opentext™

Service Desk 25.1

ZENworks Service Desk Developer Resources

January 2025

Contents

About This Guide

This document includes ZENworks Service Desk developer resources.

Audience

This guide is intended for developers.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Additional Documentation

ZENworks Service Desk is supported by other documentation that you can use to learn about and implement the product. For additional documentation, see the ZENworks Service Desk documentation site.

1 ZENworks Service Desk Developer Resources

This appendix contains some useful ZENworks Service Desk developer resources.

1.1 Rest APIs

The REST APIs can be accessed using the following URL:

https://<*zenworks_service_desk_server*>/servicedesk-apidocs/

Where *zenworks_service_desk_server* is ZENworks Service Desk server IP or hostname.

1.1.1 Add note to the request – Technician Portal

[POST] https://<zsdserver>/LiveTime/services/v1/user/requests/{requestId}/notes

Request parameters

```
Content-Type: multipart/form-data
Body:
noteData: "{
"noteTime":5,
"noteText":"Test note from rest call",
"hidden": false,
"customers": true,
"customerCcs": true,
"technicians": true,
"technicianCcs": true,
"vendors": false,
"addToRelatedIncidents": false,
"addTimeToRelatedIncidents": false,
"createKnowledge": false,
"needInfo": false,
"includeDirectLink": true,
"action":"ADDNOTE",
"existingAttachmentIds": [10,20],
"selectedTechnicianIds": [21,34],
"customerCcAddresses":"user1@zsd.com, user2@zsd.com",
"technicianCcAddresses":"user10@zsd.com, user21@zsd.com",
"parentId": 10
}"
```

NOTE: For adding attachments, you can provide any name for file and select the required file, but if additional file information (like private visibility and description) has to be provided, then use the same name for this data also. For more information, refer to the attached screenshots.

All available actions:

ADDNOTE, SOLUTION, PROPOSE, DRAFT, DELETEDRAFT

Get applicable actions for the request

[GET] https://<zsdserver>/LiveTime/services/v1/user/requests/{requestId}/requestActions

“addNote” - true (for ADDNOTE)

[GET] https://<zsdserver>/LiveTime/services/v1/user/requests/{requestId}/noteprivileges

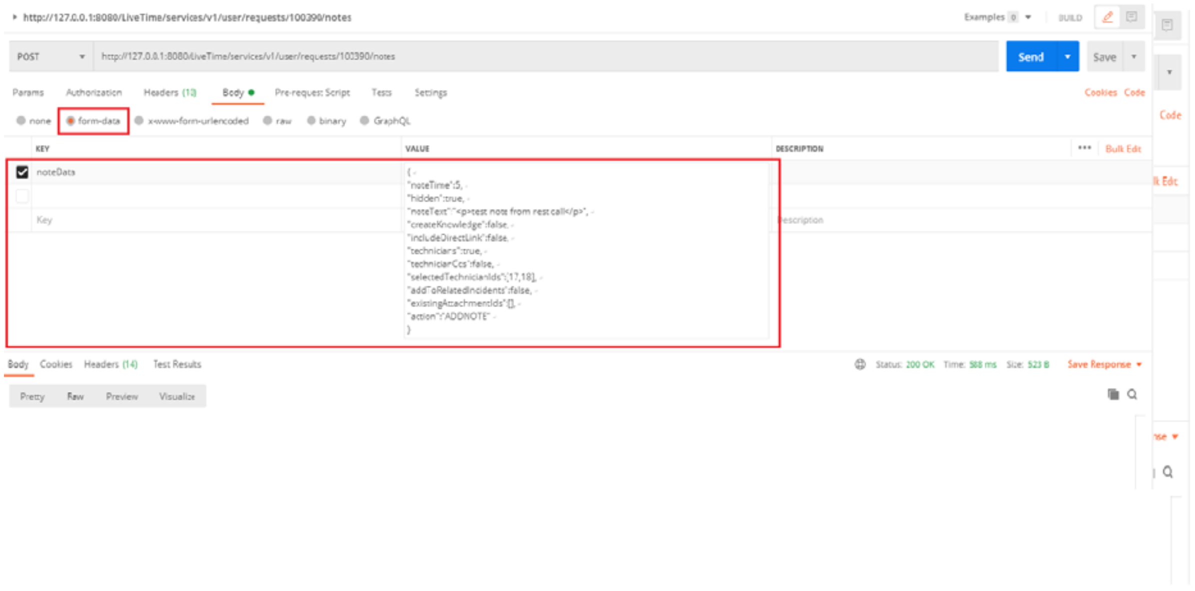
```
"isDraft"  -  true   (for DELETEDRAFT)
  "showSolutionButton" -  true (for SOLUTION)
  "showProposeButton" -  true (for PROPOSE)
```

