# Avaya Open Interfaces

Communications Control Toolkit SDK

# SOA OI CCT REST Web Service Tutorial

## *Overview*

This tutorial describes how to create a simple client application for the Avaya CCT REST Open Interface Web services using a simple Web page written in HTML and Javascript. The Dojo[1] CometD[2] event-routing bus (using Ajax Push technology known as Comet) is required and is implemented in a set of Dojo Javascript libraries.

Any text editor can be used to create the Web page (e.g. Notepad++[3]). Any Web server capable of hosting simple Web pages can be used to deploy the tutorial (e.g. Tomcat[4], Jetty HTML container[5]).

A sample Jetty-based Web server hosting the tutorial Web page is included with this tutorial at \ServerForWebClient. To start this Web server, run the start.bat file in this folder:

The HTML Web page client created in this tutorial can be hosted on a Web server by copying to the HTML file to the \ServerForWebClient\Webapp folder. These pages can be browsed to by opening a Web browser at http://localhost:8080.

**The following OI CCT REST operations are used in this tutorial:**

- Login to CCT to get a CCT session – signified by a single-signon token (SSO token)
- Subscibe for events on a specific terminal. Notifications for these events will be sent to the Web page via the Ajax Push technology (CometD implementation)
- Make a call.
- Answer a call
- Drop a call

The CCT OI REST operations not included in the tutorial are coded in the reference application available at: \ServerForWebClient\webapp\CCT_REST_OI_RefClient_v0.07.html

[1] http://www.dojofoundation.org/
[2] http://cometd.org/
[3] http://notepad-plus-plus.org/
[4] http://tomcat.apache.org/
[5] http://jetty.codehaus.org/jetty/

## *Prerequisites for the Tutorial:*

- An Avaya Aura Contact Center has been installed and the SOA license for Open Interfaces has been acquired.
- A Web server to host the client Web page must be available with access to the Dojo Javascript libraries. These are available in the Webapp folder of
- At least two CCT users need to be configured.
- At least two Contact Center addresses and terminals need to be configured.
- Each terminal needs to be mapped to at least one address and each CCT user needs to be mapped to at least one terminal.
- The client Web page requires a browser supporting cross domain HTML requests. Microsoft Internet Explorer 8 (and later) and Mozilla Firefox 3.5 (and later) are suitable.
- To observe events being received by the consumers, CCT Ref client can be used to login and logout agents as well as going ready/not ready.

# Steps

## *Step 1: Create Empty Web Page with CometD Support*

To create the shell Web page:

1. Using a text editor create a new text file and save it as CCTRestOIClient.html
2. Add the following HTML and Javascript to create the empty HTML file

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:v="urn:schemas-microsoft-com:vml">

  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
    <title>AVAYA - CCT Open Interface REST API - Test Client v0.7</title>
  </head>

  <body>
  </body>

</html>
```

3. Add Javascript to the page header to initialise the CometD Ajax Push implementation. This requires the Dojo javascript libraries to be available for import. For convenience these are included in the folder \ServerForWebClient\Webapp\js. The snippet below also includes declaration of global variables used in the client.

```
...
  <head>
    ...
    <script type="text/javascript" src="js/dojo/dojo.js"
                djConfig="debug: true, parseOnLoad: true"></script>
    <script type="text/javascript" src="js/json2.js"></script>
    <script type="text/javascript">

      dojo.require("dojo.fx");
      dojo.require("dojo.parser");
      dojo.require("dojo.io.script");
      dojo.require("dojox.cometd");

      var _CCTServerIP = "11.222.33.44";
      var _CCTRESTServerPort = "9085";
      var _CCTRESTCometDPort = "9091";
      var _invocationHistoryText;
      var _baseRestApiUrl = "http://" + _CCTServerIP + ":" +
                            _CCTRESTServerPort;
      var _baseRestApiQueryPathElement = "?ssotoken=pass";
      var _currentContactId;
      var _lastCreatedContactId;
      var _lastSsoToken;
      var _lastConsultContactId;
      var _browserType;

    </script>

  </head>
...
```

4. Configure the CometD to be communicate with the remote CCT event server. The Javascript code below goes in the page header.

