Platform SDK Developer's Guide

Creating Custom Protocols

4/11/2026

Creating Custom Protocols

Java

Overview

The External Service Protocol (ESP) was developed to simplify creation of custom protocols. It contains a minimal set of messages for exchanging information between a client and server. All messages contain a reference field to correlate the response with the request. The payload of messages is contained in key-value structures which are used as message properties. Because key-value collections can be used recursively, the total number of properties depends on the message. The custom protocol implements binary transport and obeys common rules for Genesys protocols.

Set of Messages

Message	Description
`Class Request3rdServer`	The `Request3rdServer` class holds requests for your custom server. This class extends the `Message` class and adds three additional fields: • `ReferenceId` - This integer type (32-bit) field is used to correlate this request with related events received as a server response. • `Request` - This `KeyValueCollection` type field is designed to contain a request for the server. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • `UserData` - This `KeyValueCollection` type fields is designed to have additional information related to the request. Most known protocols leave this field as is; custom protocols can use this field as desired.
`Class Event3rdServerResponse`	The `Event3rdServerResponse` class is used to send a response to clients. This class extends the `Message` class and adds three additional fields:

Message	Description
	• ReferenceId field - This integer type (32-bit) field is used to correlate this event with the related client request. • Request field - This KeyValueCollection type field is designed to contain the server response to a client request. Some ESP-based protocols such as UniversalContactServer protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • UserData field - This KeyValueCollection type fields is designed to have additional information related to the request. Most known protocols leave this field as is; custom protocols can use this field as desired.
Class Event3rdServerFault	The Event3rdServerFault class is sent to clients if the request cannot be processed for some reason. This class extends the Message class and adds two additional fields: • ReferenceId field - This integer type (32-bit) field is used to correlate this server response with the related client request. • Request field - This KeyValueCollection type field is designed to contain a reason why the error occurred. Some ESP-based protocols such as UniversalContactServer protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired.

Using ESP on the Client Side

Creating a Custom Protocol

To create the simplest ESP-based protocol, all you need to do is create a class inherited from the ExternalServiceProtocol class. However, this protocol only provides a way to send data. Your custom protocol still has to handle incoming and outgoing messages.

For example:

```
public class MessageProcessor {
    private static final String PROTOCOL_DESCRIPTION = "CustomESPProtocol";
```

```
    /**
     * Processes message which is has to be sent.
     * @param msg message to be sent.
     * @param clientSide flag indicates that the message is processing on a client side
     * @return message instance if message was processed successfully otherwise null.
     */
    static Message processSendingMessage(String msg, boolean clientSide)
    {
      Message outMsg = clientSide ? Request3rdServer.create() :
Event3rdServerResponse.create();
      KeyValueCollection request = new KeyValueCollection();
      request.addString("Protocol", PROTOCOL_DESCRIPTION);
      request.addString("Request", msg);

      if (outMsg instanceof Request3rdServer) {
          ((Request3rdServer)outMsg).setRequest(request);
      }
      else if (outMsg instanceof Event3rdServerResponse) {
          ((Event3rdServerResponse)outMsg).setRequest(request);
      }
      return outMsg;
    }

    /**
     * Handles incoming message.
     * @param message Incoming message.
     * @return Message instance if message was processed successfully otherwise null
     */
    static String processEvent(Message message)
    {
        KeyValueCollection request = null;
        if (message instanceof Event3rdServerResponse) {
            request = ((Event3rdServerResponse)message).getRequest();
        }
        else if (message instanceof Request3rdServer) {
            request = ((Request3rdServer)message).getRequest();
        }
        if (request == null) {
            return null;
        }
        String requestString = request.getString("Request");
        if (requestString == null) {
            return null;
        }
        String protocolDescr = request.getString("Protocol");
        return PROTOCOL_DESCRIPTION.equals(protocolDescr) ? requestString : null;
    }
}

public class EspExtension extends ExternalServiceProtocol {

    private static ILogger log = Log.getLogger(EspExtension.class);

    public EspExtension(Endpoint endpoint) {
        super(endpoint);
    }

    public String request(String message) throws ProtocolException
    {
      Message newMessage = MessageProcessor.processSendingMessage(message, true);
      if (newMessage != null) {
          return MessageProcessor.processEvent(request(newMessage));
      }
```

```
      if (log.isDebug()) {
          log.debugFormat("Cannot send message: '{0}'", message);
      }
      return null;
    }
}
```

