Kofax Customer Communications Manager

Core Developer's Guide

Version: 5.2

Date: 2018-11-19

Table of Contents

Preface

This guide provides information on the configuration and use of Kofax Customer Communications Manager Core (also known as CCM Core) and on the integration of CCM Core with various interfaces.

Related documentation

The documentation set for Customer Communications Manager is available here:[1]

https://docshield.kofax.com/Portal/Products/CCM/520-nz7r6s9geq/CCM.htm

In addition to this guide, the documentation set includes the following items:

Kofax Customer Communications Manager Release Notes
Contains late-breaking details and other information that is not available in your other Kofax Customer Communications Manager documentation.

Kofax Customer Communications Manager Getting Started Guide
Describes how to use Contract Manager to manage instances of Kofax Customer Communications Manager.

Kofax Customer Communications Manager Designer User's Guide
Contains general information and instructions on using Kofax Customer Communications Manager Designer, which is an authoring tool and content management system for Kofax Customer Communications Manager.

Kofax Customer Communications Manager Repository Administrator's Guide
Describes administrative and management tasks in Kofax Customer Communications Manager Repository and Kofax Customer Communications Manager Designer for Windows.

Kofax Customer Communications Manager Repository User's Guide
Includes user instructions for Kofax Customer Communications Manager Repository and Kofax Customer Communications Manager Designer for Windows.

Kofax Customer Communications Manager Repository Developer's Guide
Describes various features and APIs to integrate with Kofax Customer Communications Manager Repository and Kofax Customer Communications Manager Designer for Windows.

[1] You must be connected to the Internet to access the full documentation set online. For access without an Internet connection, see "Offline documentation" in the Installation Guide.

Kofax Customer Communications Manager Template Scripting Language Developer's Guide
Describes the CCM Template Script used in Master Templates.

Kofax Customer Communications Manager Core Developer's Guide
Provides a general overview and integration information for Kofax Customer Communications Manager Core.

Kofax Customer Communications Manager Core Scripting Language Developer's Guide
Describes the CCM Core Script.

Kofax Customer Communications Manager API Guide
Describes Contract Manager, which is the main entry point to Kofax Customer Communications Manager.

Kofax Customer Communications Manager ComposerUI for HTML5 JavaScript API Web Developer's Guide
Describes integration of ComposerUI for HTML5 into an application, using its JavaScript API.

Kofax Customer Communications Manager Batch & Output Management Getting Started Guide
Describes how to start working with Batch & Output Management.

Kofax Customer Communications Manager Batch & Output Management Developer's Guide
Describes the Batch & Output Management scripting language used in CCM Studio related scripts.

Kofax Customer Communications Manager DID Developer's Guide
Provides information on the Database Interface Definitions (referred to as DIDs), which is an alternative method retrieve data from a database and send it to Kofax Customer Communications Manager.

Getting help for Kofax products

Kofax regularly updates the Kofax Support site with the latest information about Kofax products.

To access some resources, you must have a valid Support Agreement with an authorized Kofax Reseller/Partner or with Kofax directly.

Use the tools that Kofax provides for researching and identifying issues. For example, use the Kofax Support site to search for answers about messages, keywords, and product issues. To access the Kofax Support page, go to www.kofax.com/support.

The Kofax Support page provides:

- Product information and release news
  Click a product family, select a product, and select a version number.
- Downloadable product documentation
  Click a product family, select a product, and click **Documentation**.
- Access to product knowledge bases
  Click **Knowledge Base**.

- Access to the Kofax Customer Portal (for eligible customers)

  Click **Account Management** and log in.

  To optimize your use of the portal, go to the Kofax Customer Portal login page and click the link to open the *Guide to the Kofax Support Portal*. This guide describes how to access the support site, what to do before contacting the support team, how to open a new case or view an open case, and what information to collect before opening a case.
- Access to support tools

  Click **Tools** and select the tool to use.
- Information about the support commitment for Kofax products

  Click **Support Details** and select **Kofax Support Commitment**.

Use these tools to find answers to questions that you have, to learn about new functionality, and to research possible solutions to current issues.

Chapter 1

Introduction to the functionality

CCM Core is a server application that provides the template processing and scripting functionality for CCM.

