Infor Epiphany Sales and Service Computer Telephony Integration Guide

Publication information

Release: 10.0.1

Publication Date: July 14, 2015

Contents

Introduction

1

About Infor

Infor delivers business-specific software to enterprising organizations. With experience built-in, Infor's solutions enable businesses of all sizes to be more enterprising and adapt to the rapid changes of a global marketplace. With more than 70,000 customers, Infor is changing what businesses expect from an enterprise software provider. For additional information, visit www.infor.com.

Purpose of this Guide

The *Infor Epiphany Sales and Service Computer Telephony Integration Guide* (CTI) provides the overview and configuration information needed to implement and customize an Infor CTI-enabled product. Its target audience is the implementors who deploy Infor Epiphany Sales and Service at the enterprise.

Product Documentation

The Infor Epiphany Sales and Service product documentation includes the manuals and online help systems described in this section. For a summary of features that are new for this release, late-breaking information about installation and upgrade, and information on fixed or outstanding product issues, see the Infor Epiphany Sales and Service Release Notes. For information on supported platforms, see the Documentation section of the Infor Support Portal, http://www.inforxtreme.com

Installation and Configuration Guide

The *Infor Epiphany Sales and Service Installation and Configuration Guide* includes all of the Infor-specific information required to get the Infor Epiphany Sales and Service applications running. When special configuration is required or recommended for other platform-support software (such as

WebLogic, WebSphere, JBoss, SQL Server Oracle etc.), it is also included. The audience for this book should be familiar with installing and configuring sophisticated enterprise software. They are expected to be experts in their own corporate network configurations and knowledgeable about security topics such as proxy servers and firewalls. General familiarity with database management and maintenance is also assumed.

Implementation Guide

The *Infor Epiphany Sales and Service Implementation Guide* provides procedural information relevant to individuals involved in implementing and customizing the Infor Open Architecture and the Infor Epiphany Sales and Service applications built on it. Implementers typically possess expertise in lightweight Java development, HTML, DHTML, JavaScript, and SQL. They primarily work with the Infor Studio configuration tool (and to some extent with the Designer tools). The *Infor Epiphany Sales and Service Mobile Wireless Implementation Guide* provides additional procedural information relevant to individuals involved in implementing and customizing the Infor Epiphany Sales and Service Mobile Wireless application.

Reference Guide

The *Infor Epiphany Sales and Service Architecture Reference Guide* and the *Infor Epiphany Sales and Service Application Reference Guide* contain technical reference information relevant to implementors involved in implementing and customizing Infor Epiphany Sales and Service at customer sites. These books provide the reference context for the procedural information available in the *Infor Epiphany Sales and Service Implementation Guide*. The *Infor Epiphany Sales and Service Mobile Wireless Architecture Reference Guide* provides additional technical information relevant to individuals involved in implementing the Infor Epiphany Sales and Service Mobile Wireless application.

Administrator's Guide

The *Infor Epiphany Sales and Service Administrator's Guide* provides supervisors, managers, and executives with the information to use the Infor Epiphany Sales and Service and Admin Console functionality to manage the work of their agents and salespeople. Instructions for day-today maintenance of the system are included in this book.

Computer Telephony Integration Guide

The Infor Epiphany Sales and Service Computer Telephony Integration Guide (CTI) provides the overview and configuration information needed to implement and customize an Infor CTI-enabled product. Its target audience is the implementors who deploy Infor Sales and Service at the enterprise.

Integration Guide

The Infor Epiphany Sales and Service Integration Guide provides overview and configuration information for the set of tools used to exchange data with a variety of back-end data sources, including generic SQL sources, Java and EJB-based sources, Web services, and other database types (using Infor Enterprise Application Integration, or EAI).

Online Help

Online help documentation for Infor Sales and Service is:

- Admin Console
- Sales
- Self-service
- Service
- IBRDesigner
- WorkflowDesigner
- DialogDesigner
- Studio
- Logviewer

These are available for users of the Infor Contact Center, Infor Sales, Infor Service, Infor Workflow Designer, and Infor Dialog Designer.

Viewing Release Notes and Manuals

Product release notes and manuals are part of the Sales and Service package. To view product documentation open the document of interest in Acrobat Reader.

Printing This Document

To print this document at the highest quality resolution, print to a Post-Script driver. Other drivers may not reproduce the screen shots as accurately. This document is designed to be printed on two sides of the page. If your printer is not configured for duplex printing, you may find a blank page at the end of some chapters. This is normal.

Contacting Customer Support

You may contact the Infor Customer Support center by submitting your incident via the web 24x7 at http://www.inforxtreme.com, or by placing a call during our scheduled business hours. For a complete listing of our support centers with web addresses and phone numbers, access our support site at http://www.inforxtreme.com.

Location of the Platform Support Matrix

For more information on platform support, including hardware and software requirements, see the http://www.inforxtreme.com web site.

Architecture and Overview

2

The Infor Channel system provides call and e-mail control functionality in a thin-client architecture.

This SDK is designed to enable implementors to customize the behavior of the Channel system in three ways:

- Create providers to connect Infor CTI to third-party call control systems.
- Customize or create new rules and behaviors to specify the way that incoming communications are evaluated and assigned.
- Implement custom functionality by creating extensions.

The Channels system can also be customized using metadata options.

In addition to supporting customization, the Channels SDK also includes a testing framework which enables implementors to test a CTI implementation within a simulated working environment prior to deployment. The simulator can also be used to drive the real production environment during off hours, enabling you to test using the equipment on which the system goes live.

Contents of this SDK

The CTI SDK consists of several key pieces. This document is the main part of the SDK. It provides the overview and customization information that you need to implement, deploy and test an Infor CTI-enabled product.

In addition to this document, you can also refer to the Infor Platform JavaDocs, which are installed with your Infor product. The JavaDocs contain the full specs of the interfaces, base object definitions, and some details about the implementation and utility classes. There are several key locations in the JavaDocs for getting CTI related information.

- `com.epiphany.shr.cti` - Contains the CTI platform logic
- `com.epiphany.shr.cti.application` - Contains the application helper classes that are part of the CTI interface. These are the high level interface to CTI.
- `com.epiphany.shr.cti.ctimanager` - Contains the implementation and interfaces for the CTI Manager and CTI Manager Startup.

- `com.epiphany.shr.cti.ctirules` - Contains the interfaces and implementation of rule, behavior, and the rule engine.
- `com.epiphany.shr.cti.ctiservice` - Contains the CTI Service and CTI Service Startup. These are the low level interface to CTI.
- `com.epiphany.shr.cti.middleware` - Contains the middleware implementations which are supported out-of-the-box.
- `com.epiphany.shr.tf` - Contains the CTI testing framework APIs.
- `com.epiphany.shr.cti.util` - Contains helper classes, common event definitions, common parameter definitions, and the Work Item Processor definition.

CTI Architecture

Channels Manager

The Channels Manager is the heart of the Channels system, responsible for tracking agents and connections. The bulk of the configuration and customization takes place in this component. It is also the component that is most easily extended. Within the system, it is referred to as the CtiManager.

Using Rules and Behaviors, you can extend it to include arbitrary custom logic (as long as any custom rules and behaviors do not take too long to run.)

Note: If rules and behaviors take a long time, CTI and UI functionality will not be responsive.

Users are assigned to a specific CtiManager that receives their login request, and stay with that manager for the duration of their session. This allows the CtiManager to limit its processing to events pertaining to the users that it is managing.

In the event of system or push connection failure, the users assigned to a CtiManager are moved to another CtiManager in the cluster. In general, the transition should be seamless, but depending on the user's state, they may notice a brief lapse.

Rules and Behaviors

The Manager evaluates incoming requests in a rule-based manner. Rules map requests to behaviors, which execute the requests. This allows the behavior of the system to be modified and enhanced easily. The associations between rules and behaviors are set in metadata, which enables some basic customization of call-control logic without recompiling code.

Rules and behaviors are designed and implemented using the `ManagerDataAccess` interface for managing and accessing their data. The `ManagerDataAccess` interface allows for the user's code to be running in a multi-threaded clustered environment without the user having to worry about those details. It abstracts away locking of data structures (providing speed, stability, and safety against deadlocks), and it also abstracts away fail-over support. This API is used internally for Infor-supplied rules and behaviors.

Queues and Requests

The manager has two classes of requests: synchronous requests and asynchronous requests.

- Synchronous requests are used to gather information for display or making decisions, and take actions that need to be handled before the system can continue. Examples of synchronous requests are logging in, or determining the appropriate state for the call control UI. Synchronous requests are given priority.
- Asynchronous requests are normal requests for processing. The CTI system is mostly asynchronous, so the bulk of all requests are of this type. An example of an asynchronous request is a request to place a call on hold.

Each CTI Manager has two queues, one for synchronous requests and one for asynchronous requests. There is one thread for processing both queues. When the thread is ready to do work, it first checks the synchronous request queue. If there is a pending data request it is handled. If there are no pending synchronous requests, then the first asynchronous request is handled. If there are no pending requests in the system, the processing thread will block and wait for a request to come in. Processing the synchronous requests first ensures that the CTI system does not bog the application layers down while they are waiting on synchronous requests to the CTI system. Once a request is pulled from the queue, the associations are run though in order, rules are checked, and code can be executed.

By default, there are two CTI Managers on each machine. The number of CTI Managers per machine is configurable.

CTI Application

The CTI application enables you to monitor and control a view of the information in the CTI system. The application's primary functionality includes:

- Showing agent state
- Tracking agents' phone calls
- Creating and updating interaction records
- Helping the agent associate interactions with other work that is related to a call
- Providing the user interface for the agent, and for calls and popups
- Pushing updates to the user interface when appropriate

The application interacts with the rest of the CTI system via the CTI service.

User Requests

When the user makes a request, CTI application extensions are launched. These extensions populate the appropriate Call or Agent application helper classes with data from the user's session, and use those classes to perform CTI-related activities.

Events

CTI events use the Infor Application Event Framework (AEF). The CTI AEF listener receives event calls, and creates the appropriate helper objects, updates BIOs, and uses the helper objects to push events to the user's UI.

All custom helper objects should be subclasses of `AppObject`. This base class provides basic service handle assistance.

CTI Service

There are two CTI Services, the CTIStartupService and the CTIServiceEJB. The CTIStartupService does the bulk of the work and is responsible for forwarding requests, processing complicated requests, and sending requests on to the CTIManagerStartup. The EJB service is used to send requests between server instances or to make requests from outside the server instance.

The CTI Service is the gateway of the CTI system. All requests for CTI work go through one of the CTI services.

Queues and Requests

The service layer has two classes of requests, synchronous and asynchronous. At the service level all requests are processed on the caller's thread. While the request is being forwarded, the caller is blocked. For synchronous requests, the caller is blocked until the request has been processed by the CTIManager and a value is returned. For asynchronous requests, the call returns once the work item is passed to the appropriate CTIManager for queueing.

User Assignments

The service layer is also responsible for tracking and enforcing user assignments. When a user logs into the CTI system for the first time in a session, their request is passed to the CTI Manager Startup on the local machine. The CTI Manager Startup adds the agent assignment information to the agent tracking database. All future requests by this agent, arriving anywhere in the cluster, are processed on the manager to which that agent is assigned. If a user request comes to the CTI Service on a machine other than the one to which they are assigned, it is the CTI Service's responsibility to forward the request seamlessly to the correct machine. This forwarding is done via an EJB call to the correct CTI Service EJB on the remote machine.

When a CTI system joins a running cluster, it reads the agent assignment table to determine where agents are assigned.

CTI Provider

The CTI provider enables communication between middleware systems or switches and the CTI Manager. The CTI provider translates requests from the CTI Managers into a form that the middleware provider or switch understands. It also translates the events from the middleware provider or switch into event messages that the CTI Manager understands. Finally, the provider maintains the connection to the middleware provider or switch.

At this time, Infor supports the Genesys TServer middle ware system. Support for other systems is under development.

Calls between the CTI provider and the Manager are implemented asynchronously. This prevents the processing of one component from tying up the other component. The CTI provider also tracks the users it supports. When users log in, the provider adds their information to the supported list. When users log out, they are removed from the list. Tracking user information allows the provider to quickly filter incoming messages. Filtering as many messages as possible at the provider layer greatly reduces the magnitude of the workload at the CTI Manager layer.

The CTI system is designed to enable implementors and partners write their own providers. The providers are implemented in Java and then added as the provider class in the configuration. For more information about writing providers, refer to "Writing A Provider" on page 56

The CTI Work Item types supported by the Channels system are enumerated in `com.epiphany.shr.cti.util.CtiWorkItemTypes`.

Rules, Behaviors and Associations

Rules and behaviors implement customizable logic in CTI. They enable you to support specific requirements (such as a hardware configuration which requires custom setup) in a modular, flexible way. You can extend the functionality of the system by adding your own behaviors in addition to the existing behaviors, or you can customize existing behaviors.

A rule or a behavior is a Java class implementing a standard interface. A rule contains some internal logic which is evaluated to decide whether to run a behavior. The rule is created with its configuration data and is evaluated for each CtiWorkItem. Depending on the results of the rule, the WorkItemProcessor (the base class for the Cti Managers and Cti Providers) either skips or executes the corresponding behavior.

Behaviors contain the logic to complete a piece of the work of processing a CtiWorkItem. They primarily perform two basic activities: updating data and forwarding (or stopping) CtiWorkItems. Each behavior can fire events, change data structures, and make API calls. During the execution of a behavior the code can instruct the rule engine to continue evaluation, start evaluation from the first association, or to stop evaluation. Behaviors can also create and forward new CtiWorkItems.

Rule engines are built into the CtiBaseProvider and the CtiManager implementation classes. These rule engines provide key logic allowing for customization and expandability. The rule engines maintain an ordered list of rules and associated behaviors.

Associations are entries in the rule engine evaluation list which provide a one-to-one relationship between one instance of a rule and one instance of a behavior. Associations exist in metadata, and

do not have implementation classes. They are configured in the metadata and loaded during startup. Changes to Rules, Behaviors, or Associations do not take effect until the application server is restarted.

CtiWorkItem

The CtiWorkItem object contains all the information needed to perform a CTI action (for example, Hold call). Work items are used for communication between the CTI Application, CTI Manager, and CTI Provider.

For example, a user clicks on the hold button for a call. The request goes to the CTI application extensions, which use the CTI Service to create a work item containing the agent and call information. The application then uses the CTI Service to start processing of the work item. This work item is then sent to the appropriate CTI Manager. In the Manager, the work item is processed and then forwarded to the appropriate CTI Provider. The CTI provider then forwards the item to the middleware system or switch for processing.

CtiDataAccess

CTIDataAccess handles communication between the Infor Service database, and users running on multiple instances of CTI components. Users are partitioned among the different instances of the same components, each of which has a single store for data. The data is protected by the component and a CtiDataAccess interface is passed into the rule engine. Rules and behaviors can use CtiDataAccess methods to get information. As long as the data structures are not used after the processing of a rule or behavior, this architecture does not require the data structures to have internal synchronization.

Data structures are created using `setData(agent_login, name, value)` and retrieved using `getData(agent_login, name)`.

Rule Engine

The rule engine is the place where execution of the rules and behaviors take place. During CTI startup, rule engines are created and all the rules and behaviors are loaded. The CTI Manager and each of the CTI providers contain a rule engine.

When a CtiWorkItem is passed to the rule engine, it iterates through the associations executing each in turn. The process is as follows:

1. A CtiWorkItem comes into the system.
2. The rule engine goes through the list of associations evaluating each rule. When a rule returns for the engine to evaluate the behavior, it does.
3. The rule engine evaluates the behavior and, based on the return code from the behavior, does one of the following:
   - Stops processing
   - Restarts processing at the first association
   - Continues to the next association

CTI States

Agent States

Internal state representations of the CTI system are:

- CTI_AGENT_LOGIN - the agent is currently logging into CTI
- CTI_AGENT_LOGOUT - The agent is currently logging out of CTI
- CTI_AGENT_AFTER_CALL_WORK - The agent is working on stuff related to a call that was just completed and is not ready to receive calls from splits they are logged into.
- CTI_AGENT_APP_PREFERENCE - the agent state will be set to the value, that was specified by application as a desired value after agent has been finished with after call work.
- CTI_AGENT_NOT_READY - The agent is logged in but not ready to receive calls
- CTI_AGENT_READY - The agent is logged in and ready to receive calls
- CTI_AGENT_ON_CALL - The agent is currently on a call or has a call on hold
- CTI_AGENT_OTHER_WORK - The agent is working on other things and is not ready to receive calls from splits that they are logged into
- CTI_AGENT_NULL - The agent state is invalid

For more information, refer to `cti.middleware.general.rules.AgentState`.

Connection States

- Null - the connection does not currently exist (same if data structure not found)
- Alerting - the connection is ringing
- Connected - the connection is active
- Fail - error making the connection or keeping the connection
- Hold - the connection has been placed on hold
- Initiated - dialing or prompted to take off hook
- Queued - suspended waiting for switching service
- Active - call is in a active state
- Unavailable - the current destination of the call is not available
- Unknown - call is in a unknown state

For more information refer to `cti.middleware.general.rules.ConnectionStates`

Communication between CTI components

Communication between CTI components is a combination of AEF messages, direct Java calls, and EJB calls using the service framework. The direct calls that are requests for information are simple calls that are responded to immediately. The direct calls that are items requiring processing (either

events or requests) are immediately queued and the call is returned. This mechanism allows for requests and events to be sent without tying up the threads of the component doing the requesting. This combination allows for efficient processing of work requests while still providing quick query responses. The synchronous data requests have priority over normal processing. This priority helps ensure that the screen refreshes are quick, and prevent the CTI system load from bogging down the application and UI service. EJB calls are used by the CTI service to forward requests to the machine a user is assigned to.

When running extensions based on UI actions, note that they may not be running on the same server instance as the user's CTI Manager. Extensions should be constructed to minimize server round trips.

AEF messages are used to send events into the application. These events contain updated state information that allows for the push service to be called by the application. For instance, if a call is placed on hold, a call held event is pushed to the application. The application is then responsible for updating the UI when appropriate. AEF messages are also used to keep a current list of logged in users throughout the CTI system. The CTI services keep track of all users in a cluster. The CTI Managers track users assigned to them and the CTI providers track users on the server.

Implementing CTI

This section describes information relevant to implementing CTI components.

Application

The application is fully customizable. CTI elements are standard components of the application and can be tailored to specific customer requirements. For more information on application configuration, refer to the *Infor Service Installation and Configuration Guide*.

Service

For information on CTI service configuration, refer to the *Infor Service Installation and Configuration Guide.*

Manager

The bulk of the configuration and customization takes place in the CTI Manager. This is also the component that is most easily extended. The CTI Manager is designed to make customization as easy as possible. The CTI Manager comes configured with basic functionality for the supported systems. Where appropriate, this functionality is complete with configurable parameters.

The CTI Manager can be customized in many ways. You can configure the system to use different switches and versions of the supported middleware providers. You can also customize the behavior of the application, by changing, adding or removing rules, extending or overriding behaviors, and by writing extensions. The CTI Manager is basically a rule evaluation engine with an API. The API that is used in the rules and behaviors by people customizing the system is the same API used by developers to build the entire system. The rule and behavior code is also written in Java. The rules in the CTI Manager can be based on middleware provider and switches. This allows for one CTI Manager to work with multiple switch configurations.

