Oracle® On Track Communication

Developer's Guide

Release 1 (1.0)

E20959-01

March 2011

This guide explains how to build and deploy a wide range of integrations with Oracle On Track Communication. Developers can simply embed portions of On Track in their applications, or deliver full-featured integrations using the On Track APIs. This guide explains the fundamental architecture of the server, and provides guidance on developing a range of solutions from server-side Java to client-side AJAX-based solutions.

This guide covers the following topics:

Audience

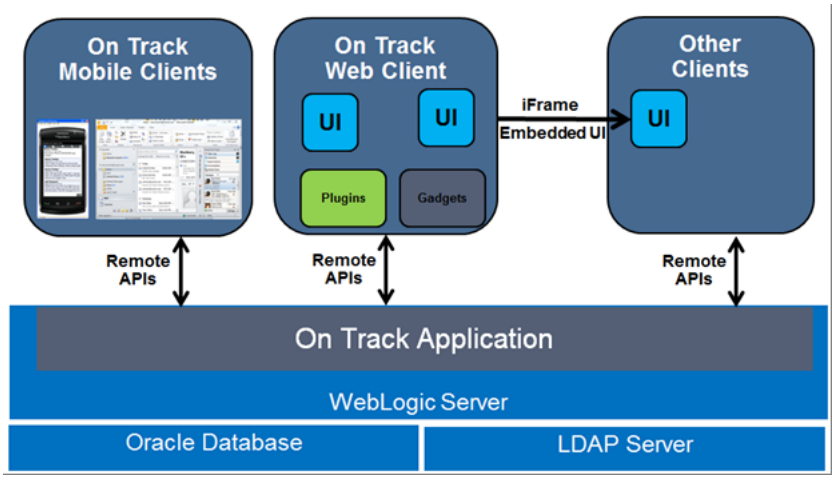
This document is intended for On Track developers who use a variety of programming languages and development environments to create valuable integrations with On Track.

Background

On Track provides open protocols and rich interfaces to expose integration capabilities across a wide range of development platforms, languages, and runtime environments. Fundamentally, On Track is a mid-tier application which runs on WebLogic Server, stores its data in Oracle Database, and can connect to LDAP for users' data and authentication.

On Track Architecture

Figure 1 On Track Architecture



As depicted in Figure 1, the On Track Application is a standard Java Application deployed on WebLogic Server.

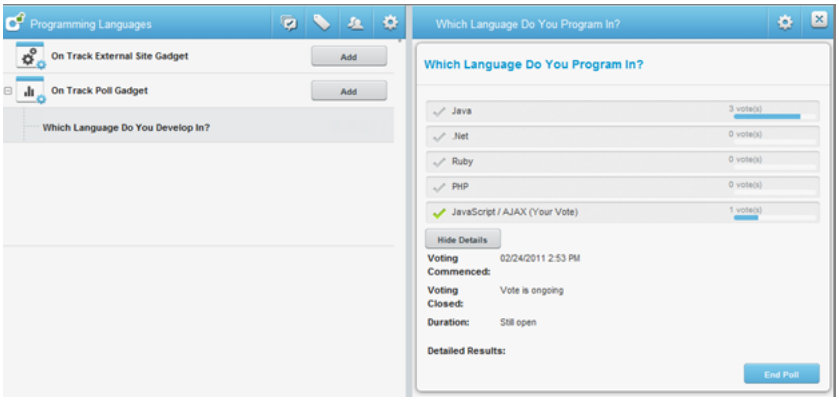
There are three principle types of clients for On Track:

- Web client: The most commonly used interface for the product, accessible through Google Chrome, Mozilla Firefox 3.6+, Apple Safari 4+, and Microsoft Internet Explorer 8+ browsers.. The Web client may embed custom developed gadgets written by developers.
- Desktop and Mobile clients: Intended to be used as a plug-in for Microsoft Outlook, or standalone iPad and iPhone applications.
- External clients: Complete new, custom-built clients that can be created to utilize the On Track application.

All three clients communicate with the On Track server using remote API invocations over HTTP. The Web client itself supports deploying and running custom developed gadgets within the On Track server as part of the gadgets tab of the Conversation pane

Additionally, the On Track Web client contains UI components, the Conversation list and Conversation pane, which can run in a standalone window and be integrated in external applications.

