Platform SDK Developer's Guide

Using the Switch Policy Library

4/5/2026

Using the Switch Policy Library

This document shows how to add simple T-Server functionality to your applications by using the Switch Policy Library.

The Platform SDK Switch Policy Library (SPL) can be used in applications that need to perform agent-related switch activity with a variety of T-Servers, without knowing beforehand what kinds of T-Servers will be used. It simplifies these applications by indicating which switch functions are available at any given time and also by showing how you can use certain switch features in your applications. However, if your application works with only one kind of T-Server, you may want to have your application communicate directly with the T-Server, rather than using SPL.

Switch Policy Library Overview

Some telephony applications need to work with more than one type of switch. Unfortunately, however, one switch may not perform a particular telephony function in the same way as another switch. This means that it can be useful to have an abstraction layer of some kind when working with multiple switches, so that you do not need custom code for each switch that is used by the application. The Switch Policy Library is designed with just this kind of abstraction in mind.

Java

Setting Up Switch Policy Library

SPL should be used by your agent desktop applications as a library, which means that it would be located within the agent desktop application shown above. The application can call SPL for guidance on how to send requests to or process events from your T-Server, as shown in the *Code Samples* section.

SPL is driven by an XML-based configuration file that supports many commonly-used switches in performing agent-related functions. Your application can query SPL to determine whether a particular feature is supported for the switch you want to work with. If a feature you need is not supported for the switches you need to work with, you can make a copy of the default configuration file and modify it as needed.

> **Important**
>
> Genesys does not support modifications to the SPL configuration file. Any modifications you make are performed at your own risk.

A copy of the default configuration file is included inside the Switch Policy Library JAR file. You can extract the XML configuration file from `switchpolicy.jar`, modify it, and pass it as an argument to the corresponding method of the `SwitchPolicyServiceFactory` factory class.

Code Samples

This section contains examples of how to perform useful functions with SPL.

These samples each require a valid instance of the `ISwitchPolicyService`, which can be created as shown here:

```
[Java]

ISwitchPolicyService service =
        SwitchPolicyServiceFactory.createSwitchPolicyService();
```

> **Tip**
>
> The `DN` classes specified below implement the `IDNContext` interface, while the `Party` classes implement the `IPartyContext` interface, and the `Call` classes implement the `ICallContext` interface.

Customizing the XML Configuration File

The following code samples create a service using the default configuration. As noted above, Genesys does not support modifications to the default SPL configuration file. Should you decide to assume the risk of creating a custom XML configuration file, your application can access this file as shown here:

```
[Java]

public void serviceCreationWithParent(ApplicationContext parent) {
        final String file_path =
                "<Path to XML Configuration File>";
        FileSystemResource resource =
                new FileSystemResource(file_path);
        ISwitchPolicyService service =
                SwitchPolicyServiceFactory.createSwitchPolicyService(parent, resource);
}
```

Get A Phone Set Configuration

On some switches, phone sets are presented as more than one *Directory Number* (DN). These DNs may also have different types, such as *Position* and *Extension*. Because these configurations vary by switch type, an application needs to know how the phone set configuration for a particular switch is structured. For example, it needs to know how many DNs are used to represent a phone set, and what their types are. To retrieve this phone set configuration information, perform the following steps:

1. Create an instance of `PhoneSetConfigurationContext`, specifying the switch type.

2. Call `ISwitchPolicyService.getPolicy`, using this `PhoneSetConfigurationContext`.
3. Analyze the returned `PhoneSetConfigurationPolicy`. The `PhoneSetConfigurationPolicy.getConfigurations` method will return all possible phone set configurations for the specified switch.

The following code snippet shows how to do this:

```
[Java]

ISwitchPolicyService service =
        SwitchPolicyServiceFactory.createSwitchPolicyService();
PhoneSetConfigurationContext context =
        new PhoneSetConfigurationContext("SwitchName");
PhoneSetConfigurationPolicy policy =
        service.getPolicy(PhoneSetConfigurationPolicy.class, context);
System.out.println(policy);
```

Get Phone Set Availability Information

When working with a phone set, additional information about the included DNs may be required. This could include information about which of the DNs should be available to the end user (for example, which ones should be visible in the user interface), which of them is callable, and which number (the `Callable Number`) the application should use to reach the agent who is logged into the phone set. To retrieve this phone set availability information, perform the following steps:

1. Create an instance of `DNAvailabilityContext` and populate it with the following required information:
    1. Specify the switch type.
    2. Specify the Agent ID.
    3. Fill the DN collection with valid implementations of `IDNContext`.
2. Call `ISwitchPolicyService.getPolicy`, using this `DNAvailabilityContext`.
3. Analyze the returned `DNAvailabilityPolicy`. The `DNAvailabilityPolicy.getDNStatuses` method will return availability information for each DN in the request.