```
...
  <head>
    ...
    <script type="text/javascript">

      /* Setup dojo CometD
         This URL must match the cometd servlet URL on the Web server.
      */
      var _connected = false;
      var _cometdUrl = "http://" + _CCTServerIP + ":" +
                       _CCTRESTCometDPort + "/ccEvents";

      function CometdConfig(){
            this.url = _cometdUrl;
            this.logLevel = "debug";
      }

      /* Initiate a CometD session - callback function is _metaConnect()*/
      var myConfig = new CometdConfig;
      dojox.cometd.configure(myConfig);
      var _metaListener =
             dojox.cometd.addListener('/meta/connect', _metaConnect);
      dojox.cometd.handshake();

      function _metaConnect(message){
            var wasConnected = _connected;
            _connected = message.successful === true;
            if (!wasConnected && _connected) {
               // handle error condition
            } else if (wasConnected && !_connected) {
               // handle error condition
            }
      }
      ...
    </script>

  </head>
...
```

## *Step 2: Log in a CCT User and Retrieve an SSO token*

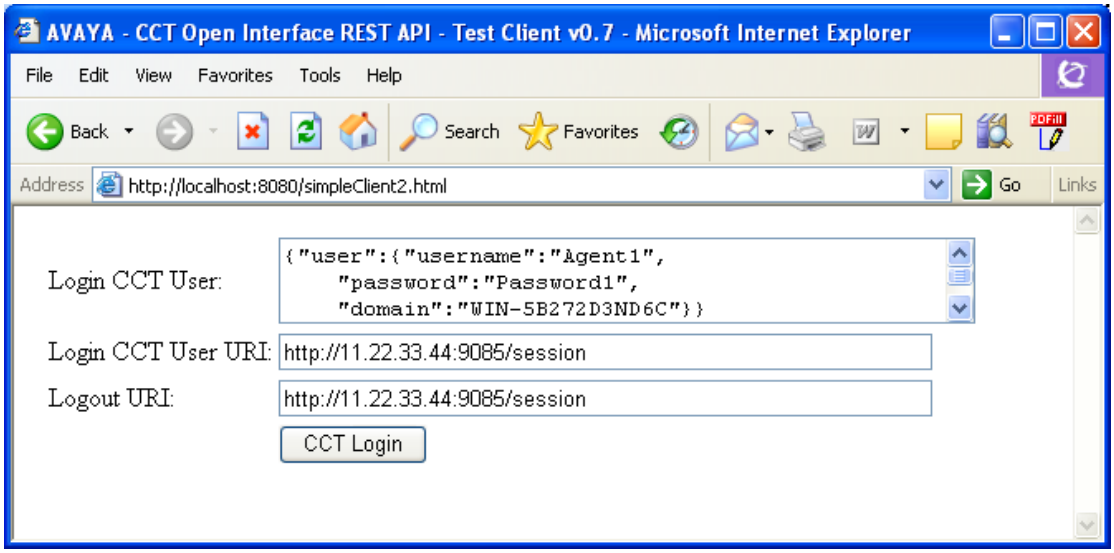
1. Internet Explorer requires the use of the XDomainRequest object to handle cross domain requests. Other browsers use the XMLHttpRequest object. Add code to determine the correct object to use for HTTP requests..

```
...
var invocation;
if (window.XDomainRequest){
    _browserType = 'ie8';
    invocation = new XDomainRequest();
} else {
    _browserType = 'generic';
    invocation = new XMLHttpRequest();
}
...
```

2. Add HTML elements for the CCT user login to the page body.

```
...
<table summary="">
   <tr>
   <td> </td>
   <td>Login CCT User:      </td>
   <td>
      <textarea name="result" rows="3"
                    cols="50" id="rawJsonCCTLoginId">
          {"user":{"username":"Agent1",
           "password":"Password1",
           "domain":"11.22.33.44"}}
      </textarea></td>
   <td></td>
   </tr>

   <td></td>
   <td>Login CCT User URI:</td>
   <td colspan="2">
   <input type="text" size="60" id="cctLoginUrlId"
          value="http://11.22.33.44:9085/session"/>   
   </td>
   </tr>

   <tr>
   <td></td>
   <td>Logout URI:</td>
   <td colspan="2">
   <input type="text" size="60" id="cctLogoutUrlId"
          value="http://11.22.33.44:9085/session"/>   
   </td>
   </tr>

   <tr>
   <td colspan="2"> </td>
   <td colspan="2">
   <input type="button" value="CCT Login"
          onClick="loginCctUser();return false"/>  
   </td>
   </tr>
 </table>
...
```

The above HTML displays in IE Explorer as:




3. Add helper code to (i) parse JSON and (ii) to handle errors. Parsing JSON to an object or array is carried out using JSON.parse( ) dojo method (included in json2.js). Error handling is trivially handled.

```
...
   /* Parse from JSON to an object */
     function jsonParse (jsonData) {
         return JSON.parse(jsonData, function(key, value){
               var type;
               if (value && typeof value === 'object') {
                     type = value.type;
                     if (typeof type === 'string' &&
                         typeof window[type] === 'function') {
                           return new (window[type])(value);
                     }
               }
               return value;
               });
         }

         function errorHandler() {
             alert('Error!');
...
```

4. A HTTP POST request is sent with the JSON data provided in the *Login CCT User* text box.