Figure 2 Poll Gadget

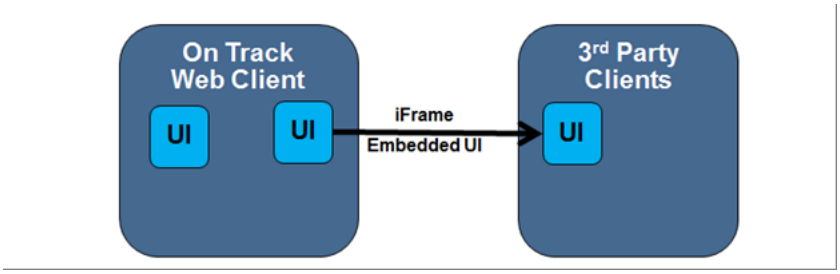


Principles Of On Track Architecture

There are several ways to integrate other applications with On Track that are covered in this guide including UI-based integrations, and programmatic integrations using Java and JavaScript. This guide will step through these options starting with the simplest and most common and progressing through more advanced scenarios. These integrations are often mixed together as part of a complete solution between On Track and other applications.

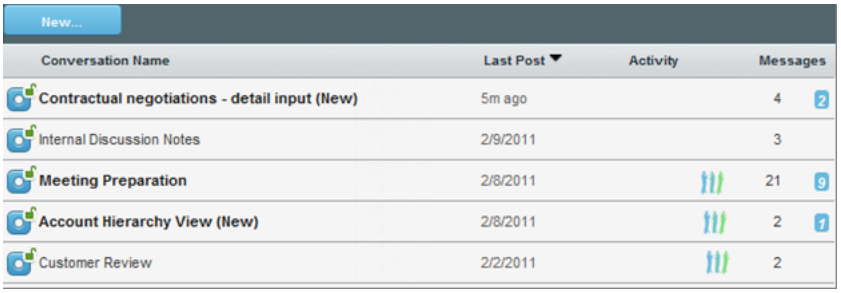
UI-Based Integrations

Figure 3 UI-Based Integration Architecture



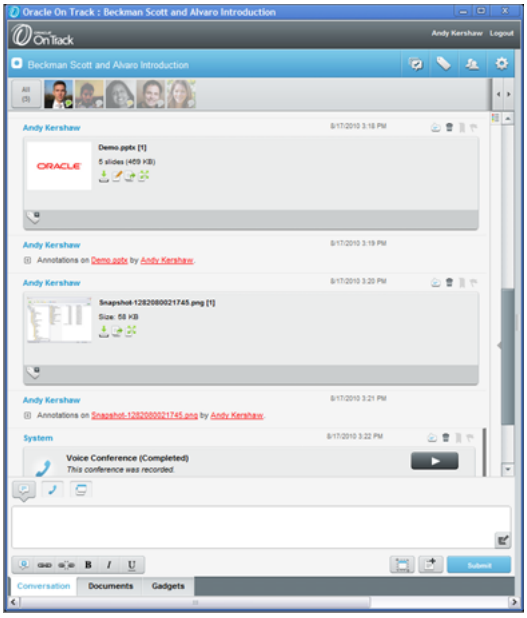
The most common method of integrating On Track in another application is to embed parts of the On Track Web client user interface in those applications. There are two principle elements of the Web client which are designed to be embedded: the Conversation list and the Conversation pane.

Figure 4 Conversation List



The Conversation list UI can be embedded in other applications, typically through an iFrame. This UI provides a list of conversations that the user is a member of. In addition, that list of conversations can be scoped to only show a subset of conversations as needed. The Conversation list itself provides real-time indicators of user activity within the conversation. Clicking on any conversation in the list opens up the conversation pane in a new pop-up window.

