Platform SDK Developer's Guide

Routing Server

4/16/2025

Routing Server

Many types of interactions enter a modern contact center, and each of them can have many possible destinations. Universal Routing Server (URS) helps them get to the right place at the right time by enabling you to create customized *routing strategies* — sets of instructions that tell URS how to handle interactions. These routing strategies can be as simple or complex as you need them to be. URS uses routing strategies to send interactions from one *target* to another, as needed, until the interactions have been successfully processed.

The Routing Platform SDK allows you to write .NET applications that combine logic from your custom application with the router-based logic of URS, in order to solve many common interaction-related tasks.

This document tells you where you can go to get more information about URS. It also contains a brief overview of the features of the Routing Platform SDK, followed by code snippets that show how to implement the basic functions you will need to write applications that work with URS.

Universal Routing Server Overview

The best way to start learning about Universal Routing Server (URS) is by getting a copy of the Universal Routing 8.1 Reference Manual. This book tells you how to work with routing strategies, objects, functions, options, and statistics. It also includes a detailed list of *Related Documentation Resources*, which discusses other sources of information that can be useful when you are working with Genesys Universal Routing.

After becoming familiar with the information in the *Universal Routing Reference Manual* and related documentation, you can start using the routing API that is exposed by the Platform SDK. As you learn about Genesys routing, it is important to keep in mind that the main purpose of the Platform SDK routing API is to work as a complement to the complex capabilities already available from URS, not to act as a replacement. This API makes it easier to resolve difficult interaction-related tasks by combining the capabilities of URS with logic from your custom application.

To create routing strategies, you use either Genesys Composer, which lets you create SCXML-based strategies, or Interaction Routing Designer (IRD), which creates strategies in the Genesys IRL routing language. Once the URS environment is established, you then use the Platform SDK routing API to give your application control over which routing strategies are selected under a given set of circumstances or what criteria URS uses to choose a particular routing target. For example, your application can select statistics for URS to use in determining which agent group would be the best one to route a particular interaction to.

Two Types of Router API Usage

Platform SDK lets you use two different methodologies in working with URS. The first method involves a **standalone router**. When you use the standalone router method, all of the interaction processing logic, including media control, is handled by the router. This method can be used by calling `RequestLoadStrategy`.

The second method is called the **router-behind** API. This method can be used when you want your

application to handle media control, such as attached data or treatments, rather than leaving that up to the router. With this method, the router is normally used only to select resources.

The code snippets in this article include some requests that work with standalone routers and some that work with the router-behind API.

Java

Connecting to Universal Routing Server

The Platform SDK uses a message-based architecture to connect to Genesys servers. In general, Genesys recommends that you use the Protocol Manager Application Block to handle these connections. We also recommend that you use the Message Broker Application Block for your message handling. However, the current version of the Protocol Manager Application Block does not support Universal Routing Server (URS). Because of this, the following code samples show how to connect to URS by using the native protocol object that is part of the Routing Platform SDK.

You can modify the Protocol Manager Application Block to support URS by following the instructions in the section on *Supporting New Protocols* in the article on Using the Protocol Manager Application Block. After doing that, you can use the material in the article on Connecting to a Server to modify your code to use the Protocol Manager Application Block. In the meantime, here is how to connect to URS using a native protocol object.

First set up `import` statements for the routing namespaces:

```
[Java]

import com.genesyslab.platform.routing.protocol.routingserver.*;
import com.genesyslab.platform.routing.protocol.routingserver.requests.*;
```

After you have set up your import statements, you need to create a `RoutingServerProtocol` object:

```
[Java]

RoutingServerProtocol protocol =
        new RoutingServerProtocol(
                new Endpoint(
                        name, host, port));
protocol.setClientName(clientName);
protocol.setClientType(clientType);
```

Then you can open your connection to URS:

```
[Java]

protocol.open();
```

Message Handling

Once you have set up your server connection, you can set up Message Broker, the application block that handles the events returned by URS. This section gives you an idea of how to do that, based on the information in the Event Handling article. First, here are the import statements:

```
[Java]

import com.genesyslab.platform.applicationblocks.commons.Action;
import com.genesyslab.platform.applicationblocks.commons.broker.BrokerServiceFactory;
import com.genesyslab.platform.applicationblocks.commons.broker.EventBrokerService;
import com.genesyslab.platform.applicationblocks.commons.broker.MessageIdFilter;
```

Now you can create an Event Broker Service object:

```
[Java]

eventBrokerService = BrokerServiceFactory
        .CreateEventBroker(protocolManagementServiceImpl.getReceiver());
```