```
...
 /* Login a CCTUser via the REST API */
 function loginCctUser(){
   if(invocation)
   {
     var loginAgentUrl, loginAgentJSON;
     loginAgentUrl = document.getElementById('cctLoginUrlId').value;
     loginAgentJSON = document.getElementById('rawJsonCCTLoginId').value;

     if (_browserType != 'ie8') {
        invocation.open('POST', loginAgentUrl, true);
        invocation.setRequestHeader('Content-Type', 'application/json');
        invocation.onreadystatechange = loginAgentHandler;
```

```
      } else {
         invocation.onload = loginAgentHandler;
         invocation.onerror = errorHandler;
         invocation.open('POST', loginAgentUrl);
      }
      invocation.send(loginAgentJSON);

    } else {
        alert('No HTTP request object available');
    }
  }
...
```

The Internet Explorer XDomainRequest object and the XMLHttpReqeuest object supported by Firefox (and others) have different object models. However, in both cases, the URL for the HTTP request and the HTTP verb – POST – are set. A call to invocation.send() sends the request. The loginCctUserHandler ( ) function is designated to handle the HTTP response. The JSON data to be sent with the request is read from the *Login CCT User* text box.

5. A designated Javascript function handles the HTTP response and extracts the SSO token if login is successful.

```
function loginCctUserHandler(evtXHR)
{
   // Get the response
   var response;
   if (_browserType == 'ie8') {
     response = invocation.responseText;
   } else {
     if (invocation.readyState == 4) {
       if (invocation.status == 200) {
         response = invocation.responseText;
       } else {
         alert('Server error: login CCT User');
         return;
       }
     } else {
        alert('Server error: login CCT User');
        return;
     }
   }

   // Parse the response
   if (response == '') {
     alert('Error: No ssotoken returned to indicate successful login');
     return;
   } else {
     // Sample response JSON: {"contact":[{"contactId":"theContactId"}]}
     ssotokenJSON = jsonParse (response);
     if (ssotokenJSON.user) {
       if(ssotokenJSON.user.ssoTokenValue) {
         _lastSsoToken = ssotokenJSON.user.ssoTokenValue;
         if (_browserType == 'ie8') {
           document.getElementById('cctLogoutUrlId').innerText =
                'http://' + _CCTServerIP + ':' +
                _CCTRESTServerPort + '/session/' + _lastSsoToken;
         } else {
           document.getElementById('cctLogoutUrlId').value =
                'http://' + _CCTServerIP + ':' +
                _CCTRESTServerPort + '/session/' + _lastSsoToken;
         }
       }
     }
   }
}
```

The loginCctUserHandler function is called back when the response to the HTTP request is received by the Web page. The first part of the function checks that a HTTP 200 response code is received. The second part parses the JSON data string returned in the response. In this case it contains an SSO token. The JSON is parsed to extract the value of the SSOToken. As subsequent URIs used to make, answer and drop calls require the SSOToken to be set as a query parameter, these user interface elements are updated now (browser-dependent code).

## *Step 3 : Subscribe for Events*

1. Add the HTML user interface elements: text boxes with default values for provider and terminal name; a text area to display received evens; and a button to subscribe. Note: The text element - *updateCallUrlId* - will be used later in the tutorial for dropping an active call.

```
<tr>
   <td> </td>
   <td>Provider Name:      </td>
   <td><input type="text" size="30" id="providerNameID"
              value="Passive"/></td>
   <td></td>
</tr>

<tr>
   <td> </td>
   <td>Terminal Name:      </td>
   <td><input type="text" size="30" id="origTermNameId"
              value="Line 3.0.0.0"/>   
       <input type="text" style="color: red;" size="10"
              id="subscribedStateId" value=""/>
   </td> <td></td>
</tr>

<tr>
   <td colspan="2"> </td>
   <td colspan="2">
     <input type="button" value="Subscribe"
            onClick="subscribeForEvents();return false"/>  
   </td>
</tr>

<tr>
   <td></td>  <td></td>
   <td colspan="2">
     <input type="text" size="60" id="subscribeUrlId"
            value="http://47.166.94.81:9085/subscriptions?ssotoken=pass"/>
               
   </td>
</tr>

<tr>
   <td> </td>
   <td>Terminal State:  </td>
   <td><input type="text" size="15" id="callStatusID" value=""/></td>
   <td></td>
</tr>

<tr>
  <td></td> <td></td>
  <td colspan="2">
  <input type="text" size="60" id="updateCallUrlId"
       value="http://47.166.94.81:9085/contacts/contactId?ssotoken=pass"/>
          
  </td>
</tr>

<tr>
   <td colspan="4"> </td>
```

```
</tr>

<tr>
   <td> </td>
   <td>Raw JSON events:  </td>
   <td>
      <textarea name="result" rows="4" cols="50"
                id="rawJsonEventsId"></textarea>
   </td>
   <td></td>
</tr>
```

2. Add the following new lines to the loginCctUserHandler() function so that the URL used to subscribe for events has the correct SSO token in its query string after CCT user login.