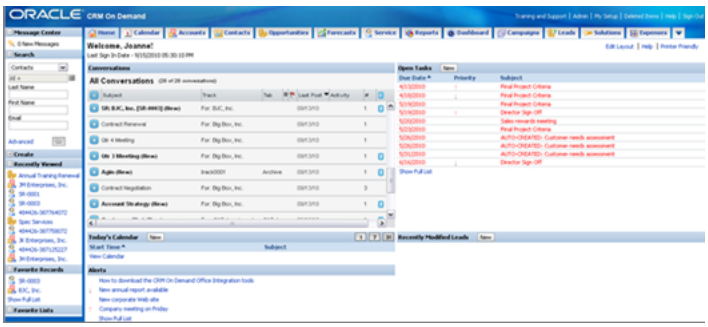
Figure 5 Conversation Pane



The Conversation pane is the primary UI component for working within On Track. When launched as a standalone window, it retains all the capabilities for working with the conversation that you have when using the full Web client.

One example of embedding a Conversation list is show in below in Figure 6, where Oracle's CRM On Demand product has been integrated with a list of On Track conversations in the page of the application.

Figure 6 Sample Embedded Conversation List



To enable the Conversation List and Conversation Pane UI to be embedded as iFrames in other applications, it is necessary for the On Track server to enable frame embedding. See *Oracle On Track Administrator's Guide* for more information. Without this setting enabled, when these components put into an iFrame the full browser window will be refreshed with the UI component, removing the iFrame and containing site.

Embedding the Conversation List

The base URL for a Conversation list is:

```
http://hostname/context/web/ConversationList.jsp
```

To create an iFrame in a containing application use the following iFrame code:

```
<iframe src="http://hostname/context/web/ConversationList.jsp" />
```

There are a few principles behind the conversations displayed in the list including:

- The returned list of conversations is limited to the conversations viewable by the authenticated user.
- If a system to provide Single Sign-On (SSO) has not been established between On Track and the containing Web application, a form-based login will be provided within the embedded UI component.

Customizing the Conversation List UI

The Conversation list can be invoked with a number of optional URL parameters which will modify the appearance of the conversation list. These parameters include:

`columns = STATE, NAME, ACTIVITY, MSG_COUNT, UNREAD_MSG_COUNT, FOLLOWUP, LASTPOST`
This is an optional parameter. If specified, the columns indicated will appear. The valid options must be separated by commas. NAME and STATE are required properties.

`sortColumn= NAME, ACTIVITY, MSG_COUNT, UNREAD_MSG_COUNT, FOLLOWUP, LASTPOST`
This is an optional parameter and can be set to one of the COLUMN values to sort the list by that value. The valid options must be separated by commas.

`sortType=ascending/descending`
This is an optional parameter that can be set when sortColumn is used to indicate the default sort order of the Conversation list.

`hideHeader`

This is used to hide the header. If omitted, the header bar will appear. 'hideHeader' is only applicable to the ConversationList when trackID is specified or secondaryID is specified (see Conversation Scoping).

Conversation Scoping

When integrating On Track with other applications, it is common to only use inline conversations that are related to a particular object that exists in the containing application. For example:

- 1-n Sales Opportunities can be related to 1-n Conversations
- 1-n Orders can be related to 1-n Conversations
- 1-n Team Sites can be related to 1-n Conversations

To enable scoping of the Conversation list to only show related conversations to an object, there are two general approaches for maintaining the reference between the related object and the conversation:

- Leverage an external system's mechanism for storing a reference to the associated On Track conversations by storing the associated conversationIDs or parent trackIDs in that external system as the relationship.
- Leverage On Track to store the references - conversations have a property on them called a secondaryID which can contain a set of references to business objects.

When using an external system to store the conversationIDs or parent trackIDs that are related to an external object, there are several parameters used to scope the Conversation list:

`conversationIDs=CONVERSATIONID,CONVERSATIONID`
This optional parameter allows one to specify a set of conversationIDs that will be displayed. A unique conversation ID exists for every conversation and can be determined programmatically through the On Track API, or is visible in the URL of any standalone conversation pane window. The list of conversationIDs must be separated by commas.