NOTE: Some parameters in the add note service are eligible if the respective value is true in the following service:

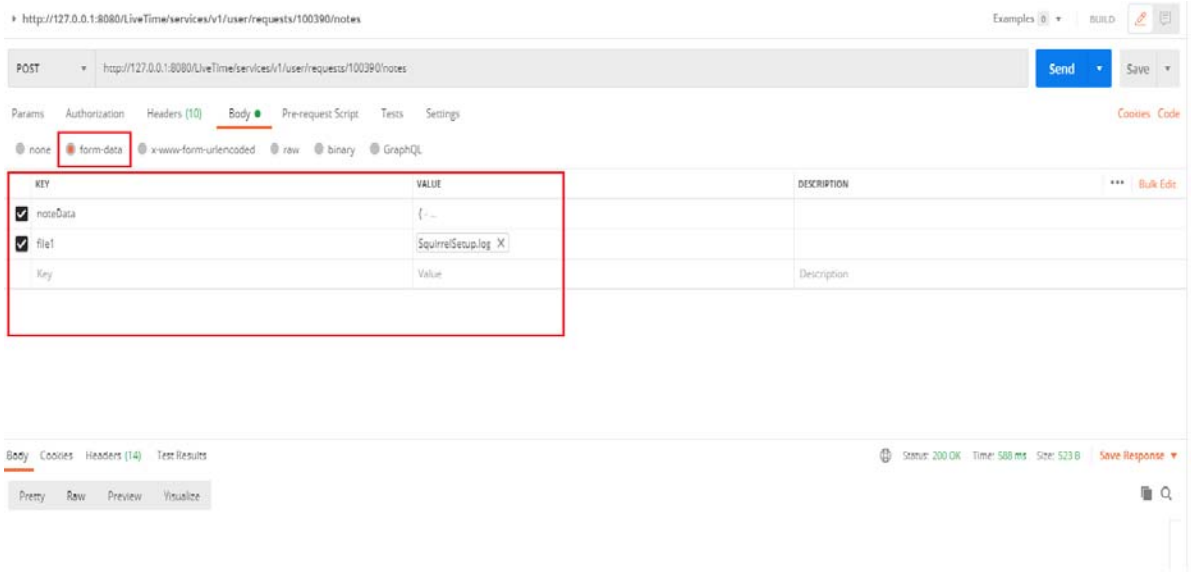
[GET] https://<zsdserver>/LiveTime/services/v1/user/requests/{requestId}/noteprivileges

For example : “createKnowledge” can be set in addnote service if “showCreateKnowledge” is true in the noteprivilege service.