CCM Core provides an extensive range of safety and load balancing features to make it a robust document production application. You can add processing capacity or remove it dynamically without interruption to the production process.

CCM Core is designed to be deployed as one or more instances in a CCM installation. In this role it receives jobs through CCM Contract Manager. A collection of APIs is provided to submit jobs directly for legacy configurations.

CCM Core is used as an internal component that implements the CCM Contract Manager functionality. Which tasks CCM can perform is determined in its **Services**. For example, a CCM Core setup can contain a task to take the parameters from a request and use them to run a specific Template. Also, a task could be to take the output of the ITP process, send it to an archiving system, convert it to PDF, and send it together with an introducing email to the customer or just send the output to a printer and send an email to the operator. For more information on Services, see CCM Core Services.

How a task is performed is determined in a **Script**, which is a series of commands. Scripts are written in the Core scripting language. CCM Core comes with a script editor to facilitate script writing. For every script, all settings defined on the Constants tab of CCM Core Administrator are accessible as global constants. For additional information on scripts, see the *Kofax Customer Communications Manager Core Scripting Language Developer's Guide*.

CCM Core is controlled with **CCM Core Administrator**. CCM Core Administrator provides information about the status of CCM Document Processor Manager and the CCM Document Processors and gives access to all necessary settings. Also, it gives you the ability to start, stop, restart, and add CCM Core and CCM Document Processors on the Servers node. CCM Core Administrator can control CCM Core remotely. In case the servers are physically difficult to access, you can install CCM Core Administrator on a workstation.

One of the tasks of CCM Core is to initiate and control the process of producing documents that incorporate data from databases by means of templates. A Service that runs template scripts is RunMdl. For more information on how to submit a request to the RunMdl Service, see Document composition .

Chapter 2

CCM Core Services

CCM Core functionality is implemented in the form of Services. CCM Contract Manager uses a standard set of Services provided with the product. You can implement additional Services with custom scripts. This chapter explains how to define Services.

Add a Service to create a script

1. Start CCM Core Administrator.
2. In the tree view, click the **Services** node.
3. On the **Services** tab, click **Add Service**.
4. Enter a Service name and click **OK**.
   The corresponding Script Editor appears.
5. Make necessary changes and click **File** > **Save**.

   Tip To write a script in multiple stages, you can simply save a script without compiling it and go back to it any time later.

6. Close the Script Editor.
7. Click **Save & Apply** to save and apply the changes.
8. New or edited instruction in a script only becomes effective after compiling the script and restarting the CCM Document processors. To compile the new script, click the new Service.
9. On the **Services** tab, map the script parameters to the request or job parameters known by their sequence number.
   The following is an example of what you may add to the list.

```
Script parameter  Value (job parameter)
Template          $1
ResultDocument    $2
```

10. In the tree view, click the **Services** node once again.
11. On the **Services** tab, click **Compile**.
    You will receive a notification about the compilation result.
12. Click **Save & Apply**.
    The affected CCM Document Processors are restarted. CCM Core might be unavailable for some time.
    To remove a Service, navigate to **Services**, click the Service to delete, and then click **Remove Service** on the toolbar. When a Service is removed, its script is not removed and remains in the Scripts folder of the CCM Core setup for future use.

Chapter 3

Requirements for printer drivers

When installing printer drivers, follow these requirements:

- Use locally installed printer drivers to create spool files. You can use the LPT1: device as the printer port.
- Avoid using spaces in the name of printers.
- Printers should never be put on hold. This blocks the creation of the spool file until the printer is released.
- Wherever available, use printer drivers provided by your printer manufacturer.

Amyuni printer drivers

The CCM installation package automatically installs Amyuni printer drivers. You have to configure CCM Core to run under a user account. You can change this account later using CCM Core Administrator or through the Windows Services control panel applet.

The Amyuni printers are named *PDFConverter ITPDP [ITP installation name] #1*, and so on. These printers may only be used by CCM Core. Other users that have accounts on the computer cannot use these printers.