`conversationIDs=CONVERSATIONID,CONVERSATIONID`
This optional parameter allows one to specify a trackID which will in turn display conversations that are a part of that track (that the user has access to see). A unique track ID exists for every conversation and can be determined programmatically through the On Track API. If trackID is specified on the URL, the following parameters must NOT exist on the URL (conversationIDs, secondaryID, and trackName)

Advanced Conversation Integrations

When On Track stores the references between conversations and external systems, there are several parameters used to scope the Conversation list. However, it should be noted that this scenario requires the programmatic development and deployment of On Track agents to listen to events. This functionality is covered in the Advanced Topic - Using Conversation List Back Channel Events section of this document. The parameters are:

`trackName=TRACKNAME`
Required parameter which is used to provide the name of the Track that will be created for new conversations that do not have a track association.

`secondaryID=SECONDARYID`

An optional parameter that, if specified, will display conversations associated with a secondary ID . The trackName parameter must be provided along with integratorName. The integration agent is responsible for creating the new conversation.

`integratorName=INTEGRATORNAME`
An optional parameter that is used for call-backs and specifies the name of the Integration user that is listening to events. It is required when specifying a secondaryID.

Standalone Conversation Pane

It is possible to directly create an On Track conversation pane as a standalone window using the following URL format:

```
http://hostname/ontrackinstance/web/?conversation=conversationID&window=standalone
```

It is recommended to embed this pane as a standalone window, as it requires more horizontal screen space to perform activities within the conversation (like opening documents or working with gadgets). When embedded as an iFrame, enough room must be provided to accommodate this horizontal expansion of the User Interface.

Programmatic Integrations and the On Track API

Integrating with On Track through custom code enables richer interaction with On Track than that provided by the integration of a standalone conversation pane. Some examples of applications that integrate with On Track APIs include:

- Automatically generating conversations for business objects in an external system
- Writing messages in an existing conversation when a business event happens in an external system
- Updating an external system's records when a message is entered into a conversation
- Having a system auto-respond to conversation messages with relevant information
- Displaying On Track information directly inside an external system, with fine-grained control of the UI
- Providing gadgets to add live business information and actions to conversations

On Track provides a remote HTTP-based API for working with the server from multiple types of clients. It is possible to work directly against the API when using a wide range of clients and languages, such as mobile client development, .NET, etc. Additionally, On Track provides a Java SDK and JavaScript library to facilitate working with those runtime environments.

API Fundamentals

The On Track API supports two fundamental types of communication paths -- the Front Channel and the Back Channel. The HTTP protocol is used for all communication across these channels. When working with the On Track API, either the XML-RPC or JSON-RPC data formats must be used.

On Track Front-Channel communications are client-to-server protocols. They are synchronous operations and are initiated on-demand by the client. These can be used to retrieve initial state information and perform updates. Examples of Front-Channel communication include posting a message to a conversation, getting the existing messages of a conversation, and posting a document.

Back-Channel communications are On Track server-to-client protocols. They are asynchronous operations that are used by the On Track server to call back the client. These can be used to provide status updates, inform the client of model data changes, or provide event notifications. Examples of Back Channel communications include listening to all messages entered into a conversation, informing a user that a contact is active in a shared conversation, or notifying a business system that a new conversation has been created.

Front-Channel and Back-Channel communications often work in tandem -- the action of a user sent over the Front Channel may result in multiple Back-Channel messages being issued to multiple clients. Likewise, Back-Channel messages often result in a set of updates through the Front Channel.

Modules

The On Track API organizes groups of common operations into Modules. These Modules are provided for working with conversations, tracks, users, groups, documents, bookmarks, and many other common functions within the product. Module APIs define both the Front-Channel and Back-Channel communications to On Track. The response to invoking a Module API is a return value or an exception; return values are scalar types including Dates, Lists, Sets, Maps, and Data Transfer Objects.

When exposed as part of the On Track Java SDK, Modules are organized as Java packages. The full list of Modules/Packages can be viewed by accessing the *Oracle On Track Communication SDK Documentation*. The Modules containing methods exposed in the SDK Documentation are identical to what is available for remote client developers using JavaScript or other mechanisms (see Java Client Development). Modules in the SDK are identified under packages named `waggle.common.modules`.