The following code snippet shows how to do this:

```
[Java]

String extDN = "1001";
String posDN = "2001";
String agentID = "9999";
// logout, in service
DNAvailabilityContext context = new DNAvailabilityContext(switchname);
context.setAgentId(agentID);
DNContextStub ext = new DNContextStub(); // implements IDNContext interface
ext.setIdentifier(extDN);
ext.setAgentStatus(AgentStatus.LOGOUT);
ext.setServiceStatus(ServiceStatus.IN_SERVICE);
ext.setType(AddressType.DN);

DNContextStub pos = new DNContextStub(); // implements IDNContext interface
pos.setIdentifier(posDN);
pos.setAgentStatus(AgentStatus.LOGOUT);
pos.setServiceStatus(ServiceStatus.IN_SERVICE);
pos.setType(AddressType.Position);
```

```
ArrayList<IDNContext> dns = new ArrayList<IDNContext>();
dns.add(ext);
dns.add(pos);
context.setDNs(dns);

// here service is correctly initialized instance of ISwitchPolicyService
DNAvailabilityPolicy policy = service.getPolicy(DNAvailabilityPolicy.class, context);
System.out.println(policy);
```

Get Function Availability Information for the Current Context

Some switches differ in when they allow certain functions to be performed. Also, some functions can always be performed on certain switches, while others may be impossible to perform. For example, `RequestMergeCalls` can never be performed on some switches. For other functions, whether or not the function can be performed varies depending on context. For example, on some switches `RequestReleaseCall` can only be used when a call is in a *Held, Dialing*, or *Established* state, while on other switches it is also possible to release a call when it is in a *Ringing* state. In addition to this, on some switches the phone set is presented as more than one *Directory Number* (DN) and each DN can have a different type, such as *Position* and *Extension*. Some functions are allowed for both types, while some other functions may be restricted to a certain DN type. To retrieve this kind of function availability information for the current context, perform the following steps:

1. Create an instance of `FunctionHandlingContext` and populate it with the following required information:
    1. Specify the switch type.
    2. Specify the request by calling the `setMessage` method.
    3. Describe the context as fully as possible.
2. Call `ISwitchPolicyService.getPolicy`, using this `FunctionHandlingContext`.
3. Analyze the returned `FunctionAvailabilityPolicy`. If the specified request is possible in the given context, the `getIsFunctionAvailable` method will return true. However, if the request is not supported, SPL will return null.

The following code snippet shows how to do this:

```
[Java]

DNContextStub dn = new DNContextStub();// implements IDNContext
dn.setIdentifier("1001");
dn.setType(AddressType.DN);
dn.setAgentStatus(AgentStatus.READY);
dn.setServiceStatus(ServiceStatus.IN_SERVICE);

DNContextStub otherdn = new DNContextStub();
otherdn.setIdentifier("2001");
otherdn.setType(AddressType.DN);
otherdn.setAgentStatus(AgentStatus.READY);
otherdn.setServiceStatus(ServiceStatus.IN_SERVICE);

PartyContextStub mainparty = new PartyContextStub();// implements IPartyContext
mainparty.setIdentifier("9841");
mainparty.setStatus(PartyStatus.ESTABLISHED);
mainparty.setIsConferencing(true);
mainparty.setIsTransferring(true);
```

```
mainparty.setDN(dn);

PartyContextStub otherParty = new PartyContextStub();
otherParty.setIdentifier(mainparty.getIdentifier());
otherParty.setStatus(PartyStatus.ESTABLISHED);
otherParty.setIsConferencing(true);
otherParty.setIsTransferring(true);
otherParty.setDN(otherdn);

CallContextStub call = new CallContextStub();// Implements ICallContext
call.setStatus(CallStatus.ESTABLISHED);
call.setDestination(mainparty);
call.setOrigination(otherParty);
call.setIdentifier(mainparty.getIdentifier());
call.setConferencing(true);
call.setTransferring(true);
call.setParties(Arrays.<IPartyContext> asList(mainparty, otherParty));

for (String swtype : new String[] { swtypeA4400Classic, swtypeA4400Emul, swtypeA4400Subs }) {
        for (CallType callType : GEnum.valuesBy(CallType.class)) {
                FunctionHandlingContext context = new FunctionHandlingContext(swtype);
                context.setMessage(RequestHoldCall.create());
                context.setDN(dn);
                mainparty.setCallType(callType);
                otherParty.setCallType(callType);
                call.setCallType(callType);
                context.setParty(mainparty);
                context.setCall(call);
                FunctionAvailabilityPolicy policy =
service.getPolicy(FunctionAvailabilityPolicy.class, context);
                System.out.println(policy);
        }
}
```