Reinstall Amyuni printer drivers

The Amyuni printer drivers may accidentally be damaged or removed. CCM Core provides two methods to reinstall missing Amyuni printer drivers:

1. In CCM Core Administrator, remove the CCM Document Processor whose printer driver is damaged and then add the CCM Document Processor again. This automatically recreates the Amyuni printer driver for that CCM Document Processor.
2. Use the command line program CreateAmyuniPrinter.exe that resides in the directory *bin/DocToPDF* of the CCM Core program folder. This program can be used to install and uninstall Amyuni printer drivers for a CCM Document Processor. The program requires the following parameters: either "install" (to install an Amyuni printer driver) or "uninstall" (to uninstall an Amyuni printer driver), the name of the CCM Core installation, and the number of the CCM Document Processor whose Amyuni printer driver is to be installed or uninstalled.

   Once an Amyuni printer driver is reinstalled, restart the corresponding CCM Document Processors.

Chapter 4

Log and setup reports

Logging is set on the CCM Core level. The settings apply to all servers and CCM Document Processors in your setup and to the CCM Document Processor Manager and Watcher. It is possible to log to a file or to the Windows event log.

Watcher and CCM Document Processor Manager log

The Watcher and CCM Document Processor Manager also produce logs. To view the logs:

1. In CCM Core Administrator, on the menu, click **View**.
2. Click **Watcher log** or **CCM Document Processor Manager log**, respectively.

Change the size limit of the log file

You can change the size limit of the log file for each CCM Document Processor.

1. Start CCM Core Administrator.
2. Under **Servers**, click a server.
3. In the right pane, select the **Logging** tab and make a selection where needed.
   - To make the logging size unlimited, select **Keep all log files.**
   - To limit the logging size, select **Rotate log files.**

     This option has the following additional settings.

     To limit the size of the log file per CCM Document Processor, specify the size in KB in **Maximum size of the log files**. The default is 10000 KB (10 files of 1000 KB each).

     To configure the number of log files to keep, specify the number in **Keep the last log files**. During startup, CCM Core rotates existing log files and creates a new log file. Log files are also rotated if the maximum size of a log file is reached. Old files are named *[[logfiles] [number]].log*.

     Tip To keep all log files but still limit the total size, you can select a high number of files, such as 9999, in combination with a small size log file, such as 1000 KB. The total log size per CCM Document Processor is then limited to the number of files multiplied by the maximum log file size.
4. Click **Save & Apply**.

Error messages

If a problem occurs, check the logs for error messages. There are three logs you can analyze -- the CCM Core log, the CCM Document Processor log, and the Watcher log. You may want to view the Watcher log when a CCM Document Processor log shows that the processor keeps shutting down.

Location of the log files

The log files are text files written to the log directory of the CCM installation. Each CCM Code instance creates a structure of log files matching the installed components.

Example If CCM Core version 5.2 is installed on a server named `TestServer`, log files for CCM Core instance 3 with two CCM Document Processors would reside with the following structure at `C:\CCM \Work\5.2\Instance_03\core\Log`.

```
TestServer
  ITPDP [core_03_5.2] #1
  ITPDP [core_03_5.2] #2
  ITPDPWatcher [core_03_5.2]
  ITPServer [core_03_5.2]
```

These folders contain the logs for their respective components. By default, the logs are rotated. The most recent logs reside in the unnumbered file with the .log extension.

You can view logs in a text editor.

Note Ensure that the account that runs the CCM Core Processes has sufficient authorization to write to the folder that the log files are written to. If CCM Core does not have sufficient authorization in the log folder, it is not able to start any of its Services. If no log files are created by CCM Core, check the Windows event log for error messages.

CCM Document Processor Manager example log

The following example log is based on the result of a request to the RunMdl Service as part of the CCM Core instance 3 on a computer called `TestServer`.

```
09:36:18.994[2] CCM Core: Received request [TestJobID] for processing.
```

`[TestJobID]` is the job ID. Use it to track the job.

```
 09:36:19.975[3] ITPDP [core_03_5.2] #1@TestServer: Assigned job TestJobID to server
 ITPDP [TestServer] #1@MACHINE_1.
```

The job is assigned to the first CCM Document Processor on the `TestServer` setup on `MACHINE_1`.

```
09:36:20.156[2] ITPDP [core_03_5.2] #1@TestServer: ||| [TestJobID] Started.
09:36:23.400[3] ITPDP [core_03_5.2] #1@TestServer: <<< [TestJobID] Starting download
 file c:\temp\result.doc.
```