```
function loginCctUserHandler(evtXHR){
...
    [Existing lines]
    if (_browserType == 'ie8') {
       document.getElementById('cctLogoutUrlId').innerText =
       'http://' + _CCTServerIP + ':' +
       _CCTRESTServerPort + '/session/' + _lastSsoToken;
    [new line]
        document.getElementById('subscribeUrlId').innerText =
        'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
        '/subscriptions?ssotoken=' + _lastSsoToken;
    [Existing line]
    } else {
       document.getElementById('cctLogoutUrlId').value =
       'http://' + _CCTServerIP + ':' +
       _CCTRESTServerPort + '/session/' + _lastSsoToken;
    [new line]
       document.getElementById('subscribeUrlId').value =
       'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
       '/subscriptions?ssotoken=' + _lastSsoToken;
    }
...
}
```

3. Add the Javascript functions to subscribe for events occurring against a specific terminal. Examples of events are ACTIVE, RINGING and HANGUP.

```
function subscribeForEvents(){
  var terminalName = document.getElementById('origTermNameId').value;
  var providerName = document.getElementById('providerNameID').value;

  // Note: valid channel names must start with a '/'
  var channel = '/terminal/' + providerName + '::' + terminalName;
  _cometDSubscription = dojox.cometd.subscribe(channel,
                            terminalEventsNotificationtionHandler);

  sendSubscriptionRequest(terminalName, providerName, channel);
}
```

4. CometD requires that a ‘channel’ be created between the Web page and the server providing the events. The Web page is responsible for naming this channel and sending it to the server. Note: valid channel names must start with a forward slash.

5. The call to the CometD subscribe function must supply a callback function where CometD events sent to the Web page will be processed. This function is called: terminalEventsNotificationtionHandler. It is shown below. In this example, the full JSON of the event is displayed in a test area. The event type is used to update the callStatus text box.

```
function terminalEventsNotificationtionHandler(msg){
  var eventType, contactID, providerName,terminalName,
      calledAddressName, callingAddressName;
  var data = msg.data;

  if (data) {
    // Write the JSON incoming event to a text area
    document.getElementById('rawJsonEventsId').value = data;
    var myData = jsonParse (data);
    if (myData.event) {
      _currentContactId = (myData.event.params.contactID) ?
                          myData.event.params.contactID : null;

      if (_lastConsultContactId != _currentContactId) {

        providerName = (myData.event.params.providerName) ?
                       myData.event.params.providerName : null;
        terminalName = (myData.event.params.terminalName) ?
                       myData.event.params.terminalName : null;
        calledAddressName = (myData.event.params.calledAddressName) ?
                            myData.event.params.calledAddressName : null;
        callingAddressName = (myData.event.params.callingAddressName) ?
                             myData.event.params.callingAddressName : null;

        eventType = (myData.event.type) ? myData.event.type : null;

        if (eventType == 'CALL_ACTIVE')
          document.getElementById('callStatusID').value = 'ACTIVE';
        if (eventType == 'CALL_IDLE')
          document.getElementById('callStatusID').value = 'IDLE';
        if (eventType == 'CALL_RINGING')
          document.getElementById('callStatusID').value = 'RINGING';
        if (eventType == 'CALL_ANSWERED')
          document.getElementById('callStatusID').value = 'ANSWERED';
        if (eventType == 'CALL_HANGUP')
          document.getElementById('callStatusID').value = 'HANGUP';
      } else {
        // NOTE:
        // Not displaying events for supervised transfer consult
        // calls on the originating client for now
      }
    }
  }
}
```

6. The sendSubscriptionRequest function is shown below:

```
function sendSubscriptionRequest(terminalName, providerName, channel) {
  if(invocation)
  {
    var makeSubscriptionUrl;
    makeSubscriptionUrl=document.getElementById('subscribeUrlId').value;

    var jsonSubscription=createJSONForSubscription(terminalName,
                                                    providerName, channel);

    if(jsonSubscription != "" && makeSubscriptionUrl != "") {

      if (_browserType == 'ie8') {
         invocation.onload = createSubscriptionHandler;
         invocation.onerror = errorHandler;
         invocation.open('POST', makeSubscriptionUrl);
      } else {
         invocation.open('POST', makeSubscriptionUrl, true);
         invocation.setRequestHeader('Content-Type', 'application/json');
         invocation.onreadystatechange = createSubscriptionHandler;
      }
      invocation.send(jsonSubscription);
    } else {
      if(jsonSubscription == "")
         alert('The JSON payload to create a
                susbcription is an empty string.');
    }
   } else {
      _invocationHistoryText = "No invocation of the
                                subscription URL took place.";
   }
}
```

7. The JSON to be sent with HTTP request is created in the function createJSONForSubscription, shown below. The subscription is for events on the terminal identified by the provider name and terminal name provided in the HTML edit boxes on the Web page.