Get Instructions On How To Implement a Feature

Some switches differ in how certain features can be accessed. The majority of their features may map directly to individual switch functions, but this is not always so. For example, for some switches it is not possible to log the agent out while the agent is in the ready state. So, the feature which implements agent logout for these switches would require two steps:

1. Make sure the agent is in a `NotReady` state
2. Log the agent out

SPL implements a *feature handler* for each feature that it supports. To create and run a feature handler, perform the following steps:

1. Create a new instance of `FunctionHandlingContext` and populate it with the following required information:
    1. Specify the switch type.
    2. Specify the request by calling the `setMessage` method. This step can be omitted if a feature handler is going to be created by using the `featureName` parameter in the `ISwitchPolicyService.createFeatureHandler(String featureName, FunctionHandlingContext context)` method.
    3. Provide a valid Protocol instance by calling the `setProtocol` method.

4. Describe the context as fully as possible.

2. Call `ISwitchPolicyService.createFeatureHandler` and pass this `FunctionHandlingContext`, either alone or with the name of the feature.
3. Call the `beginExecute` method of `IFeatureHandler` on the returned handler, passing the same instance of `FunctionHandlingContext`.
4. The remainder of the processing depends on the implementation, but the general approach is to perform the following actions while the status of the handler is `Executing`:
    1. Receive event from T-Server.
    2. Update `FunctionHandlingContext` based on the received event.
    3. Assign the received event by calling the `setMessage` method of your `FunctionHandlingContext` instance.
    4. Call the handle method of `IFeatureHandler` passing with it the updated `FunctionHandlingContext`.

The following code snippet shows how to do this:

```
[Java]

public void LogoutReadyAgent(Protocol protocol,
        ISwitchPolicyService service, String thisDN, String switchType)
        throws IllegalStateException, InterruptedException,
        SwitchPolicyException {

        FunctionHandlingContext context =
                new FunctionHandlingContext(switchType);
        context.setMessage(RequestAgentLogout.create(thisDN));
        DNContextStub dn = new DNContextStub(); // implements IDNContext interface
        dn.setAgentStatus(AgentStatus.READY);
        dn.setAgentWorkMode(AgentWorkMode.Unknown);
        dn.setIdentifier(thisDN);
        dn.setType(AddressType.DN);
        dn.setServiceStatus(ServiceStatus.IN_SERVICE);
        context.setDN(dn);
        context.setProtocol(protocol);

        IFeatureHandler handler = service.createFeatureHandler(context);
        if (handler != null) {
                handler.beginExecute(context);
                while (handler.getStatus() == FeatureStatus.EXECUTING) {
                        Message message = (Message) protocol.receive();
                        // update context due to received message
                        // .....
                        context.setMessage(message);
                        handler.handle(context);
                }
        }
}
```

Get Instructions On How To Accomplish Complex Functionality

Your application may sometimes need access to functionality that depends on the switch type. For example, when an application receives events from the T-Server, the way a given event's fields are used can depend on both the call scenario and the switch type. To retrieve this information, perform the following steps:

1. Create a `MessageHandlingContext` and populate it with the following required information:
    1. Name of switch.
    2. Name of handler.
2. Call `ISwitchPolicyService.createMessageHandler`, pass this context into it, and receive the resulting `IMessageHandler`.
3. Call the `IMessageHandler.handle` method on the received handler.