Providers

The Channels system is designed to enable people to implement their own providers in the field. Custom providers need to support the required Java interfaces with the same asynchronous communication semantics.

In addition to the supported Genesys and Avaya providers, the Channels system also includes a base provider. In standard integration mode, it provides a simple simulation of a working CTI integration, which is useful for demos and testing. In non-integrated mode, it supports inbound and outbound interactions without CTI integration, which is useful in situations in which hardware interaction is controlled by an external application. For more information about the base provider, refer to "Configuring the Base Provider" on page 95.

You can use C++ classes to communicate with the middleware system by using a Java - C++ bridge.

Metadata

The CTI system metadata provides the number of CTI Managers to start on each server, the number and types of providers to start on each server (and their connection information), application configuration, and configuration information for the CTI Manager.

The provider metadata defines what provider classes to load, where the classes should get their configuration information, and the switch configurations for the instances of the switches they are talking to.

The provider instance metadata allows for specific overrides of provider type metadata to allow for configuration differences between instances of the middleware subsystems. The CTI Manager metadata defines the rules, behaviors, configuration, and logging information to the Cti Manager. It also defines the validation of work item parameters at the CTI Manager and provider level.

The validation types are represented in `cti.util.CtiMetadataParams`. However, all metadata, including validation, is defined in Infor Studio. The metadata params class is an internal reference to the variables in the metadata that Channels can access. For instance, Rules have a configuration parameter named value_storage, defined in the `CtiMetadataParams` class, which is used to help avoid hard-coding strings and aids in refactoring.

agent_cti_extension

The agent_cti_extension is a definition of all the current active user's individual extensions. It has the following properties:

Table 1: agent_cti_extension Properties

Properties	Requirements	Data Mappings
agent_cti_extension_id	GUID for the database	
extension_number	The agent phone number	Direct
revision_number standard		
obsolete flag standard		

agent_cti_split

The agent_cti_split has multiple entries per agent and each entry adds agent's membership in a split. It has the following data mappings:

Table 2: agent_cti_split

Properties	Description
agent_id	FK to agent (the ID of the agent from the CTI_agent table)
split	STRING 15. This is the actual CTI split the agent belongs to. It is typically a phone number.
agent_cti_split_id	
revision_number	
obsolete flag	

Extensions

The UI Event Handlers rely on the following helper classes to encapsulate much of their work:

- `Agent`
- `AppObject`
- `Call`
- `CtiAppUtil`

For more information, refer to the JavaDocs for `com.epiphany.shr.cti.application`.

Common Event Fields

Common event fields are contained in the `CtiParameters` interface (`com.epiphany.shr.util.CtiParameters`). This interface defines fields to contain most information about objects and events related to CTI interactions.

You can add additional fields with any name at any time, and the system will pass the values on as long as any additional fields do not conflict with the original fields. Infor stores string constants in `CtiParameters`. If you add any additional fields, you should also store them in an interface.

AEF Events

The AEF event handlers catch events sent from the CTI Manager and use the Push service to cause selective redraws of parts of the screen.

Implementation is done through a base class for the event handlers, from which you derive the basic `CtiCallExtension` and `CtiAgentExtension`. These in turn do the bulk of the work for all the various event handlers.

Internationalization

Most of the CTI system is implemented using Java Unicode strings. When talking to the middleware providers, care must be taken to ensure that the middleware providers provide correct Unicode strings, or that the strings retrieved from them are converted.

Enumerations is a key area where internationalization has an impact on CTI. Enumerations must support expansion and display values in multiple languages. For example, the agent state of unavailable should not be hard-coded in the application. It should be presented based on the user's language and locale. The customer code attached to calls is maintained as a Unicode string.

The UI button labels for CTI are loaded from metadata based on the agent’s locale and pushed to the screen. CTI errors in Genesys are handled the same way.

When the CTI system uses the normal logging mechanism, it follows the correct international protocols.

Channels Configuration

3

Enabling CTI

By default, CTI is disabled. To use this feature, you must enable it in Studio:

1 In Studio, go to **Administration > User Preference Templates**.

2 Search for a preference name of `*show_account_summary`. Set this property to **true.**

3 Restart Infor Service for your changes to take effect.

The show_account_summary setting controls the visibility of the call control buttons.

Configuring Agents and Splits

Out of the box, the Channels system does not have any splits configured. You must create entries for each of the splits to which you want to add agents. In the agent interface, splits are referred to as skill groups.

Configuring Splits

To configure a split:

1 Open Studio and connect to the database.

2 In the Guide Bar, expand **Physical and Logical Schema**.

3 Open **Lookups**.

4 Click on `agent_cti_split`.

5 Add a new row, and enter the split information defined in your telephone system. For the demo provider, you can use any values in these fields. Fill in the following fields:

- **Lookup text**: This is the text that is displayed in the dropdown menu that appears when an administrator is configuring agents in the Web UI. This can be a numeric value, or a more descriptive string.
- **Code String**: This is the actual value of the split. Other fields are filled in by default.

6 Save this record to the database.

7 If the server is not already running, start the server. The split changes do not take effect until the server is restarted, or you refresh the metadata.

Split entries are used to populate the **Skill Groups** field in the Administration interface.

Configuring Agents

Once you have defined splits and reloaded the metadata (or restarted the server), you must configure each agent for CTI.

To configure an agent:

1 Start the server.

2 Open a new browser and log into the web UI as a user with administrator privileges.

3 In the Administration navigation bar, select the agent you want to configure.

4 Select the **Telephony** tab for that agent.

5 Add a **Telephony Username** and **Telephony Password** and save the record.

6 Add a telephony extension by clicking **New Extension**. On the screen that appears, type in the phone number that the agent logs in to.

7 Click **Save**.

8 Add a split by clicking **New Split**. Pick the agent's split from the dropdown. Currently the system only supports one extension per agent. The next time this agent logs in to Infor Service, CTI will be configured for them.

Disabling Call Pop-Ups

Out of the box, CTI creates call pop-ups for incoming calls. When the call is answered, a new tab is created. This behavior is controlled by the New-is answered, a new tab is created. This behavior is controlled by the `New-CallPopupAlert` global setting. Setting this to false creates tabs immediately for incoming calls, without creating a pop-up. To configure this behavior:

1 Open Studio and connect to your database.

2 In the Guide Bar, select **Administration > Global Settings**.

3 Search for a global setting named `NewDispatchedItemPopupAlert`.

4 Set `NewDispatchedItemPopupAlert` to **true** to enable pop-ups, or **false** to disable pop-ups.

5 In Studio, under **System Administration**, click **Refresh Global Settings** to deploy the changes into metadata on the running server.

Disabling Active Dispatching Pop-Ups

Out of the box, Infor Service creates pop-ups for items that are actively dispatched to agents. When the agent accepts the actively dispatched item, a new tab is created. This behavior is controlled by the `NewDispatchedItemPopupAlert` global setting. Setting this to false creates tabs immediately for actively dispatched items, without creating a pop-up.

1. Open Studio and connect to your database.
2. In the Guide Bar, select **Administration > Global Settings**.
3. Search for a global setting named `NewDispatchedItemPopupAlert`.
4. Set `NewDispatchedItemPopupAlert` to **true** to enable pop-ups, or **false** to disable pop-ups.
5. In Studio, under **System Administration**, click **Refresh Global Settings** to deploy the changes into metadata on the running server.

Ensuring Call Pop-Ups Take the Foreground

If the Internet browser is in the background when the inbound phone call arrives, the pop-up may blink on the taskbar and does not take the foreground. If this happens, do the following:

1. Download Tweak UI (Tweakui.exe) from this URL:http://www.microsoft.com/ntworkstation/downloads/PowerToys/Networking/ NTTweakUI.asp
2. Run `Tweakui.exe`.
3. Start Control Panel (**Start > Settings > Control Panel**) and click **Tweak UI**.

   In the Tweak UI dialog, click the **General** tab and uncheck **Prevent applications from stealing focus**.
4. Click **Apply** and then click **OK**.
5. Restart Internet Explorer.

   Once you perform these steps, the call pop-ups will show up in the foreground.

Setting the Call Pickup Number

The call pickup button dials a predefined phone number. This phone number is configured in metadata, on a per-provider basis

To change the call pickup number:

1. Open Studio and connect to your database
2. In the Guide Bar, click on Administration.
3. Expand **CtiManager > Providers**, and select **Parameters**.

4 In the main window, add a new parameter by setting the parameter type to `CallPickupAccessCode`. Click in a different row to make your changes take effect.
5 In the Guide Bar, select **CtiManager**.
6 In the Properties window, expand **Providers**.
7 Expand the provider that you want to configure. Beside the `CallPickupAccessCode` property, enter the new call pickup number.

Configuring Email Active Dispatching

Infor Service can actively dispatch e-mails to agents, one message at a time. Active Dispatching monitors a queue and sends E-mail messages from the queue to the agent's desktop, giving them a screen pop. The screen pop provides them with some information about the e-mail and allows them to accept it.

Active Dispatching is enabled by configuring queues to be actively dispatched, and having agents that belong to those queues.

To configure a queue to be actively dispatched:

1 Start the server.
2 Log in to the web UI as an administrator.
3 In the Administration Nav Bar, click on **Group Queues Admin**, and search for the group queue you want to configure.
4 In the detail view for the group queue that you are configuring, select the Distribution Rules tab.
5 Click on the Interaction work item type.
6 For the work item type that you want to actively dispatch (such as Request, Task or Interaction), set the Distribution Mode to Least Busy.
7 Save your changes.

Changing the CTI Landing Screen

By default, when an agent answers an incoming call, they are taken to the CTI Customer Search screen. You can modify this behavior to do the following:

- Always skip the search screen
- Skip the search screen when the customer has been identified

Skipping the Search screen means that when an agent queues an incoming call, they will be taken directly to the Detail screen. If the Search screen is not skipped, then once the agent selects an account from the list, they are taken to the Detail screen for that account. The final destination for the new interaction is always the Detail screen.

In 6.5.0, you can specify which Detail screen the agent lands on when receiving an incoming call via a global setting. This procedure is described in more detail below.

Skipping the Search Screen

By default, the customer search screen is shown for every interaction.

To skip the search screen:

1 In Studio, go to **Administration > Global Settings**.

2 To skip the search screen:

- When the customer is identified - set NewCallSkipSearchScreenWhenCustomerIdentified to true. If a single customer record is found matching the caller ID for a given interaction, the search screen is automatically skipped.

  Note: If you use the Cisco ICM (Intelligent Contact Management) provider, setting `NewCallSkipSearchScreenWhenCustomerIdentified` to true does not work.

- For every interaction - set `NewCallSkipSearchScreenAlways` to true. When set to true, this overrides the value set for the " when customeridentified" option. Default value is false .

  You can also update global settings in the web application UI.

Specifying the Detail Landing Screen

New call tabs no longer arrive at the interaction Detail screen. Instead, they arrive at the individual detail screen. To change the default landing screen:

1 Go to **Administration > Global Settings**.

2 Locate the `NewCallDetailLandingScreen` setting.

3 Select a value from the property list. Possible values are individual, organization, contact, order, product_instance, request, and task.

  It is also possible for attached data from the middle ware to override the detail landing screen value specified in the global settings. This occurs when the attached data from the middle ware contains a key-value entry where the key is DesiredDetailLandingScreen. The value of the keyvalue pair can be any of the possible values for the NewCallDetailLandingScreen global setting.

Tracing a Single User, Extension, and/or Place

You can log information about incoming and outgoing work items for a single user or extension in Production mode. Enabling log tracing for a single user allows you to track detailed information about

the operation of the CTI system, where enabling the same level of tracing for all users might flood the log with messages and degrade system performance.

1 In Studio, go to **Administration > Global Settings**.

2 Locate the property for the level of tracing that you want to configure:

- To enable tracing in all CTI layers other than the provider (i.e.,CtiManager, CtiService and CTI application), locate TraceFullyQualifiedUsername, and enter the username to trace.
- To enable tracing in the Genesys TServer provider layer, locate TraceTelephonyExtension, and enter the extension to trace.
- To enable tracing in the Genesys AIL provider layer, locateT raceTelephonyPlace , and enter the Genesys Place name to trace.

  You can also configure global settings through the Administration

3 In Studio, refresh your meta. If you made the changes in the Administration interface, click **Hot Deploy**.

4 In a browser, open http://<server>/<instance>/soap/SetPriority and set the log level to INFO for the appropriate class:

- For tracing in the CTI application layer: `com.epiphany.common.channels.extension.helper.ChannelsApplicationTraceByUsername`
- For tracing in the CtiService layer `com.epiphany.shr.cti.ctiservice.CtiServiceTrace By- Username`
- For tracing in the CtiManager layer `com.epiphany.shr.cti.ctimanager.CtiManager TraceBy- Username`
- For tracing in the Genesys TServer provider layer: `com.epiphany.shr.cti.middleware.tserver.base.provider. TServerTraceByExtension`
- For tracing in the Genesys AIL provider layer: `com.epiphany.shr.cti.middleware.genesysail. base.provider.GenesysAilTraceByPlace`

  Once enabled, activity for that username or extension is logged, and stored in the general application log file.

Performance Considerations

When tuning your CTI integration to guarantee optimal performance, please consider the following:

- Performance analysis of a CTI integration must be done on each element of the end-to-end transaction to identify where bottlenecks are occurring which are preventing the optimal target response time of the transaction from being reached.
- Be aware that if a fat client is being used to set a benchmark for ideal response time, this client should receive calls along the same telephony and network path as the call flowing into Epiphany. Further, if Epiphany is making calls to the database or external applications (via EAI, Web Services,

MQ, etc.) and the fat client does not simulate those behaviors, then the time spent with those activities needs to be considered for a meaningful comparison.

- It is essential to ensure that the times (and delays) measured along various points in the call flow path are measured against the same clock (e.g. if these times are extracted from the logs, the servers need to all be set to the atomic clock, so meaningful calculations can be made). Further, if logging is enabled specifically to track the times various events happen, the overhead of logging itself can introduce delays, which should be included as an assumption of the performance analysis effort.

Please contact Infor Customer support for assistance if you are having difficulty tuning your CTI system to reach optimal target performance goals.

Channels Application Parameters

The following settings affect the behavior of Channels functionality on agents' browsers. To adjust these setting, run Infor Studio and go to the location specified in the table. You must restart your Infor Server or refresh your metadata before the change takes effect.

Table 3: Call Management Application Properties

Property	Purpose	Location in Studio
show_account_summary	Determines whether CTI is enabled for the user. This property can be pre-set in the user preference template.	**Administration > User Preference Templates**
NewCallDetailLandingScreen	Specifies the type of detail screen displayed when a new call interaction tab is generated. If the new tab first lands on a search screen, then this detail screen does not appear until a customer is selected from the search screen. If the new tab skips the search screen, then this detail screen appears immediately. The following types of detail screens are supported values for this parameter: individual, contact, organization, order, product_instance, request, and task.	**Administration > Global Settings**
NewCallPopupAlert	Determines whether a popup alert appears to the user when a new call comes in. Set to true to enable pop-ups.	**Administration > Global Settings**

Property	Purpose	Location in Studio
NewCallSkipSearchScreenAlways	Determines whether a new call interaction tab should always skip showing the search screen. Set to true to skip the search screen and go immediately to a detail screen.	**Administration > Global Settings**
NewCallSkipSearchScreenWhenCustomerIdentified	Determines whether a new call interaction tab should skip showing the search screen whenever a single customer has been identified (based on the caller's phone number). Set to true to skip the search screen and go immediately to a detail screen.	**Administration > Global Settings**
NewDispatchedItemPopupAlert	Determines whether a popup alert appears to the user when a new dispatched item comes in. Set to true to enable pop-ups.	**Administration > Global Settings**
TraceFullyQualifiedUsername	Facilitates trace logging in production mode for the specified user at all CTI layers other than the provider (i.e., CtiManager, CtiService and CTI application). This setting has no effect unless the log level is set to INFO for one or more of the appropriate CTI trace classes. Once enabled, the designated CTI layer traces all incoming and outgoing work items for the specified user. The value for this parameter should match the value that appears in the Login ID field of the out-of-the-box User Detail View (for example, EPIPHANY\demo for NTLM, uid=demo,ou=people,dc=epiphany,dc=com for LDAP, and so on.)	**Administration > Global Settings**
TraceTelephonyExtension	Facilitates trace logging in production mode for the specified extension at the Genesys TServer provider layer. This setting has no effect unless the log level is	**Administration > Global Settings**

Property	Purpose	Location in Studio
	set to INFO for the TServer provider trace class. Once enabled, the TServer provider layer traces all incoming and outgoing work items, as well as incoming Genesys events and outgoing Genesys requests, for the specified telephony extension. The value for this parameter should match the Telephony Extension value which appears in the **Telephony** tab of the out-of-the-box User Detail View.	
TraceTelephonyPlace	Facilitates trace logging in production mode for the specified Genesys Place name at the Genesys AIL provider layer. This setting has no effect unless the log level is set to INFO for the AIL provider trace class. Once enabled, the AIL provider layer traces all incoming and outgoing work items, as well as incoming Genesys events and outgoing Genesys requests, for the specified Genesys Place name. The value for this parameter should match the Telephony Extension value which appears in the Telephony tab of the out-of-the-box User Detail View. (In the case of the AIL provider, the Telephony Extension field in the User's Telephony tab has been overloaded to take the Genesys Place name.)	**Administration > Global Settings**

Rules, Behaviors and Associations

4

Rules and behaviors enable you to customize or add logic to the Channel Manager during the implementation phase of a project. This section describes the process of adding a new piece of functionality to the system, by creating an example behavior and rule, and configuring an association to trigger them.

The example developed in this section changes the landing screen based on attached data from the middle ware.

This section assumes that you have the Channels system installed, with the Base Provider configured and functioning. Infor supplies the Base Provider so that you can demonstrate and test CTI functionality if you do not have a fully-functional provider installed. For more information, refer to "Configuring the Base Provider" on page 95.

The stages in creating new rule engine functionality are:

1. Design: In this example, we develop and implement a simple class that alters the CTI landing screen in response to information received from the middle ware, overriding the landing screen setting in the Global Settings.

   In this example, we develop and implement a simple class that alters the CTI landing screen in response to information received from the middle ware, overriding the landing screen setting in the Global Settings.

2. Planning: If the enhancement needs to interact with other parts of the system, such as data storage or the UI, determine how it will do so, and implement any necessary changes in those components.

   - Choose a location to store any data that the extension needs. In this example, we are getting data from the middleware, so there is no need to store anything.
   - Modify the UI to display the new information. In this example, we use standard UI elements, so the UI is not modified.

3. Implementation: After performing any UI modifications, create and implement the behavior.

   In this example, the behavior depends on out-of-the-box UI elements, so no UI modifications are needed

4. Configuration: Once the behavior is defined, you must create an association to link it with a rule. That rule defines the conditions under which the behavior should be executed. It is always a good practice to define the rule as precisely as possible to minimize unnecessary execution of the behavior.

After each step in this process, you should be able to compile all code changes without any errors. If you make changes to the configuration information located in the CtiManager settings in the Administration panel in Studio, you must restart the server for these changes to take effect.