For each message you want to receive, you must set up a handler class. Here is a sample:

```
[Java]

class EventInfoHandler implements Action {

        public void handle(Message obj) {
                EventInfo eventInfo = (EventInfo) obj;
                if (eventInfo != null) {
                        System.out.println("EventInfo:\n"
                                + eventInfo.toString());
                }
                ...
        }
}
```

This handler processes any `EventInfo` messages you receive. The handler classes for other messages will have a similar structure. In order to work properly, each handler must be registered with the Event Broker Service. Here is how to register your `EventInfo` handler:

```
[Java]

eventBrokerService.register(new EventInfoHandler(),
        new MessageIdFilter(EventInfo.ID));
```

Working with URS

As mentioned above, there are two basic methods for using the Platform SDK to work with URS. This section contains examples of both the standalone router and router-behind APIs.

Standalone Router

The Routing Platform SDK allows you to control which routing strategy is executed on a given routing point, while leaving everything else to the routing server. To use this methodology, which is known as

"standalone router," issue a RequestLoadStrategy that specifies the routing point and the associated T-Server, and also the location of the routing strategy. Once the routing strategy has been loaded, all interactions arriving on the specified routing point will be processed with that strategy.

The following snippet shows how to do this:

```
[Java]

RequestLoadStrategy requestLoadStrategy = RequestLoadStrategy.create();
requestLoadStrategy.setTServer("TheT-Server");
requestLoadStrategy.setRoutingPoint("TheRoutingPoint");
requestLoadStrategy.setPath("<Path to the strategy>");

Message response = protocol.request(requestLoadStrategy);
```

URS will respond to your request with an EventInfo, an example of which is shown here:

```
'EventInfo' (2) attributes:
                R_Message [str] = "ATTENTION: Strategy has been loaded from ooo-file."
                R_cdn_status [int] = 1 [Loaded]
                R_cdn [str] = "RP_sip1"
                R_ErrorCode [int] = 0 [NoError]
                R_tserver [str] = "TServerSip1"
                R_refID [int] = 1
                R_time [str] = "06/30/2011 10:00:29"
                R_path [str] = "<Path>"
```

You can use RequestNotify to check which routing points have been loaded:

```
[Java]

RequestNotify requestNotify =
    RequestNotify.create();

protocol.send(requestNotify);
```

This request will also return an EventInfo similar to the one shown above.

When you want to stop using the routing strategy you have loaded, for example, if you want to start using a different one, you can issue a RequestReleaseStrategy:

```
[Java]

RequestReleaseStrategy requestReleaseStrategy =
    RequestReleaseStrategy.create();
requestReleaseStrategy.setTServer("TheT-Server");
requestReleaseStrategy.setRoutingPoint("TheRoutingPoint");

Message response = protocol.request(requestReleaseStrategy);
```

Router-Behind API

The router-behind method allows your application code to handle media control. The following example shows how to execute a strategy using RequestExecuteStrategy. This request is different from RequestLoadStrategy in that it only executes a strategy one time, instead of associating a particular strategy with a routing point. To use RequestExecuteStrategy, specify the routing strategy you want to execute and the tenant (contact center) in whose environment the strategy is to be executed, as shown here:

```
[Java]

RequestExecuteStrategy requestExecuteStrategy =
        RequestExecuteStrategy.create();
requestExecuteStrategy.setStrategy("TheRoutingStrategyName");
requestExecuteStrategy.setTenant("TheTenant");

Message response = protocol.request(requestExecuteStrategy);
```

It is important to remember that it can often take a considerable amount of time to process a routing strategy. If your request is correctly formatted and can be executed by URS, then you will immediately receive an `EventExecutionInProgress`. This is a very simple event that only returns the reference ID of your request, as you can see here:

```
'EventExecutionInProgress' ('199')
message attributes:
R_refID [int]   = 2
```

Once your request has successfully executed, you will receive an `EventExecutionAck`. Here is an example of the kind of output you might receive from an EventExecutionAck:

```
'EventExecutionAck' ('200')
message attributes:
R_result [bstr] = KVList:
      'DN' [str] = "701"
      'CUSTOMER_ID' [str] = "TenantForTest"
      'TARGET' [str] = "701_sip@StatServer1.A"
      'SWITCH' [str] = "SipSwitch"
      'NVQ' [int] = 1
      'PLACE' [str] = "701"
      'AGENT' [str] = "701_sip"
      'ACCESS' [str] = "701"
      'VQ' [str] = "1234"
```