The following code snippet shows how to do this:

```
[Java]

        EventRinging msgRinging =
                EventRinging.create(TimeStamp.create(1249566176, 312000));

        KeyValueCollection p = new KeyValueCollection();
        p.addInt("BusinessCall", 0);
        p.addInt("GCTI_BUSINESS_CALL", 0);
        p.addString("GCTI_SUB_THIS_DN", "18101");
        p.addString("GCTI_SUB_OTHER_DN", "18100");
        p.addString("GCTI_OTHER_DEVICE_NAME", "18100");
        p.addString("GCTI_PARTY_NAME", "18100");
        msgRinging.setExtensions(p);
        msgRinging.setEventSequenceNumber(0x0000000000000399);
        msgRinging.setOtherDN("11100");
        msgRinging.setOtherDNRole(DNRole.RoleOrigination);//
        msgRinging.setOtherTrunk(521);
        msgRinging.setThisDNRole(DNRole.RoleOrigination);//
        msgRinging.setThisDN("11101");
        msgRinging.setDNIS("18101");
        msgRinging.setThisTrunk(522);
        msgRinging.setCallUuid("BTMT3AJVT17QPE364J2DV9V5I000005P");
        msgRinging.setConnID(new ConnectionId("022701b746b29021"));
        msgRinging.setCallID(3648);
        msgRinging.setCallType(CallType.Internal);
        msgRinging.setNetworkCallID(0x1ee07a4a400e0100l);
        msgRinging.setCallState(0);
        msgRinging.setAgentID("18101");
        msgRinging.setPropagatedCallType(CallType.Internal);

        MessageHandlingContext context = new MessageHandlingContext(Switchname);
        context.setHandlerName("OtherDN");
        IMessageHandler othDNH = service.createMessageHandler(context);
        String otherDN = (String) othDNH.handle(msgRinging); System.out.println(otherDN);
```

Add Logging Support

You can add support for logging by providing an application context with a registered `ILogger` bean. This logger will be used by the Switch Policy Library. Here is a code sample:

```
[Java]

AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext();
context.register(ConsoleLogger.class);

// ConsoleLogger implements ILogger interface
context.refresh();
```

```
ISwitchPolicyService service =
        SwitchPolicyServiceFactory.createSwitchPolicyService(context);
```

.NET

Setting Up Switch Policy Library

SPL should be used by your agent desktop applications as a library, which means that it would be located within the agent desktop application shown above. The application can call SPL for guidance on how to send requests to or process events from your T-Server, as shown in the *Code Samples* section.

SPL is driven by an XML-based configuration file that supports many commonly-used switches in performing agent-related functions. Your application can query SPL to determine whether a particular feature is supported for the switch you want to work with. If a feature you need is not supported for the switches you need to work with, you can make a copy of the default configuration file and modify it as needed.

> **Important**
>
> Genesys does not support modifications to the SPL configuration file. Any modifications you make are performed at your own risk.

A copy of the default configuration file is included inside the Switch Policy Library DLL. There is also a copy in the Bin directory of the Platform SDK installation package. If you need to modify the configuration file, you can use the `app.config` file for SPL to point to your copy.

Code Samples

This section contains examples of how to perform useful functions with SPL.

These samples each require a valid instance of the `ISwitchPolicyService`, which can be created as shown here:

```
[C#]

ISwitchPolicyService policyService =
        SwitchPolicyFactory.CreateSwitchPolicyService();
```

> **Tip**

> The DN classes specified below implement the IDNContext interface, while the Party classes implement the IPartyContext interface, and the Call classes implement the ICallContext interface.

Get A Phone Set Configuration

On some switches, phone sets are presented as more than one *Directory Number* (DN). These DNs may also have different types, such as *Position* and *Extension*. Because these configurations vary by switch type, an application needs to know how the phone set configuration for a particular switch is structured. For example, it needs to know how many DNs are used to represent a phone set, and what their types are. To retrieve this phone set configuration information, perform the following steps:

1. Create an instance of PhoneSetConfigurationContext, specifying the switch type.
2. Call ISwitchPolicyService.GetPolicy, using this PhoneSetConfigurationContext.
3. Analyze the returned PhoneSetConfigurationPolicy. The PhoneSetConfigurationPolicy.Configurations property will contain all possible phone set configurations for the specified switch.