Design

Attached data represents any custom-defined data attached to the call (such as an account number) that can be used to take a specific action. The behavior developed in this example uses attached data to determine which screen the agent should be presented with when a call is answered.

Planning

The example behavior needs to know the format of the attached data to be able to make sense of it. In addition, the behavior needs to set the proper values in the application data associated with the Call object, in order for the application to present the right screen to the user.

Creating the Behavior

There are two main pieces of functionality for which the example behavior will be responsible. First, it should retrieve the numeric code - used to determine the correct landing screen - from the incoming data.

Second, the behavior is also responsible for converting this code into a string that the application recognizes, and storing this screen in the application data associated with the Call object.

Creating the Behavior Class

The first step in creating a new behavior is to create a new class. For the example, the class is in the `com.epiphany.shr.cti.middleware.general.rules` package. You can use any package name for your behaviors, as long as the package name exactly matches the directory structure in which your Java code is stored.

The class must be a subclass of `CtiRuleBehaviorCommonImpl`. Also, all behaviors must implement the `CtiBehavior` interface. For this example behavior, the class name is BhvrTranslateLandingScreenData .

The behavior class consists of:

- A static map used for conversion between numeric codes and strings.
- String constants identifying a unique key in the application data, which the application looks for to determine which landing screen should be displayed to an end user.
- A universal `evaluate()` method, which is called by the rule engine
- Helper methods.

Importing Classes

Begin by importing the relevant Java classes:

```
import java.util.HashMap;
import com.epiphany.shr.cti.ctimanager.CtiManagerDataAccess;
import com.epiphany.shr.cti.ctirules.*
import com.epiphany.shr.cti.util.*;
import com.epiphany.shr.util.logging.*;
```

`java.util.HashMap` is a standard utility class supplied by the JDK.

The `cti.ctirules` package includes the `CtiBehavior` interface and the `CtiRuleBehavior CommonImpl` base class.

The `cti.util` package brings in the `CtiParameters` class.

```
private static HashMap _landingScreensForEnteredDigits =
                  new HashMap();
static
{
_landingScreensForEnteredDigits.put("1","order");
_landingScreensForEnteredDigits.put("2","product_instance");
_landingScreensForEnteredDigits.put("3","request");
_landingScreensForEnteredDigits.put("4","task");
}
```

This map defines 1:1 relationship between numeric codes received from the middle ware, and the CTI landing screen names used by the application extensions.

```
private static final String
             ATTACHEDDATAKEY_DESIRED_DETAIL_LANDING_SCREEN = "Desired
DetailLand
               ingScreen";
```

This is a simple constant variable that defines the application data key for the name of the landing screen that you want to display.

The Evaluate Method

The evaluate method performs a sanity check on the incoming data, retrieves the numeric code from the incoming data (using a helper method), converts it into the name of the landing screen, and stores the resulting screen in the application data (using another helper method).

```
public int evaluate(CtiWorkItem workItem, CtiDataAccess data)
{
if (!CtiManagerDataAccess.class.isAssignableFrom(data.getClass()))
{
_log.error("LOG_CTI_MANAGERDATAACCESS_NOT_ASSIGNABLE", "The type of
    data access object being used to evaluate this CTI manager behavior

    cannot be converted to a CTI manager type data access object (item-

     Type,behavior) ({0},{1})", SuggestedCategory.CTI, workItem.get-
     Type(), getClass().toString());

return RESULT_EVALUATE_PRECONDITION_FAIL;
}
CtiManagerDataAccess managerData = (CtiManagerDataAccess)data;

// translate CED into desired landing screen
String desiredLandingScreen = getLandingScreenFromCED(workItem,
managerData);

// set app data with desired landing screen
setLandingScreenInAppData(workItem, managerData,
desiredLandingScreen);

return RESULT_EVALUATE_NEXT_RULE;
}
```

The return value of the behavior is used by the rule engine to determine how to proceed with further rule processing. There are five possible return values. The first three control evaluation (next, stop, and first) the last two are used for integrity checking (pre-condition and post-condition). Possible return values are:

- RESULT_EVALUATE_NEXT_RULE: Go on the next association in the rule engine
- RESULT_STOP_EVALUATION: Stop evaluating rules. This should only be done in extreme circumstances when further processing should be stopped.
- RESULT_EVALUATE_FIRST_RULE: Start the evaluation at the first association in the rule engine. This is useful if the work item has been changed by this behavior and the system should start processing all over again. When using this return value, take care to avoid creating infinite loops.
- RESULT_EVALUATE_PRECONDITION_FAIL: Indicates that one of the conditions required for this rule to succeed has not been met. For example, this might be a return value for a behavior that requires an active call to place on hold, if no active call is present.
- RESULT_EVALUATE_POSTCONDITION_FAIL: Indicates that some expected result state was not achieved - for example, if an action was attempted and the result was not appropriate.

Adding Helper Methods

The first helper method is **getLandingScreenFromCED**. This method is used to retrieve the numeric code specifying the desired CTI landing screen from the data supplied by the midyear. The method returns **NULL**.

```
     protected String getLandingScreenFromCED(CtiWorkItem workItem,
CtiMan-
          agerDataAccess managerData)
     {
     // get caller entered digits (CED)
    String callerEnteredDigits = workItem.getStringParameter(CtiParame

        CtiParameters.EPNY_PARAM_CALL_CED);
     if (callerEnteredDigits == null)
     {
     return null;
     }

     String desiredLandingScreen =
            (String)_landingScreensForEnteredDigits.get(callerEntered
Digits);
     return desiredLandingScreen;
     }
```

The other helper method is setLandingScreenInAppData. This method retrieves the Call object associated with the current phone call, gets access to the application data stored in that object (creating a new instance of application data, if necessary), and adds the desired landing into the application data, using the appropriate data key. Notice the sanity checks in the method. If desired landing screen value is not valid, or Call object cannot be retrieved, the method does not do anything harmful, returning immediately.

```
protected void setLandingScreenInAppData(CtiWorkItem workItem, CtiMan
         agerDataAccess managerData, String desiredLandingScreen)
         {
         if (desiredLandingScreen == null)
         {
         return;
         }
         CtiCallObject callObj = managerData.getCallObject(workItem,
false);
         if (callObj == null)
         {
         return;
         }
         HashMap appData = callObj.getApplicationData();
         if(appData == null)
         {
```

```
          appData = new HashMap();
          callObj.setApplicationData(appData);
          }
          appData.put(ATTACHEDDATAKEY_DESIRED_DETAIL_LANDING_SCREEN,
desiredLandingScreen);
          }
```

Putting it all together, we get:

```
package com.epiphany.shr.cti.middleware.general.rules;
import java.util.HashMap;
import com.epiphany.shr.cti.ctimanager.CtiManagerDataAccess;
import com.epiphany.shr.cti.ctirules.*;
import com.epiphany.shr.cti.util.*;
import com.epiphany.shr.util.logging.*;

// Sample behavior to translate attached data from the middleware which
// specifies the desired detail landing screen.
//
public class BhvrTranslateLandingScreenData
extends CtiRuleBehaviorCommonImpl
implements CtiBehavior
{
// This reference to the logger instance allows for writing
// log statements. Log statements are useful for tracking the
// flow of execution during troubleshooting.
//
private static ILoggerCategory _log = LoggerFactory.getInstance(Bhvr-
  TranslateLandingScreenData.class;

// This is a map of desired detail landing screens for various
// possibilities of digits entered by the caller. For simplicity,
// we will assume that only one digit will be entered by the caller.
//
private static HashMap _landingScreensForEnteredDigits
= new HashMap();
   static
   {
_landingScreensForEnteredDigits.put("1","order");
_landingScreensForEnteredDigits.put("2","product_instance");
_landingScreensForEnteredDigits.put("3","request");
_landingScreensForEnteredDigits.put("4","task");
   }
// The key to the value for desired detail landing screen, as
// expected by the application extensions. The key-value pair
// is stored inside the call's application data.
//
private static final String
   ATTACHEDDATAKEY_DESIRED_DETAIL_LANDING_SCREEN = "DesiredDetailLanding
Screen";
```

```
// This is the main method of the behavior. It currently
// translates the caller entered digits into a string that
// represents the desired detail landing screen. It then sets
// be accessed later from the application extensions.
//
public int evaluate(CtiWorkItem workItem, CtiDataAccess data)
{
if(!CtiManagerDataAccess.class.isAssignableFrom(data.getClass()))
      {
_log.error("LOG_CTI_MANAGERDATAACCESS_NOT_ASSIGNABLE", "The type of
      data access object being used to evaluate this CTI manager
behavior
      cannot be converted to a CTI manager type data access object
(item-
      Type,behavior) ({0},{1})", SuggestedCategory.CTI, workItem.get-
      Type(), getClass().toString());

return RESULT_EVALUATE_PRECONDITION_FAIL;
       }
       CtiManagerDataAccess managerData = (CtiManagerDataAccess)data;

       // translate CED into desired landing screen
      String desiredLandingScreen = getLandingScreenFromCED(workItem,
managerData);
       // set app data with desired landing screen
       setLandingScreenInAppData(workItem, managerData,
desiredLandingScreen);

       return RESULT_EVALUATE_NEXT_RULE;
     }
// Extract the caller entered digits (CED) from the work item.
// Then obtain the desired detail landing screen that corresponds
// to the digits entered.
//
protected String getLandingScreenFromCED(CtiWorkItem workItem, CtiMan
   agerDataAccess managerData)
    {
       // get caller entered digits (CED)
       String callerEnteredDigits = workItem.getStringParameter(
CtiParameters.EPNY_PARAM_CALL_CED);

if (callerEnteredDigits == null)
{
return null;
}
String desiredLandingScreen =
  (String)_landingScreensForEnteredDigits.get(callerEnteredDigits);

return desiredLandingScreen;
   }
// Set the detail landing screen value inside the call's
// application data, so that it can be accessed later from
// the application extensions.
```

```
//
protected void setLandingScreenInAppData(CtiWorkItem workItem,
CtiManagerDataAccess managerData,
String desiredLandingScreen)
   {
if (desiredLandingScreen == null)
{
return;
}

CtiCallObject callObj = managerData.getCallObject(workItem,
false);
if (callObj == null)
{
return;
}
HashMap appData = callObj.getApplicationData();
if (appData == null)
{
appData = new HashMap();
callObj.setApplicationData(appData);
}

appData.put(ATTACHEDDATAKEY_DESIRED_DETAIL_LANDING_SCREEN,
desiredLandingScreen);
    }
```

You can compile the code at this point, and should not receive any errors

Configuring the Behavior in Metadata

Once the behavior is written and compiles without errors, configure the new behavior in the manager metadata.

Note: The location of the behavior must be included in your classpath, and the directory structure should match the package name that you have chosen.

To configure the behavior:

1. Open Studio and connect to the database
2. In the Guide Bar, click **Administration**.
3. Open **Services > CtiManager > Parameters**.
4. Click on **Behaviors**.
5. In the last entry in the row, set the Parameter Name to Behavior.
6. Click Save to make your changes take effect.
7. In the Guide Bar, select **CtiManager**.

8 In the Properties view, expand Behaviors, and locate the new behavior. In the entry for the new behavior, enter the behavior name. For the example, this is `BhvrTranslateLandingScreenData` .

9 The name of the behavior is the class name. The value that you enter must match the class name exactly, and is case-sensitive. Do not include any file name suffixes (such as .class or .java).

10 Expand the properties for the behavior (by clicking the + beside the behavior name). For the ImplClass property, enter the full package and class name. For the example behavior, this is: `com.epiphany.shr.cti.middleware.general.rules.BhvrTran slateLandingScreenData.`

11 The class name and package names are case-sensitive and must exactly match the class and package name defined in the behavior.

Creating the Initial Association

At this point, the behavior has been defined, but it cannot be activated until it is linked with a rule by an association. For now, we set it up using the AlwaysRun rule. This calls the behavior for every work item. Later, when all of the logic works, we can enhance the rule to filter out work items more efficiently.

The new association should be between the AlwaysRun rule and the new behavior. It should occur immediately before the forward association (the association with the highest number, which is processed last). You may need to adjust the ordering of behaviors.

Configure the new Association in the manager metadata:

1 Open Studio and connect to the database

2 In the Guide Bar, click on **Administration**.

3 Open **Services > Parameters**.

4 Click on **Associations**.

5 In the last entry in the row, set the parameter name to Association.

6 Under Value , set the list order to a value lower than the forward-work-item association (which forwards work items to their final destination, and must be processed last).

7 Click Save to make your changes take effect.

8 In the Guide Bar, click on **CTI Manager**. In the Properties view, expand Associations. A list of Associations appears, with your new entry at the bottom. Expand the new entry, and set the following properties:

- Set Behavior to BhvrTranslateLandingScreenData
- Set the Rule to RuleAlwaysRun
- Set Enabled to true

Note: When you configure associations, be careful not to remove any existing associations. If you modify an association, you must replace it with equivalent functionality. Associations are identified by a number which determines the order in which they run, so take care when changing association numbers. The forward association must always be last.

Testing the Behavior

At this point, you can log in to the server and see the initial behavior working. (If you have not configured CTI yet, refer to "Configuring the Base Provider" on page 95 for instructions for setting up the base provider.)

To see the new control:

1. Start the server.
2. Open a new browser and log in as the configured agent
3. Click the **Login** button for CTI (located in the Agent Slot on the bottom of the page).
4. The user's state should now be shown as logged in with a status of unavailable.
5. Click the **Available** button.
6. 5 seconds later, a call should come in from the configured customer.
7. Click **Answer Call**. Instead of default individual landing screen, either product_instance or request screen should appear.

Creating Rules

While the RuleAlwaysRun rule works for this use case, a more specific rule would be more efficient. Since a large number of work items go through the system, processing should be as efficient as possible. This section describes the most commonly used rule in the out-of-the-box application, RuleTypeFilter.

RuleTypeFilter fires a specified behavior for any work item that matches the configured supported types. When the system is started up, the rule reads in configuration information that determines which work items should trigger the behavior. This information is stored in a HashMap. As work items come in, the rule checks to see if the work item is listed in the supported work item types. If it is, the behavior fires; if not, the behavior is skipped.

The `RuleTypeFilter` class implements the filtering described above, and since behaviors share helper methods between implementations of logic for multiple work item types, this filter is useful for cleanly restricting the work item types for which a behavior is called.

As with the behavior example, the package used is arbitrary (`com.epiphany.shr.cti.middleware.general.rules`). You are free to use any package that you want, as long as it matches the directory structure of the project.

Like behaviors, rules are also subclasses of `CtiRuleBehaviorCommonImpl` . Rules must implement `CtiRule` . This interface only has two methods: initialize and evaluate. The initialize method is called once for every instance of the rule when the system is started up. This is a place where you can load configuration and setup common variables. The evaluate method is called for each work item that gets to the association that the rule is a part of. It decides if the behavior should be evaluated or not.

The initialize method is responsible for reading in the configuration information and setting up the HashMap of supported rules.

```
Object strTypes = param.get(PARAM_SUPPORTED_TYPES);
```

First the supported types string is retrieved. This is a comma separated list of supported types.

```
StringTokenizer stkn = new StringTokenizer(strTypes.toString(), ",");

HashMap translationClassEnumMap = (HashMap) allTypes;
```

As each token is read, the system looks to see if there is a mapping available for that token. This allows the use of variable names when defining this list. We use enumeration interfaces to allow the variables to differ from the values. If it becomes necessary to change the runtime value, all we have to do is make the change in one place.

```
Object suppValue = translationClassEnumMap.get(nextType);

if (suppValue != null)
             _wiTypes.put(suppValue, suppValue);

else
             _wiTypes.put(nextType, nextType);
```

If a token enumeration entry is found that matches the string of the supported type, then the translated value is entered.

Now that the HashMap has been created, the evaluation method is simple. All the evaluate method has to do is check to see if the type of the work item being evaluated matches one of the supported types in the HashMap.

```
public int evaluate(CtiWorkItem workItem, CtiDataAccess data)
{

int retV = RESULT_SKIP_BEHAVIOR;

Object mapVal = _wiTypes.get(workItem.getType());

if (mapVal != null)
```

```
retV = RESULT_RUN_BEHAVIOR;
return retV;
}
```

Below is the complete source code for RuleTypeFilter. It is a simple and powerful rule, as well as a good example of the kind of thing that can be done with the rule engine.

```
package com.epiphany.shr.cti.middleware.general.rules;
import java.util.*;
import org.apache.commons.collections.FastHashMap;
import com.epiphany.shr.cti.ctirules.CtiRule;
import com.epiphany.shr.cti.ctirules.CtiRuleBehaviorCommonImpl;
import com.epiphany.shr.cti.util.CtiDataAccess;
import com.epiphany.shr.cti.util.CtiWorkItem;
import com.epiphany.shr.util.exceptions.EpiException;
import com.epiphany.shr.util.logging.ILoggerCategory;
import com.epiphany.shr.util.logging.LoggerFactory;
// This rule filters incoming work items by work item type. Work items
// of types, specified in filtering condition, are to be passed to the
// associated behavior, and all other work items are to be blocked.
// Parameter "supportedTypes" (the only parameter for this rule)
// allows to specify multiple types, separated by commas. Parameter
// parser is intelligent enough to recognize constants from
// CtiWorkItemTypes enumerations and resolve them to
// associated values.  However, at this point, custom user-defined
// enumerations are not supported (although later we may add ability
// to define list of additional interfaces, containing custom
```

```
// enumerations).
//
public class RuleTypeFilter
    extends CtiRuleBehaviorCommonImpl
    implements CtiRule
{
// Work item filtering condition (list of types that are
// supposed to be passed to associated behavior).
//
protected FastHashMap _wiTypes = new FastHashMap();
public static final String PARAM_SUPPORTED_TYPES = "supportedTypes";
public static final String PARAM_ADDITIONAL_TRANSLATION_CLASSES =
        "additionalTranslationClasses";
    // ------------------ CtiRule implementation start ---------------
public boolean initialize(String name, Map param, Map allTypes)
{
boolean retV = super.initialize(name, param, allTypes);
try
{
if (retV)
{
Object strTypes = param.get(PARAM_SUPPORTED_TYPES);
if (strTypes != null)
{
StringTokenizer stkn =
                        new StringTokenizer(strTypes.toString(), ",");
```

```
HashMap translationClassEnumMap = (HashMap) allTypes;
while (stkn.hasMoreTokens())
{
String nextType = stkn.nextToken();
Object suppValue = translationClassEnumMap.get(nextType);
// if we found a value for the specified key,
// add this value to our list of work items,
// otherwise add the key itself (this is not
// a key, but a directly specified value)
//
if (suppValue != null)
_wiTypes.put(suppValue, suppValue);
else
_wiTypes.put(nextType, nextType);
}
}
}
}
catch (Throwable boo)
{
_log.error(
             new EpiException(
                    "EXP_CTI_RULETYPEFILTER_INITIALIZE",
                    "Exception in RuleTypeFilter.initialize()",
                    boo));
retV = false;
}
```

```
return retV;

}

public int evaluate(CtiWorkItem workItem, CtiDataAccess data)

{

int retV = RESULT_SKIP_BEHAVIOR;

Object mapVal = _wiTypes.get(workItem.getType());

if (mapVal != null)

retV = RESULT_RUN_BEHAVIOR;

return retV;

}


protected ILoggerCategory _log = LoggerFactory.getInstance(getClass());

}
```

You can compile the code at this point, and should not receive any errors.

