistical databases, and financial tickers). To do this, it extracts data from various data sources and converts the data into feeds. Software AG MashZone offers a host of visualization components for this purpose, such as bar graphs, pie charts, pyramid charts, funnel charts, and maps.

Visualizing Event Streams on the Eclipse Console

The Event Bus Console utility enables you to subscribe to topics on the event bus and view the traces for each event stream on the Eclipse console. The trace displays the traffic on the event bus, such as what kind of events occur and how often they occur.

The Event Bus Console runs as a console view in the **Events Development** perspective of Software AG Designer. For more information on using the Event Bus Console, see *webMethods Event Processing Help*.

A Troubleshooting NERV

NERV is Software AG's solution for routing and transformation of events within an event-enabled environment. This chapter provides information about troubleshooting your applications which use NERV's capabilities to send and consume events.

Starting the OSGi Console for a Single OSGi Profile

NERV is a framework which enables you to use events within the Software AG Common Platform. Below you will find explanations about how to start an OSGi console for a single OSGi profile in your installation.

To start the OSGi console for a single OSGi profile

1. In your Software AG installation, navigate to the configuration directory of the profile to which you want to connect, for example *Software AG_directory*/profiles/SPM/configuration.
2. Using a text editor, open for editing the *config.ini* file.
3. Enable the OSGi console.
   - To connect remotely to the OSGi console using telnet, add the following line to the *config.ini* file:

     ```
     osgi_console=<user-defined_portnumber>
     ```
   - To connect directly to the OSGi console, leave the `osgi.console` property empty:

     ```
     osgi.console=
     ```
4. Restart the profile.
   - If you chose to connect to the console remotely using telnet, use the restart or shutdown/startup scripts.
   - If you chose to connect directly to the console, use the shutdown script to shut down the profile, and then use the console startup script to restart it.
5. Connect to the OSGi console.
   - If you chose to use a remote connection to the OSGi console, open a telnet session and type the commands.
   - If you chose to connect directly to the OSGi console, type the commands in the runtime console.

When you are connected to the OSGi console, you can use the following commands:

- **ss** - to list all available bundles. Use a filter to narrow down the list, for example:

  ```
  ss <user_filter>
  ```

  The command returns a list of bundles with their id, bundle name, and current state (ACTIVE, INSTALLED, or RESOLVED).

- **bundle <bundle_id>** - to obtain information about a specific bundle. The command returns a list of services used and exposed by the bundle, imported and exported packages, and other information.
- **diag <bundle_id>** - to diagnose a specific bundle in the Common Platform. The command returns a list of any unresolved dependencies for the bundle.
- **help** - to obtain the list of all available console commands.

OSGi Commands Provided by NERV

NERV runs within the Software AG Common Platform. Below you can find a list of NERV-related OSGi commands.

OSGi Command	Description
`nerv`	Displays all NERV commands.
`nervDiag`	Displays diagnostics information about NERV, such as the process D of the current Common Platform instance, the location of the Event Type Store and the NERV assets directory. This command also lists all OSGi services exposed by NERV.
`nervDefaultJms`	Displays the URL of the default JMS provider, as well as information whether it was configured using the properties file or by deploying a configuration bundle.
`nervDefinedComponents`	Displays information about bundles that contain a component.xml file - the location of the bundles on the file system, and the beans that are defined in the component.xml files. **Note:** This command only lists the component definitions but does not provide information about whether the components are active or not. For information about active NERV components, use the `nervActiveComponents` command.
`nervDefinedEmit`	Displays information about bundles that contain an emit.xml file - the location of the bundles on the file system and the routes defined in the emit.xml files. **Note:** This command only lists the route definitions but does not provide information about whether the routes

OSGi Command	Description
	are active or not. For information about active NERV routes, use the `nervRouteInfo` command.
`nervDefinedConsume`	Displays information about bundles that contain a consume.xml file - the location of the bundles on the file system and the beans defined in the consume.xml files. **Note:** This command only lists the route definitions but does not provide information about whether the routes are active or not. For information about active NERV routes, use the `nervRouteInfo` command.
`nervActiveComponents`	Displays a list of the currently active components with their names, types, and information about the bundle that declared them.
`nervRouteInfo`	Displays information about the currently active routes - the route id, the context where they are defined, the current state of the route, and information about the number of exchanges (events) that passed through the route. The command returns information about the full configuration of the route - the start endpoint, the intermediate processors, and the endpoint.
`nervContextInfo`	Displays diagnostics information about the currently active Camel contexts in the system.