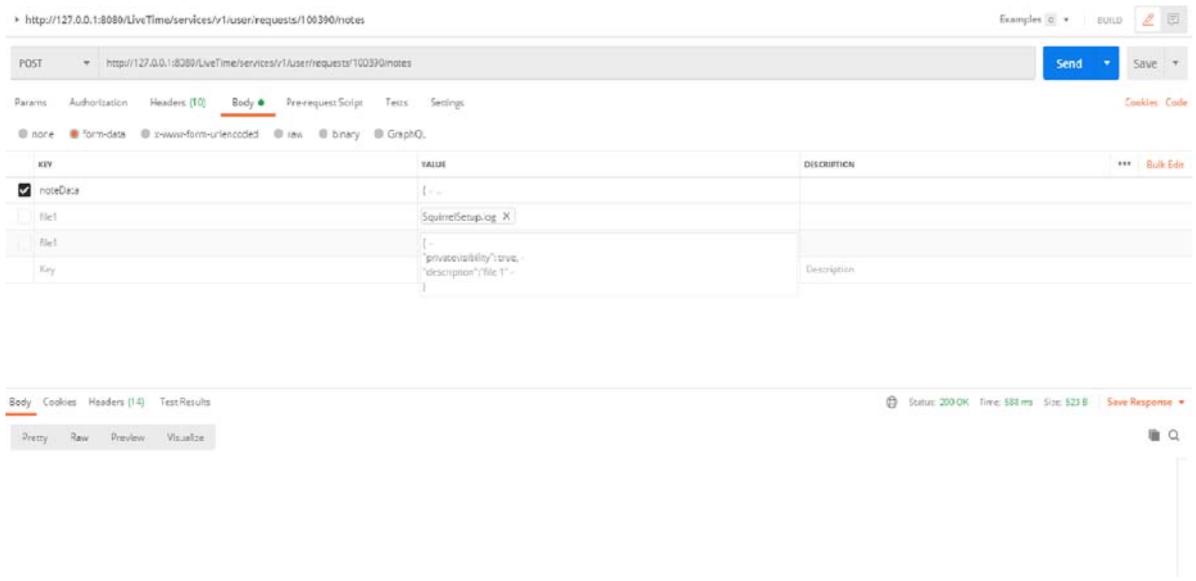
Examples



```
▸ http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/notes

POST ▾   http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/notes      Send ▾   Save ▾

Params   Authorization   Headers (13)   Body ●   Pre-request Script   Tests   Settings      Cookies  Code

○ none   ● form-data   ○ x-www-form-urlencoded   ○ raw   ○ binary   ○ GraphQL

KEY          VALUE                                              DESCRIPTION        ••• Bulk Edit
☑ noteData   { -
             "noteTime":5, -
             "hidden":true, -
             "noteText":"<p>test note from rest call</p>", -
             "createKnowledge":false, -
             "includeDirectLink":false, -
             "technicians":true, -
             "technicianCcs":false, -
             "selectedTechnicianIds":[17,18], -
             "addToRelatedIncidents":false, -
             "existingAttachmentIds":[], -
             "action":"ADDNOTE" -
             }
  Key                                                            Description

Body   Cookies   Headers (14)   Test Results      Status: 200 OK   Time: 588 ms   Size: 523 B   Save Response ▾
Pretty   Raw   Preview   Visualize
```



```
▸ http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/notes

POST ▾   http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/notes      Send ▾   Save ▾

Params   Authorization   Headers (10)   Body ●   Pre-request Script   Tests   Settings      Cookies  Code

○ none   ● form-data   ○ x-www-form-urlencoded   ○ raw   ○ binary   ○ GraphQL

KEY          VALUE                  DESCRIPTION        ••• Bulk Edit
☑ noteData   { - ...
☑ file1      SquirrelSetup.log X
  Key        Value                  Description

Body   Cookies   Headers (14)   Test Results      Status: 200 OK   Time: 588 ms   Size: 523 B   Save Response ▾
Pretty   Raw   Preview   Visualize
```



POST http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/notes

Body: form-data

KEY	VALUE	DESCRIPTION
✓ noteData	{ ...	
file1	SquirrelSetup.log	
file1	{ "privatevisibility":true, "description":"file 1" }	

Status: 200 OK Time: 588 ms Size: 523 B

1.1.2 Bulk Request Update (Close, Reopen and Delete) – Technician Portal

[PUT] https://<zsdserver>/LiveTime/services/v1/user/requests/bulkUpdate

Query Parameters

```
"currentPage" : String ( Allowed values - mytask, incident, service,
change, problem)
"filterId" : int
"includeAll" : Boolean
"includedExcludedIds" : List<Integer>
```

Request Parameters

```
Content-Type: multipart/form-data
Body:
data: "{
"action":"CLOSE",
"searchCriteria" : {
  "slaBreachedOnly": false,
  "escalatedOnly": false,
  "awaitingInfoOnly": false,
  "awaitingMyApprovalOnly": false,
  "searchTerm" : "test",
      "requestFilterInfo" : {
"type": "incident", (incident/service/change/problem)
"scope": "My Tasks", (My Tasks, My Team Tasks)
"incidentId": "100003",
"groupName": "test",
"loggedBy": "User1",
"technicians": [12, 15],
"teams": [2, 8],
"workflowId": 7,
"statusSearchType": 0, (Allowed values: 0/1/2/3)
"statusTypes": [12,14],
"problemtypeId": 10,
"customClassString": "test",
"priorityTypes": [1, 2],
"solId": 1,
"waId": 2,
"firstName": "first name",
"lastName": "last name",
"email": "user1@gmail.com",
"orgUnit": "My Company",
"clientRoom": "10",
"clientCity": "bangalore",
"incidentLocation": "Chennai",
},
"requestAdditionalSearchInfo" : {
  "incidentId": 100120,
  "item": "test item",
  "itemType": "Software",
  "itemStatus": "Available",
  "team": "Incident team",
  "workflow": "Incident workflow"
}
}
}"
```

All Available Actions

CLOSE, REOPEN and DELETE.