Configuring the Rule in Metadata

Once code for the rule is complete, it needs to be configured in the metadata This process is similar to creating the behavior earlier, and just provides the name of the rule, the class, and any configuration parameters.

To configure the new rule in the manager metadata:

1. Open Studio and connect to the database.
2. In the navigation bar, click on Administration.
3. Open **Services > CtiManager > Parameters**.
4. Click on **Rules**.
5. In the last entry in the row, set the parameter name to Rule. Click on a different row to make your changes take effect.
6. In the Guide Bar, click on **CtiManager**.
7. In the Properties view, locate the new rule (it should be the last entry) and expand it. Set the following properties:

- Beside the rule entry, enter the name of the rule, RuleIncomingCall.
- Beside ImplClass, enter the full package and class name for the rule. For this example, it is: `com.epiphany.shr.cti.middleware.general.rules.RuleTypeFilter`
- Beside the first parameter, enter **supportedTypes**.
- Set the value of the first parameter to be a comma separated list of supported event types. In the case of this example, enter the following type
- CALL_RINGING
- For a list of the out-of-the-box types, refer to the Javadocs for `com.epiphany.shr.cti.util.CtiWorkItemTypes`.

Note: Enter the name of the event exactly as it appears. Do not enclose the list in quotes or include spaces between entries.

Updating the Association

Now that the rule is defined, we need to update the association that we created earlier

To reconfigure the association:

1. Open Studio and connect to the database
2. In the Guide Bar, click on Administration.
3. Open **Services > CtiManager > Parameters**.
4. Click on **Associations**.
5. Locate and expand the Association that you created earlier.
6. In the Properties view, change the Rule from RuleAlwaysRuns to **RuleIncomingCall**.

Testing the New Logic

The example behavior should now be fully-functional. At this point, we can restart the server and test the new logic. One scenario is answering a call via a pop-up. Below are the instructions for testing that scenario.

To see the new control:

1. Start the server.
2. Open a new browser and log in as the configured agent
3. Click the **Login** button for CTI (located in the Agent Slot on the bottom of the page).
4. The user's state should now be shown as logged in with a status of unavailable.
5. Click the **Available** button.
6. 5 seconds later, a call should come in from the configured customer.
7. Click **Answer Call**. Instead of the default individual landing screen, either the product instance screen or the request screen should appear.

8 Hang up.

9 Click the **Unavailable** button.

10 Repeat steps 4-7 several times.

Writing a Provider

5

The Infor Channels system is designed to be easily extended. It enables you to connect to middle ware systems that Infor does not support out-of-the-box by writing your own provider (using the API that Infor uses internally to write providers). While this is a very powerful option, it should not be undertaken lightly. CTI providers have to be fast, capable of handling large loads, and stable under error conditions.

This section assumes that you have read and followed the Design and Planning sections starting on "page 4–2" on page 34. The information in that section will be useful when testing the system and customizing the system to handle your new provider, and provides good background material for the event handling that is required in a building a provider.

Before writing your own provider, contact Infor to verify that we are not already working on the one you need, and to let us know what you are trying to do. Custom providers are typically supported through the partner that constructed them instead of directly by Infor.

The CTI system was designed to support providers with varying levels of functionality. If there is logic that you do not or cannot support, it can be removed. If there are extra features that you would like to support, they can be quickly added throughout the system.

Getting Started

Once you have decided to write a new provider, you should:

- Verify that you have the latest copy of this SDK.
- Get API documentation for connecting to the middle ware system you are supporting.
- Review "Design" on page 34.
- Review this entire section for instructions, tips, advice, and sample code.

Design Considerations

The easiest way to design your provider is to use the same style as the Infor-supplied providers. This means that:

- Your provider should subclass the following class: `com.epiphany.shr.cti.middleware.general.provider.BaseProvider`
- Your provider should use rules and behaviors to implement logic.
- Almost all processing should be done asynchronously. There is one thread for processing provider work items. Since there is only one thread, it is very important that you do not tie up that thread for a long period of time. You should not do anything in a rule or behavior that will have a significant delay or take a long time. If there is a section of code that takes a long time to complete, your provider will become non-responsive while it is processing that item. If the delays are long, users will notice delays and complain.

The only true requirement for writing a provider is that your class must implement `com.epiphany.shr.cti.middleware.general.provider.CtiProvider`, and honor the work item types and parameters. The system is capable of having all of the work item types completely customized in the field. There is no requirement that the work item processor model be followed.

The work item processor model was designed with a few key concepts in mind:

- Performance and scalability: An asynchronous communications model enables us to process at high rates of speed without waiting for other components and simplifies the locking logic. (Only the processing thread has access to the internal data.)
- The ability to query information from a work item processor in real-time. The Base Provider implements a queuing mechanism that blends real-time blocking requests and bulk processing.
- Extensibility: The work item processor model is a rule based system that supports full customization of rule types, behaviors, and associations (evaluation order).

Building Out The Components

We recommend that you build providers in the same order in which the samples are constructed. The samples are built in the way that we internally build providers, enabling good project tracking and quick initial results, and start with the easiest piece so that you can get familiar with the APIs involved.

For a CTI provider we typically:

- Create the implementation class
- Establish communication with the middleware system
- Implement and test agent control:
  - Login
  - Logout
  - Change state
- Implement and test basic call control:
  - Call
  - Pickup
  - Answer

  - Hold
  - Retrieve
  - End call
- Implement and test advanced call control:
  - Mute transfer
  - Assisted transfer
  - Assisted conference
  - Cancel consult
  - Compete conference
  - Complete transfer
- Perform full functionality tests
- Perform cluster testing
- Perform stability tests
- Perform scalability tests

Expected Request and Event Flow

When your provider sends events from the telephony middle ware provider to the Channels Manager in the form of work items, the Channels Manager has very specific expectations regarding which work items should be sent.

The order of events varies depending on the use case involved. For instance, if the agent attempts to dial out using the teleset or using the browser, the Channels Manager expects the provider to send a CALL_DIALING work item. However, if the agent attempts to transfer a call to another agent, the Channels Manager expects the provider to send a CALL_CONSULTATION_INITIATED work item rather than the CALL_DIALING event you might expect.

For more information about work item types, refer to the JavaDocs for the

```
com.epiphany.shr.cti.util.CtiWorkItemTypes class.
```

This section lists the request and event work items that your provider implementation is expected to handle. The list is grouped by functionality, and lists each request work item, followed by the event work item that the Channels Manager expects to receive as an asynchronous response

Request Work Item	Expected Event Work Item
LOGIN_AGENT	AGENT_LOGGED_ON (or PARTY_ERROR)
SET_AGENT_STATE	AGENT_MODE_CHANGED (or PARTY_ERROR)
LOGOUT_AGENT	AGENT_LOGGED_OUT (or PARTY_ERROR)

Request Work Item	Expected Event Work Item
MAKE_CALL	CALL_DIALING or CALL_DESTINATION_BUSY (or CALL_ERROR)
ANSWER_CALL	CALL_ESTABLISHED (or CALL_ERROR)
HOLD_CALL	CALL_HELD (or CALL_ERROR)
RETRIEVE_CALL	CALL_RETRIEVED (or CALL_ERROR)
HANGUP_CALL	CALL_DISCONNECTED (or CALL_ERROR)
PICKUP_CALL	• CALL_RINGING (or CALL_ERROR) • CALL_ESTABLISHED, if preceded by CALL_RINGING Only required if your switch supports pickup functionality.

Request Work Item	Expected Event Work Item
CANCEL_CALL	CALL_DISCONNECTEDfor the consultation call (or CALL_ERROR)
SINGLE_STEP_TRANSFER	CALL_HELD for the original call (or CALL_ERROR). CALL_CONSULTATION_INITIATED , if CALL_ERROR did not occur. CALL_PARTY_CHANGED, if CALL_ERROR did not occur. CALL_TRANSFERRED , if CALL_ERROR did not occur.
INITIATE_TRANSFER	CALL_HELDfor the original call (or CALL_ERROR). CALL_CONSULTATION_INITIATED , if not preceded by CALL_ERROR .
COMPLETE_TRANSFER	CALL_PARTY_CHANGED (or CALL_ERROR). CALL_TRANSFERRED , if not preceded by CALL_ERROR .
INITIATE_CONFERENCE	CALL_HELD for the original call (or CALL_ERROR). CALL_CONSULTATION_INITIATED , if not preceded by CALL_ERROR .
COMPLETE_CONFERENCE	CALL_CONFERENCED (or CALL_ERROR) CONFERENCE_INFO , with list of extensions currently conferenced

Request Work Item	Expected Event Work Item
HANGUP_CALL	CALL_DISCONNECTED or CALL_CONFERENCE_PARTY_REMOVED (or CALL_ERROR) This request/event work item pair should also be implemented under basic call control functionality. However, the implementation must be expanded in the context of advanced call control. If the hang-up request is submitted by an agent currently involved in a conference, then the Channels Manager expects to receive CALL_CONFERENCE_PARTY_REMOVEDinstead of CALL_DISCONNECTED , unless the agent is the last party remaining in the conference with the customer, in which case it expects CALL_DISCONNECTED .
ASSOCIATE_DATA	CALL_INFORMATION (or CALL_ERROR). This pair is required to make use of Infor CTI features. For more information, refer to “Handling ASSOCIATE_DATA ” below.

Handling ASSOCIATE_DATA

The ASSOCIATE_DATA / CALL_INFORMATION request/event work item pair are normally optional for a custom provider implementation. However, if you are implementing advanced call control functionality, and you want to take full advantage of the capabilities of Infor CTI solution, you should handle these work items.

This request/event work item pair represents the mechanism used to pass the customer context from one agent to another, especially as it relates to a multi-site call center. When the Channels Manager sends ASSOCIATE_DATA to the provider, the work item contains a HashMap parameter called EPNY_PARAM_APPLICATION_DATA. Within this parameter, there is (at minimum) one key-value pair where the key is Customer_ID and the value is a String representing the customer context for the current call. Your provider is expected to send back a CALL_INFORMATION work item, which should contain the exact same key-value pair within the EPNY_PARAM_APPLICATION_DATA work item parameter, which provides confirmation that the middle ware successfully attached this data to the call.

If the original agent performs a transfer or conference across call center sites, the middle ware can use this parameter to pass the customer context along with the inbound consultation call event when it occurs at the recipient’s desk.

Configuring Your Provider

Once you have created the new provider, you need to configure it in Studio. For more information, refer to.

When You Are Done

Before going to production deployment, the system should be capable of running for days with twice the expected load without errors. This is a good indication that your system is capable of handling the load placed on it, and is reliable in the production environment. During your tests, you should have as complete and realistic test of the system as you can. This should include running the test using the actual hardware that would be used in a deployment if at all practical. For a CTI provider, testing just the CTI component should be sufficient. In an implementation of a system, however, you should test CTI with all of the other subsystems running under normal load.

Things to Look Out For

When developing a provider, take note of the following:

- Errors and warnings are usually an indication that you are doing something wrong. When working on a provider and running tests, make sure they are free of errors and warnings.
- When testing the provider, note whether the UI update time is consistent. If an action takes a long time sometimes and less time others, or if some actions take much longer than others, there is a change that there is a rule or behavior in the provider that can take a long time to run. You should profile the provider and determine where the time is spent.
- Make sure the application flow matches your business needs. The application should conform to how you want it to run. Go through your scenarios and make sure they work the way they should.
- It is a good idea to watch the CPU load of a CTI-only system test. For a CTI test without a running UI, if the expected number of users is simulated, the CPU load should not be more than a few percent. If it is significantly higher than that, then you may need to optimize the CTI setup, or add more hardware.

Writing A Provider

In this section, we will construct a simple version of the base provider. The base provider is capable of handling agent control, and simple call control. Initially it will not support transfers, conferences, or cluster mode (all instances will run independently). As with the example developed in Design and Planning sections starting on "Design" on page 34, we will add additional functionality incrementally.

The example provider is designed to demonstrate the basic concepts involved in creating a provider. It is the "Base" provider from which all of our other providers are derived. In the process of implementing a provider, the functionality of the sample provider is overridden.

Work Items and System Flow

The typical flow of the system is as follows: a request comes in from the application, goes through the service and channel manager and arrives at the provider's queue. When the provider processes this item, where a real provider would normally make a call out to the middleware system, the base provider creates the corresponding response work item(s) itself.

The provider has two options for what to do with the new work items. If a delay is required, there is a simple inner class that will send the work item after a 5 second delay. This will be used to send calls to agents after 5 second pauses. To add more realism, you could customize the base provider to create a new thread with each request, add a random sleep statement, and true asynchronous non-deterministic behavior would result. Since the system routinely works in a truly asynchronous mode, developing this functionality is not a priority for the base provider.

When the worker thread pulls the item off the queue, normal work item processing begins. While most of the base provider logic involves triggering events in response to requests, the handling of incoming calls is a special case. Incoming calls are one of the truly asynchronous parts of the channels system. To simulate traffic in a this simple simulated environment, incoming calls will arrive at a regular interval when the user is logged in and available.

Since there is no phone present, hardware-driven events (such as physically pressing the phone’s Hold button) are not supported in the base provider.

Implementing the Sample Provider

Creating the Base Class

The first step in creating a provider is creating the initial base class. This main class will be the one that the CTI Manager interfaces with. It is responsible for receiving all of the work items from the CTI Manager. Once the work items are received, they can be processed any way that you want.

CTI Providers must implement the `CtiProvider` interface. This is used to communicate with the CTI Manager. Aside from this implementation, you can use any mechanism for implementing the provider logic. Infor uses the WorkItemProcessor functionality. All of our providers are built based on this class, and our metadata is configured for this class. It should help you get started and help you avoid many of the performance and threading issues that are commonly encountered in working on this type of code.

This section assumes that the provider is a subclass of BaseProvider. It includes some logic that extends what the base provider itself provides. This provider is intended for demonstration and testing purposes.

The first thing to do is get the base class created with the key methods stubbed out. Since this provider does not require much flexibility, most of the logic in the example is hard-coded. Typically, when developing a production provider you would want to build as much flexibility as is required into the rules and behaviors, which can be configured without modifying the provider code.

In this example, we construct the provider logic in an Infor package, `com.epiphany.shr.cti.middleware.general`. The class will be `BaseProvider`. It implements `CtiProvider`, and should be used as the parent class for custom providers.

You can use any name or package for your implementation, provided that the directory and package structure are consistent with each other. The base implementation with the method stubs appears below.

```
protected static final Object[] _keysToCopy =
{
   CtiParameters.EPNY_PARAM_AGENT_ID,
   CtiParameters.EPNY_PARAM_AGENT_USER_ID
};
protected static final Object[] _keysToCopyWithDN =
{
   CtiParameters.EPNY_PARAM_AGENT_ID,
   CtiParameters.EPNY_PARAM_AGENT_USER_ID,
   CtiParameters.EPNY_PARAM_PARTY_NUMBER
};
```

These two data structures are used to specify standard parameters to copy. Since all of the generated work items in this provider are generated from copying key attributes from the incoming work items we have created variables to store the keys that should be copied. For work items that should include the DN, we use _keysToCopyWithDN. For all other work items, we use _keysToCopy.

In addition to the two listed above, the following method stub is also required. You do not need to add anything to it.

```
public boolean isValidatingAgentIDs()
return checkAgentID;
```

Helpers

These three methods are helpers that simplify the processing for individual methods. All three of the helper methods use the original work item as source material. Each creates a different type of initial work item.

- `createAppWorkItem` creates work items that copy the primary parameters with DN. It also sets the destination of the work item to be the application. This ensures that after the manager processes it, it will be sent on to the application.

- `createSampleAgentStateChange` is used to create agent state change events. The only parameters that you need to pass in are the original work item, and the new state that you want for the agent. The system then creates the agent state change event work item and gets it ready for sending to the manager.
- `createSampleCall` is used to generate new call work items. It creates ringing work items that have 5551212 as the default phone number. These are useful for creating fake calls.

These three methods together make the code for handling events much smaller. They are a good way to get some of the implementation details out of the event handling code, which makes the handler code cleaner.

```
protected CtiWorkItem createAppWorkItem(CtiWorkItem item, String type)
{
  CtiWorkItem result = createWorkItem(type);
  result.copyParameters(item, _keysToCopyWithDN);

  result.getDestination().setLocationType
   (CtiLocation.LOCATION_TYPE_APPLICATION);

  return result;
}
protected CtiWorkItem createSampleAgentStateChange(CtiWorkItem item,
     String agentState)
{

   CtiWorkItem makeUnavailable = createWorkItem(CtiWorkItem-
    Types.AGENT_MODE_CHANGED);


   makeUnavailable.copyParameters(item, _keysToCopy);
   makeUnavailable.setParameter(CtiParameters.
    EPNY_PARAM_AGENT_MODE, agentState);


   makeUnavailable.setParameter(CtiParameters.
    EPNY_PARAM_AGENT_GROUP, AGENT_SPLIT);


   makeUnavailable.getDestination().setLocationType(CtiLocation.
    LOCATION_TYPE_APPLICATION);

  return makeUnavailable;
}
protected CtiWorkItem createSampleCall(CtiWorkItem item)
{
 CtiWorkItem newCall = createWorkItem(CtiWorkItemTypes.CALL_RINGING);

 newCall.copyParameters(item, _keysToCopyWithDN);
 newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_ID,"callid1");
 newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_ANI,"5551212");
 newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_DIALED_NUMBER,
```

```
  "800-555-1212");

newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_DNIS, "800-555-
  1212");

 newCall.getDestination().setLocationType(CtiLocation.
  LOCATION_TYPE_APPLICATION);

 return newCall;
}
```