NERV Troubleshooting Information

Troubleshooting NERV Component Configuration Bundles

Problem: I have deployed a NERV Component bundle which overrides the default NERV component, but my messages are still not sent to the specified JMS provider

Solution:

1. Connect to the OSGi console where you expect the configuration bundle to be active.

   For more information about how to connect to the OSGi console for a single profile, see "Starting the OSGi Console for a Single OSGi Profile" on page 50.

2. Use the `nervDefaultJms` command to check which is the currently active JMS provider.

If the command output indicates that the default JMS provider is the one specified in your user-defined configuration bundle, then events should be sent to that JMS provider by default, and you should check whether events are actually being sent and if the receiving side is configured correctly.

If the command output indicates that the default JMS provider is different than the one that you specified in your user-defined configuration bundle, then either the component did not get activated, or was not configured correctly.

3. Locate your configuration bundle in the Common Platform profile and make a note of its ID.
   - Use the `nervDefinedComponents` command to list bundles that define a component in a component.xml file following the convention for NERV configuration bundles.
   - Use the `ss` command to list bundles that declare components not following the convention for NERV configuration bundles (for example, programmatically or using declarative services).
4. Depending on the state of your configuration bundle, do the following:
   - If your bundle is not in an active state, use the `diag <bundle_id>` command to diagnose why it was not started, resolve any dependency issues, then use the `start <bundle_id>` command to start your bundle.
   - If your bundle is in an active state, use the `bundle <bundle_id>` command to list all services that the bundle uses and exposes.

     To successfully expose a component for usage by NERV, your bundle must expose at least one service of type *org.apache.camel.Component* with a *componentId=nervDefaultJMS* property (properties for the services are listed right after the service type).

     If the `bundle <bundle_id>` command output lists an *org.apache.camel.Component* service type with a correct *componentId=nervDefaultJMS* property, then a race condition might exist inside the OSGi registry which prevented NERV from picking up the service registration. Use the `stop <bundle_id>` and `start <bundle_id>` commands to restart your bundle. Alternatively, you can also restart the whole Common Platform instance. If this resolves the problem, you should decrease the bundle start level, so it starts later than NERV.

     If the `bundle <bundle_id>` command output does not list a component service, diagnose why the service was not started.

     i. Check the content of your log file, located by default in the log directory under *Software AG_directory*/profiles/<profile_name>.

     ii. In the OSGi console, restart your bundle using the `stop <bundle_id>` and `start <bundle_id>` commands, then check the content of the log file again. It should contain information about the start-up of your bundle and some indications why the bundle was not instantiated and exposed correctly.

     Some of the most common reasons are:

Common reasons	Possible solutions
An XML parsing error or exception indicates that the component configuration file is not grammatically correct.	The parser usually prints out the number of the line which causes the error or exception. Locate it in the component configuration file and fix it.
A ClassNotFoundException or a NoClassDefFoundError indicate that the bundle imports are not correct, and that the Common Platform cannot locate a certain class while instantiating your bundle.	Make sure that the Import-Package statement of your MANIFEST file includes the *org.apache.camel* and *org.apache.camel.component.jms* classes from the Apache Camel package, the *org.springframework.jndi* and *org.springframework.jms.connection* Spring classes, as well as the specific JMS provider classes.
A Blueprint-related or a Spring-related exception or warning message indicates that the framework instantiating and exposing the component service encountered an exception.	Examine the stack trace and try to fix the issue. Check whether another service with the same componentId is not already declared in the same Common Platform instance.