The result document is made, and you can move it to the required location.

```
09:36:24.792[2] ITPDP [core_03_5.2] #1@TestServer: <<< [TestJobID] Transferring 8192
 bytes of data.
```

```
09:36:24.812[2] ITPDP [core_03_5.2] #1@TestServer: >>> [TestJobID] Client confirmed
 transfer.
```

The user clicked OK in the file download confirmation window.

```
09:36:24.822[2] ITPDP [core_03_5.2] #1@TestServer: <<< [TestJobID] Completed.
09:36:24.852[2] ITPDP [core_03_5.2] #1@TestServer: Job TestJobID finished (0:00:04.867)
09:36:24.852[3] ITPDP [core_03_5.2] #1@TestServer: Requesting job (-1,-1).
09:36:24.852[3] CCM Core: Closing socket 1.
09:36:24.852[3] CCM Core: Releasing resources on socket 1.
```

CCM Document Processor example log

You can search a specific CCM Document Processor log. In this case, the log of the first CCM Document Processor in the CCM Core instance is similar to the following log.

```
 09:36:19.985[3] LoadBalancer: Received LB_JOB
09:36:19.985[2] LoadBalancer: Submitted request
TestJobID for processing by service RunMdl.
```

`LoadBalancer` is the DP Manager; it sends a request to process a job. You can use the Job ID to search for its first appearance in the CCM Document Processor log.

```
09:36:20.085[2] Processor: Processing job TestJobID for
service RunMdl.
```

```
The request is accepted and the RunMdl Service starts
processing.
09:36:20.085[3] [TestJobID]: Subst [$1] to "listeng".
09:36:20.085[3] [TestJobID]: Subst [$2] to "c:\temp\result.doc".
09:36:20.085[3] [TestJobID]: Subst [$3] to default.
09:36:20.095[3] [TestJobID]: Subst [$4] to default.
09:36:20.095[3] [TestJobID]: Subst [$5] to default.
```

The script parameters are substituted for the past parameters. In this case, the Master Template and the result document with its path are passed in the request.

```
09:36:20.095[3] [TestJobID]: Running script RunMdl.
09:36:20.095[4] [TestJobID]: Parameter Model = "listeng";
09:36:20.095[4] [TestJobID]: Parameter Result = "c:\temp\result.doc";
09:36:20.095[4] [TestJobID]: Parameter Keys = "";
09:36:20.105[4] [TestJobID]: Parameter Extras = "";
09:36:20.105[4] [TestJobID]: Parameter Flags = "";
09:36:20.105[4] [TestJobID]: Constant ModelDir = "C:\Program Files\ITPWORK - TestServer
\Models";
09:36:20.105[4] [TestJobID]: Constant DPItpTmpDir = "C:\TEMP\MACHINE_1\ITPDP
 [TestServer] #1\ITPTemp";
09:36:20.105[4] [TestJobID]: Constant DPItpConfigDir = "C:\TEMP\MACHINE_1\ITPDP
 [TestServer] #1\Config";
09:36:20.115[4] [TestJobID]: Constant DPItpDataDir = "C:\TEMP\MACHINE_1\ITPDP
 [TestServer] #1\Data";
09:36:20.115[4] [TestJobID]: Constant ServiceConfig = "C:\TEMP\MACHINE_1\ITPDP
 [TestServer] #1\ITPTemp\itp.cfg";
09:36:20.115[4] [TestJobID]: Constant ServiceResult = "C:\TEMP\MACHINE_1\ITPDP
 [TestServer] #1\Data\result.doc";
09:36:20.115[4] [TestJobID]: Constant ServiceModel = "C:\Program Files\ITPWORK -
 TestServer\Models\listeng.itp";
09:36:20.125[4] [TestJobID]: Built-in Delete.
09:36:20.125[4] [TestJobID]:   File ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\ITPTemp
\itp.cfg")
```

```
09:36:20.125[3] [TestJobID]: Deleted file C:\TEMP\MACHINE_1\ITPDP [TestServer]
 #1\ITPTemp\itp.cfg.
09:36:20.125[4] [TestJobID]: Built-in Copy.
09:36:20.135[4] [TestJobID]:   Src ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Config
\itp.cfg")
09:36:20.135[4] [TestJobID]:   Dest ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\ITPTemp
\itp.cfg")
09:36:20.135[3] [TestJobID]: Copied file C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Config
\itp.cfg to C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\ITPTemp\itp.cfg.
```