Handling the Work Item

The main logic of the base provider is in the receive work item method. We will go through each of the pieces in the appropriate sections. This method is copied in its entirety below, for reference.

```
// Process incoming work item (from manager or service) here
// Default implementation unconditionally reply on the
// following requests:
// EPNY_AGENT_LOGIN -> CTI_AGENT_LOGGED_ON
// EPNY_AGENT_LOGOUT -> CTI_AGENT_LOGGED_OFF
//
protected void receiveWorkItem(CtiWorkItem item)
{
  String iType = item.getType();


String agentUserID = item.getStringParameter(CtiParameters.
  EPNY_PARAM_AGENT_USER_ID);
if (iType.equals(CtiWorkItemTypes.LOGIN_AGENT))
{
  proceedRawWorkItem(item);

  CtiWorkItem reply = createWorkItem(CtiWorkItem-
     Types.AGENT_LOGGED_ON);
             reply.copyParameters(item, _keysToCopy);


  reply.getDestination().setLocationType(CtiLocation.
      LOCATION_TYPE_APPLICATION);

   processWorkItem(reply);
   // This is used for demoing a base provider. It creates a
   // fake call after the user is made available.
   CtiWorkItem makeUnavailable = createSampleAgentStateChange(item,
     AgentStates.CTI_AGENT_NOT_READY);
```

```
   processWorkItem(makeUnavailable);
}
else if (iType.equals(CtiWorkItemTypes.LOGOUT_AGENT))
{
proceedRawWorkItem(item);

   CtiWorkItem reply = createWorkItem(CtiWorkItem-
       Types.AGENT_LOGGED_OFF);
         reply.copyParameters(item, _keysToCopy);

   reply.getDestination().setLocationType(CtiLocation.
      LOCATION_TYPE_APPLICATION);
         processWorkItem(reply);
}
else if (iType.equals(CtiWorkItemTypes.SET_AGENT_STATUS))
{
   if (item.getParameter(CtiParameters.EPNY_PARAM_AGENT_MODE) ==
        AgentStates.CTI_AGENT_READY)
    {
    // Create a new incoming call pop event
    CtiWorkItem newCall = createSampleCall(item);

    CtiWorkItem makeAvailable = createSampleAgentStateChange(item,
       AgentStates.CTI_AGENT_READY);


        processWorkItem(makeAvailable);

        Thread sendCall = new DemoPopThread(this, newCall, null);
        sendCall.start();
    }
    else
    {
     // reflect back the request.
     CtiWorkItem stateChange = createSampleAgentStateChange(item,
      (String)item.getParameter(CtiParameters.
       EPNY_PARAM_AGENT_MODE));

       processWorkItem(stateChange);
     }
}
else if (iType.equals(CtiWorkItemTypes.MAKE_CALL) ||
         iType.equals(CtiWorkItemTypes.PICKUP_CALL))
{
    // This is the signal that there the users wants an
    // immediate call.
    CtiWorkItem newCall = createSampleCall(item);
    newCall.setType(CtiWorkItemTypes.CALL_DIALING);

    // The phone number, if provided, should be used. Availability
    // is not modified.
    newCall.setParameterIfNotNull(CtiParameters.EPNY_PARAM_CALL_ANI,
      item.getParameter(CtiParameters.EPNY_PARAM_CALL_ANI));
```

```
    // Send the item for immediate dispatch
    processWorkItem(newCall);


    CtiWorkItem established = createAppWorkItem(item, CtiWorkItem-
      Types.CALL_ESTABLISHED);

    processWorkItem(established);

    CtiWorkItem makeBusy = createSampleAgentStateChange(item, Agent-
      States.CTI_AGENT_ON_CALL);

    processWorkItem(makeBusy);
}
else if (iType.equals(CtiWorkItemTypes.ANSWER_CALL))
{

    CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
      Types.CALL_ESTABLISHED);

    processWorkItem(result);


    CtiWorkItem makeBusy = createSampleAgentStateChange(item, Agent-
     States.CTI_AGENT_ON_CALL);

    processWorkItem(makeBusy);
}
else if (iType.equals(CtiWorkItemTypes.HOLD_CALL))
{


    CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
      Types.CALL_HELD);
    processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.RETRIEVE_CALL))
{

    CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
      Types.CALL_RETRIEVED);
    processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.HANGUP_CALL))
{
     CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
        Types.CALL_DISCONNECTED);

     processWorkItem(result);


     result = createSampleAgentStateChange(item, Agent-
```

```
        States.CTI_AGENT_AFTER_CALL_WORK);


      processWorkItem(result);
}
}
```

This method is basically a big switch statement based on the type of work item passed in. Each section is responsible for sending the response work item. Since this provider is very simple, all of the responses for each event can be composed from the information in the incoming work item. While this example does not do anything that is useful in a practical sense, it does serve as a convenient example of the kind of flow that the manager is expecting. It provides a good template for testing application changes and allows the provider developer to see what events should be generated. Later, we will go through the individual pieces of the handler code.

DemoPopThread

DemoPopThread is an inner class. It is a helper class that is used to create a delay when firing events. It is designed to take two events and the base provider. After 5 seconds, it sends the first event to the base provider. Ten seconds after that, it sends the second event (if any) to the base provider.

```
class DemoPopThread extends Thread
{
  DemoPopThread(BaseProvider base, CtiWorkItem firstItemToSend,
       CtiWorkItem secondItemToSend)
 {
  this.base = base;
  this.firstItemToSend = firstItemToSend;
  this.secondItemToSend = secondItemToSend;
 }

  BaseProvider base;
  CtiWorkItem firstItemToSend;
  CtiWorkItem secondItemToSend;


  public void run()
  {
    try
    {

     sleep(5000);
    }
    catch (Exception ex)
    {
    }

    base.processWorkItem(firstItemToSend);
```

```
    if (secondItemToSend != null)
    {
     try
      {
        sleep(10000);
     }
     catch (Exception ex)
     {
     }

     base.processWorkItem(secondItemToSend);
     }
   }
}
```

Login Methods

The login methods are intended to be used by provider implementations to track the agents that are either logged into the provider, or are in the process of logging in to the provider. These methods abstract away the tracking responsibility, and enable you to check whether users are currently logged in, or are in the process of logging in.

- `beginLogin`

  - Call when an agent login process is started

- `commitLogin`

  - Call when an agent login process is complete

- `commitLogout`

  - Call when an agent logout process is complete

- `isLoggedIn`

  - Determine if a specified user is logged into this provider

- `isLoginPending`

  - Determine if a specified user is in the process of logging in

- `registerDnAgentLogin`

  - This adds an agent to the DN -> agent mapping

```
// Adds agent to the _myAgents map
protected void beginLogin(Object agentID, Object agentUserID)
{
  _pendingLogins.put(agentID, agentUserID);
}

// Adds agent to the _myAgents map
protected void commitLogin(Object agentID)
{

  Object agentUserID = _pendingLogins.remove(agentID);
  _myAgents.put(agentID, agentUserID);
_log.info("LOG_CTI_BASEPROVIDER_NEWLOGIN", "Agent is logging on
    (agent,user,provider)({0},{1},{2})", SuggestedCategory.CTI, agent
ID,
    agentUserID, _type);
}

// Removes agent from the _myAgents map
protected void commitLogout(Object agentID)
```

```
{
  Object agentUserID = _myAgents.remove(agentID);
   if (agentUserID != null)
{
  _log.info("LOG_CTI_BASEPROVIDER_NEWLOGOUT", "Agent is logging out
   (agent,user,provider)({0},{1},{2})", SuggestedCategory.CTI, agent
ID,
      agentUserID, _type);
}
}

// Checks whether the agent is in the _myAgents map
protected boolean isLoggedIn(Object agentID)
{
return _myAgents.containsKey(agentID);
}

// Checks whether the agent is in the _pendingLogins map
protected boolean isLoginPending(Object agentID)
{
return _pendingLogins.containsKey(agentID);
}
// Register DN -> Agent Login mapping
protected void registerDnAgentLogin(Object DN, Object agentLogin)
{
   _dnAgentLogins.put(DN, agentLogin);
   _log.info("LOG_CTI_BASEPROVIDER_AGENT_DEVICE_MAPPING", "Agent-Device

    mapping is registered (agent,device,provider)({0},{1},{2})",
SuggestedCategory.
    CTI, agentLogin, DN, _type);
}
protected Object getAgentLoginByDN(Object DN)
{
    return _dnAgentLogins.get(DN);
}
```

proceedWorkItem

```
proceedWorkItem
```

is a main helper method which provides the structure for calling other work item methods that are intended to be overridden. All work items go through this method when they are pulled off the queue. You can override this method if necessary.

```
// Helper function called by worker thread
```

```
//
protected void proceedWorkItem(CtiWorkItem item) throws EpiException
{
   try
   {
     _log.info("LOG_CTI_PROVIDER_PROCEED_WORK_ITEM", "CTI Provider
       proceed work item {0}", SuggestedCategory.CTI, item.get-
         Type());

     // For work items, generated in provider (raw items) we
     // do proceed raw item, first then, calling overloadable
     // convertWorkItem method that should prepare another work
     // item to be forwarded to the CTI Manager (it can return
     // reference to the same raw work item, or return null
     // if we decide to cancel sending work item to the manager.
     // For all other items - we proceed here and don't send anything


     if (item.getSource().getLocationType() != null &&
item.getSource().getLocationType().equals(CtiLocation.
LOCATION_TYPE_PROVIDER))
     {

        proceedRawWorkItem(item);
        CtiWorkItem wiToSend = convertWorkItem(item);

        if (wiToSend != null)
        {
           wiToSend.getDestination().setLocationType(CtiLocation.
            LOCATION_TYPE_APPLICATION);

           _mgr.processWorkItem(wiToSend);
         }
      }
      else
      {
      receiveWorkItem(item);
      }
}
catch (Throwable excpt)
{
 _log.error("LOG_CTI_PROVIDER_CRITICAL_ERROR", "unhandled exception
  in CtiProvider thread {0}", SuggestedCategory.CTI,
    Thread.currentThread().getName());


throw new EpiException("
    EXP_CTI_PROVIDER_CRITICAL_ERROR_EPIWRAPPER", "Exception
      in CtiProvider", excpt);
   }
 }
```

Handling the Work Item

The work item methods are at the heart of the processing structured in

```
proceedWorkItem:
```

- `proceedRawWorkItem`

  - used for handling raw work items from the middle ware system.

- `convertWorkItem`

  - used as a placeholder to allow for translating the work item before it is sent on to the manager. This is a good place to change codes, look up error strings, or convert parameters to the manager definition.

- `forwardWorkItem`

  - sends the work item on to the manager.

  All of these methods are protected, allowing for customization.

```
public void forwardWorkItem(CtiWorkItem item)
{
    try
    {
     // possibly, more complicated logic should be there, but by
     // default - just send the item to the manager
    _mgr.processWorkItem(item);
    }
    catch (Throwable boo)
    {
       _log.error(new EpiException("EXP_BASEPVD_FORWARD_WORKITEM",
         "Exception in BaseProvider.forwardWorkItem()", boo));
    }
}

     // This method is to be called by worker thread when the work
     // item is just retrieved from the queue. The default
implementation
    // specifically processes CTI_AGENT_LOGGED_ON and CTI_AGENT_LOGGED_
OFF
    // events, and calls commitLogin() and commitLogout() respectively

     //
    protected void proceedRawWorkItem(CtiWorkItem item)
```

```
    {
      String iType = item.getType();
      Object agentID = item.getParameter(CtiParameters.
         EPNY_PARAM_AGENT_ID);

      if (iType.equals(CtiWorkItemTypes.AGENT_LOGGED_ON))
      {
          commitLogin(agentID);
      }
      else if (iType.equals(CtiWorkItemTypes.AGENT_LOGGED_OFF))
      {
         // commitLogout is going to destroy agentID - agentUserID
         // for this agent, therefore, we need to retrieve
         // agentUserID now
         Object agentUserID = _myAgents.get(agentID);
         if (agentUserID != null)
            item.setParameter(CtiParameters.EPNY_PARAM_AGENT_USER_ID,
                   agentUserID);


          commitLogout(item.getParameter(CtiParameters.
              EPNY_PARAM_AGENT_ID));
          }
        }
        // Defines additional conversion of workitem before sending
        // it to the manager.
        //
        protected CtiWorkItem convertWorkItem (CtiWorkItem item)
        {
         // default implementation - add agent user ID, based on agent
 ID
          Object agentID = item.getParameter(CtiParameters.
            EPNY_PARAM_AGENT_ID);


          Object agentUserID = _myAgents.get(agentID);

           if (agentUserID != null)
               item.setParameter(CtiParameters.EPNY_PARAM_AGENT_USER_
ID, agentUserID);

    return item;
}
```

Testing The Initial Version

At this point, you can configure the base provider in Studio. For help with this step, refer to “Configuring the Base Provider”"Configuring the New Provider" on page 92. Once the example provider is configured to be the default provider, you must start or restart the server for the changes to take effect.

Adding Agent Control

Now that the basic class is complete, we need to create the logic to handle agent control. This logic should be able to handle the events and request for login, logout, and state change. The base handler logic does not do much more than just send the response event thought the normal channel simulating the provider response.

Three important work items for agent control are:

- Login
- Logout
- State Change

Login requests should respond with the logged in event, and logout requests should respond with the logged out event. Also, when an agent logs in, the provider should send a state change event to make them Not Available. In general, the provider is responsible for making sure the event flow matches what the CTI Manager is expecting. If the event flow is different for your provider, you either need to fix it to conform, or change the CTI Manager to handle your modified event flow.

One area where the events may not be intuitive is the agent state events. Some providers and switches provide very verbose events, while others do not. For instance, in most applications agents are considered busy when they are on the phone with a customer. Some middleware systems send an agent state change event when the call is answered to change the agent's state to busy. Other providers (such as Genesys JTele) require the provider to create the agent state event and fire it at the correct time. In their interface, some agent state changes are inferred from events. An example of this is an agent transitioning to the busy state when a call is answered.

After handling login and logout requests, we will implement State Change. State change requests trigger state changed responses. Also, if the state change is to Available, then a call should be assigned to the agent.

Login and Logout

For agent login, we call `proceedRawWorkItem`. This method automatically handles the login processing. We then create a new application work item showing that the agent has logged in. After the agent login is processed, we send an agent not ready event. This will update the current agent state in the UI.

```
if (iType.equals(CtiWorkItemTypes.LOGIN_AGENT))
{
    proceedRawWorkItem(item);

    CtiWorkItem reply = createWorkItem(CtiWorkItem-
      Types.AGENT_LOGGED_ON);

    reply.copyParameters(item, _keysToCopy);
    reply.getDestination().setLocationType(CtiLocation.
      LOCATION_TYPE_APPLICATION);
```

```
    processWorkItem(reply);

    CtiWorkItem makeUnavailable = createSampleAgentStateChange(item,
      AgentStates.CTI_AGENT_NOT_READY);

processWorkItem(makeUnavailable);
}
```

Agent logout is similar. proceedRawWorkItem handles the automatic logout processing. An agent logged off work item is then sent to the manager.

```
else if (iType.equals(CtiWorkItemTypes.LOGOUT_AGENT))
{

   proceedRawWorkItem(item);

   CtiWorkItem reply = createWorkItem(CtiWorkItem-
     Types.AGENT_LOGGED_OFF);

  reply.copyParameters(item, _keysToCopy);
  reply.getDestination().setLocationType(CtiLocation.
   LOCATION_TYPE_APPLICATION);

   processWorkItem(reply);
}
```

State Change

The set status work item is used to change the agents status in the middleware. Because this system does not have complicated logic for generating incoming calls, we trigger a simple delayed call in this event. If the agent state is being set to ready, we create a sample call and send it on a separate thread with a delay. In the meantime, we make the agent state available and send that right away. For all other states, we just send a state event saying that the agent has changed to the requested state.

```
else if (iType.equals(CtiWorkItemTypes.SET_AGENT_STATUS))
{
  if (item.getParameter(CtiParameters.EPNY_PARAM_AGENT_MODE) == Agent-

    States.CTI_AGENT_READY)
{
   // Create a new incoming call pop event
   CtiWorkItem newCall = createSampleCall(item);

   CtiWorkItem makeAvailable = createSampleAgentStateChange(item,
```

```
      AgentStates.CTI_AGENT_READY);

   processWorkItem(makeAvailable);

   Thread sendCall = new DemoPopThread(this, newCall, null);
    sendCall.start();
}
else
{
   // Reflect back the request
   CtiWorkItem stateChange = createSampleAgentStateChange(item,
   (String)item.getParameter(CtiParameters.
   EPNY_PARAM_AGENT_MODE));

   processWorkItem(stateChange);
  }
}
```

Test the server. You should be able to login, logout, change your state. The UI for the agent state should look and behave normally.

Basic Call Control

Once agent control is in place, the provider can start to handle call events. This section covers basic call control - dial, pickup, answer, hold, retrieve, and end call (all of the call functions that do not relate to transfer and conference). Transfer and conference are more difficult to simulate since the agents may not be in the same manager. We typically rely on the data passing mechanisms of the middleware server to relay call information for us. This version of the sample provider does not support transfer and conference. The version of the base provider that ships with the server supports showing one side of the transfer and conference. With a real provider these can be developed in a similar fashion as the sections that are demonstrated in the sample provider.

We need a basic behavior to handle the work items that are basic call control related. This behavior should be able to handle the events and request for dial, pickup, answer, hold, retrieve, and end call. The handler methods typically send the response events through the normal channel, simulating the provider response.

The important work items for call control are:

MAKE_CALL

PICKUP_CALL

ANSWER_CALL

HOLD_CALL

RETRIEVE_CALL

HANGUP_CALL

In this example, we will implement handlers for all of the basic call control events. Each handler will be briefly explained.

In addition, to follow the model of agent state, we need to send agent state change events during steps of the call. When the call is first dialed or answered, the agent state should be changed to busy. When the call is ended, the agent state should change to wrap-up. Later, when the agent is done with the interaction, their state should be changed to available or unavailable depending on their current value of the Application_Preferred agent state. The Application_Preferred agent state defaults to *Available*

In general, the provider is responsible for making sure the event flow matches what the CTI Manager is expecting. If the event flow is different for your provider, you either need to fix it to conform, or change the CTI Manager to handle your modified event flow.

Make call and pickup call are the same code in this provider. They both create an outbound call. If the provided phone number is not null, then it is copied into the outbound call as the number dialed. If the value is null then 5551212 is used (it is specified in the helper method). After the new call event is processed, we immediately process an established event. This is required for full call control. Most switches do not allow for call control when a call is not yet established. We then change the agent state to busy. (Currently, the phone number is only sent during dialing.)

```
else if (iType.equals(CtiWorkItemTypes.MAKE_CALL) ||
iType.equals(CtiWorkItemTypes.PICKUP_CALL))
{
    // This is the signal that there the users wants an immediate call.

     CtiWorkItem newCall = createSampleCall(item);
     newCall.setType(CtiWorkItemTypes.CALL_DIALING);

   // The phone number, if provided, should be used. Availability
   // is not modified.
   newCall.setParameterIfNotNull(CtiParameters.EPNY_PARAM_CALL_ANI,
    item.getParameter(CtiParameters.EPNY_PARAM_CALL_ANI));

   // Send the item for immediate dispatch.
   processWorkItem(newCall);

   CtiWorkItem established = createAppWorkItem(item, CtiWorkItem-
   Types.CALL_ESTABLISHED);

   processWorkItem(established);

   CtiWorkItem makeBusy = createSampleAgentStateChange(item, Agent-
States.CTI_AGENT_ON_CALL);
processWorkItem(makeBusy);

}
```

The last few handlers are simple. Answer call generates a call answered event and an agent state change to on-call. The hold call request generates a call held event. The retrieve call request generates a call retrieved event. The hang up call request generates a disconnected event and an agent state change to after call work.

```
else if (iType.equals(CtiWorkItemTypes.ANSWER_CALL))
{
   CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
    Types.CALL_ESTABLISHED);

   processWorkItem(result);

   CtiWorkItem makeBusy = createSampleAgentStateChange(item, Agent-
    States.CTI_AGENT_ON_CALL);

   processWorkItem(makeBusy);
}
else if (iType.equals(CtiWorkItemTypes.HOLD_CALL))
{
   CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
    Types.CALL_HELD);

   processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.RETRIEVE_CALL))
{
   CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
    Types.CALL_RETRIEVED);

    processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.HANGUP_CALL))
{
   CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
    Types.CALL_DISCONNECTED);

   processWorkItem(result);
   result = createSampleAgentStateChange(item, Agent-
    States.CTI_AGENT_AFTER_CALL_WORK);

processWorkItem(result);
}
```

Testing the Enhanced Provider

At this point, you can test the server by logging in, making yourself available, getting a call and answering it, placing it on hold and retrieving it, hanging it up, clicking on Save and Close to close the interaction and the interaction screen tab. The agent state should be available before the call, busy when the call is in progress, wrap up after the call is over before the tab is closed, and back to available when the tab is closed.