Problem: I have deployed a NERV Component bundle, which declares a component, but when I try to use it, it doesn't work

Solution:

1. Connect to the OSGi console where you expect the configuration bundle to be active.

   For more information about how to connect to the OSGi console for a single profile, see "Starting the OSGi Console for a Single OSGi Profile" on page 50.

2. Use the `nervActiveComponents` command to list the currently active component bundles on the system.

   If the command output lists your component bundle, then it is currently active and available for usage. Double check the configuration for using the component to make sure it is set correctly.

   If the command output does not list your component bundle, then it is not currently active, it was not properly instantiated, or it has not been defined correctly.

3. Locate your configuration bundle in the Common Platform and mark its ID.

   - Use the `nervDefinedComponents` command to list bundles that define a component in a component.xml file following the convention for NERV configuration bundles.

- Use the `ss` command to list bundles that declare components not following the convention for NERV configuration bundles (for example, programmatically or using declarative services).

4. Depending on the state of your configuration bundle, do the following:
   - If your bundle is not in an active state, use the `diag <bundle_id>` command to diagnose why it was not started, resolve any dependency issues, then use the `start <bundle_id>` command to start your bundle.
   - If your bundle is in an active state, use the `bundle <bundle_id>` command to list all services that the bundle uses and exposes.

     To successfully expose a component for usage by NERV, your bundle must expose at least one service of type *org.apache.camel.Component*.

     If the `bundle <bundle_id>` command output does not list a component service, diagnose why the service was not started.

     i. Check the content of your log file, located by default in the log directory under *Software AG_directory*/profiles/<profile_name>.

     ii. In the OSGi console, restart your bundle using the `stop <bundle_id>` and `start <bundle_id>` commands, then check the content of the log file again. It should contain information about the start-up of your bundle and some indications why the bundle was not instantiated and exposed correctly.

     Some of the most common reasons are:

Common reasons	Possible solutions
An XML parsing error or exception indicates that the component configuration file is not grammatically correct.	The parser usually prints out the number of the line which causes the error or exception. Locate it in the component configuration file and fix it.
A ClassNotFoundException or a NoClassDefFoundError indicate that the bundle imports are not correct, and that the Common Platform cannot locate a certain class while instantiating your bundle.	Make sure that the Import-Package statement of your MANIFEST file includes the *org.apache.camel* and *org.apache.camel.component.jms* classes from the Apache Camel package, the *org.springframework.jndi* and *org.springframework.jms.connection* Spring classes, as well as the specific JMS provider classes.
A Blueprint-related or a Spring-related exception or warning message indicates that the framework instantiating and	Examine the stack trace and try to fix the issue. Check whether another service with the same componentId

Common reasons	Possible solutions
exposing the component service encountered an exception.	is not already declared in the same Common Platform instance.

Troubleshooting NERV Emit Configuration Bundles

Problem: I have a bundle, which overrides the default NERV emit configuration, but it does not work

Solution:

1. Connect to the OSGi console where you expect your emit configuration bundle to be active.

   For more information about how to connect to the OSGi console for a single profile, see "Starting the OSGi Console for a Single OSGi Profile" on page 50.

2. Locate your emit configuration bundle in the Common Platform and mark its ID.
   - Use the `nervDefinedEmit` command to list bundles that declare an emit configuration using an emit.xml configuration file.
   - Use the `ss` command to list bundles where the emit logic is defined not following the convention for NERV emit configuration bundles (for example, programmatically or using declarative services).
3. Depending on the state of your bundle, do the following:
   - If your bundle is not in an active state, use the `diag <bundle_id>` command to diagnose why it was not started, resolve any dependency issues, then use the `start <bundle_id>` command to start your bundle.
   - If your bundle is in an active state, use the `bundle <bundle_id>` command to list all services that the bundle uses and exposes.

     To successfully emit events with NERV, your bundle must:
     - Use services from other bundles - either an instance of the *org.apache.camel.Component* service, or some of the other NERV services.
     - Expose some services - either an Apache Camel-related class, or an instance of a container or a class (for example, an instance of the *BlueprintContainer* or the *DelegatedExecutionOsgiBundleApplicationContext* classes).