Setting Required Headers

When working with the On Track API, several required HTTP Headers must be provided to establish communication with the server. Table 1 lists the different headers needed:

Table 1

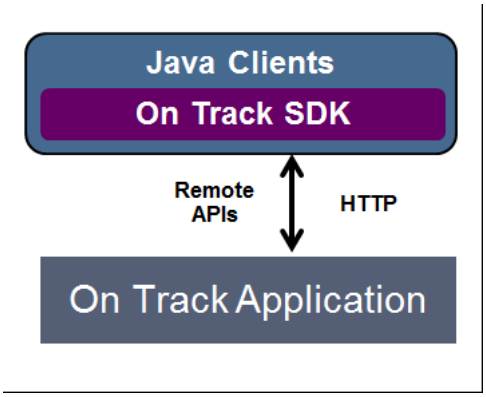
Name	Description	Value
X-OnTrack-APIVersion	Minimum version of the On Track API that the client expects to use; will be utilized by future versions of the On Track server for client compatibility checks.	1005

Table 1 (Cont.)

Name	Description	Value
X-OnTrack-ClientID	The ClientID is a long integer that uniquely identifies an instance of the client program within the scope of an HTTP session. For example, if two or more tabs are opened in Firefox and the Web client is loaded into each tab, then each instance of Web client must have a unique ClientID value. The recommended practice is to set the ClientID to the current time in milliseconds when its client is first loaded.	Long integer
X-OnTrack-Agent	The Agent header should be set to a string that identifies the client program. The format is free-form, but providing a full client name and version is recommended strongly as a best practice to assist with troubleshooting on the server.	String, for example "Oracle On Track Client for Windows Mobile Phone 1.2.5"
X-OnTrack-RandomID	Also known as the cross-site request forgery (CSRF) token, this helps prevent CSRF security exploits. This is a secret key that is issued to the client after login, and has to be supplied with each subsequent API request made by the client. The intent is to prevent a malicious script from calling the On Track server on behalf of the client by piggy-backing on a logged-in session that's maintained by the browser. RandomIDs are unique to each user session.	77e3574512ce8ca516dc7a4e639dacb1

Java Client Development

On Track ships with a Java SDK to provide developers a means to work with On Track APIs without using raw HTTP calls.



The On Track SDK takes care of making remote calls (over HTTP) to the On Track server. The main library for the SDK is named ontrack.jar. This file is located under the `MIDDLEWARE_HOME/Oracle_ONTRACK1/OnTrack/ClientSdk/ontrack-sdk/lib` directory of an On Track installation. In addition to ontrack.jar, this directory contains a variety of other utility libraries which need to be in the classpath of any client code which uses this class.

Java-Based Front-Channel Interactions

Below is a sample Front-Channel Java application which returns the display name of the logged-in user. This sample demonstrates creating a client interface to On Track, logging on, performing a front-channel request, and then logging out and shutting down the client.

```
import waggle.client.main.XClientMain;
import waggle.common.modules.connect.XConnectModule;
import waggle.common.modules.connect.infos.XLoginCredentialsInfo;
import waggle.common.modules.connect.infos.XLoginInfo;
import waggle.common.modules.user.XUserModule;
import waggle.common.modules.user.infos.XUserInfo;
import waggle.core.api.XAPI;
import waggle.core.api.XAPIManager;
import waggle.core.api.exceptions.XAPIException;
import java.util.Properties;

public class BasicSample {
    static String fHost = "http://localhost:8080";
    static String fContext = "/ontrack";
    static String fUserName = "";
    static String fUserPassword = "";

    public static void main(String[] args) {
        try {
        // initialize the client
        Properties properties = new Properties();
        XClientMain.initAppWithJar(properties);

        // get an API instance, and set the session host, context, and agent name
        XAPI fAPI = XAPIManager.newInstance();
        fAPI.setSession(fHost, fContext);
        fAPI.setOnTrackAgent("On Track API Example");

        // perform login and set the session random ID
        XLoginCredentialsInfo credentials = new XLoginCredentialsInfo();
```

```
        credentials.UserName = fUserName;
        credentials.UserPassword = fUserPassword;
        XLoginInfo loginInfo =
fAPI.call(XConnectModule.Server.class).login(credentials);
        fAPI.setRandomID(loginInfo.APIRandomID);

        // Get information about the currently logged in user and print it out
        XUserInfo userInfo = fAPI.call(XUserModule.Server.class).getMe();
        System.out.println("Users Display Name is " + userInfo.DisplayName);

        // perform logout and shutdown the API library
        fAPI.call(XConnectModule.Server.class).logout();
        XClientMain.shutdown();
           }
           catch (XAPIException ex) {
        System.out.println("The Server replied with an Error : " + ex);
           }
           catch (Exception ex) {
        ex.printStackTrace();
           }
    }
}
```