Complete Example

The completed example looks like this:

```
package com.epiphany.shr.cti.middleware.general.provider;

import com.epiphany.shr.cti.ctimanager.CtiManager;

import com.epiphany.shr.cti.util.*;
import com.epiphany.shr.cti.ctirules.CtiRuleEngine;

// This class is only imported to get the global constant
// out of AgentInfo

import com.epiphany.shr.cti.middleware.general.rules.AgentInfo;
import com.epiphany.shr.util.logging.ILoggerCategory;
import com.epiphany.shr.util.logging.LoggerFactory;
import com.epiphany.shr.util.logging.SuggestedCategory;
import com.epiphany.shr.util.exceptions.EpiException;
import com.epiphany.shr.util.math.IntegerConstants;
import com.epiphany.shr.data.server.IPThread;
import com.epiphany.shr.data.server.Application;
import com.epiphany.shr.sf.*;
import com.epiphany.shr.aef.element.Event;
import com.epiphany.shr.cti.middleware.general.rules.AgentStates;

import org.apache.commons.collections.FastHashMap;
import java.util.ArrayList;
import java.util.*;

// Base provider. Should be used as a starting point for creation
// of other providers.
//
public class BaseProvider extends CtiWorkItemProcessorImpl implements
   CtiProvider
{
   protected ArrayList validAgentID;
```

```
 protected boolean checkAgentID = false;

 // Reference to the CTI Manager
 //
 protected CtiManager _mgr;

// String of "Providers.ProviderName." format that should be used
// for building of keys to access configuration
//

protected String _cfgBaseKey;

// Unique name of the provider
//

 protected String _name;

 // Type of the provider
 //
 protected String _type;


 // Version of the provider
 //
 protected String _version;

// Map of the required fields for work items
//

 private Map _requiredFields;
 // List of agents, assigned to this server. We don't need
 // any infromation about agents assigned to different servers
 // in a cluster, because we don't do any routing to another
 // servers, we only do filter incoming events.
 //
 private FastHashMap _myAgents;


 // Contains mapping between agentIDs and agentUserIDs for
 // pending logins. As soon as login is committed, associated
 // map entry is to be migrated to _myAgents map.
 //
 private FastHashMap _pendingLogins;


 // Contains mapping betweeng DNs and Agent Logins (DOMAIN\User)
 //
 private FastHashMap _dnAgentLogins;


 // Local ServerRef
 //
 private ServerRef _serverRef;
```

```
   public static final String AGENT_SPLIT = "3124";


  // -------------- CtiProvider implementation --------


  // When overloading this method, call the superclass' method
  // first and then perform additional initialization
  //
  public void initialize(String name, String type, String version,
                   CtiManager mgr, CtiConfig cfg, String baseKey,
                   Map requiredFields, CtiRuleEngine ruleEng)
    throws EpiException
  {
_log.info("LOG_CTI_BASEPROVIDER_STARTINIT", "Initializing provider
  (name,type,version)({0},{1},{2})", SuggestedCategory.CTI, name,
   type, version);


       _name = name;
       _componentId = name;

       _type = type;
       _version = version;
       _mgr = mgr;
       _cfg = cfg;
       _cfgBaseKey = baseKey;

       _myAgents = new FastHashMap();
       _pendingLogins = new FastHashMap();
       _requiredFields = requiredFields;

       _dnAgentLogins = new FastHashMap();
       _myAgents.put(AgentInfo.AGENT_SERVERSTATE_MONITOR,
       AgentInfo.AGENT_SERVERSTATE_MONITOR);

     EpnyServiceManager epnyMgr = EpnyServiceManagerFactory.getInstance
();

      _serverRef = epnyMgr.getLocalServerRef();

     init(cfg, null, false, _name, ruleEng);

     String checkID = _cfg.getStringValue(baseKey + CtiMetadata-
        Params.CHECKAGENTID_STORAGE);


     if (checkID.equalsIgnoreCase("true"))
        {
          _log.info("LOG_CTI_BASEPROVIDER_INIT_CHECKID", "Agent IDs
will
```

```
          be verified for Provider {0}", SuggestedCategory.CTI,
          _name);

      checkAgentID = true;
      validAgentID = new ArrayList();
      String validIDList = _cfg.getStringValue(baseKey + CtiMetadata
Params.
        VALIDAGENTIDS_STORAGE);

      convertCommaSeparatedRangeStringToList(validIDList, validAgent
ID);
      }
else
      {
         _log.info("LOG_CTI_BASEPROVIDER_INIT_NO_CHECKID", "Agent IDs

          will not be verified for Provider {0}", SuggestedCategory.
          CTI, _name);
      }
}

    public boolean isValidAgentID(Object agentID)
{

       if (checkAgentID)
{
          return validAgentID.contains(agentID);
    }
      return true;
   }


    public boolean isValidatingAgentIDs()
   {
         // XXX Auto-generated method stub
         return checkAgentID;
   }

    public void startup() throws EpiException
{
        if (_workerThread != null)
        {
          _workerThread.start();
        }
        else
        {
          _log.error("LOG_CTI_BASEPROVIDER_STARTUPFAIL", "Can't run
          the worker thread because it is not created yet", Suggested
Category.
            CTI);
        }
    }
    public void shutdown() throws EpiException
```

```
   {
try
{
}
catch (Throwable excpt)
{
throw new EpiException("EXP_CTI_SHUTDOWN", "Exception",
excpt);
}
}
// The default implementation creates the workitem and
// sends it to itself.
//
public void initRuleEngine() throws EpiException
{
CtiWorkItem item = createWorkItem(CtiWorkItem-
Types.INITIALIZE_RULE_ENGINE);
item.getSource().setLocationType(CtiLocation.
LOCATION_TYPE_MANAGER);
item.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_PROVIDER);
item.setParameter(CtiParameters.EPNY_PARAM_CONFIG, _cfg);
item.setParameter(CtiParameters.EPNY_PARAM_CONFIG_BASE_KEY,
_cfgBaseKey);
processWorkItem(item);
}
public CtiWorkItem createWorkItem(String type)
{
CtiWorkItem item = _mgr.createWorkItem(type);
item.getSource().setLocationType(CtiLocation.
LOCATION_TYPE_PROVIDER);
item.getSource().setProvider(_name);
item.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
return item;
}
public void processWorkItem(CtiWorkItem item)
{
super.processWorkItem(item, null, _requiredFields);
}
public String getName()
{
return _name;
}
public String getType()
{
return _type;
}
public String getVersion()
{
return _version;
}
// Returns list of agents known to the provider (from _myAgents
```

```
// and _pendingLogins map)
//
public List getKnownAgents()
{
int nSize = _myAgents.size() + _pendingLogins.size();
ArrayList retV = new ArrayList(nSize);
// _myAgents go first
retV.addAll(_myAgents.keySet());
// then pending logins
retV.addAll(_pendingLogins.keySet());
return retV;
}
public void forwardWorkItem(CtiWorkItem item)
{
try
{
// More complicated logic should be here, but by default
// this just sends the item to the manager
_mgr.processWorkItem(item);
}
catch (Throwable boo)
{
_log.error(new EpiException("EXP_BASEPVD_FORWARD_WORKITEM",
"Exception in BaseProvider.forwardWorkItem()", boo));
}
}
public void setValidationRules(Map validationRules)
{
_log.info("LOG_CTI_PROVIDER_SET_VALIDATION_RULE", "Replace validation
rule for provider", SuggestedCategory.CTI);
_requiredFields = validationRules;
}
public void setRuleEngine(CtiRuleEngine ruleEngine)
{
super.setRuleEngine(ruleEngine);
}
// ------------- Protected methods (for overloading) -----------
// This method is to be called by worker thread when work
// item is just retrieved from the queue. Default implementation
// specifically process CTI_AGENT_LOGGED_ON and
// CTI_AGENT_LOGGED_OFF events, and calling commitLogin() and
// commitLogout() respectively.
//
protected void proceedRawWorkItem(CtiWorkItem item)
{
String iType = item.getType();
Object agentID = item.getParameter(CtiParameters.
EPNY_PARAM_AGENT_ID);
if (iType.equals(CtiWorkItemTypes.AGENT_LOGGED_ON))
{
commitLogin(agentID);
}
else if (iType.equals(CtiWorkItemTypes.AGENT_LOGGED_OFF))
```

```
{
// commitLogout is going to destroy agentID -
// agentUserID for this agent, therefore, we need to
// retrieve agentUserID now
Object agentUserID = _myAgents.get(agentID);
if (agentUserID != null)
item.setParameter(CtiParameters.EPNY_PARAM_AGENT_USER_ID,
agentUserID);
commitLogout(item.getParameter(CtiParameters.
EPNY_PARAM_AGENT_ID));
}
}
// Defines additional conversion of the workitem before sending
// it to the manager.
//
protected CtiWorkItem convertWorkItem (CtiWorkItem item)
{
// default implementation - add agent user ID, based on agent ID
Object agentID = item.getParameter(CtiParameters.
EPNY_PARAM_AGENT_ID);
Object agentUserID = _myAgents.get(agentID);
if (agentUserID != null)
item.setParameter(CtiParameters.EPNY_PARAM_AGENT_USER_ID,
agentUserID);
return item;
}
protected static final Object[] _keysToCopy =
{
CtiParameters.EPNY_PARAM_AGENT_ID,
CtiParameters.EPNY_PARAM_AGENT_USER_ID
};
protected static final Object[] _keysToCopyWithDN =
{
CtiParameters.EPNY_PARAM_AGENT_ID,
CtiParameters.EPNY_PARAM_AGENT_USER_ID,
CtiParameters.EPNY_PARAM_PARTY_NUMBER
};
// Process incoming work item (from manager or service) here
// Default implementation unconditionally reply on the following
// requests:
// EPNY_AGENT_LOGIN -> CTI_AGENT_LOGGED_ON
// EPNY_AGENT_LOGOUT -> CTI_AGENT_LOGGED_OFF
//
protected void receiveWorkItem(CtiWorkItem item)
{
String iType = item.getType();
String agentUserID = item.getStringParameter(CtiParameters.
EPNY_PARAM_AGENT_USER_ID);
if (iType.equals(CtiWorkItemTypes.LOGIN_AGENT))
{
proceedRawWorkItem(item);
CtiWorkItem reply = createWorkItem(CtiWorkItem-
Types.AGENT_LOGGED_ON);
```

```
reply.copyParameters(item, _keysToCopy);
reply.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
processWorkItem(reply);
CtiWorkItem makeUnavailable = createSampleAgentStateChange(
item, AgentStates.CTI_AGENT_NOT_READY);
processWorkItem(makeUnavailable);
}
else if (iType.equals(CtiWorkItemTypes.LOGOUT_AGENT))
{
proceedRawWorkItem(item);
CtiWorkItem reply = createWorkItem(CtiWorkItem-
Types.AGENT_LOGGED_OFF);
reply.copyParameters(item, _keysToCopy);
reply.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
processWorkItem(reply);
}
else if (iType.equals(CtiWorkItemTypes.SET_AGENT_STATUS))
{
if (item.getParameter(CtiParameters.EPNY_PARAM_AGENT_MODE) ==
AgentStates.CTI_AGENT_READY)
{
// Create a new incoming call pop event
CtiWorkItem newCall = createSampleCall(item);
CtiWorkItem makeAvailable = createSampleAgentStateChange(
item, AgentStates.CTI_AGENT_READY);
processWorkItem(makeAvailable);
Thread sendCall = new DemoPopThread(this, newCall, null);
sendCall.start();
}
else
{
// reflect back the request.
CtiWorkItem stateChange = createSampleAgentStateChange(
item, (String)item.getParameter(CtiParameters.
EPNY_PARAM_AGENT_MODE));
processWorkItem(stateChange);
}
}
else if (iType.equals(CtiWorkItemTypes.MAKE_CALL) ||
iType.equals(CtiWorkItemTypes.PICKUP_CALL))
{
// This is the signal that there the users wants an
// immediate call.
CtiWorkItem newCall = createSampleCall(item);
newCall.setType(CtiWorkItemTypes.CALL_DIALING);
// The phone number, if provided, should be used.
// Availability is not modified.
newCall.setParameterIfNotNull(CtiParameters.
EPNY_PARAM_CALL_ANI, item.getParameter(CtiParameters.
EPNY_PARAM_CALL_ANI));
// Send the item for immediate dispatch
```

```
processWorkItem(newCall);
CtiWorkItem established = createAppWorkItem(item, CtiWorkItem-
Types.CALL_ESTABLISHED);
processWorkItem(established);
CtiWorkItem makeBusy = createSampleAgentStateChange(item,
AgentStates.CTI_AGENT_ON_CALL);
processWorkItem(makeBusy);
}
else if (iType.equals(CtiWorkItemTypes.ANSWER_CALL))
{
CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
Types.CALL_ESTABLISHED);
processWorkItem(result);
CtiWorkItem makeBusy = createSampleAgentStateChange(item,
AgentStates.CTI_AGENT_ON_CALL);
processWorkItem(makeBusy);
}
else if (iType.equals(CtiWorkItemTypes.HOLD_CALL))
{
CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
Types.CALL_HELD);
processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.RETRIEVE_CALL))
{
CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
Types.CALL_RETRIEVED);
processWorkItem(result);
}
else if (iType.equals(CtiWorkItemTypes.HANGUP_CALL))
{
CtiWorkItem result = createAppWorkItem(item, CtiWorkItem-
Types.CALL_DISCONNECTED);
processWorkItem(result);
result = createSampleAgentStateChange(item, Agent-
States.CTI_AGENT_AFTER_CALL_WORK);
processWorkItem(result);
}
}
protected CtiWorkItem createAppWorkItem(CtiWorkItem item,
String type)
{
CtiWorkItem result = createWorkItem(type);
result.copyParameters(item, _keysToCopyWithDN);
result.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
return result;
}
protected CtiWorkItem createSampleAgentStateChange(
CtiWorkItem item, String agentState)
{
CtiWorkItem makeUnavailable = createWorkItem(CtiWorkItem-
Types.AGENT_MODE_CHANGED);
```

```
makeUnavailable.copyParameters(item, _keysToCopy);
makeUnavailable.setParameter(CtiParameters.
EPNY_PARAM_AGENT_MODE, agentState);
makeUnavailable.setParameter(CtiParameters.
EPNY_PARAM_AGENT_GROUP, AGENT_SPLIT);
makeUnavailable.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
return makeUnavailable;
}
protected CtiWorkItem createSampleCall(CtiWorkItem item)
{
CtiWorkItem newCall = createWorkItem(CtiWorkItem-
Types.CALL_RINGING);
newCall.copyParameters(item, _keysToCopyWithDN);
newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_ID,
"callid1");
newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_ANI,
"5551212");
newCall.setParameter(CtiParameters.
EPNY_PARAM_CALL_DIALED_NUMBER, "800-555-1212");
newCall.setParameter(CtiParameters.EPNY_PARAM_CALL_DNIS, "800-
555-1212");
newCall.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
return newCall;
}
class DemoPopThread extends Thread
{
DemoPopThread(BaseProvider base, CtiWorkItem firstItemToSend,
CtiWorkItem secondItemToSend)
{
this.base = base;
this.firstItemToSend = firstItemToSend;
this.secondItemToSend = secondItemToSend;
}
BaseProvider base;
CtiWorkItem firstItemToSend;
CtiWorkItem secondItemToSend;
public void run()
{
try
{
sleep(5000);
}
catch (Exception ex)
{
}
base.processWorkItem(firstItemToSend);
if (secondItemToSend != null)
{
try
{
sleep(10000);
```

```
}
catch (Exception ex)
{
}
base.processWorkItem(secondItemToSend);
}
}
}
// Adds agent to the _myAgents map
//
protected void beginLogin(Object agentID, Object agentUserID)
{
_pendingLogins.put(agentID, agentUserID);
}
// Adds agent to the _myAgents map
//
protected void commitLogin(Object agentID)
{
Object agentUserID = _pendingLogins.remove(agentID);
_myAgents.put(agentID, agentUserID);
_log.info("LOG_CTI_BASEPROVIDER_NEWLOGIN", "Agent is logging on
(agent,user,provider)({0},{1},{2})", SuggestedCategory.CTI,
agentID, agentUserID, _type);
}
//Removes agent from the _myAgents map
//
protected void commitLogout(Object agentID)
{
Object agentUserID = _myAgents.remove(agentID);
if (agentUserID != null)
{
_log.info("LOG_CTI_BASEPROVIDER_NEWLOGOUT", "Agent is logging
out (agent,user,provider)({0},{1},{2})", SuggestedCategory.
CTI, agentID, agentUserID, _type);
}
}
// Checks whether the agent is in the _myAgents map
//
protected boolean isLoggedIn(Object agentID)
{
return _myAgents.containsKey(agentID);
}
// Checks whether the agent is in the _pendingLogins map
//
protected boolean isLoginPending(Object agentID)
{
return _pendingLogins.containsKey(agentID);
}
// Register DN -> Agent Login mapping
//
protected void registerDnAgentLogin(Object DN, Object agentLogin)
{
_dnAgentLogins.put(DN, agentLogin);
```

```
_log.info("LOG_CTI_BASEPROVIDER_AGENT_DEVICE_MAPPING", "Agent-
Device mapping is registered (agent,device,provider)({
0},{1},{2})", SuggestedCategory.CTI, agentLogin, DN,
_type);
}
protected Object getAgentLoginByDN(Object DN)
{
return _dnAgentLogins.get(DN);
}
// Helper function called by worker thread
//
protected void proceedWorkItem(CtiWorkItem item)
throws EpiException
{
try
{
_log.info("LOG_CTI_PROVIDER_PROCEED_WORK_ITEM", "CTI Provider proceed
work item {0}", SuggestedCategory.CTI, item.getType());
// For work items, generated in provider (raw items) we
// do proceed raw item, first then, calling overloadable
// convertWorkItem method that should prepare another work
// item to be forwarded to the CTI Manager (it can return
// reference to the same raw work item, or return null
// if we decide to cancel sending work item to the manager.
// For all other items - we proceed here and
// don't send anything.
if (item.getSource().getLocationType() != null && item.get-
Source().getLocationType().equals(CtiLocation.
LOCATION_TYPE_PROVIDER))
{
proceedRawWorkItem(item);
CtiWorkItem wiToSend = convertWorkItem(item);
if (wiToSend != null)
{
wiToSend.getDestination().setLocationType(CtiLocation.
LOCATION_TYPE_APPLICATION);
_mgr.processWorkItem(wiToSend);
}
}
else
{
receiveWorkItem(item);
}
}
catch (Throwable excpt)
{
_log.error("LOG_CTI_PROVIDER_CRITICAL_ERROR", "unhandled
exception in CtiProvider thread {0}", SuggestedCategory.
CTI, Thread.currentThread().getName());
throw new EpiException("
EXP_CTI_PROVIDER_CRITICAL_ERROR_EPIWRAPPER", "Exception
in CtiProvider", excpt);
}
```

```
}
// Convert a string of the form 1,2,3,5-7,12-45 into an
// ArrayList of String values.
//
private void convertCommaSeparatedRangeStringToList(
String sourceList, ArrayList list)
{
if (sourceList != null)
{
StringTokenizer stkn = new StringTokenizer(sourceList, ",");
while (stkn.hasMoreTokens())
{
String token = stkn.nextToken();
StringTokenizer stknconsecutive = new StringTokenizer(
token, "-");
int begin = 0,end = 0;
// if this is a ###-### range, expand it to include
// the individual values.
if (stknconsecutive.countTokens() == 2)
{
begin = Integer.parseInt(stknconsecutive.nextToken());
end = Integer.parseInt(stknconsecutive.nextToken());
for (int i = begin; i<= end; i++)
list.add(IntegerConstants.getInteger(i).toString();
}
else
list.add(token);
}
}
}
}
```

Common Mistakes

When writing a provider, rules, or behaviors, keep the following in mind:

- All processing is typically done inside of WorkItemProcessor subclasses.
- Each WorkItemProcessor has one thread for processing. Do not use this thread for long, unless you are on a special work item processor. Each WorkItemProcessor is designed to handle many short requests. Handling requests quickly ensures that the UI stays in synch with the outside world (such as the phones). If a rule or behavior takes a long time to complete, all of the work waiting for all users on that provider is blocked.
- In work item processors, it is also very important to not have code that produces periodic long delays. If this is the case, then the system will appear halting to the user and it will be hard to track down. If you have some code that you are concerned about, apply some defensive programming to make sure that a call does not take too long, put the call on another thread, or find some fixed time algorithm.