Using ESP on the Server Side

Class ExternalServiceProtocolListener

The `ExternalServiceProtocolListener` class provides server-side functionality based on the Platform SDK `ServerChannel` class, and implements External Service Protocol. The simplest server side logic is shown in the following example:

```
public class EspServer {

    private static final ILogger log = Log.getLogger(EspServer.class);

    private final ExternalServiceProtocolListener listener;

    public EspServer(Endpoint settings)
    {
        listener = new ExternalServiceProtocolListener(settings);

        listener.setClientRequestHandler(new ClientRequestHandler() {

            @Override
            public void processRequest(RequestContext context) {
                try {
                    EspServer.this.processRequest(context);
                } catch (ProtocolException e) {
                    if (log.isError()) {
                        log.error("Message processing error:\n" +
context.getRequestMessage(), e);
                    }
                }
            }
        });
    }


    public ExternalServiceProtocolListener getServer() {
        return listener;
    }

    private void processRequest(RequestContext context) throws ProtocolException {
        //TODO: Return to client reversed source request

        Message requestMessage = context.getRequestMessage();
        if (requestMessage == null) {
            return;
        }
        if (log.isDebug()) {
            log.debugFormat("Request: {0}", requestMessage);
        }
        String msg = MessageProcessor.processEvent(requestMessage);
        if (msg != null)
```

```
        {
          String reversedMsg = new StringBuilder(msg).reverse().toString();
          Message outMsg = MessageProcessor.processSendingMessage( reversedMsg, false);
          if (outMsg instanceof Referenceable
                  && requestMessage instanceof Referenceable) {

((Referenceable)outMsg).updateReference(((Referenceable)requestMessage).retreiveReference());
          }
          if (log.isDebug()) {
              log.debugFormat("Request: {0}", requestMessage);
          }
          context.respond(outMsg); // or context.getClientChannel().send(outMsg);
        }
    }
}
```

Testing Your Protocol

A simple test example is shown below:

```
public class TestEsp {

    final String REQUEST = "Hello world!!!";

    @Test
    public void testMirrorSerializedMessage() throws ChannelNotClosedException,
ProtocolException, InterruptedException
    {
      String response = null;
      ExternalServiceProtocolListener server = new EspServer(new
WildcardEndpoint(0)).getServer();
      server.open();
      InetSocketAddress ep = server.getLocalEndPoint();
      if (ep != null) {
          EspExtension client = new EspExtension(new Endpoint("localhost", ep.getPort()));
          client.open();
        response = client.request(REQUEST);
        client.close();
      }
      server.close();

      System.out.println("Request: \n" + REQUEST);
      System.out.println("Response: \n" + response);
      Assert.assertNotNull(response);

      String expected = new StringBuilder(REQUEST).reverse().toString();
      Assert.assertEquals(response, expected);
  }
}
```

.NET

Overview

The External Service Protocol (ESP) was developed to simplify creation of custom protocols. It contains a minimal set of messages for exchanging information between a client and server. All messages contain a reference field to correlate the response with the request. The payload of messages is contained in key-value structures which are used as message properties. Because key-value collections can be used recursively, the total number of properties depends on the message. The custom protocol implements binary transport and obeys common rules for Genesys protocols.

Set of Messages

Message	Description
`Class Request3rdServer`	The `Request3rdServer` class serves for request the server. This class contains the base set of properties inherent to the `IMessage` interface and three additional fields. • `ReferenceId` field - This integer type (32-bit) field is used to correlate this request with related events received as a server response. • `Request` field - This `KeyValueCollection` type field is designed to contain a request to server. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • `UserData` field - This `KeyValueCollection` type field is designed to have additional information related to the request. Most known protocols leave it as is; custom protocols can use this field as desired.
`Class Event3rdServerResponse`	The `Event3rdServerResponse` class is used to send a response to clients. This class contains the base set of properties inherent to the `IMessage` interface and three additional fields: • `ReferenceId` field - This integer type (32-bit) field is used to correlate this event with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain a server response to a client request. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols

Message	Description
	can use this field as desired. • `UserData` field - This `KeyValueCollection` type field is designed to have additional information of request. Most known protocols leave it as is; custom protocols can use this field as desired.
`Class Event3rdServerFault`	The `Event3rdServerFault` class is sent to clients if the request cannot be processed for some reasons. This class contains the base set of properties inherent to the `IMessage` interface and two additional fields: • `ReferenceId` field - This integer type (32-bit) field is used to correlate this server response with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain a reason why the error occurred. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired.

Using ESP on the Client Side

Creating a Custom Protocol

To create the simplest ESP-based protocol, all you need to do is create a class inherited from the `ExternalServiceProtocol` class. However, this protocol only provides a way to send data. Your custom protocol still has to handle incoming and outgoing messages.