At the heart of the above example is the location where the On Track User Module is executed to return the user object for the currently logged-in user.

```
XUserInfo userInfo = fAPI.call(XUserModule.Server.class).getMe();
```

By passing in different Modules as part of the fAPI.call, it is possible to invoke the full set of Modules provided by the Java API. A more extensive set of Java-based tutorials is available from the On Track product Web site at

http://www.oracle.com/goto/ontrack

Java-Based Back-Channel Interactions

Back-Channel Module APIs provide a rich way for client developers to extend On Track. Using the Java API, developers can register to have their clients receive callbacks from the server when certain events happen. This is accomplished by writing a class which implements a client events Module. For example, the `XChatModuleClientEvents` module provides a number of callbacks to the client including:

- notifyChatCreated
- notifyChatBookmarkClosed
- notifyChatBookmarkOpened
- notifyChatDeleted
- notifyChatFollowupClosed
- notifyChatFollowupOpened
- notifyChatUpdated
- notifyChatsRead
- notifyTyping

As an example, it is possible to write a client that implements the `notifyChatCreated(XChatInfo arg0)` method. Once messages are posted into a

conversation, these contents are sent to the client, which then acts on the callback. These actions may include automatically parsing and posting a response to the conversation, modifying an external system.

Clients that support Back-Channel communication must:

- Implement a ClientEvents Module
- After login, invoke the `XHiveModule.Server.class.enterHive()` method to register as an agent
- Register themselves with `XEventsManager.register(this)`
- Instantiate and start a `XMessageReceiverThread` thread using `XMessageReceiverThreadJSON(fAPI);`
- Loop continually waiting for and acting on message callbacks
- When complete, unregister themselves with `XEventsManager.unregister(this);` and stop the `XMessageReceiverThread` thread

Back-Channel Agent Requirements

Agents need to be logged into On Track to be able to receive callbacks from the server. The user name they are logged in as must have the 'On Behalf Of' privilege which can be granted in On Track Administration Console. See *Oracle On Track Administrator's Guide* for more details. Additionally, the On Track client listener operates as the user it logs in as, and has the same visibility as that user has within the application. In order to get call backs when messages are added to a conversation, the user the agent is running as must be added to the conversations that are of interest.

It is possible to design this integration in a number of ways; for example the agent could be programmatically added to a conversation when a system creates that conversation for the first time, or the agent can be designed so that it joins a Conversation when it is buzzed by a user.

Java Back-Channel Agent samples are available on the Oracle On Track product Web site at

`http://www.oracle.com/goto/ontrack`

Advanced Topic - Using Conversation List Back Channel Events

The embeddable conversation list User Interface has a mode that allows users to both see existing conversations and create new conversations.

Figure 7 Conversation List with New Conversation

To support the creation of new conversations and provide for other interactions, the Conversation list creates Back-Channel events that a client agent can listen and respond to. This provides the client agent the ability to take additional actions over the

Front-Channel, such as adding additional people or messages to the newly created conversation.

As detailed in the Advanced Conversation Integrations section of this document, the Conversation list must be invoked with three URL parameters: secondaryID, integratorName, and trackName for it to send Back-Channel messages. Then, when the Conversation list is viewed or the **New** button is pressed, an event is created. This event contains the user ID of the user looking at the conversation list, as well as one of the following two possible message payloads:

- GetConversations – Generated whenever the Conversation List is displayed in the user's browser.
- CreateConversation – Generated when the user clicks the **New** button in the Conversation List, provides a Conversation Name and presses **OK**.

Client agents must implement the `XbackChannelModuleClientEvents` class and its `messageSent` method. The messageSent method returns a payload of strings formatted as a JSON object, which can be parsed by the client agent into a list of name/value pairs containing information about the conversation list.