- When changing the work items to either require more parameters for evaluation, require less parameters, or different parameters be sure to update the validation rules under the service configuration in Infor Studio.

Configuring a New Provider

6

The basic steps for configuring a new provider in Studio are as follows:

1. Configure a new module.
2. Create a new provider type.
3. Create a new provider.
4. Configure a new provider
5. Disable the base provider.
6. Save the changes to the new module.

   The following sections outline each of these steps in more detail.

Setting Up the New Provider

Before configuring the new provider, you need to enable the System Maintenance options in Studio, and create a new metadata module in which to store the provider configuration.

1. Enable the System Maintenance Tab in the Guide Bar.
   a. Open SQL Query Analyzer against the machine where your Epiphany databases resides.
   b. Select epiphany_meta from the pull down, in order to specify the metadata database.
   c. Paste in the following script, and then execute it:

```
delete from tsy_users
delete from tsy_users_groups
delete from tsy_groups where group_id = 'epny'
INSERT INTO
tsy_groups(group_id,app_module_id,group_name,group_des
cription) VALUES('epny','STUDIO','EPNY','Epiphany
Engineering')
```

   d. Open Studio, and verify that the System Maintenance heading appears in the Guide Bar. It is located underneath the Customization & Business Processes bar.

2 Create a new module and set it as the default.
   a Click on **Application Development > Modules** in Guide Bar View.
   b Select **New > Module** from the **File** menu.
   c Fill in values in the main window frame as follows:

      Name
      Enter the name for the new provider.

      Version
      Enter the provider version.

      Folder Name
      Enter the path to which you want to save your new module file.

   d Select **Save** from the **File** menu.
   e Go to the Module dropdown on the Toolbar, and select aspect_provider from dropdown menu.
   f Click the **Set Current** button on the Toolbar next to the **Module** dropdown.

Creating A New Provider Type

1 Create the new provider type:
   a From the Guide Bar, select **System Maintenance > System Maintenance > Service Configurations > Service Definitions > CtiManager > Service Configurations > Providers > Service Configurations > Provider > Service Configurations > Type**.
   b Add a new row as follows:

      Module
      The name of your new module.

      Parameter Name
      The name of the new provider type. This is the value that is used to identify the provider on other configuration screens.

      Property Value Type
      STRING

   c Click outside of the new row to commit your changes.

2 Add parameters to the new provider type:
   a From the Guide Bar, select **System Maintenance > System Maintenance > Service Configurations > Service Definitions > CtiManager > Service Configurations > Providers > Service Configurations > Provider > Service Configurations > Type > Service Configurations**.
   b Click on the new row that you added for the Aspect Type, in the top portion of the main window frame

c Add a new row in the lower (for example, second) portion of the main window frame (Service Configurations tab) as follows:

Module
Enter the name of the module that you created for this provider.

Parameter Name
Enter the name of the new custom parameter

Property Value Type
Enter the type for the new parameter.

d Click outside of the new row to commit your changes.

e Repeat this procedure for any additional parameters.

Creating a New Provider

1 Create the new provider:

a From the Guide Bar, select **Administration > Services > CtiManager > Parameters**.

b Click on the Providers row, located in the top portion of the main window frame.

c In the **Service Configurations** tab in the lower pane, add a new row with the following information:

Module
Enter the name of the module that you created for this provider.

Parameter Name
Enter **Provider**.

d Click outside of the new row to commit your changes.

2 Add parameters to the new provider:

a From the Guide Bar, select **Administration > Services > CtiManager > Parameters > Providers (1) > Parameters > Provider - (6)**. Note that this provider does not yet have a name.

b Right-click on each of the following rows, and select **Delete**.

- Associations
- Behaviors
- Rules

c Add new rows for any parameters that you need to configure for the new provider. For example, to configure a parameter for a backup middle ware server:

Module
Enter the name of the module that you created for this provider.

Parameter Name
Enter **Server**.

d Click outside of the new row to commit your changes.

Configuring the New Provider

1 From the Guide Bar, select **Administration > Services > CtiManager > Parameters > Providers (1)**.
2 Set the name of the provider:
   a Select the Properties pane, and edit the following parameter:
   Provider (6): Enter the name of the new provider.
   b Click the **Save** icon at the top of the Properties view.
3 In the Properties pane, expand Provider (6) (the provider that you just named), and set the following properties:
   - **Enabled**: Set this to true.
   - **ImplClass**: Enter `com.test.provider.TestProvider`.
   - **IntegrationType**: Select standard.
   - **Type**: Enter the name of the new provider type that you created.
4 Expand Server (3) and edit the following properties:
   - **Host**: Enter the hostname of your primary middle ware server.
   - **Port**: Enter the port for your primary middle ware server.
5 Expand Server (15) and edit the following properties:
   - **Host**: Enter the hostname of your backup middle ware server.
   - **Port**: Enter the port for your backup middle ware server.
6 Expand **Type > ProviderName** (where ProviderName is the name of your provider type), and edit the property values for any custom properties that you added when you created the new provider type.
7 Click the **Save** icon at the top of the Properties view.

Disabling The Base Provider

1 From the Guide Bar, select **Administration > Services > CtiManager > Parameters > Providers (1)**.
2 Expand Provider (1) (the Base Provider), and edit the following properties:
   - **Enabled**: Set this to false.
   - **ImplClass**: Enter **com.epiphany.shr.cti.middle ware.general.provider.BaseProvider.In**.

Saving Changes to the New Module

1 From the Guide Bar, select **Application Development > Modules > provider_name**.

2 In the main window pane, click **Save Module**.

Configuring the Base Provider

7

Infor supplies a base provider, which is similar to the sample provider described in this SDK. This provider can be used for simple demos and to test UI changes without requiring telephony hardware. It is not intended to replace providers specific to the telephony hardware. When you use the base provider, keep the following limitations in mind:

- It does not offer all the features of a real switch
- It is only intended to handle one call at a time. If multiple calls are made (for example, if you have a simulated inbound customer call and an agent-initiated outbound call), the two calls are independent and the interaction tabs are not sychronized.
- The simulated calls only represent one side of the call. If agent A calls agent B, only agent A will see CTI call controls for the call.

Configuring the Base Provider

The base provider has two integration types:

- noIntegration: This mode supports creating inbound and outbound interactions without CTI integration. (Hardware interaction is controlled by an external application.)
- standard: This integration type simulates a typical provider's behavior.

1. Enable the base provider.
2. Configure an agent to use CTI.
3. Enable the CTI UI components.
4. Test the provider configuration.

   The following sections cover these steps in more detail.

Enabling the Base Provider

To enable the base provider:

1. Open Studio and connect to the database.
2. In the Guide Bar, expand Administration.
3. Select **Services > CTI Manager > Parameters**.
4. Click on **Providers**.
5. In the Properties window, expand **Base**.
6. Set **Enabled** to true.
7. Set the **Integration Type**:
   - **noIntegration**: This mode allows for creating inbound and outbound interactions without CTI integration. (Hardware interaction is not controlled or monitored by Infor.)
   - **standard**: This integration type simulates a typical provider's behavior.
8. If you have previously enabled another provider, set all of their enabled flags to false.

Configuring Agents for The Base Provider

Once the Base Provider is configured, you need to configure your agent for CTI

1. In Studio, configure a split. For help with this, see “Configuring Splits” on page 17
2. Open a browser and log in to the server as an administrator.
3. Navigate to the User Administration screen, and find the agent you want to enable.
4. Give the user a **CTI Telephony Username**. For the demo provider, this can be any number. For example: 3138.
5. Click **Save**.
6. Set the DN to 3108. (In a working provider, this would normally be the agent's phone number.)
7. In the Skill Group field, select the split number that you entered earlier from the dropdown. (For this example, 3124.) Split and DN changes take effect on the agent’s next login.
8. Create an individual customer, or find one and set their primary phone number to 5551212 (with no area code).
9. Close the browser.

Enabling the CTI UI Components

1. In Studio, select the **Administration** tab.
2. Select the User Preference template.
3. Locate the `show_account_summary` property, and set it to true.

Testing the Base Provider Configuration

Testing the Standard Integration Case

1. Open a new browser and log in as the configured agent.
2. In the CTI section of the Agent Slot on the bottom of the page, click **Login**.
3. The user's state should now be shown as logged in with a status of Unavailable.
4. Click the **Available** button.

   5 seconds later a call should come in from the configured customer
5. Click **Answer Call**. The customer verification screen should appear.
6. Click on the customer's name in the list. The CTI interaction screen appears.
7. You should be able to perform basic operations on the active call (such as Hold, Retrieve, End Call, Save and Queue, and Save and Close).

Testing the noIntegration Case

1. Open a new browser and log in as the configured agent.
2. In the CTI section of the Agent Slot on the bottom of the page, click **New Inbound**.

   A new inbound interaction tab should appear.
3. In the CTI section of the Agent Slot on the bottom of the page, click **New Outbound**.

   A new outbound interaction tab should appear.

Configuring Genesys AIL Provider

8

Infor integrates with Genesys TServer via AIL (Agent Interaction Layer). A direct integration with TServer (without AIL) is no longer supported. The Genesys integration enables monitoring and control over agents and phone calls. Also, Infor can track hardware events and events triggered by other soft phones.

The thin-client soft phone can be used to control the telephone and agent. All requests for telephone and agent control go from the Infor browser-based UI, to the Infor server, and are then forwarded to Genesys. The Infor UI is not updated until Genesys has had a chance to respond to the request. For example, when an agent sets their status to ‘Available’, we send that request from the client to the UI framework on the server, to the CTI Manager, to the CTI Genesys provider, to the Genesys server. At this point, the action is complete for Infor CTI. The Genesys server then works with the switch to change the agent state to ‘Available; in the switch. Once this occurs, the Genesys server sends a notification to the provider that the agent state has changed. The provider forwards the information to the CTI Manager, which then uses AEF scripting to forward the event to a CTI application extension, which updates the UI. In this example, it is not the button press that updates the UI, but the event that comes from the Genesys server.

Genesys AIL

Agent Interaction (Java API) is Genesys' primary interface for developing agent desktop applications. The library fully integrates with Genesys Configuration Layer objects (such as Agent, Place, and DN). Its Java API presents a common development interface for all interaction and media types, with application support for Voice, Email, Contact History, Chat, Outbound Campaigns, Expert Contact, Voice Callback.

Configuring Genesys AIL

The Genesys Agent Interaction Layer is a library that helps you to manage multimedia interactions issued or intended for an Agent. It consolidates Genesys models coming from various servers such as the agent model, multimedia interaction data, and contact data, as well as events issued from Genesys servers and Databases.

The core of the Interaction SDK library is designed to handle the specific features of switches, but the API presents an abstraction of switch features, a uniform interface for all switches supported by the core. Information, requests, and events are consolidated and forwarded when they go through the AIL to or from the Genesys Servers.

AIL keeps an execution context coming from those servers.

Currently, the Genesys AIL provider supports the following operations:

- answer
- dial
- pickup
- hang-up
- hold
- retrieve
- cancel
- mute transfer
- assisted transfer
- assisted conference

The latest version of Genesys AIL that is supported with Genesys 8.1 Framework is AIL7.6.

Configuring TServer for Infor

A direct TServer Provider is no longer supported in Infor CTI integration. However, for Genesys AIL integration, a TSever Application is still required. This is because the AIL application connects to the TServer via the TServer Application which is configured in the Configuration manager.

Consult your T-Server documentation for more information about the settings described in this section.

1. Open the Genesys Configuration Manager and go to **Configuration > Environment > Applications**.
2. Right-click the entry for your T-Server application, and select **Properties**. A dialog box appears.
3. In the application dialog box, go to the **Options** tab. Double-click the TServer section, and then do the following:
   a. Double-click second-call-as-consult, and change the option value to true, so that Call Management can support transfers and conferences initiated from the telephone handset.
   b. Double-click consult-user-data , and change the option value to joint', so that Call Management can pass the customer record during transfers and conferences.
   c. (Optional) Double-click preserve-collected-digits, and change the option value to 'udata'.

   This setting is necessary only if you want CTI to have visibility into the original string of digits which were entered while the customer was navigating through the IVR. Normally these digits would be used by Genesys Universal Routing as it executes its routing strategies. This setting is necessary only if you want CTI to have visibility into the original string of digits which were entered while the customer was navigating through the IVR. Normally these digits would be used by Genesys Universal Routing as it executes its routing strategies.

Instead of using this T-Server setting to have the entered digits passed down to Infor, it may be easier to customize Genesys Universal Routing so that it translates these digits into human-readable key/ value pairs which would then be accessible when CTI generates a screen pop.

Configuring AIL for Infor

1. Open the Configuration Manager and go to **Configuration > Environment > Applications**.
2. Create a new application:

   The name of this application can be determined using the following entries from the installer.properties file (<S&S ROOT>\shared\etc\installer.properties):

   app.server

   jmsjndi

   jndi1

   The syntax is: <app.server>_<jmsjndi/ jndi1>_epiphany_ail_server. or jndi1 entries contain any non alpha-numeric characters other than an underscore, they should be removed when generating the name.

   For example, if the installer.properties has the following entries:

   ```
   app.server=mymachine1.infor.com
   jmsjndi=server1
   jndi1= mymachine1.infor.com:2810
   ```

   Then the application name would be:

   ```
   mymachine1.infor.com_server1_epiphany_ail_server
   ```

   If 'jmsjndi' was not present, then the application name would be:

   ```
   mymachine1.infor.com_mymachine1inforcom2810_epiphany_ail_server
   ```

3. In the **General** tab, select the `Agent_Interaction_server_763`.
4. In the **Connection** tab, add the TServer application that AIL should communicate with.
5. In the **Options** tab, expand the voice section, and then change the enable-possible-changed-event option to false (default value is true). If this option is not changed to false; the Infor AIL provider will regularly receive events which are unexpected from the perspective of the AIL provider.

   These events will confuse the provider and negatively impact the propagation of UI updates to the CTI agent's browser.
6. In the Configuration Manager, go to **Configuration > Resource > Person**.
7. Create an agent with the correct Login ID and password. In the **Agent Info** tab, add the corresponding agent from the switch.
8. In the Configuration Manager, go to **Configuration > Resource > Place**.
9. Create a new place object with the name of the extension.
10. Find the extension under the switch and add it to the newly-created Place Object

Configuring Infor For AIL

1 Locate the following .JAR files in your Genesys AIL SDK installation, and add them to the class path. Locate folder `<S&S ROOT>\shared\lib\3rdparty\ail` and copy these files to it:

```
activation.jar,AIL.jar,chat_api.jar,commonLib.jar,commons.jar,concurren
t.jar,configurationprotocol.jar,connection.jar,flexlm.jar,jaxqname.
jar,
jaxbapi.jar,jaxbimpl.jar,kvlists.jar,log4j.jar,mail.jar,namespace.
jar,o
penmediaprotocol.jar,protocol.jar,relaxngDatatype.jar,StatLib.jar,
syste
m.jar,tkv.jar,ucsapi.jar,voiceprotocol.jar,xercesImpl.jar,xkv.jar,xml
Pa
rserAPIs.jar,xsdlib.jar
```

   If any of these JAR files are missing from the class path, a 'Class Not Found' exception occurs.

   Check the manifest file in ail_classpath.jar (`<S&S ROOT>\shared\lib\3rdparty`) to make sure all these JARS are listed.

2 In Studio, go to **Administration > Services > CtiManager > Providers > Base Provider**, and make sure the base provider is disabled.

3 Select **Administration > Services > CtiManager > Providers > AIL**.

4 In the Properties pane, set the enabled property to true.

5 In the username and password fields, enter the username and password that you use to log in to the Genesys Configuration Manager

6 Set the Server Host and Port with the server name and port for the Genesys Configuration Server.

7 If your configuration includes a backup server and port, set that also.

8 Run `SS_regenerateServiceEar.sh/bat` to regenerate the service.ear from `<S&S ROOT>/shared/bin`

9 If your application server is WebSphere (any version), do the following:

   a cd `<S&S ROOT>/shared/bin/websphere`

   b execute `admin_deploy.bat`

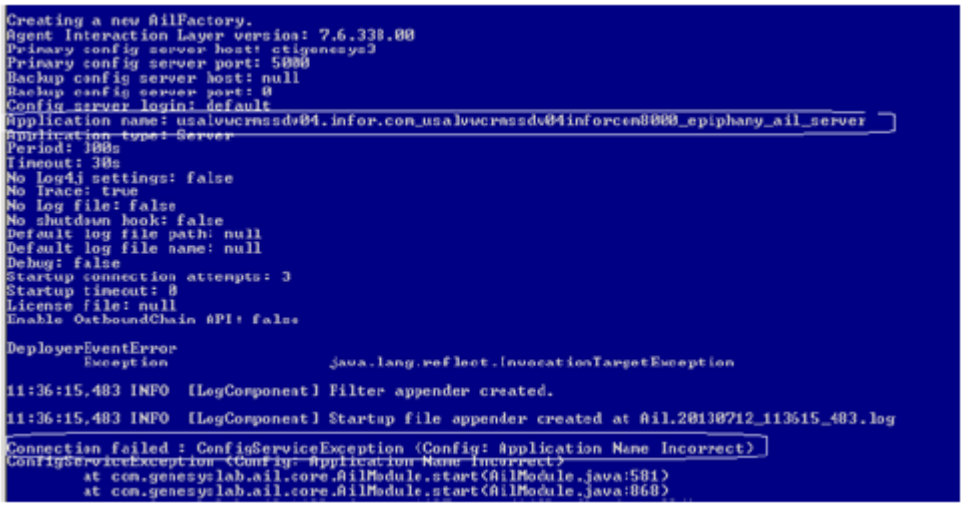
   with the needed parameters (for a UNIX environment, enter: ./admin_deploy.sh) admin_deploy will pick up the updated service.ear and deploy it to the WebSphere Deployment Manager or to the stand-alone server node. In a Deployment Manager Environment with multiple cluster nodes, allow for 15 minutes for all the updates to automatically propagate to the nodes.