The following code snippet shows how to do this:

```
[C#]

PhoneSetConfigurationContext context =
        new PhoneSetConfigurationContext("SomeSwitch");
PhoneSetConfigurationPolicy policy =
switchPolicyService.GetPolicy<PhoneSetConfigurationPolicy>(context);
foreach (PhoneSetConfiguration configuration in policy.Configurations)
{
        Console.WriteLine(configuration);
}
```

Get Phone Set Availability Information

When working with a phone set, additional information about the included DNs may be required. This could include information about which of the DNs should be available to the end user (for example, which ones should be visible in the user interface), which of them is callable, and which number (the Callable Number) the application should use to reach the agent who is logged into the phone set. To retrieve this phone set availability information, perform the following steps:

1. Create an instance of DNAvailabilityContext and populate it with the following required information:
   - Specify the switch type.
   - Specify the Agent ID.
   - Fill the DN collection with valid implementations of IDNContext.
2. Call ISwitchPolicyService.GetPolicy, using this DNAvailabilityContext.
3. Analyze the returned DNAvailabilityPolicy. The DNAvailabilityPolicy.DNStatuses property will contain availability information for each DN in the request.

The following code snippet shows how to do this:

```
[C#]

private static void DemonstrateDNAvailability(ISwitchPolicyService service)
{
        DNAvailabilityContext dnacontext =
                new DNAvailabilityContext("SomeSwitch");
        dnacontext.AgentId = "AgentLogin1000";
        dnacontext.DNs.Add(new Dn
        {
                AgentStatus = AgentStatus.Ready,
                        Identifier = "1000",
                        ServiceStatus = ServiceStatus.InService,
                        Type = AddressType.DN
        });
        dnacontext.DNs.Add(new Dn
        {
                AgentStatus = AgentStatus.Ready,
                        Identifier = "2000",
                        ServiceStatus = ServiceStatus.InService,
                        Type = AddressType.Position
        });

        DNAvailabilityPolicy dnpolicy =
                service.GetPolicy<DNAvailabilityPolicy>(dnacontext);
        DisplayInColor(dnpolicy, ConsoleColor.Red);
}
```

Get Function Availability Information for the Current Context

Some switches differ in when they allow certain functions to be performed. Also, some functions can always be performed on certain switches, while others may be impossible to perform. For example, `RequestMergeCalls` can never be performed on some switches. For other functions, whether or not the function can be performed varies depending on context. For example, on some switches `RequestReleaseCall` can only be used when a call is in a Held, Dialing, or Established state, while on other switches it is also possible to release a call when it is in a Ringing state. In addition to this, on some switches the phone set is presented as more than one *Directory Number* (DN) and each DN can have a different type, such as *Position* and *Extension*. Some functions are allowed for both types, while some other functions may be restricted to a certain DN type. To retrieve this kind of function availability information for the current context, perform the following steps:

1. Create an instance of `FunctionHandlingContext` and populate it with the following required information:
   - Specify the switch type.
   - Specify the request by setting the `Message` property.
   - Describe the context as fully as possible.
2. Call `ISwitchPolicyService.GetPolicy`, using this `FunctionHandlingContext`.
3. Analyze the returned `FunctionAvailabilityPolicy`. If the specified request is possible in the given context, the `IsFunctionAvailable` property will be true. However, if the request is not supported, SPL will return null.

The following code snippet shows how to do this:

```
[C#]

foreach (string switchType in new[] { swTypeA4400Classic, swTypeA4400emul, swTypeA4400Subs })
{
        DNContext dn = new DNContext //implements IDNContext
        {
                Identifier = "1001",
                Type = AddressType.DN,
                AgentStatus = AgentStatus.Ready,
                ServiceStatus = ServiceStatus.InService,
                DndStatus = FunctionStatus.Off,
                ForwardStatus = FunctionStatus.Off
        };

        DNContext otherDN = new DNContext
        {
                Identifier = "2001",
                Type = AddressType.DN,
                AgentStatus = AgentStatus.Ready,
                ServiceStatus = ServiceStatus.InService,
                DndStatus = FunctionStatus.Off,
                ForwardStatus = FunctionStatus.Off
        };

        foreach (CallType callType in Enum.GetValues(typeof(CallType)))
        {
                PartyContext mainParty = new PartyContext //implements IPartyContext
                {
                        Identifier = "1002",
                        Status = PartyStatus.Established,
                        CallType = callType,
                        IsConferencing = true,
                        IsTransferring = true,
                        DN = dn
                };

                PartyContext otherParty = new PartyContext
                {
                        Identifier = "1002",
                        CallType = callType,
                        DN = otherDN,
                        IsConferencing = true,
                        IsTransferring = true,
                        Status = PartyStatus.Established
                };
                CallContextStub ccontext = new CallContextStub //implements ICallContext
                {
                        CallType = callType,
                        Destination = mainParty,
                        Origination = otherParty,
                        Identifier = "1002",
                        IsConferencing = true,
                        IsTransferring = true,
                        Parties = new List<IPartyContext>{mainParty,otherParty},
                        Parent = null//no parentCall - our call is solitary call.
                };

                FunctionHandlingContext context = new FunctionHandlingContext(switchType)
                {
                        Message = RequestHoldCall.Create(),
                        DN = dn,
                        Party = mainParty,
                        Call = ccontext
```

```
                };
                FunctionAvailabilityPolicy policy =
service.GetPolicy<FunctionAvailabilityPolicy>(context);

                Console.WriteLine(policy);
        }
}
```