In the preceding block, the folders are set by the startup script of CCM Core.

```
09:36:20.135[4] [TestJobID]: OnError: Ignoring errors.
09:36:20.146[4] [TestJobID]: Built-in WriteFile.
09:36:20.146[4] [TestJobID]:   File ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\ITPTemp
\itp.cfg")
09:36:20.146[4] [TestJobID]:   Message (";Model-specific settings")
09:36:20.146[4] [TestJobID]: Var Setting = "".
09:36:20.166[4] [TestJobID]: OnError: Using built-in error handler.
09:36:20.166[4] [TestJobID]: OnError: Installing ITPError as error handler.
```

`ITPError` is a standard error handler command provided by CCM Core.

```
09:36:20.166[4] [TestJobID]: Model("listeng")
09:36:20.166[4] [TestJobID]: Built-in ITP.
09:36:20.166[4] [TestJobID]:   Model ("C:\Program Files\ITPWORK - TestServer\Models
\listeng.itp")
09:36:20.176[4] [TestJobID]:   Configuration ("C:\TEMP\MACHINE_1\ITPDP [TestServer]
 #1\ITPTemp\itp.cfg")
09:36:20.176[4] [TestJobID]:   Result ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Data
\result.doc")
09:36:20.176[4] [TestJobID]:   Overwrite (True)
09:36:20.176[4] [TestJobID]:   Keys ("")
09:36:20.176[4] [TestJobID]:   Extras ("")
09:36:20.176[4] [TestJobID]:   Flags ("")
09:36:23.380[4] [TestJobID]: Built-in SendFile.
09:36:23.380[4] [TestJobID]:   Src ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Data
\result.doc")
09:36:23.380[4] [TestJobID]:   Dest ("c:\temp\result.doc")
09:36:24.812[4] [TestJobID]: Built-in Delete.
09:36:24.812[4] [TestJobID]:   File ("C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Data
\result.doc")
09:36:24.812[3] [TestJobID]: Deleted file C:\TEMP\MACHINE_1\ITPDP [TestServer] #1\Data
\result.doc.
09:36:24.822[4] [TestJobID]: Script RunMdl terminated OK.
```

The result document resides in `[drive]:\=temp`.

Read performance statistics from logs

If CCM Core is configured to log progress information (level 3 or higher), job statistics are written to the itpserver.log file. This information can help you analyze the performance and load of the CCM Core configuration.

You can identify the statistics by the prefix `STATS:` on the line. Information is written as KEY[VALUE] pairs.

An example is provided here.

```
16:19:31.890[2] ITPDP [Sample] #1@HOST: STATS:
SUB[16:19:08.734] STR[16:19:08.734] END[16:19:31.890] ID[StH] OL[0] SRV[Test] STS[4]
 IID[3]
```

The status line contains the following information:

- `16:19:31.890`
  Time the statistics line was written to the log.
- `[2]`
  Log level (only shown if CCM Core is configured to log all information).
- `ITPDP [Sample] #1@HOST`
  Identification of the specific Document Processor @ Server that executed the job.
- `SUB[16:19:08.734]`
  Time the job was received and queued by CCM Core.
- `STR[16:19:08.734]`
  Time the job was started by the designated CCM Document Processor.
- `END[16:19:31.890]`
  Time the job finished.
- `ID[StH]`
  Job ID.
- `OL[0]`
  Identifies if the job is interactive or batch. This flag is 1 if the job is submitted through the CCM ComposerUI for HTML5; otherwise, the flag is 0.
- `SRV[Test]`
  Service.
- `STS[4]`
  Completion status. Status 3 indicates successful completion, status 4 indicates that the script reported an error. Other values indicate that the job failed to run.
- `IID[3]`
  Internal identification of the job. This attribute is based on the order in which jobs are submitted to CCM Core. However, this attribute cannot be used as a unique identification because scheduled jobs and maintenance jobs generate a STATS: line every time the job runs on a CCM Document Processor.