Context = ComplexClass(OperationContext):

```
      UserData [bstr] = KVList:
             'ServiceObjective' [str] = ""
             'ServiceType' [str] = "default"
             'CBR-Interaction_cost' [str] = ""
             'RTargetTypeSelected' [str] = "0"
             'CBR-IT-path_DBIDs' [str] = ""
             'RVQDBID' [str] = ""
             'RTargetPlaceSelected' [str] = "701"
             'RTargetAgentSelected' [str] = "701_sip"
             'CBR-actual_volume' [str] = ""
             'RStrategyName' [str] = "##GetTarget"
             'RRequestedSkillCombination' [str] = ""
             'RTargetRuleSelected' [str] = ""
             'RStrategyDBID' [str] = ""
             'RRequestedSkills' [bstr] = KVList:

             'CustomerSegment' [str] = "default"
             'RTargetObjSelDBID' [str] = "984"
             'RTargetObjectSelected' [str] = "701_sip"
             'RTenant' [str] = "TenantForTest"
             'RVQID' [str] = ""
             'CBR-contract_DBIDs' [str] = ""
R_refID [int]   = 0
```

If you have any syntax errors, your request will not execute and you will receive an `EventError`. Here

is an example of an EventError:

```
'EventError' (1) attributes:
                R_cdn_status [int] = 0 [Released]
                R_cdn [str] = ""
                R_ErrorCode [int] = 4 [NotAvailable]
                R_tserver [str] = ""
                R_refID [int] = 1
                R_time [str] = ""
                R_path [str] = "<Path>"
```

If, on the other hand, URS has a problem executing your request, you will receive an EventExecutionError, an example of which is shown here:

```
'EventExecutionError' ('201')
message attributes:
R_result [bstr] = KVList:
                                'Reason' [str] = "Rejected"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'PegRejected' [int] = 1
R_refID [int]   = 2
```

There may be times when you want URS to pick a routing target for you. You can use RequestFindTarget for that purpose. As shown in the sample below, you can use a statistic to aid in this selection:

```
[Java]

RequestFindTarget requestFindTarget =
    RequestFindTarget.create();
requestFindTarget.setTenant("TheTenant");
requestFindTarget.setTargets("TheTargetList");
requestFindTarget.setTimeout(5);
requestFindTarget.setStatistic("TheStatistic");
requestFindTarget.setStatisticUsage(StatisticUsage.Max);
requestFindTarget.setVirtualQueue("TheQueue");
requestFindTarget.setPriority(1);
requestFindTarget.setMediaType("TheMediaType");

Message response = protocol.request(requestFindTarget);
```

You can also have URS fetch statistical information for you directly, in case you want to know more about the current conditions in your contact center, perhaps in preparation for a RequestFindTarget. The following example shows how to do this, using RequestGetStatistic:

```
[Java]

RequestGetStatistic requestGetStatistic =
        RequestGetStatistic.create();
requestGetStatistic.setTenant("TheTenant");
requestGetStatistic.setTargets("TheTargetList");
requestGetStatistic.setStatistic("StatAgentsBusy");

Message response = protocol.request(requestGetStatistic);
```

Both RequestFindTarget and RequestGetStatistic return the same messages as RequestExecuteStrategy.

Closing the Connection

When you are finished communicating with URS, you should close the connection, in order to minimize resource utilization:

[Java]

```
protocol.close();
```

.NET

Connecting to Universal Routing Server

The Platform SDK uses a message-based architecture to connect to Genesys servers. In general, Genesys recommends that you use the Protocol Manager Application Block to handle these connections. We also recommend that you use the Message Broker Application Block for your message handling. However, the current version of the Protocol Manager Application Block does not support Universal Routing Server (URS). Because of this, the following code samples show how to connect to URS by using the native protocol object that is part of the Routing Platform SDK.

You can modify the Protocol Manager Application Block to support URS by following the instructions in the section on *Supporting New Protocols* in the article on Using the Protocol Manager Application Block. After doing that, you can use the material in the article on Connecting to a Server to modify your code to use the Protocol Manager Application Block. In the meantime, here is how to connect to URS using a native protocol object.