Get Instructions On How To Implement a Feature

Some switches differ in how certain features can be accessed. The majority of their features may map directly to individual switch functions, but this is not always so. For example, for some switches it is not possible to log the agent out while the agent is in the ready state. So, the feature which implements agent logout for these switches would require two steps:

1. Make sure the agent is in a NotReady state
2. Log the agent out

SPL implements a feature handler for each feature that it supports. To create and run a feature handler, perform the following steps:

1. Create a new instance of FunctionHandlingContext and populate it with the following required information:
   - Specify the switch type.
   - Specify the request by setting the Message property. This step can be omitted if the feature handler is created by using the featureName parameter in the ISwitchPolicyService.CreateFeatureHandler(String featureName, FunctionHandlingContext context) method.
   - Provide a valid IProtocol instance as the value of the Protocol property.
   - Describe the context as fully as possible.
2. Call the ISwitchPolicyService.CreateFeatureHandler and pass this FunctionHandlingContext, either alone or with the name of the feature.
3. Call the BeginExecute method on the returned handler, passing the same instance of FunctionHandlingContext.
4. The remainder of the processing depends on the implementation, but the general approach is to perform the following actions while the status of the handler is Executing:
   1. Receive event from TServer.
   2. Update FunctionHandlingContext based on the received event.
   3. Assign the received event to the Message property of your FunctionHandlingContext instance.
   4. Call the Handle method of IFeatureHandler passing with it the updated FunctionHandlingContext.

The following code snippet shows how to do this:

```
[C#]

private static void LoginReadyAgent(IProtocol protocol,
        ISwitchPolicyService service, string thisdn, string agentID)
```

```
{
        FunctionHandlingContext context = new FunctionHandlingContext("SomeSwitch");
        RequestAgentLogin requestAgentLogin = RequestAgentLogin.Create();
        requestAgentLogin.ThisDN = thisdn;
        requestAgentLogin.AgentID = agentID;
        requestAgentLogin.AgentWorkMode = AgentWorkMode.AutoIn;
        context.Message = requestAgentLogin;
        context.Protocol = protocol;

        IFeatureHandler loginHandler = service.CreateFeatureHandler(context);

        if(loginHandler == null)
        {
                protocol.Send(requestAgentLogin);
                // Process the incoming events for the scenario
                return;
        }

        // Processing feature handler
        loginHandler.BeginExecute(context);
        while (loginHandler.Status == FeatureStatus.Executing)
        {
                context.Message = context.Protocol.Receive();
                // Update the context based on the received T-Server event
                loginHandler.Handle(context);
        }
}
```

Get Instructions On How To Accomplish Complex Functionality

Your application may sometimes need access to functionality that depends on the switch type. For example, when an application receives events from the T-Server, the way a given event's fields are used can depend on both the call scenario and the switch type. To retrieve this information, perform the following steps:

1. Create a `MessageHandlingContext` and populate it with the following required information:
    - Name of switch.
    - Name of handler.
2. Call `ISwitchPolicyService.CreateMessageHandler`, pass this context into it, and receive the resulting `IMessageHandler`.
3. Call the `IMessageHandler.Handle` method on the received handler.