You can derive the timing for a job from the following information:

- (STR - SUB) is the time the job spent queued waiting for assignment to an available CCM Document Processor.
- (END - STR) is the time the CCM Document Processor spent executing the job.

Time in the STATS: line is based on a 24-hour clock read from the clock of the computer. The frequency of this clock varies, but could be as low as 60Hz.

STATS: lines processing

The STATS: line is intended for automated processing. Applications that parse this line must adhere to the following guidelines:

- All attributes are written in the format KEY[VALUE].

- The order of the attributes is not fixed and can change between versions. Applications should ignore unknown attributes.
- As time is based on a 24-hour local clock, applications must account for jobs running past midnight or changes due to daylight saving time.

Chapter 5

Notifications

You can configure CCM Core to notify an operator through the Event Log or by email if any CCM Core job exceeds the configured time. To activate this feature, add the following settings to the `[Configuration]` section of the dp.ini file in the CCM Core setup.

Note Ensure that all CCM Core Administrators are closed before adjusting the file dp.ini. After any adjustments are made, the CCM Core CCM Document Processors need to be restarted in order to apply the changes.

```
; Notification settings
Notification=
NotificationEvent=
NotificationTo=
NotificationFrom=
NotificationSMTP=
```

The following is a description of the setting.

`Notification`

Amount of time before a notification is sent.

`NotificationEvent`

Windows server where the event is logged. ITPServerMSG.DLL contains the definition for the events and must be installed on the host.

`NotificationFrom`

Email address used to send emails. This address is only relevant if the email cannot be delivered to the addresses specified in the `NotificationTo` setting.

`NotificationTo`

A list of email addresses for recipients of the notification.

`NotificationSMTP`

SMTP server used to send the notification.

Note There are no defaults for these settings. If one of them is omitted, the appropriate notifications are not sent. Notifications are sent while the job is still being processed. The status of the job is not affected. If the job finishes while the notifications are still being sent, the CCM Document Processor waits until these notifications are completed.

The following parameters are used to write events to the Windows Event log:

- Log: Application
- Type: Error
- Source: ITPEvent
- Event ID: 24

The following is a list of the Event parameters:

- Server
- CCM Document Processor
- Job ID
- Timeout in seconds
- Time that the timeout was triggered

An example is provided here.

```
; Notification settings
Notification=10
NotificationEvent=someserver
NotificationTo=operator@somewhere.com,administrator@somewhere.com
NotificationFrom=sender@somewhere.com
NotificationSMTP=smtp.somewhere.com
```

Chapter 6

Monitor and Watcher

Use the Monitor and Watcher functionalities to monitor CCM Core and CCM Document Processors.

Monitor functionality

CCM Core can be monitored either using the Monitor program or using a browser and enabling the HTTP capabilities of CCM Core. You can remove, hold, and release individual jobs. Also, you can retrieve information about the interfaces in use by CCM Core and retrieve the CCM Document Processor Manager log.

You should opt for HTTP Monitor if the following is true in your situation: if you are sharing the program with other people, and/or if access from outside the LAN is required. Otherwise, use Monitor.

For more information on the Monitor and HTTP Monitor, see Monitor application and HTTP Monitor application, respectively.

Monitor application

You can run Monitor on the same machine as CCM Core or on a remote machine.

1. To configure the Monitor settings, click the CCM Core node and select the **Monitor** tab.
   You must set the port and security settings. For more information on the security settings, see the next chapter.

   Note The port must not be in use by another instance or application. Implementing a port requires the CCM Document Processor Manager to be restarted. When you click **Save & Apply**, the port is saved, and the Manager is restarted.

2. To start the program, navigate to `<deploy root>\CCM\Programs\<version>\ITP Server\DPMonitor` and start the DSMonitor application.
   Use the port that you specified in Step 1.

HTTP Monitor application

To use HTTP Monitor, you need to have an Internet browser installed on the monitoring computer. This browser communicates with CCM Core using HTTP.

1. To configure the HTTP Monitor settings, click the CCM Core node and select the **HTTP Monitor** tab.
   You must set the port and security settings.