First set up using statements for the routing namespaces:

[C#]

```
using Genesyslab.Platform.Routing.Protocols;
using Genesyslab.Platform.Routing.Protocols.RoutingServer;
using Genesyslab.Platform.Routing.Protocols.RoutingServer.Events;
using Genesyslab.Platform.Routing.Protocols.RoutingServer.Requests;
```

After you have set up your using statements, you need to create a `RoutingServerProtocol` object:

[C#]

```
RoutingServerProtocol protocol =
        new RoutingServerProtocol(
                new Endpoint(
                        name, host, port));
protocol.ClientName = clientName;
protocol.ClientType = clientType;
```

Then you can open your connection to URS:

[C#]

```
protocol.Open();
```

Message Handling

Once you have set up your server connection, you can set up Message Broker, the application block that handles the events returned by URS. This section gives you an idea of how to do that, based on the information in the Event Handling article.

First, here is the using statement:

```
[C#]

using Genesyslab.Platform.ApplicationBlocks.Commons.Broker;
```

Now you can create an Event Broker Service object:

```
[C#]

eventBrokerService =
        BrokerServiceFactory.CreateEventBroker(protocol);
```

For each message you want to receive, you must set up a handler. Here is a sample:

```
[C#]

private void OnEventExecutionAck(IMessage theMessage)
{
        EventExecutionAck eventExecutionAck =
                theMessage as EventExecutionAck;
        if (eventExecutionAck != null)
        {
                writeToLogArea("EventExecutionAck:\n"
                        + eventExecutionAck + "\n");
                ...
        }
}
```

This handler processes any EventExecutionAck messages you receive. The handler classes for other messages will have a similar structure. In order to work properly, each handler must be registered with the Event Broker Service. Here is how to register handlers for three of the URS events:

```
[C#]

eventBrokerService.Register(this.OnEventExecutionAck, new
MessageIdFilter(EventExecutionAck.MessageId));
eventBrokerService.Register(this.OnEventInfo, new MessageIdFilter(EventInfo.MessageId));
eventBrokerService.Register(this.OnEventError, new MessageIdFilter(EventError.MessageId));
```

Working with URS

As mentioned above, there are two basic methods for using the Platform SDK to work with URS. This section contains examples of both the standalone router and router-behind APIs.

Standalone Router

The Routing Platform SDK allows you to control which routing strategy is executed on a given routing

point, while leaving everything else to the routing server. To use this "standalone router" methodology, issue a RequestLoadStrategy that specifies the routing point and the associated T-Server, and also the location of the routing strategy. Once the routing strategy has been loaded, all interactions arriving on the specified routing point will be processed with that strategy.

The following snippet shows how to do this:

```
[C#]

RequestLoadStrategy requestLoadStrategy =
    RequestLoadStrategy.Create();
requestLoadStrategy.TServer = "TheT-Server";
requestLoadStrategy.RoutingPoint = "TheRoutingPoint";
requestLoadStrategy.Path = "<Path to the strategy>";

IMessage response = protocol.Request(requestLoadStrategy);
```

URS will respond to your request with an EventInfo, an example of which is shown here:

```
'EventInfo' (2) attributes:
                R_Message [str] = "ATTENTION: Strategy has been loaded from ooo-file."
                R_cdn_status [int] = 1 [Loaded]
                R_cdn [str] = "RP_sip1"
                R_ErrorCode [int] = 0 [NoError]
                R_tserver [str] = "TServerSip1"
                R_refID [int] = 1
                R_time [str] = "06/30/2011 10:00:29"
                R_path [str] = "<Path>"
```

You can use RequestNotify to check which strategies have been loaded to routing points:

```
[C#]

RequestNotify requestNotify =
    RequestNotify.Create();

protocol.Send(requestNotify);
```

This request will also return an EventInfo similar to the one shown above.

When you want to stop using the routing strategy you have loaded, for example, if you want to start using a different one, you can issue a RequestReleaseStrategy:

```
[C#]

RequestReleaseStrategy requestReleaseStrategy =
    RequestReleaseStrategy.Create();
requestReleaseStrategy.TServer = "TheT-Server";
requestReleaseStrategy.RoutingPoint = "TheRoutingPoint";

IMessage response = protocol.Request(requestReleaseStrategy);
```

Router-Behind API

The router-behind method allows your application code to handle media control. The following example shows how to execute a strategy using RequestExecuteStrategy. This request is different from RequestLoadStrategy in that it only executes a strategy one time, instead of associating a particular strategy with a routing point. To use RequestExecuteStrategy, specify the routing

strategy you want to execute and the tenant (contact center) in whose environment the strategy is to be executed, as shown here:

```
[C#]

RequestExecuteStrategy requestExecuteStrategy =
        RequestExecuteStrategy.Create();
requestExecuteStrategy.Strategy = "TheRoutingStrategyName";
requestExecuteStrategy.Tenant = "TheTenant";

IMessage response = protocol.Request(requestExecuteStrategy);
```