Additionally, any URL query parameter added to the Conversation List URL is added to this list as well which allows for additional context to be passed to the client agent.

The following is a sample Conversation List URL:

```
http://hostname/ontrackcontext/web/ConversationList.jsp?integratorName=STANDARD_
AGENT&secondaryID=SR-1234&trackName=Service%20Request%201234&myOwnParameter=SomeVa
lue
```

The resulting message payload sent to the client agent when a user views that Conversation list (causing the URL to be invoked) will be:

```
{
  "MessageType":"GetConversations",
  "integratorName":"<OnTrack Agent User>",
  "secondaryID":"SR-1234",
  "trackName":"Service Request 1234",
  "integratorID":"<OnTrack Agent User ID>",
  "senderName":"<OnTrack User>" ,
  "myOwnParameter":"SomeValue" ,
}
```

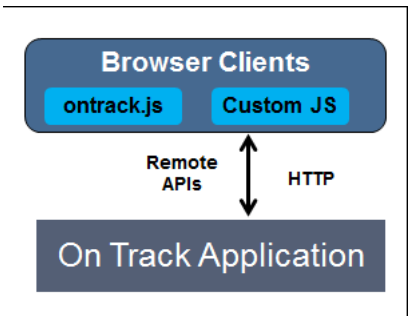
If the user presses the **New** button in the Conversation List and provides a Conversation Name in the dialog, the following message payload is sent to the agent:

```
{
  "MessageType":"CreateConversation",
  "Name":"Conversation Name",
  "integratorName":"<OnTrack Agent User>",
  "secondaryID":"SR-1234",
  "trackName":"Service Request 1234",
  "integratorID":"<OnTrack Agent User ID>",
  "senderName":"<OnTrack User>" ,
  "myOwnParameter":"SomeValue" ,
}
```

Notice how the custom URL query parameter "myOwnParameter" is part of the message as well.

JavaScript Client Development

On Track supports the development of custom client-side JavaScript which can run in the browser of a user and perform operations against the On Track Module API over HTTP using AJAX. This allows the creation of rich Web clients which can leverage both Front and Back-Channel interactions with an On Track application. Any standard JavaScript library can be used by client developers, from jQuery to DOJO, or a developer can use their own JavaScript code. A convenient JavaScript library, ontrack.js, is provided to simplify this process, but is not strictly necessary to use the On Track API.



The ontrack.js file can be found under the `MIDDLEWARE_HOME/Oracle_ONTRACK1/OnTrack/ClientSdk/ontrack-sdk/lib` directory of the installed product and can be copied to a place where it is co-located with your custom JavaScript and accompanying HTML files.

The ontrack.js file contains methods for logging into the server and executing Module APIs. The full JavaScript documentation for ontrack.js is provided in the *Oracle On Track JavaScript Documentation*.

The following is a jQuery code snippet which invokes the ot.execute method of ontrack.js to call the Tracks module and return a listing of tracks for the current user. This track list is then looped over and the track names are written out.

```
ot.execute( 'Track', 'getTracks',  function(success, trackInfos) {
for(var ix = 0; ix < trackInfos.length; ix++) {
                var trackInfo = trackInfos[ix];
                document.write(trackInfo.Name) } }
```

JavaScript Client Deployment Requirements

JavaScript development is complicated by the fact that the same-origin policy prevents a document or script loaded from one origin to get or set properties of a document from another origin. When the browser checks the origin of a JavaScript file, it verifies that it is hosted on the same named server as the remote APIs it invokes. As an example, if On Track is deployed on ontrack.company.com, any JavaScript invocations against that server must have their JavaScript and HTML files served from ontrack.company.com as well to run in the browser.

There are a number of ways to handle this same-origin policy with custom JavaScript clients that are written for On Track. The first option is to host the HTML and JavaScript files on the same server as On Track. This can be done by authoring a new Web Application Archive (WAR) file which is deployed on On Track servers (in addition to the On Track application). This effectively co-locates the files in the same origin.

A second option is to deploy the custom JavaScript to another server, masking that server and the On Track server through a proxy to appear as if they have the same origin.