The following code snippet shows how to do this:

```
[C#]

private static void DemonstrateMessageHandler(ISwitchPolicyService service)
{
        EventRinging message = EventRinging.Create();
        message.ThirdPartyDN = "12345";
        message.DNIS = "18009870987";
        message.CallType = CallType.Internal;
        message.OtherDN = "9875";
        MessageHandlingContext context35 =
                new MessageHandlingContext("AlcatelA4400DHS3::Classic")
                { HandlerName = "OtherDN" };
        IMessageHandler handler = service.CreateMessageHandler(context35);
```

```
        string res = (string)handler.Handle(message);
        DisplayInColor(res, ConsoleColor.Yellow);
}
```

Add Logging Support

To add logging support, carry out the following steps:

1. Create an instance of `IUnityContainer` and register an anonymous instance or type mapping for the `ILogger` interface.
2. Pass the `IUnityContainer` created during the previous step to the factory method, which creates an instance of `ISwitchPolicyService`.

The following code snippet shows how to do this:

```
[C#]

IUnityContainer root = new UnityContainer();
root.RegisterInstance(new ConsoleLogger());
ISwitchPolicyService service =
        SwitchPolicyFactory.CreateSwitchPolicyService(root);
```

SPL also provides the following options:

- Your application can log the topmost messages into a distinct log. To use this option, call the `CreateSwitchPolicyService(IUnityContainer container, ILogger logger)` method of the `SwitchPolicyServiceFactory` class. The passed logger (if it is not null) will be used for logging the topmost messages.
- You can configure any switch container to use a specific logger. Objects created by the Unity container (feature handlers, policy providers and so on) can use the container to resolve the `ILogger` for further logging.

> **Tip**
>
> the classes provided by SPL resolve the `ILogger` (if there is one) at creation time. So, if your application changes the `ILogger` resolution rule for the root container that was previously passed into the `SwitchPolicyService` constructor after the corresponding method call, this will not affect:
>
> - Existing instances.
> - Objects which are created in the container(s), for which special `ILogger` mapping rule is configured.

Supported Functions

As mentioned above, SPL is driven by a configuration file that makes it possible to support a wide variety of switch functions. The following table shows functions that are supported by SPL at

installation time, using the default configuration file.

Switch Functions Supported by SPL At Installation Time

Switch Function	Description
DN and Agent Functions	
RequestAgentLogin	Logs in the agent specified by the AgentId parameter to the ACD group specified by the parameter.
RequestAgentLogout	Logs the agent out of the ACD group specified by the Queue parameter.
RequestAgentNotReady	Sets a state in which the agent is not ready to receive calls. The agents telephone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestAgentReady	Sets a state in which the agent is ready to receive calls. The agents phone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestCallForwardCancel	Sets the Forwarding feature to Off for the telephony object that is specified by the DN parameter.
RequestCallForwardSet	Sets the Forwarding feature to On for the telephony object that is specified by the DN parameter.
RequestCancelMonitoring	A request by a supervisor to cancel monitoring the calls delivered to the agent. If this request is successful, T-Server distributes EventMonitoringCancelled to all clients registered on the supervisor's and agent's DNs.
RequestMonitorNextCall	A request by a supervisor to monitor (be automatically conferenced in as a party on) the next call delivered to an agent. Supervisors can request to monitor one subsequent call or all calls until the request is explicitly canceled. If a request is successful, EventMonitoringNextCall is distributed to all clients registered on the supervisor's and agent's DNs. Supervisors start monitoring each call in Mute mode. To speak, they must execute the function
RequestSetDNDOff	Sets the Do-Not-Disturb (DND) feature to Off for the telephony object specified by the DN parameter.
RequestSetDNDOn	Sets the Do-Not-Disturb (DND) feature to On for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
Call Handling Functions	
RequestAlternateCall	On behalf of the telephony object specified by the DN parameter, places the active call specified by