     Note: If your emit configuration bundle also defines a component, you might not need to use any other services.

     If the `bundle <bundle_id>` command output does not list any exposed services, diagnose why they are not used.

     i. Check the content of your log file, located by default in the log directory under *Software AG_directory*/profiles/<profile_name>.

ii. In the OSGi console, restart your bundle using the `stop <bundle_id>` and `start <bundle_id>` commands, then check the content of the log file again. It should contain information about the start-up of your bundle and some indications why the bundle was not instantiated and exposed correctly.

Some of the most common reasons are:

Common reasons	Possible solutions
An XML parsing error or exception indicates that the component configuration file is not grammatically correct.	The parser usually prints out the number of the line which causes the error or exception. Locate it in the component configuration file and fix it.
A ClassNotFoundException or a NoClassDefFoundError indicate that the bundle imports are not correct, and that the Common Platform cannot locate a certain class while instantiating your bundle.	Make sure that the Import-Package statement of your MANIFEST file includes the classes from the Apache Camel package, the Spring classes, as well as any other classes you might need.
A Blueprint-related or a Spring-related exception or warning message indicates that the framework instantiating and exposing the component service encountered an exception.	Examine the stack trace and try to fix the issue. Check whether another service with the same componentId is not already declared in the same Common Platform instance.

Troubleshooting NERV Consume Configuration Bundles

Problem: I have a bundle, which consumes events, but it does not work

Solution:

1. Connect to the OSGi console where you expect your consume bundle to be active.

   For more information about how to connect to the OSGi console for a single profile, see "Starting the OSGi Console for a Single OSGi Profile" on page 50.

2. Locate your consume bundle in the Common Platform and mark its ID.

   - Use the `nervDefinedConsume` command to list bundles that consume events using an consume.xml configuration file.
   - Use the `ss` command to list bundles where the consume logic is defined not following the convention for NERV consume configuration bundles (for example, programmatically or using declarative services).

3. Depending on the state of your bundle, do the following:

   - If your bundle is not in an active state, use the `diag <bundle_id>` command to diagnose why it was not started, resolve any dependency issues, then use the `start <bundle_id>` command to start your bundle.
   - If your bundle is in an active state, use the `bundle <bundle_id>` command to list all services that the bundle uses and exposes.

     To successfully consume events with NERV, your bundle must use services from other bundles - either an instance of the *org.apache.camel.Component* service, or some of the other NERV services.

     Note: If your consume configuration bundle also defines a component, you might not need to use any other services.

     If the `bundle <bundle_id>` command output does not list any exposed services, diagnose why they are not used.

     i. Check the content of your log file, located by default in the log directory under *Software AG_directory*/profiles/<profile_name>.

     ii. In the OSGi console, restart your bundle using the `stop <bundle_id>` and `start <bundle_id>` commands, then check the content of the log file again. It should contain information about the start-up of your bundle and some indications why the bundle was not instantiated and exposed correctly.

     Some of the most common reasons are:

Common reasons	Possible solutions
An XML parsing error or exception indicates that the component configuration file is not grammatically correct.	The parser usually prints out the number of the line which causes the error or exception. Locate it in the component configuration file and fix it.
A ClassNotFoundException or a NoClassDefFoundError indicate that the bundle imports are not correct, and that the Common Platform cannot locate a certain class while instantiating your bundle.	Make sure that the Import-Package statement of your MANIFEST file includes the classes from the Apache Camel package, the Spring classes, as well as any other classes you might need.
A Blueprint-related or a Spring-related exception or warning message indicates that the framework instantiating and	Examine the stack trace and try to fix the issue. Check whether another service with the same componentId is not already declared in the same Common Platform instance.

Common reasons	Possible solutions
exposing the component service encountered an exception.	

B NERV Configuration Properties

You can configure the default transport layer, the guaranteed delivery for EDA events with Ehcache, as well as the error handling options that NERV uses during runtime.

- When NERV is used by emit applications in the Software AG Common Platform, modify the configuration properties as described in Modifying the Transport Layer Configuration.
- When NERV is used by the Event Bus Console, modify the properties available in the com.softwareag.eda.nerv.properties file, located in the *Software AG_directory*\common\conf directory.