NOTE: 1.If “includeAll “ is false then action will be performed on the requests provided in the “includedExcludedIds “. (The other request search parameters will not be considered)

2. If “includeAll “ is true, then the action will be performed on the requests depends on the search criterias provided. In this case the request ids provided in the “includedExcludedIds” will be excluded from the results.
3. If any request is not eligible for the provided action, that request will be skipped.

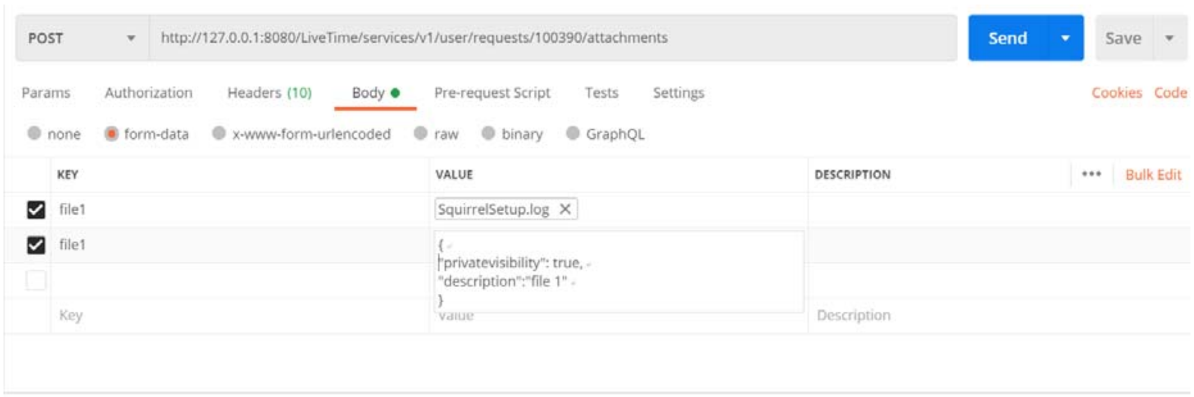
1.1.3 Add Attachment to the Request/Note – Technician Portal

[POST] https://<zsdserver>/LiveTime/services/v1/user/requests/100002/attachments

Request Parameters

```
Content-Type: multipart/form-data
Body:
  "file1": Selected File,
"file1": {
  "privatevisibility": true,
  "description": "Request Attachment 1"
},
  "file2": Selected File,
"file2": {
"privatevisibility": false,
  "description": "Request Attachment 2"
}
```

NOTE: For adding attachments, you can provide any name for file and select the required file, but if additional file information (like private visibility and description) has to be provided, then use the same name for this data also. For more information, refer to the above example and the following screenshot.



http://127.0.0.1:8080/LiveTime/services/v1/user/requests/100390/attachments

	KEY	VALUE	DESCRIPTION
✓	file1	SquirrelSetup.log	
✓	file1	{ "privatevisibility": true, "description":"file 1" }	
	Key	Value	Description

1.2 Developer

- Section 1.2.1, “Integrations,” on page 12
- Section 1.2.2, “Connectors,” on page 13

1.2.1 Integrations

SOAP APIs are no longer supported. It is recommended that you use REST APIs. For more information, see Rest APIs or contact OpenText Customer Support.

1.2.2 Connectors

service desk AMIE engine (Asset Management Integration Engine) provides a unique mechanism to connect to any third party Asset Management system.By default service desk provides many built-in connectors. This is expanding every release. One of the unique properties of connectors built using AMIE is that they readily adapt to schema changes on the foreign host so in most cases there is no need to change them between upgrades.

Users have the ability to create their own mapping files and we provide some great examples on how to do this step by step in AMIE Part I and AMIE Part II.

1.3 Localizing Service Desk ITSM

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first i and the last n.

The Service Desk product suite is an internationalized program that has the following characteristics:

- With the addition of localized data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hard coded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require re-compilation.
- Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- Service Desk can be localized quickly.

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text. The term localization is often abbreviated as l10n, because there are 10 letters between the l and the n.