For example:

```
  internal static class MessageProcessor
  {
    private const string ProtocolDescriptionStr = "CustomESPProtocol";

    /// <summary>
    /// Processes message which is has to be sent.
    /// </summary>
    /// <param name="msg">message to be sent</param>
    /// <param name="clientSide">flag indicates that message is processing on client
side</param>
    /// <returns>IMessage instance if message was processed successfuly otherwise
null</returns>
    internal static IMessage ProcessSendingMessage(string msg, bool clientSide)
    {
      IMessage outMsg = clientSide ? Request3rdServer.Create() :
```

```
Event3rdServerResponse.Create() as IMessage;
      var request = new KeyValueCollection {{"Protocol", ProtocolDescriptionStr}, {"Request",
msg}};
      var req = outMsg as Request3rdServer;
      if (req != null) req.Request = request;
      var evt = outMsg as Event3rdServerResponse;
      if (evt != null) evt.Request = request;
      return outMsg;
    }
    /// <summary>
    /// Handles incoming message
    /// </summary>
    /// <param name="message">Incoming message</param>
    /// <returns>IMessage instance if message was processed successfuly otherwise
null</returns>
    internal static string ProcessEvent(IMessage message)
    {
      var response = message as Event3rdServerResponse;
      var req = message as Request3rdServer;
      KeyValueCollection request = null;
      if ((response != null) && (response.Request != null))
      {
        request = response.Request;
      }
      else
      {
        if ((req != null) && (req.Request != null))
          request = req.Request;
      }
      if (request == null) return null;
      var requestStr = request["Request"] as String;
      if (String.IsNullOrEmpty(requestStr)) return null;
      var protocolDescr = request["Protocol"] as string;
      return (ProtocolDescriptionStr.Equals(protocolDescr))?requestStr:null;
    }
  }

  public class EspExtension : ExternalServiceProtocol
  {

    public EspExtension(Endpoint endPoint) : base(endPoint) { }
    public string Request(string message)
    {
      var newMessage = MessageProcessor.ProcessSendingMessage(message, true);
      if (newMessage != null)
      {
        return MessageProcessor.ProcessEvent(Request(newMessage));
      }
      if ((Logger != null) && (Logger.IsDebugEnabled))
        Logger.DebugFormat("Cannot send message: '{0}'", message);
      return null;
    }
  }
```

Using ESP on the Server Side

Class ExternalServiceProtocolListener

The `ExternalServiceProtocolListener` class provides server-side functionality based on the Platform SDK `ServerChannel` class, and implements External Service Protocol. The simplest server side logic is shown in the following example:

```
  public class EspServer
  {
    private readonly ExternalServiceProtocolListener _listener;
    public EspServer(Endpoint settings)
    {
      _listener = new ExternalServiceProtocolListener(settings);
      _listener.Received += ListenerOnReceived;
    }
    public ExternalServiceProtocolListener Server { get { return _listener; } }

    private void ListenerOnReceived(object sender, EventArgs eventArgs)
    {
      //TODO: Return to client source request
      var duplexChannel = sender as DuplexChannel;
      var args = eventArgs as MessageEventArgs;
      if ((duplexChannel == null) || (args == null)) return;
      Console.WriteLine(args.Message);
      var msg = MessageProcessor.ProcessEvent(args.Message);
      if (msg != null)
      {
        var outMsg = MessageProcessor.ProcessSendingMessage(new
string(msg.Reverse().ToArray()), false);
        var source = args.Message as IReferenceable;
        var dest = outMsg as IReferenceable;
        if ((source!=null) && (dest!=null))
         dest.UpdateReference(source.RetrieveReference());
        Console.WriteLine(outMsg);
        duplexChannel.Send(outMsg);
      }
    }
  }
```

Testing Your Protocol

A simple test example is shown below:

```
  [TestClass]
  public class TestEsp
  {
    [TestMethod]
    public void TestMirrorSerializedMessage()
    {
      const string request = "Hello world!!!";
      string response = null;
      var server = new EspServer(new WildcardEndpoint(0)).Server;
      server.Open();
      var ep = server.LocalEndPoint as IPEndPoint;
      if (ep != null)
```

```
      {
        var client = new EspExtension(new Endpoint("localhost", (server.LocalEndPoint as
IPEndPoint).Port));
        client.Open();
        response = client.Request(request);
        client.Close();
      }
      server.Close();

      Console.WriteLine("Request: \n{0}", request);
      Console.WriteLine("Response: \n{0}", response);
      Assert.IsNotNull(response);
      Assert.IsTrue(request.Equals(new string(response.Reverse().ToArray())));    }
  }
```