You can modify the values for the following NERV configuration properties:

- **com.softwareag.eda.nerv.dead.letter.channel.uri** - specifies the location where NERV sends any undelivered events after the maximum number of redelivery attempts has been reached. The default value is `@url:sag.install.area/common/nerv/dlc`. When NERV runs in the Common Platform, the `@url:sag.install.area` token is dynamically resolved to point to your Software AG installation directory.
- **com.softwareag.eda.nerv.in.memory.channel.size** - specifies the maximum number of messages contained in the in-memory channel used by NERV. This property is global and applies to all event types. The default value is `1000`.
- **com.softwareag.eda.nerv.maximum.redelivery.attempts** - defines the maximum number of attempts that NERV makes to redeliver events, in case the initial delivery attempt fails. The default value is `100`. Set to `-1` for infinite redelivery attempts.

  Note: When you set up the guaranteed delivery level to `MAXIMUM_STRONG` or `MAXIMUM_EVENTUAL`, the value you have defined for the `com.softwareag.eda.nerv.maximum.redelivery.attempts` property is disregarded. Instead, NERV attempts to redeliver events infinitely to prevent the loss of events, as well as to ensure that their order is kept.

- **com.softwareag.eda.nerv.cache.configuration.location** - specifies the location of the Ehcache configuration files. It is used together with the `com.softwareag.eda.nerv.guaranteed.delivery.level` property.
- **com.softwareag.eda.nerv.default.jms.provider** - defines the default transport layer which NERV uses to connect to the Event Bus. The value of this parameter is populated during installation. The default value is `nsp://<host_name>:9000`.

  Note: This property supports only the use of webMethods Universal Messaging or webMethods Broker as JMS providers. If you want to use another JMS provider, you must deploy a custom NERV component configuration bundle for the respective provider.

- **com.softwareag.eda.nerv.jms.asynch.subscription** - defines the way NERV subscribers consume events. The default value is `false`. If set to `true`, NERV subscribers consume events asynchronously.
- **com.softwareag.eda.nerv.jms.auto.generate.topics** - enables NERV to create topics automatically on the specified JMS provider. The default value is `true`. If set to `false`, users must create topics manually on the JMS provider.

  Note: This property supports only the use of webMethods Universal Messaging or webMethods Broker as JMS providers. When NERV is used in a Universal Messaging cluster environment, this property must be set to `false`.

- **com.softwareag.eda.nerv.guaranteed.delivery.level** - specifies the guaranteed delivery level for storing events. Possible values are `NONE`, `MAXIMUM_STRONG` (default), and `MAXIMUM_EVENTUAL`.
- **com.softwareag.eda.nerv.configbundles.location** - contains the path to the location where NERV expects to find its component configuration bundles. The default value is `@path:sag.install.area/common/nerv/bundles`. When NERV runs in the Common Platform, the `@path:sag.install.area` token is dynamically resolved to point to your Software AG installation directory.
- **com.softwareag.eda.nerv.eventtypestore.location** - contains the path to the Event Type Store location. The value of this parameter is populated during installation. The default value is:`@path:sag.install.area/common/EventTypeStore`. When NERV runs in the Common Platform, the `@path:sag.install.area` token is dynamically resolved at runtime to point to your Software AG installation directory.
- **com.softwareag.eda.nerv.redelivery.delay.ms** - defines the interval (in milliseconds) at which NERV makes the redelivery attempts. The default value is `3000`.
- **com.softwareag.eda.nerv.security.file.location** - contains the path to the NERV security file. This file contains the encrypted secret key used by NERV for encrypting and decrypting passwords specified in the route bundles making connections which require password authentications. The default value is: `@path\:sag.install.area/common/conf/nerv/nerv-security.xml`. When NERV runs in the Common Platform, the `@path:sag.install.area` token is dynamically resolved at runtime to point to your Software AG installation directory.

  Note: When NERV runs outside the Common Platform, the default value is not taken into account.