Switch Function	Description
	thecurrent_conn_id parameter on hold and connects the call specified by the held_conn_id parameter.
RequestAnswerCall	Answers the alerting call specified by the conn_id parameter.
RequestAttachUserData	On behalf of the telephony object specified by the DN parameter, attaches the user data structure specified by the user_data parameter to the T-Server information that is related to the call specified by the conn_id parameter.
RequestClearCall	Deletes all parties, that is, all telephony objects, from the call specified by conn_id and disconnects the call.
RequestCompleteConference	Completes a previously-initiated conference by merging the held call specified by the held_conn_id parameter with the active consultation call specified by the current_conn_id parameter on behalf of the telephony object specified by the DN. Assigns the held_conn_id to the resulting conference call. Clears the consultation call specified by the current_conn_id parameter.
RequestCompleteTransfer	On behalf of the telephony object specified by the DN parameter, completes a previously initiated two-step transfer by merging the held call specified by the conn_id parameter with the active consultation call specified by the current_conn_id parameter. Assigns held_conn_id to the resulting call. Releases the telephony object specified by the DN parameter from both calls and clears the consultation call specified by the current_conn_id parameter.
RequestDeleteFromConference	A telephony object specified by DN deletes the telephony object specified by dn_to_drop from the conference call specified by conn_id. The client that invokes this service must be a party on the call in question.
RequestDeletePair	On behalf of the telephony object specified by the DN parameter, deletes the key-value pair specified by the key parameter from the user data attached to the call specified by the conn_id parameter.
RequestDeleteUserData	On behalf of the telephony object specified by the DN parameter, deletes all of the user data attached to the call specified by the conn_id parameter.
RequestHoldCall	On behalf of the telephony object specified by the DN parameter, places the call specified by the conn_id parameter on hold.
RequestInitiateConference	On behalf of the telephony object specified by the DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the

Switch Function	Description
	destination parameter with the purpose of a conference call.
RequestInitiateTransfer	On behalf of the telephony object specified by the DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the destination parameter for the purpose of a two-step transfer.
RequestListenDisconnect	On an existing conference call, sets Deaf mode for the party specified by the listener_dn parameter. For example, if two agents wish to consult privately, the subscriber may temporarily be placed in Deaf mode.
RequestListenReconnect	On an existing conference call, cancels Deaf mode for the party defined by the listener_dn parameter.
RequestMakeCall	Originates a regular call from the telephony object specified by the DN parameter to the called party specified by the Destination parameter.
RequestMakePredictiveCall	Makes a predictive call from the thisDN DN to the otherDN called party. A predictive call occurs before any agent-subscriber interaction is created. For example, if a fax machine answers the call, no agent connection occurs. The agent connection occurs only if there is an actual subscriber available on line.
RequestMergeCalls	On behalf of the telephony object specified by the DN parameter, merges the held call specified by the held_conn_id parameter with the active call specified by the current_conn_id parameter in a manner specified by the merge_type parameter. The resulting call will have the same conn_id as the held call.
RequestMuteTransfer	Initiates a transfer of the call specified by the conn_id parameter from the telephony object specified by the DN parameter to the party specified by the destination parameter; completes the transfer without waiting for the destination party to pick it up. Releases the telephony object specified by the DN parameter from the call.
RequestQueryCall	Requests the information specified by info_type about the telephony object specified by conn_id. If the query type is supported, the requested information will be returned in EventPartyInfo.
RequestReconnectCall	Releases the telephony object specified by the DN parameter from the active call specified by the current_conn_id parameter and retrieves the previously held call, specified by the held_conn_id parameter, to the same object. This function is commonly used to clear an active call and to return to a held call, or to cancel a consult call (due to

Switch Function	Description
	lack of an answer, because the device is busy, and so on) and then to return to a held call.
RequestRedirectCall	Requests that the call be redirected, without an answer, from the party specified by the DN parameter to the party specified by the dest_dn parameter.
RequestRegisterAddress	Registers for a DN. Your application must register the DN before sending the RequestAgentLogin.
RequestReleaseCall	Releases the telephony object specified by the DN parameter from the call specified by the conn_id parameter.
RequestRetrieveCall	Connects the held call specified by the conn_id parameter to the telephony object specified by the DN parameter.
RequestSendDtmf	On behalf of the telephony object specified by the DN parameter, sends the digits that are expected by an interactive voice response system.
RequestSetCallInfo	Changes the call attributes. Warning: Improper use of this function may result in unpredictable behavior on the part of the T-Server and the Genesys Framework. If you have any doubt on how to use it, please consult with Genesys.
RequestSetMessageWaitingOff	Sets the Message Waiting indication to off for the telephony object specified by the DN parameter.
RequestSetMessageWaitingOn	Sets the Message Waiting indication to on for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
RequestSingleStepConference	Adds a new party to an existing call and creates a conference.
RequestSingleStepTransfer	Transfers the call from a specified directory number DN that is currently engaged in the call specified by the conn_id parameter to a destination DN that is specified by the destination parameter.
RequestUnregisterAddress	Unregisters a DN.
RequestUpdateUserData	On behalf of the telephony object specified by the DN parameter, updates the user data that is attached to the call specified by the conn_id parameter with the data specified by the user_data parameter.