The Service Desk internationalized framework is very intuitive and flexible, allowing Users to localize it for a particular language and character encoding scheme.

- Section 1.3.1, "Getting Started," on page 14
- Section 1.3.2, "Customizing Strings," on page 14
- Section 1.3.3, "Simple Messages," on page 14
- Section 1.3.4, "Compound Messages," on page 15
- Section 1.3.5, "Locales," on page 15

1.3.1 Getting Started

The primary task of Users performing localization of the Service Desk application is to translate the user interface elements in the external text file – LiveTime.properties.

The following steps guide you through the localization process. We will assume a French localization for this example, which uses the ISO-8859-1 character set.

1. Ensure you have the appropriate character set installed on your Desktop's Operating System. The localization file and rendering will both require the character set to be available for data entry and data presentation.
2. Ensure your RDBMS has been created using the appropriate character set, usually UTF-8. This is required so that French data entry can be successfully saved and retrieved from the database.
3. Specify the correct character set encoding for Service Desk. Edit the 'Properties' file (/ CONTAINER_APPS_PATH>/LiveTime/WEB-INF/LiveTime.woa/Contents/Resources folder) and specify the character set to use for presentation to the end user. This character set will be sent to the end users to control presentation in both the user interface and outbound email.

1.3.2 Customizing Strings

The LiveTime.properties file (/CONTAINER_APPS_PATH>/LiveTime/WEB-INF/LiveTime.woa/ Contents/Resources folder) stores the externalized user interface content. This file is stored in plain text. The names of the localized files do follow a standard. In our example of French, livetime_fr.properties was the resultant file name. The fr component of this name comes from iso 639-1. The default (English) is encoded using the ISO-8859-1 character set. You can create alternative versions of this file in any text editor, that allows the file to be saved in the necessary character set.

The first step in creating a local version of this file is to save a copy of it representing the appropriate language. Being a French translation, there is no need to worry about the file encoding, so simply save a copy of the file called livetime_fr.properties.

1.3.3 Simple Messages

The copy of this file consists of a collection of name-value pairs, each are broken up into clusters representing user interface components. For example:

```
ClassName.greetings=Hello

ClassName.farewell=Goodbye

ClassName.inquiry=How are you?
```

The name or key section (ClassName.greetings) of these strings is of little relevance to the end user. They are simply markers inside the application representing where each string goes.

The value component of these pairs (Hello) represents the corresponding user interface value. The (French) translator can safely edit these values and they will appear within Service Desk.

```
ClassName.greetings=Bonjour.

ClassName.farewell=Au revoir.
ClassName.inquiry=Comment allez-vous?
```

1.3.4 Compound Messages

Compound messages contain variable data – that is data that Service Desk will substitute into the message at run time. For example, given the message Hello Sunshine, the string Sunshine may vary. This message is difficult to translate because the position of the string in the sentence is not the same in all languages, or the value may need to be generated by Service Desk based on some business logic.

Where this substitution is necessary, the localized strings will have a set of braces containing an id number (a substitution variable). These blocks will be replaced at run time by Service Desk and are identified by their presence. The localized version of a given String must contain the same number of substitution variables.

For example:

```
ClassName.helloSunshine=hello {0}
```

This will render as hello sunshine; however we could modify this message to read sunshine, hello by doing this:

```
ClassName.helloSunshine={0}, hello
```

1.3.5 Locales

The LiveTimeLocales.xml file (/CONTAINER_APPS_PATH>/LiveTime/WEB-INF/LiveTime.woa/Contents/Resources folder) defines the language code and charset used in order to process the strings defined in the external properties file.

Typically, this will be the charset used when saving the external properties file. By default the following language locale is selected:

```
<LOCALE language="en" country="" charset="ISO-8859-1"/>
```

To enable our French external properties file to behave correctly, we will need to uncomment the following line:

```
<LOCALE language="fr" country="" charset="UTF-8"/>
```

This will enable the application to use both English and French based on the user's browser and operating system settings. For convenience, many values are already supplied in this file, you need to uncomment the necessary entry.

If an entry for your locale language doesn't exist, then you can define it in this file by supplying the language code, country code, and charset, as illustrated in the above examples.

Instructions for setting browser locales is outside the scope of this document as each browser/OS combo handles this differently.