```
function createJSONForSubscription(terminalName, providerName, channel) {

  susbcriptionDetails = new Object;
  susbcriptionDetails.type = "terminal";

  // This creates an array with one element
  // var a = []; creates an empty extensible array
  var entityNames = [terminalName];
  susbcriptionDetails.entityNames = entityNames;

  var subscription = new Object;

  // Use either the bayeux channel name or the REST endpoint URI.
  subscription.channelName = channel;
  // subscription.eventEndpointUri = "http://' + _CCTServerIP + ':'
              + _CCTRESTServerPort + '/eventEndpoint";
  subscription.providerName = providerName;
  subscription.subscriptionDetails = susbcriptionDetails;

  subscriptionRequest = new Object;
  var subscriptions = [subscription];
  subscriptionRequest.subscription = subscriptions;

  return JSON.stringify(subscriptionRequest);
}
```

8. The callback handler for the response to the HTTP subscription request is shown below. If a HTTP 200 status response is received, the subscription has been successful and a text box is updated with a red *SUBSCRIBED.*

```
function createSubscriptionHandler(evtXHR)
{
  if (_browserType == 'ie8') {
    document.getElementById('subscribedStateId').value = 'Subscribed';
  } else {
    if (invocation.readyState == 4) {
      if (invocation.status == 200) {
        document.getElementById('subscribedStateId').value = 'Subscribed';
      } else {
         alert("Create subscription Occured " + invocation.readyState +
               " and the status is " + invocation.status);
      }
    } else {
      dump("currently the application is at" + invocation.readyState);
    }
  }
}
```

## *Step 4: Make a Call and Observe Incoming Events*

1. **NOTE:** This section describes how to make a call using the REST API. In the following step, drop-call functionality will be added. Until then each time a call is made using the REST API, it will need to be released using another mechanism e.g. the CCT Ref Client deployed with each CCT Server. Calls created using the REST API can be observed on the CCT Ref Client.
2. Making a call follows the same pattern as subscribing for an event. A JSON string with the data required for the call is created. A HTTP POST message is sent to the CCT server. If the HTTP response has status of 200, the response contains the ID of the new call. At the same time an event is sent from the CCT server via the CometD mechanism to indicate that the call is ACTIVE.
3. The HTML required to call the Make Call REST operation is shown below.

```
<tr>
  <td> </td>
  <td>Calling Address:  </td>
  <td><input type="text" size="60" id="origAddressId" value="2000"/></td>
  <td></td>
</tr>
<tr>
  <td> </td>
  <td>Called Address:  </td>
  <td><input type="text" size="60" id="destAddressId" value="2001"/></td>
  <td></td>
</tr>

<tr>
  <td colspan="2"></td>
  <td colspan="2">
  <input type="button" value="Make Call"
         onClick="createContact();return false"/>
            
  <input type="text" style="color: red;" size="40"
         id="createContactJsonResponseId" value=""/>
  </td>
</tr>

<tr>
  <td></td>  <td></td>
  <td colspan="2">
  <input type="text" size="60" id="makeCallUrlId"
         value="http://47.166.94.81:9085/contacts?ssotoken=pass"/>
     
  </td>
</tr>
```

4. A small change is required to the loginCctUserHandler function to update the default text of the makeCallUri text box with a valid sso token after a successful CCT login.

```
function loginCctUserHandler(evtXHR){
...
    [Existing lines]
    if (_browserType == 'ie8') {
        ...
        document.getElementById('subscribeUrlId').innerText =
        'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
        '/subscriptions?ssotoken=' + _lastSsoToken;
    [new lines]
        document.getElementById('makeCallUrlId').innerText =
        'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
        '/contacts?ssotoken=' + _lastSsoToken;
    }
    [Existing lines]
    } else {
       ...
       document.getElementById('subscribeUrlId').value =
       'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
       '/subscriptions?ssotoken=' + _lastSsoToken;
    [new line]
        document.getElementById('makeCallUrlId').value =
        'http://' + _CCTServerIP + ':' + _CCTRESTServerPort +
        '/contacts?ssotoken=' + _lastSsoToken;
    }
...
}
```

5. The Javascript function to send the HTTP POST message to make a call is shown below:

```
/* Make a REST Call to create a contact */
function createContact(){
  if(invocation) {
    var makeCallUrl;
    makeCallUrl = document.getElementById('makeCallUrlId').value;

    if(makeCallUrl != "") {
      if (_browserType != 'ie8') {
        invocation.open('POST', makeCallUrl, true);
        invocation.setRequestHeader('Content-Type', 'application/json');
        invocation.onreadystatechange = createContactHandler;
      } else {
        invocation.onload = createContactHandler;
        invocation.onerror = errorHandler;
        invocation.open('POST', makeCallUrl);
      }
      invocation.send(createJSONForCreateContact());
    } else {
      alert('The create call URL is blank');
    }
  } else {
    _invocationHistoryText = "No Invocation took place at all";
  }
}
```

6. The Javascript function to create the JSON for the createContact HTTP POST request is provided below.

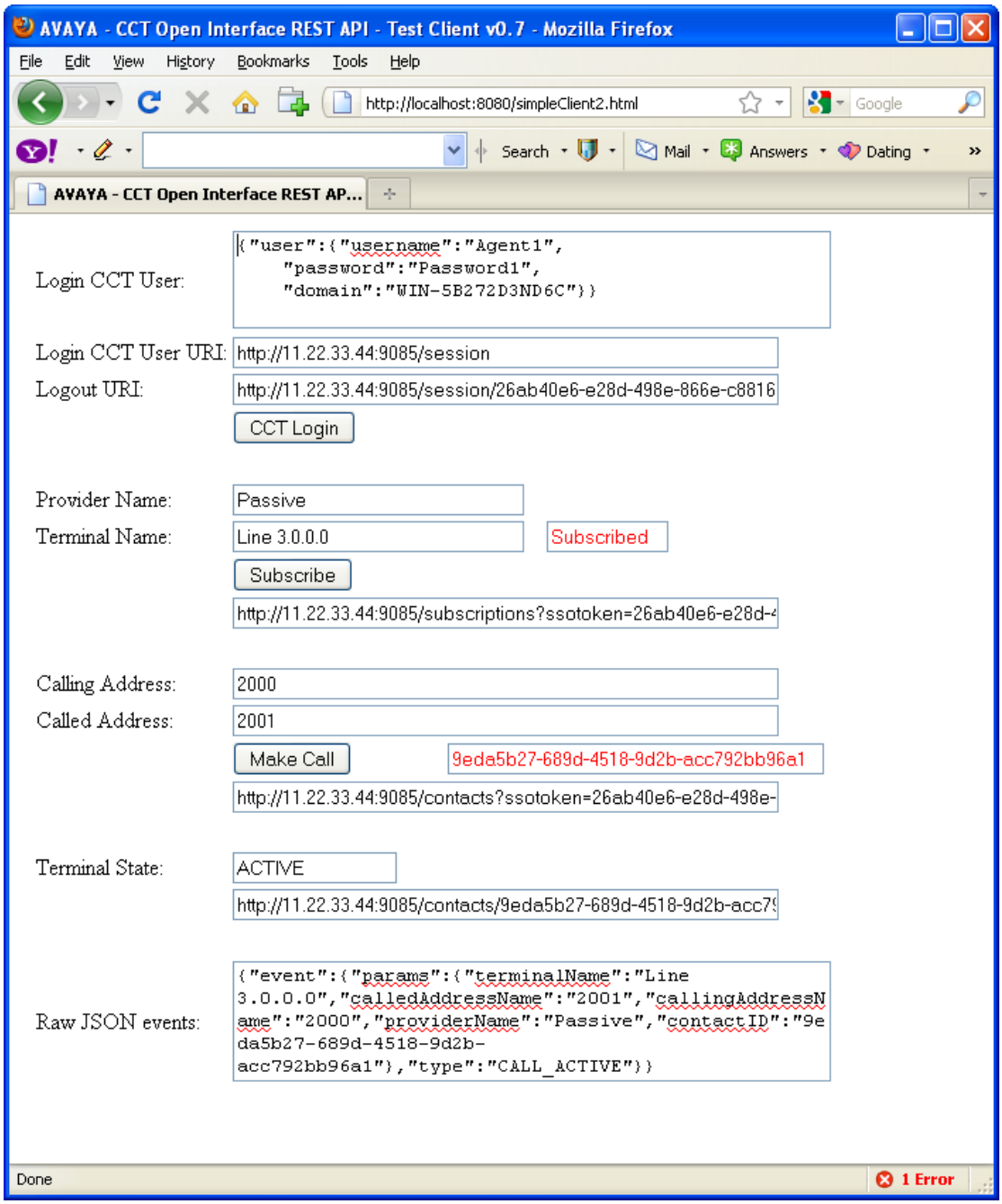
```
function createJSONForCreateContact() {
  var contact = new Object;
  contact.origAddressName =
      document.getElementById('origAddressId').value;
  var destAddressNames =
      [document.getElementById('destAddressId').value];
  contact.destAddressNames = destAddressNames;
  contact.providerName =
      document.getElementById('providerNameID').value;
  contact.origTerminalName =
      document.getElementById('origTermNameId').value;

  createContactRequest = new Object;
  createContactRequest.contact = contact;
  var jsonResult = JSON.stringify(createContactRequest);
  return jsonResult;
}
```

7. The Javascript function to handle the HTTP response for the create-contact HTTP request is provided below.