   Note: Step 9 is not required for Weblogic and JBoss platforms.

10 Restart the application server(s).

Troubleshooting

1 If Sales & Service fails to start with the error message `Application Name Incorrect`, then copy the correct (expected) application name that the server is trying to connect to (this is listed in the startup log) and use that name to create the AIL application from the Configuration Manager. Then restart Sales & Service.



2 After restarting the server, if you see the error below then make sure ‘jaxbimpl. jar’ does not exist more than once in the classpath.

```
javax.xml.bind.JAXBException
- with linked exception:
java.lang.SecurityException: class
"com.sun.xml.bind.DatatypeConverterImpl"'s signer information does
not
match signer information of other classes in the same package]
```

Configuring Cisco ICM Provider

9

Configuring Infor for ICM

Note: The Cisco JavaCIL library requires JDK version 1.4.

Before enabling Cisco ICM (Intelligent Contact Management) provider,you need to enable and configure Infor CTI, see “Enabling CTI” on page 17 in Chapter 3, “Channel Configuration”.

To enable Cisco ICM provider, do the following:

1 Copy the jar file JavaCIL.jar provided by Cisco

- For application server WebSphere 5.X, copy JavaCIL.jar to this directory:`<WebSphereinstall dir>\AppServer\installedApps\<host name>Network\service.ear\lib\`
- For application server WebLogic or WebSphere 4.X, do the following:

a Copy JavaCIL.jar to this directory: `<Infor install dir>\shared\lib\`

b For Windows, add **<**Infor install dir**>**\shared\lib\JavaCIL.jar to the CLASSPATH environment variable specified in file **startWebLogic.cmd** or **startManagedWebLogic.cmd**, as shown below:

```
set CLASSPATH=C:\Program Files\Infor\shared\etc;C:\Program
Files\Infor\shared\lib\js.jar;%WEBLOGIC_CLASSPATH%;%POINTBA
SE_CLASSPATH%;%JAVA_HOME%\jre\lib\rt.jar;%WL_HOME%\server\l
ib\webservices.jar;C:\Program Files\Infor\shared\lib\classpath.
jar;C:\Program Files\Infor\service\lib\ip.jar;C:\Program
Files\Infor\shared\lib\dialogs.jar;C:\Program
Files\Infor\shared\lib\JavaCIL.jar;%CLASSPATH%
```

You can find files **startWebLogic.cmd** and **startManagedWeb-Logic.cmd** from the directory where WebLogic was installed. For UNIX systems, add `<Infor_Global_Dir>/shared/lib/JavaCIL.jar` to the CLASSPATH environment variable specified in file **startWebLogic.sh** or **startManagedWebLogic.sh**.

2 For Windows, run `<Infor install dir>\shared\bin\servicegen.bat`. For UNIX systems, run **<**`Infor_Global_Dir>/shared/bin/servicegen.sh`

3 Start Infor Studio, which connects to the operational database.

4 In the Guide Bar, expand Administration

5 Expand **CTI Manager > Parameters**.

6 Click **Providers**.
7 In the properties window, expand ICM
8 Set Enabled to **true**
9 Under the first Server entry, specify the primary Cisco CTIOS server's host and port number.
10 Under the second Server entry, specify its host and port number if a secondary Cisco CTIOS server (a backup and load-balance to the primary) exists.
11 Expand **Type > ICM** in the properties windows.
12 Set the Heartbeat value; the default is 3
13 Set the PeripheralID value to match the IPCC CallManager's PeripheralID ;the default is 5001.
14 If you do not have a second Cisco CTIOS server, do the following:
   a Go back to the Guide Bar and expand **Providers > Parameters**.
   b Click **Provider > ICM**.
   c In the list view, delete the second entry.
15 In the Guide Bar, click **Providers**. Make sure that all other providers have their enabled flag set to false. By default, the Base Provider is enabled.
16 In the Guide Bar, expand User Interface
17 Expand **Forms > Other Forms > CtiAgentControlForm > Widgets**.
18 Click **CallPickup**.
19 In the properties window, set readonly to **true**. This will disable the pickup button in the agent control panel. Cisco ICM does not support pickup.
20 Restart Infor server.

Configuring ICM for Infor

Infor ICM provider makes use of ECC (Expanded Call Context) variables in ICM to pass and retrieve data in a call; therefore, you have to follow Cisco's instructions to setup ECC variables.

How to Set the Enable Expanded Call Context Option

1 Within the Configuration Manager, select **Tools > Miscellaneous Tools > System Information**. The System Information window appears.
2 Select **Enable Expanded Call Context**.
3 Click **Save** to apply your changes.

How to Define an Expanded Call Context (ECC) Variable

1 Within the Configuration Manager, select **Tools > List Tools > Expanded Call Variable List**. The Expanded Call Variable List window appears.
2 In the Expanded Call Variable List window, enable Add by clicking Retrieve.

3 Click **Add**. The Attributes property tab appears.

4 Complete the Attributes Property tab.

5 Click **Save** to apply your changes.

The variable name must be defined in the form of`user.<Company-Name>.<Variable Description>`. For Infor's variable, `user.epny.Customer_ID` must be defined and set the size to 210.

Declaring ECC Variables in CTIOS Server

The ECC variables are declared in the registry of each server on the following location.

`[HKEY_LOCAL_MACHINE\SOFTWARE\Cisco Systems, Inc.\CtiOs\<CTIOS InstanceName>\<CTIOSServerName>\ EnterpriseDesktopSettings\All Desktops\ECC\Name\Variable Name]`

To declare a given ECC variable either named array or scalar, you must define a subkey under \ECC\Name\ without the prefix user. However,when creating ECC variables using the ICM Expanded Call Variable List tool, the ECC name must start with the prefix "user.". The subkey name is case sensitive.

For example, to declare the scalar variables user.CustomerID and user.CustomerType in CTI OS you must enter **HKEY_LOCAL_MACHINE\SOFTWARE\...\ECC\Name\CustomerID] "Data"="ECC" (Note: enter this line without the quotes)**

[HKEY_LOCAL_MACHINE\SOFTWARE\...\ECC\Name\Customer-Type] "Data"="ECC" (Note: enter this line without the quotes)

To declare a named array user.CustomerAccount with 27 elements, you must enter**[HKEY_LOCAL_MACHINE\SOFTWARE\...\ECC\Name\CustomerAccount]"Data"="ECCARRAY[27]" (Note: enter this line without the quotes)**

For Infor's variable, user.epny.Customer_ID must be defined as **[HKEY_LOCAL_MACHINE\SOFTWARE\...\ECC\Name\Customer_ID] Data=ECC**in the registry on all CTIOS servers

Note: For details, please refer to these two PDF documents:

To configure ECC in ICM, see Chapter 10 of the ICM Configuration Guide in the following PDF document:

http://www.cisco.com/univercd/cc/td/doc/product/icm/ icmentpr/icm60doc/coreicm6/config60/icme60cg.pdf

To configure in CTIOS, see Chapter 4 of the CTIOS System Manager's Guide in the following PDF document:

http://www.cisco.com/univercd/cc/td/doc/product/icm/icmentpr/icm60doc/icm6cti/ctios60/cti60mgr.pdf

Integrating with Avaya

10

The Infor-supplied Avaya provider integrates with Avaya Interaction Center. This provider uses a client-side integration, in which the Avaya client application controls the agent state, calls and e-mail, and Infor CTI records interactions and creates tabs displaying the customer information for the contact. The integration is implemented by adding and modifying Interaction Center scripts, and is dependent on the script modifications (not dependent on the UI modifications).

You can customize the UI and limit the posts to Infor by altering the script changes.

Configuring Avaya For Infor

Configuring the Avaya CDL File

The Avaya client uses a special configuration file, the CDL file, to lay out the user interface. Infor provides a modified CDL fileavaya_agent_en_customize.cdl file, which adds the Infor components. You can also customize an existing CDL file to modify the UI.

To customize the CDL file:

1 Open the CDL file in a text editor.

2 Locate the QSection part of the CDL file. Add an Infor QSection.

The QSection contains the configuration information for enabling the integration and data necessary to contact the Infor server. Add the following fields:

- **Enabled**: Set to TRUE to enable the Infor/Avaya integration,FALSE to disable it.
- **EAIUrl**: Specify the URL to reach the Infor EAI service. Avaya uses EAI to contact the Infor server. EAI needs to be configured and enabled for this integration. Refer to the *Infor Platform Integration Guide* for more information. The EAI URL takes the following form:

  http://<web server name>:<web server port>/<instance name>/ ecs/EAI

  Note: The EAI URL in the Avaya CDL file must match the actual EAI URL.

- **UserName** and **Password**.

The configured section should look like this:

```
<QSection Name="EpiphanyApp"
Enabled="TRUE"
WebServer="epiphanyserver.mydomain.com"
WebServerPort="80"
EAIUrl="/epiphany/ecs/EAI?"
UserName="USER_NAME=myuser"
Password="PASSWORD=mypassword"
/>
-----------------------------------------------------------------
```

3 If your implementation requires that the integration apply to some agents but not others, add a section like the following for each agent:

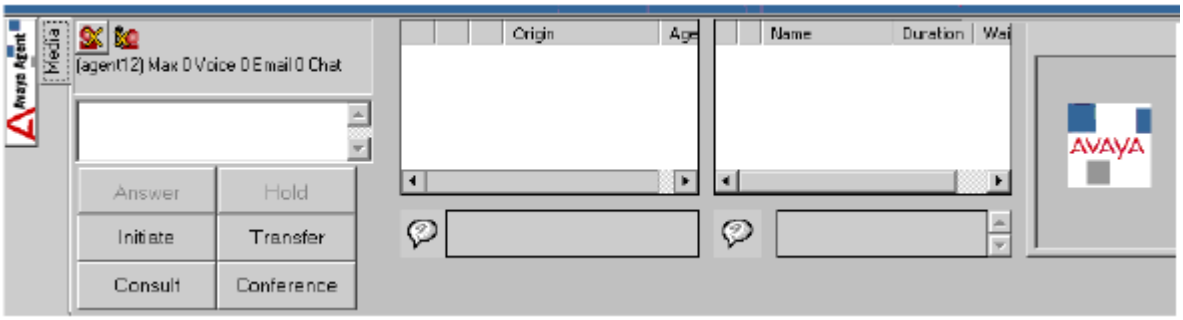
```
<QSection Name="agent1"
WebServer="epiphanyserver.mydomain.com"
WebServerPort="80"
EAIUrl="/epiphany/ecs/EAI?"
UserName="USER_NAME=myUser"
Password="PASSWORD=myPassword"
/>
```

4 If the Avaya agent supports email, and the email icon is not visible, then change the line in the Avaya_agent_en_customize_71.cdl file:

`<QControl Name="MailEngine" ProgID="AvayaEmail.MailEngineCtrl.71" Visible="FALSE">`to<QControl Name="MailEngine" ProgID="AvayaEmail.MailEngineCtrl.71" Visible="TRUE">

Rearranging the UI

For an Infor/Avaya integration, the right panel of the Avaya client screen is moved to the bottom panel, and the Contact and Prompter panes are both removed from the bottom panel.



The default bottom panel dimensions are 175 pixels high by 1000 pixels wide.

In the Infor-supplied CDL file, the relevant panel is configured as follows:

- The `<QFrame Name="Right_Frame" Width="200"Orientation="RIGHT">` is taken out. The following applies for Avaya 6x:
- The deleted QFrame entry is replaced with `<QFrame`
- Name="Bottom_Frame" Height="175" Orientation="BOTTOM"
- `Visible="TRUE">.`

The following applies for Avaya 7x:

- The deleted **QFrame entry is replaced with <QFrame Name="Bottom_Frame"**
- **Height="205" Orientation="BOTTOM" Visible="TRUE">.**

Table 4: CDL Customization

	Original Configuration	New Configuration
Call Control	`StatusCtrl Left="1" Top="1" Width="193" Height="51"`	`StatusCtrl Left="1" Top="1" Width="200" Height="51"`
	`CallListCtrl Left="2" Top="55" Width="190" Height="43"`	`CallListCtrl Left="2" Top="55" Width="200" Height="43"`
	`Name="tbAnswer" TelButtonCtrl Left="3" Top="100" Width="94" Height="32"`	`Name="tbAnswer" TelButtonCtrl Left="3" Top="100" Width="100" Height="32"`
	`Name="tbHold" TelButton Ctrl Left="97" Top="100" Width="94" Height="32"`	`Name="tbHold" TelButton Ctrl Left="110" Top="100" Width="100" Height="32"`
	`Name="btnInitiate" Prog ID="Avaya.AvayaButton Ctrl.71" Left="3" Top=`	`Name="btnInitiate" Prog ID="Avaya.AvayaButton Ctrl.71" Left="3" Top=`

	Original Configuration	New Configuration
	"132" Width="94" Height= "32"	"132" Width="94" Height= "32"
	Name="btnTransfer" Prog ID="Avaya.AvayaButton Ctrl.71" Left="97" Top= "132" Width="94" Height= "32"	Name="btnTransfer" Prog ID="Avaya.AvayaButton Ctrl.71" Left="97" Top= "132" Width="94" Height= "32"
	Name="btnConsult" Prog ID="Avaya.AvayaButton Ctrl.71" Left="3" Top= "164" Width="94" Height= "32"	Name="btnConsult" Prog ID="Avaya.AvayaButton Ctrl.71" Left="3" Top= "164" Width="94" Height= "32"
	Name="btnConference" ProgID="Avaya.AvayaBut-tonCtrl.71" Left="97" Top="164" Width="94" Height="32"	Name="btnConference" ProgID="Avaya.AvayaBut-tonCtrl.71" Left="97" Top="164" Width="94" Height="32"
Email Control	TaskListCtrl Left="2" Top="180" Width="190" Height="160"	TaskListCtrl Left="220" Top="1" Width="200" Height="160"
Chat Control	ChatListCtrl Left="2" Top="356" Width="190" Height="127"	ChatListCtrl Left="430" Top="1" Width="200" Height="160
EDU Viewer	EduViewerCtrl Left="490" Top="0" Width="300" Height="141" Visible= "TRUE"	EduViewerCtrl Left="640" Top="0" Width="300" Height="161" Visible= "TRUE"

Script Hooks

1 Add the Epiphany Scripts into the database:

Place the Epiphany.qsc file in the epiphany directory under Qconsole.

Place the **Avaya_agent_en_customize_71.cdl** file in the epiphany directory under Qconsole.

In the IC Script Editor load one of the Epiphany scripts into the editor window. Click on the '...' button for the filename. Locate the epiphany directory under qconsole. Highlight the epiphany_71.qsc file. Click the open button. Next click upload.In the IC Script Editor load one of the Epiphany scripts into the editor window. Click on the '...' button for the filename. Locate the epiphany directory under qconsole. Highlight the epiphany_71.qsc file. Click the open button. Next click upload.

You may have to enter the login/password for the database access. Once uploaded build the Avaya Windows application.

a Get Epiphany.qsc and the .cdl file.

For Avaya 7x, retrieve from : `<install_dir>\service\etc\cti\avaya\epiphany_71.qsc`

For Avaya 6x, retrieve from:

`<install_dir>\service\etc\cti\avaya\epiphany.qsc`

Move them to an installed Avaya qconsole directory. If you are usingAvaya IC 60, then the directory path would be: `c:\program files\Avaya\IC60\design\qconsole\epiphany`

Else, if you are using Avaya IC 71, the path would be:`c:\program files\Avaya\IC71\design\qconsole\epiphany`

b Add the file to the preloaded script files section in the Database Designer. In the Database Designer, click on the ccq.adl file in the viewer, and add the Epiphany.qsc.

c In the Database Designer, select **File > Generate Windows Application**. In the window that appears, check IC Scripts and click **OK**.

2 Add Epiphany Hooks to the IC Scripts:

a In the Database Designer, go to IC Script Editor.

b Make sure that the Infor scripts are in the database. Note: All the files start with “Epiphany.”

c Go to the script PhoneEngine_OnTelephonyStateChanged.

d Add the following line before the script exits.

3 Declare Function Epiphany_ProcessTelephonyEvent iTelephonyEvent.

a Save and upload the script.

b Go to the Script Shared_OnSelec

c Add the following line before the script exits:

4 Declare Function Epiphany_ProcessSelectEvent(sEduId As String,lMediaType As Long)

a Save and upload the script.

5 Add the following script hooks:

a Agent Hook. Add this hook to BlenderClient:**<QScript Event="ADUChange" Name="Epiphany_ProcessAgentEvent"/>**

b Email Script Hook. Add this line to the Script dictionary of the TaskListCtrl.**<QScript Event="Change" Name="Epiphany_ProcessEmailEvent"/>**

c Persist the changes.

d In the Database Designer, select **File > Generate Windowsapplication**.

e Check Avaya Agent Layout, and click **OK**.

Configuring Infor For Avaya

The Avaya provider is a Java-based client to server integration using Infor Enterprise Application Integration (EAI) framework. In the Infor integration, the Avaya Interaction Center is customized to post messages to Infor over HTTP. This enables Infor CTI to monitor agent state, email messages, and

phone calls. This integration provides a one-way integration, and does not include agent or call control in the Infor user interface.

Before enabling the Avaya provider, CTI must be enabled. For more information, refer to"Configuring the Base Provider" on page 95on page 7-2. Agents must also have agent IDs configured. DNs and splits (skill groups) are not required.

To enable the Avaya Provider:

1. Open Studio and connect to the database.
2. In the Guide Bar, expand Administration.
3. Select **CTI Manager > Parameters > Providers**.
4. In the Properties window, expand Avaya.
5. Set **Enabled** to true.
6. In the Guide Bar, click on **Providers**. Make sure that all of the other providers have their enabled flag set to false. By default, the Base Provider comes enabled.
7. Restart the Infor server for the changes to take effect.

About Client-Side Integration

The Avaya provider is a client-side integration. There are two major components, the Avaya integration itself, and the client-side connector.

The out-of-the-box Avaya IC integration has both the Infor UI and the Avaya UI on the screen. The Avaya UI takes the bottom fifth of the screen, and is always on top. The Infor UI occupies the rest of the screen. Phone calls and e-mail messages are tracked through Avaya. When a call or e-mail message gets focus in the Avaya application, the appropriate events are sent to the Infor server using an EAI connector via a HTTP `post` operation. When the selection in the Avaya application is changed, the UI tabs in the Infor application are switched to match the Avaya selection. When a call is closed, Infor presents the Save and Close and Save and Queue buttons.

The EAI connector is more general. It was designed to allow for an external system to drive simple CTI and e-mail interactions. Since Infor is receiving data from the Avaya CTI system in a one way connection, all CTI agent and call controls are disabled. Infor CTI supports receiving new calls, new e-mail messages, changing the current focus (e-mail or call), end call, and agent state change. From the Infor UI, the user only has the option of whether or not to connect to the Avaya integration. When not connected, the server does not send messages to the Infor client. The Avaya interface is not affected.

Index

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

W