1.4 ZSD Extensions

ZENworks Service Desk (ZSD) has the ability to add customized extensions to request workflow state transitions and item lifecycle state transitions. In order to implement custom functionality on these state transitions, there are several steps a user must follow:

1.4.1 Building the Extension

Two java interfaces are provided, in a standalone jar file ‘livetime-listen.jar’, which can be found in:

```
{ZSD Installation Folder}/LiveTime.woa/Contents/Resources/Java/
```

This jar file contains two interfaces, each of which has two methods:

```
WorkflowListener
public Map<String, String> stateEntered(Map<String, Object>argsMap) throws Exception
public Map<String, String> stateExited(Map<String, Object>argsMap) throws Exception
LifecycleListener
public Map<String, String> stateEntered(Map<String, Object>argsMap) throws Exception
public Map<String, String> stateExited(Map<String, Object>argsMap) throws Exception
```

Naturally the WorkflowListener is used for integrating with the Request Workflow, whilst the LifecycleListener is used for integrating with the Item Lifecycle.

The task for the developer is to create a java class that implements the appropriate interface to achieve the integration objective. The two methods exist to provide flexibility to the developer implementing the integration, by allowing them to perform tasks based on a state being ‘exited’ and then a state being ‘entered’.

The implementing class is required to return a Map for all methods. Non-implemented interfaces can return null, and this will be treated as a no-op internally. Methods that provide functionality should return a map which will be parsed for two parameters:

- ‘success’ : ‘true’ or ‘false’
- ‘message’ : Description to be stored against the history of the request or item

Each method is passed a Map of parameters that relate to the request or item being updated. ZSDZSD will pass a String for all values, the method takes for future extensions that may require the use of Objects. Implementations should check the object is a String prior to casting to future-proof the implementation.

1.4.2 WorkflowListener Arguments

The parameter Map passed in to the WorkflowListener interface methods consists of:

Parameter	Description
triggerStatusId	The ID of the status that has triggered the listener event: ◆ statusExited: originalStatus ◆ statusEntered: newStatus
triggerStatusName	The name of the status that triggered the listener event
requestId	The ID of the request being updated ◆ 1000 = Incident ◆ 2000 = Problem ◆ 3000 = Change Request ◆ 7000 = Service Request ◆ 9000 = Deployment Task
requestType	The type of request being updated ◆ 1000 = Incident ◆ 2000 = Problem ◆ 3000 = Change Request ◆ 7000 = Service Request ◆ 9000 = Deployment Task
statusId	The current state of the request: ◆ statusExited: newStatus ◆ statusEntered: same as triggerStatus
statusName	The name of the state defined by statusId above
classificationId	The ID of the classification assigned to the request
customerId	The ID of the customer assigned to the request
customerFirstName	The first name of the customer assigned to the request
customerLastName	The last name of the customer assigned to the request
customerEmail	The email address of the customer assigned to the request
orgUnitId	The ID of the Org Unit associated with the request
orgUnitName	The name of the Org Unit associated with the request
itemNumber	The item number of the CI associated with the request

As this is a first iteration of the call-out interface, these parameters have been deemed sufficient to allow a developer to perform non-trivial tasks like update an external system that may require request updates.

1.4.3 LifecycleListener Arguments

The parameter Map passed in to the LifecycleListener interface methods consists of:

Parameter	Description
triggerStatusId	The ID of the status that has triggered the listener event: ◆ statusExited: originalStatus ◆ statusEntered: newStatus
triggerStatusName	The name of the status that triggered the listener event
itemNumber	The Item Number assigned to the CI
itemStatusId	The current state of the request: ◆ statusExited: newStatus ◆ statusEntered: same as triggerStatus
itemStatusName	The name of the state defined by statusId above
itemtypeId	The Item Type ID
itemtypeName	The name of the Item Type the Item is an instance of
categoryId	The Category ID
categoryName	The name of the Category the Item Type belongs to

These are the initial parameters deemed appropriate to allow a developer to perform non-trivial tasks like (for example) to feed state changes into a monitoring tool to reset an alert trigger.

1.4.4 Implementing a Listener

This requires some Java knowledge, to either define the entire functionality, using these entry points, or to create a JNI wrapper to page out to code written in an alternate language, although this does have platform implications. A JNI (Java Native Interface) implementation is outside the scope of this document, and the following sample focuses on a Java Implementation.

The user needs to create a class, for example ‘MyWorkflowListener’ which implements the WorkflowListener Interface, or MyLifecycleListener, which implements the LifecycleListener interface. These methods then need to be made to perform some work. A non-trivial example would be one where the listener calls a web service to an external system. Consider the following usage scenario.