```
 function createContactHandler(evtXHR){
   var response;
   if (_browserType == 'ie8') {
     response = invocation.responseText;
   } else {
     if (invocation.readyState == 4) {
       if (invocation.status == 200) {
         response = invocation.responseText;
       } else {
         alert("Make Call error. Ready state: " +
         invocation.readyState + "; HTTP status: " + invocation.status);
         return;
       }
     }
   }

   if (response != '') {
     createContactResponseJSON = jsonParse (response);
     if (createContactResponseJSON.contact) {
       if(createContactResponseJSON.contact.contactId) {
          _lastCreatedContactId =
                       createContactResponseJSON.contact.contactId;
          if (_browserType == 'ie8') {
             document.getElementById('createContactJsonResponseId').
innerText = _lastCreatedContactId;

             document.getElementById('updateCallUrlId').innerText =
                  'http://' + _CCTServerIP + ':' +
                  _CCTRESTServerPort + '/contacts/' +
                 _lastCreatedContactId  + '?ssotoken=' + _lastSsoToken;
           } else {
             document.getElementById('createContactJsonResponseId').value
= _lastCreatedContactId;

             document.getElementById('updateCallUrlId').value =
                   'http://' + _CCTServerIP + ':' +
                   _CCTRESTServerPort + '/contacts/' +
                   _lastCreatedContactId  + '?ssotoken=' + _lastSsoToken;
         }
       }
       document.getElementById('createContactJsonResponseId').value =
                                              _lastCreatedContactId;
     }
   } else {
     alert('Error: Empty response. Expected contactID');
   }
 }
```

8. The HTTP response to the create-contact request is expected to include a JSON string that contains the ID of the new contact. If a conact ID is returned, it is used (i) to update the URI used for subsequent updates to the call state and (ii) to update the contents of an edit box.
9. After a CCT user login, then subscribe has been performed, the Make Call button is clicked. The text box to the right of the button should be updated with the unique ID of the new contact. The text area labelled *Raw JSON Events* should be updated with an

event from the CCT server indicating that the call has become active. This should also be reflected in the contect of the Terminal State text box, whose value should be *ACTIVE*. A sceenshot of the Web page with an active call is shown below.



## *Step 5: Drop an Active Call*

1. Add the following HTML under the HTML for the Terminal State text box.
   (In the example the Notification Producer uses the Consmer2Impl.java)

```
<tr>
  <td colspan="2"> </td>
  <td colspan="2">
  <input type="button" value="Drop"
         onClick="dropContactLeg();return false"/>  
</td>
</tr>
```

2. Add the following Javascript for to create the drop call HTTP request.

```
function dropContactLeg(){
  if(invocation)
  {
   if (_currentContactId) {
     var dropContactLegUrl;
     dropContactLegUrl = document.getElementById('updateCallUrlId').value;
     console.log('drop contact URL = : ' + dropContactLegUrl);