It is important to remember that it can often take a considerable amount of time to process a routing strategy. If your request is correctly formatted and can be executed by URS, then you will immediately receive an `EventExecutionInProgress`. This is a very simple event that only returns the reference ID of your request, as you can see here:

```
'EventExecutionInProgress' ('199')
message attributes:
R_refID [int]   = 2
```

Once your request has successfully executed, you will receive an `EventExecutionAck`. Here is an example of the kind of output you might receive from an `EventExecutionAck`:

```
'EventExecutionAck' ('200')
message attributes:
R_result [bstr] = KVList:
                                'DN' [str] = "701"
                                'CUSTOMER_ID' [str] = "TenantForTest"
                                'TARGET' [str] = "701_sip@StatServer1.A"
                                'SWITCH' [str] = "SipSwitch"
                                'NVQ' [int] = 1
                                'PLACE' [str] = "701"
                                'AGENT' [str] = "701_sip"
                                'ACCESS' [str] = "701"
                                'VQ' [str] = "1234"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'ServiceObjective' [str] = ""
                                                'ServiceType' [str] = "default"
                                                'CBR-Interaction_cost' [str] = ""
                                                'RTargetTypeSelected' [str] = "0"
                                                'CBR-IT-path_DBIDs' [str] = ""
                                                'RVQDBID' [str] = ""
                                                'RTargetPlaceSelected' [str] = "701"
                                                'RTargetAgentSelected' [str] = "701_sip"
                                                'CBR-actual_volume' [str] = ""
                                                'RStrategyName' [str] = "##GetTarget"
                                                'RRequestedSkillCombination' [str] = ""
                                                'RTargetRuleSelected' [str] = ""
                                                'RStrategyDBID' [str] = ""
                                                'RRequestedSkills' [bstr] = KVList:

                                                'CustomerSegment' [str] = "default"
                                                'RTargetObjSelDBID' [str] = "984"
                                                'RTargetObjectSelected' [str] = "701_sip"
                                                'RTenant' [str] = "TenantForTest"
                                                'RVQID' [str] = ""
                                                'CBR-contract_DBIDs' [str] = ""
R_refID [int]   = 0
```

If you have any syntax errors, your request will not execute and you will receive an EventError. Here is an example of an EventError:

```
'EventError' (1) attributes:
                R_cdn_status [int] = 0 [Released]
                R_cdn [str] = ""
                R_ErrorCode [int] = 4 [NotAvailable]
                R_tserver [str] = ""
                R_refID [int] = 1
                R_time [str] = ""
                R_path [str] = "<Path>"
```

If, on the other hand, URS has a problem executing your request, you will receive an EventExecutionError, an example of which is shown here:

```
'EventExecutionError' ('201')
message attributes:
R_result [bstr] = KVList:
                                'Reason' [str] = "Rejected"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'PegRejected' [int] = 1
R_refID [int]   = 2
```

There may be times when you want URS to pick a routing target for you. You can use RequestFindTarget for that purpose. As shown in the sample below, you can use a statistic to aid in this selection:

```
[C#]

RequestFindTarget requestFindTarget =
    RequestFindTarget.Create();
requestFindTarget.Tenant = "TheTenant";
requestFindTarget.Targets = "TheTargetList";
requestFindTarget.Timeout = 5;
requestFindTarget.Statistic = "TheStatistic";
requestFindTarget.StatisticUsage = StatisticUsage.Max;
requestFindTarget.VirtualQueue = "TheQueue";
requestFindTarget.Priority = 1;
requestFindTarget.MediaType = "TheMediaType";

IMessage response = protocol.Request(requestFindTarget);
```

You can also have URS fetch statistical information for you directly, in case you want to know more about the current conditions in your contact center, perhaps in preparation for a RequestFindTarget. The following example shows how to do this, using RequestGetStatistic:

```
[C#]

RequestGetStatistic requestGetStatistic =
    RequestGetStatistic.Create();
requestGetStatistic.Tenant = "TheTenant";
requestGetStatistic.Targets = "TheTargetList";
requestGetStatistic.Statistic = "StatAgentsBusy";

IMessage response = protocol.Request(requestGetStatistic);
```

Both RequestFindTarget and RequestGetStatistic return the same messages as RequestExecuteStrategy.

Closing the Connection

When you are finished communicating with URS, you should close the connection, in order to minimize resource utilization:

```
[C#]

protocol.Close();
```