A company has (for whatever reason) two ZSD instances, and they are needing to, on occasion feed updates to the secondary system on request state changes. ZSD has an inbound web services interface, and now, an outgoing interface for communicating changes to third parties.

The SOAP WSDL's can be used to generate Java classes to make calls into the secondary system, which can now be called from the outgoing interface. Generating the SOAP equivalent java classes, and calling them from a WorkflowListener might yield a listener class that looks something like the following.

```
package com.livetime.sample;
import com.livetime.ws.listen.WorkflowListener;
import java.util.Map;
public class WorkflowListenerImpl implements WorkflowListener
{
public WorkflowListenerImpl() {}
public Map<String, String> statusEntered(Map<String, Object>argsMap) throws Exception
{BaseRequest baseRequest = new BaseRequest();
String requestId = (String)argsMap.get("requestId");
String statusName = (String)argsMap.get("triggerStatusName");
Map temp = new HashMap();
temp.put("subject", "Request Created via Outbound WebServices Call");
temp.put("description", "Request #" + requestId + " has entered status " + statusName);
return baseRequest.createRequest(temp);
}
public Map<String, String> statusExited(Map<String, String>argsMap) throws Exception
{
BaseRequest baseRequest = new BaseRequest();
String requestId = (String)argsMap.get("requestId");
String statusName = (String)argsMap.get("triggerStatusName");
Map temp = new HashMap();
temp.put("subject", "Request Created via Outbound WebServices Call");
temp.put("description", "Request #" + requestId + " has exited status " + statusName);
return baseRequest.createRequest(temp);
}
}
```

With the createRequest method in the class BaseRequest looking like this:

```
public Map<String, String> createRequest(Map<String, String>
properties)
{
// try and connect, if successful, create request and logout
 if(connect()) {
java.net.URL requestURL = null;
HashMap response = new HashMap();
try {
// Service endpoint
requestURL = new
java.net.URL(props.getRequestServiceURL());
// Get handle to the service
Request_Service service = new
Request_ServiceLocator();
Request_PortType port = service.getRequest(requestURL);
// Call BaseClient method to populate persistent
headers populateHeaders((javax.xml.rpc.Stub)port);
String subject = (String)properties.get("subject");
String description = (String)properties.get("description");
response = port.createIncident(props.getTargetItemNumber(),
props.getTargetClassificationId(), subject,
description, new HashMap());
}
catch(Exception ex)
{
```

```
return buildErrorMessage("An error occurred whilst creating");
}
if(disconnect())
{
return response;
}
else {
return buildErrorMessage("An error occurred whilst disconnecting");
}
}
else {
return buildErrorMessage("An error occurred whilst connecting");
}
}
```

In this example, the listener simply creates a new request in the second system, stating the nature of the change, but this highlights some of the possibilities of outbound functionality.

NOTE: This example has been heavily truncated to illustrate the key functionality.

1.4.5 Making the Extension Available to ZSD

This is a rather simple process for users with an install on their own infrastructure.

In order for the class to be accessible to ZSD, the compiled code needs to be on the ZSD classpath. In this case, this means the compiled jar, along with any associated components, need to be copied into the lib folder of the ZSD installation ({ZSD Path}/WEB-INF/lib), and the ZSD instance needs to be restarted, so these resources are picked up by the classloader.

1.4.6 Configuring ZSD to Use the Extension

At this point a new class (or classes) exist and have been loaded, now ZSD simply needs to be configured to use them. This is a two part process. Firstly the option needs to be enabled by a system administrator to enable outbound web services. This option can be found in the Administrator portal, under **Setup** > **Privileges** > **System** (Outbound Web Services).

Once set to ‘On’ and saved, a new field appears in both the Workflow State and Lifecycle State Editors respectively. The ‘Listener Class’ can now be populated per workflow (or lifecycle) state, allowing each state to call the same, or different implementations as necessary. This allows different workflows to behave per the designers’ requirements.

1.5 ZSD Store Extensions

ZENworks Service Desk (ZSD) has the ability to add the customized store extensions for the auto assignment of store item. In order to implement custom functionality on the assignment state follow:

1.5.1 Building the Extension

Java interface provided in a standalone jar file `livetime-listen.jar`, and it is available at:

`{ZSD Installation Folder}/LiveTime.woa/Contents/Resources/Java/`

The `livetime-listen.jar` file contains the `ExternalStoreExtension` interface, with the following method:

`public Map<String, String> statusEntered(Map<String, Object> argsMap)` throws `Exception`;

The developer task is to create a java class that implements above interface to achieve the integration objective.