     if (_browserType != 'ie8') {
       invocation.open('POST', dropContactLegUrl, true);
        invocation.setRequestHeader('Content-Type', 'application/json');
        invocation.onreadystatechange = dropContactLegHandler;
     } else {
        invocation.onload = dropContactLegHandler;
        invocation.onerror = errorHandler;
        invocation.open('POST', dropContactLegUrl);
     }
     invocation.send(createJSONForDropContactLeg());
   } else {
     alert('There is no active call to drop.');
   }
  } else {
    _invocationHistoryText = "No Invocation took place";
  }
  console.log('leave: dropContactLeg');
}
```

3. Add the following Javascript to create the JSON for the drop call request.

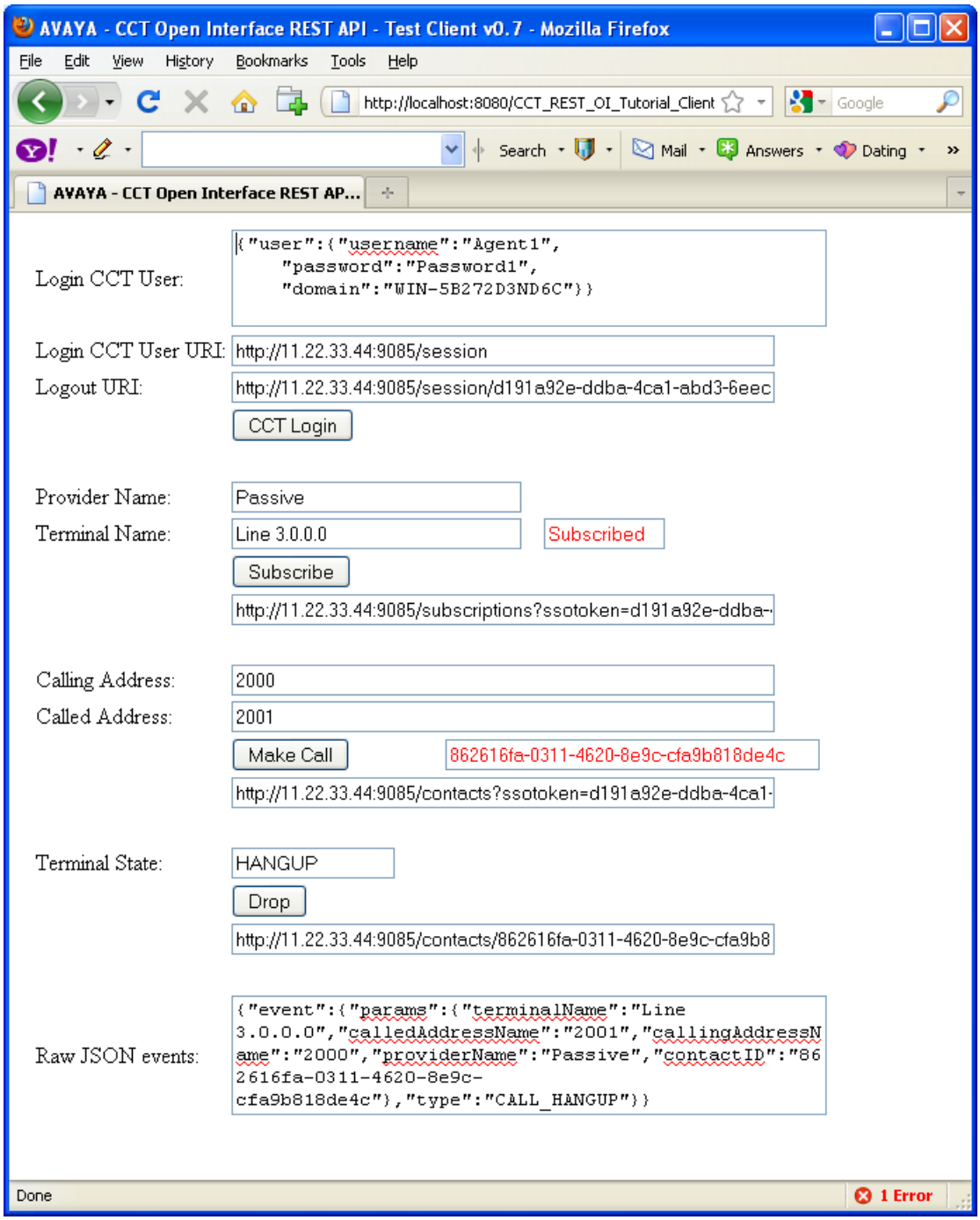
```
function createJSONForDropContactLeg() {
  var contact = new Object;
  contact.mode = 'drop';
  contact.providerName = document.getElementById('providerNameID').value;
  contact.origTerminalName =
                        document.getElementById('origTermNameId').value;
```

```
  var dropContactLegRequest = new Object;
  dropContactLegRequest.contact = contact;

  var jsonResult = JSON.stringify(dropContactLegRequest);
  return jsonResult;
}
```

4. Add the following Javascript to handle the HTTP response for the drop call request.

```
function dropContactLegHandler(evtXHR)
{
  var response;
  if (_browserType == 'ie8') {
    response = invocation.responseText;
  } else {
    if (invocation.readyState == 4) {
      if (invocation.status == 200) {
        response = invocation.responseText;
      } else {
        console.log('Server error: Drop contact leg');
        alert('Server error: Drop contact leg');
        return;
      }
    } else {
      // Error handling
    }
  }
}
```

A screenshot of the Web page after the Drop button was clicked is shown below. Note that the event JSON shows that a CALL_HANGUP event was received. This is also reflected in the value of the Terminal State edit box – HANGUP.




Note that it is useful to open two instances of the client Web page side by side. On one, subscribe for events against one terminal (and associated address). On the other subscribe for events for the second terminal. This way calls can be made on one Web client and answered on another.

**Complete Source for the Tutorial**

The complete source for the Web Client created in the steps above is available at \ServerForWebClient\Webapp\CCT_REST_OI_Tutorial_Client.html

## Adding Answer Call, Agent Login, Blind Transfer and Supervised Transfer

A sample reference client with this functionality encoded is available at: ServerForWebClient\webapp\CCT_REST_OI_RefClient_v0.07.html

The approach to of Web service invocation and the handling of responses for this functionality follows the same pattern described for the operations included in this tutorial.