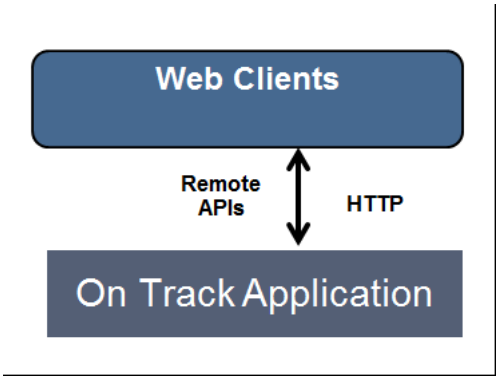
These options can make development of JavaScript applications more difficult to deploy and test. However, it is possible to selectively disable the same origin policy in browsers such as Google Chrome and Mozilla Firefox for development purposes. For example, developers can download the developer channel edition of Google Chrome which allows the browser to be invoked with the command line argument `-disable-web-security`. This argument disables the same origin policy. Clearly, this is not a recommended procedure for users of On Track in production where the two permitted configurations are co-locating the files in WebLogic or using a proxy server.

JavaScript samples are available on the Oracle On Track product Web site at

`http://www.oracle.com/goto/ontrack`

Other Remote Client Development

On Track supports the development of a variety of remote clients, from desktop clients using .Net libraries to native mobile applications written in Objective C.



The base URLs for accessing the On Track API are:

- Front Channel

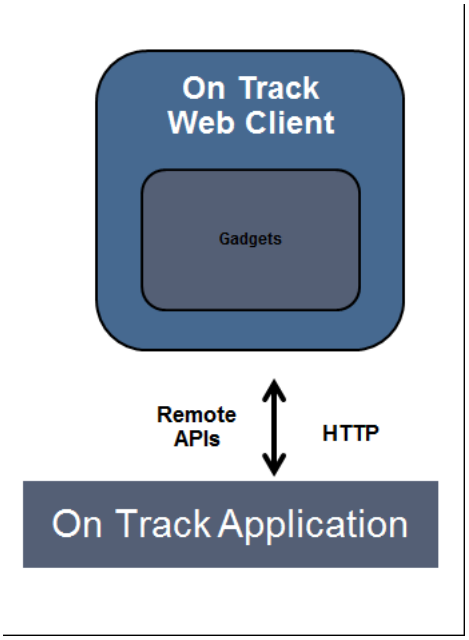
  ```
  http://<host:port>/ontrackcontext/ontrack/<method-name>
  ```

- Back Channel

  ```
  http://<host:port>/ontrackcontext/message/<method-name>
  ```

Gadget Development

On Track provides Gadgets by way of an implementation of the OpenSocial Core Gadget Specification. Developers can build Gadgets and register them with the On Track application so they can be added to conversations. On Track provides a base set of Open Social Features, which are common JavaScript functions that are loaded into the user's browser and be invoked by code in the Gadget. These functions include some base OpenSocial Features, as well as Features which are particular to On Track. Additionally, users may register their own Features in the On Track application.



Authoring and Registering Gadgets

Information on authoring Open Social gadgets can be found at http://code.google.com/apis/gadgets/. Gadgets are defined in XML files, which can be registered using the On Track administration console. Details on Gadget registration are provided in the *Oracle On Track Administration Console Online Help*. Some base Open Social Core Gadgets Features are supported including setting Gadget preferences and title.

Gadget samples are available on the Oracle On Track product Web site at

http://www.oracle.com/goto/ontrack

On Track Specific Feature

On Track provides a feature to enable communication between Gadgets and their containing On Track conversations. This feature is documented in *Oracle On Track Communication Gadget Feature Documentation*.

Custom Features

On Track developers can also write their own Open Social Features and register them with On Track. Registration is accomplished using the On Track administration console, details are provided in the *Oracle On Track Administration Console Online Help*.

Additional Resources

Additional development resources (including samples, blog posts and online discussion forums) are available from the Oracle On Track product Web site at

http://www.oracle.com/goto/ontrack

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/us/corporate/accessibility/index.html.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Oracle On Track Communication Developer's Guide, Release 1 (1.0)
E20959-01