The implementing class is required to return a Map. Method that provide functionality should return a map which will be parsed for the following parameters:

- `success`: '`true`' or '`false`'
- `message`: Description to be stored against the history and note of the store request.

The method is passed a Map of parameters that relate to the store request being updated. ZSD will pass a string for all the values, this method takes for future extensions that might require the use of objects. Implementations should check whether the object is a String prior to casting to future-proof the implementation.

1.5.2 ExternalStoreExtension Arguments

The parameter Map passed in to the `ExternalStoreExtension` interface method consists:

Parameter	Description
`triggerStatusId`	ID of the status that has triggered the extension: ◆ `statusEntered`: `newStatus`
`triggerStatusName`	Name of the status that triggered the extension.
`requestId`	ID of the request being updated.
`requestType`	Type of the request being updated. ◆ 7000 = Service Request
`statusId`	Current state of the request same as assignment state.
`statusName`	Name of the state defined by statusId above.
`classificationId`	ID of the classification assigned to the request.
`customerId`	ID of the customer assigned to the request.
`customerFirstName`	The first name of the customer assigned to the request.

Parameter	Description
`customerLastName`	The last name of the customer assigned to the request.
`customerEmail`	Email address of the customer assigned to the request.
`orgUnitId`	ID of the Org Unit associated with the request.
`orgUnitName`	Name of the Org Unit associated with the request.
`itemNumber`	Item number of the CI associated with the request.

As this is the first iteration of the call-out interface, these parameters have been deemed sufficient to allow a developer to perform tasks such as update an external system that might require request updates.

1.5.3 Implementing the Store Extension

Requires Java knowledge, to either define the entire functionality, using these entry points, or to create a Java Native Interface (JNI) wrapper to page out to code written in an alternate language, although this does have platform implications. A JNI implementation is outside the scope of this document, and the following sample focuses on a Java Implementation.

The user needs to create a class, for example ‘`MyStoreExtensionImpl`’ which implements the `ExternalStoreExtension` interface. Implemented method then needs to be made to perform some work. For example, the extension calls a web service to an external system. Consider the following usage scenario:

```
package com.microfocus;

import java.util.HashMap;
import java.util.Map;
import com.novell.store.extension.external.ExternalStoreExtension;

public class BusinessCardExtensionImpl implements ExternalStoreExtension
{
       /**
     * This method will be called at the entry of the state, i.e if workflow has state
hierarchy like state1-> Assignment State-> State2 then this will be called on transition
of state1 to Assignment State
     * @param argsMap
     * @return a map. This map must contain 2 entries.
     *         key=message value=A user friendly message summarizing the result of action
performed by this extension.
     *         key=success value=true|false in String. this denotes the status of the
extension process.
     * @throws Exception
     */
       @Override
     public Map<String, String> statusEntered(Map<String, Object> arg0)
      throws Exception {
    boolean success = doSomething(); // Your actual business logic goes here.
    Map<String, String> response = new HashMap<>(); //This message will be added as part
of the Note and Audit Trail. E.g

    if(success)
             response.put("message", "Order for Business card has been successfully
placed.");
    else
             response.put("message", "Order for Business card has been failed.");
```

```
            response.put("success", String.valueOf(success));

            return response;
  }

  private boolean doSomething() {
    //
    //Do Something. Invoke external services handling order for business cards.
    //
    boolean success = true;
    return success;
    }
}
```

1.5.4 Making the Store Extension Available to ZSD

In order for the class to be accessible to ZSD, the compiled code needs to be on the ZSD classpath. In this case, the compiled jar, along with any associated components, need to be copied into the lib folder of the ZSD installation (`{ZSD Path}/WEB-INF/lib`), and the ZSD instance needs to be restarted, so these resources are picked up by the classloader.

1.5.5 Configuring ZSD to Use the Extension

At this point a new class or classes exist and have been loaded, now ZSD needs to be configured to use them.

- Enable the Store feature. For information, see *Enabling Store*.
- Create the Store Extension. For information, see *Creating a Store Extension*.
- Assign the extension to applicable Item Category or Item Type.