Note The port must not be in use. If no port is set, the HTTP capabilities of CCM Core are switched off. Implementing a port requires the CCM Document Processor Manager to be restarted. When you click **Save & Apply**, the port is saved, and the Manager is restarted.

1. To monitor CCM Core, start an Internet browser.
2. Use the following URL with the port that you specified in Step 1.

```
http://<host name>:<port>/queue
```

You can also check logs and service information on this link.

Watcher functionality

CCM Document Processors can be monitored using the Watcher functionality. Every server that runs CCM Document Processors has a Watcher. If a CCM Document Processor shuts down, Watcher can restart it a given number of times within the set interval. If the CCM Document Processor keeps shutting down, the Watcher can stop restarting it to avoid an indefinite loop. To specify the number of seconds after which Watcher restarts Document Processors, and a maximum number of times to attempt a restart, follow these steps.

Note The Windows Service Manager and third-party tools can provide similar functionality. To avoid conflicts between these tools, we recommend that you only use one method to restart CCM Core Services.

1. Start CCM Core Administrator.
2. In the tree view, click the **CCM Core** node.
3. In the right pane, select the **Advanced** tab.
   - In the **Watcher** section, to turn on the Watcher, select **Restart the Document Processors when they have shut down**. Specify the necessary numbers in the corresponding fields.

     Tip Watcher produces its own log. If a problem occurs with CCM Document Processors, this log can provide additional information to the log file of the CCM Document Processors. To view the log, on the menu, click **View** > **CCM Core Manager log**.
4. Click **Save & Apply**.

Chapter 7

Security issues

Proper security is essential to the CCM Core external interface, the internal interface between CCM Document Processors and CCM Document Processor Manager, and the settings for the account used to run CCM Document Processor Manager and CCM Document Processors. This chapter provides detailed information on how to adjust the relevant security settings.

External interface

A CCM Core setup uses an external interface for requests. You can limit the access to this interface.

Grant or restrict network access

You can restrict access to the local host or to the machines on the access list, or grant access to all machines. The access list is a comma-separated list of IP numbers of the machines that must have access.

Note Restricting access to the local host only works if requests are submitted from the same machine that CCM Core runs on.

1. Start CCM Core Administrator.

   Note If jobs are submitted through CCM Contract Manager, the servers hosting CCM Contract Managers must be granted access to this interface.

2. In the tree view, click the **CCM Core** node.
3. In the right pane, select the **DP Manager** tab.
4. Locate the **TCP/IP Interface** section and make a selection under **Security**.
   - To restrict access to the local host, select **Access is restricted to the local host**.
   - To allow access to all machines, select **No access restrictions**.
   - To grant access to particular machines, select **Network access list** and specify the IP numbers in the corresponding field.

     Example 10.0.0.0/24

     This setting grants access to IP numbers in the range from 10.0.0.0 to 10.0.0.255.
5. Click **Save & Apply**.

Adjust internal interface settings

The internal interface is used by CCM Document Processor Manager to communicate with CCM Document Processors in the CCM Core setup. This is a TCP/IP interface, which can use similar security settings to the external interface.

By default, the security is set to public. If all of the CCM Core setups run on one machine, you might consider changing the security setting to local. When the servers running CCM Document Processors are known, you can set an access list. To adjust the settings, follow these steps.

1. Start CCM Core Administrator.
2. In the tree view, click the **Servers** node.
3. On the **Security** tab, make a selection under **Security**.
   - To restrict access to the local host, select **Access is restricted to the local host**.
   - To set an access list for particular machines, select **Network access list** and specify the IP numbers.

     Example 10.0.0.0/24

     This setting grants access to all IP numbers in the range from 10.0.0.0 to 10.0.0.255.
4. To save and apply the changes, click **Save & Apply**.

Requirements for a service account to install and run CCM Core

You should create a service account to install and run CCM Core. It ensures that all CCM Document Processors can access their common configuration file, and that CCM Document Processor Manager can update the configuration file. This account must neither expire or have a password expiration set. As Windows stores credentials for this account as part of the Service configuration, you cannot disable this service account or have password expiration enabled.

You should add the account to the Local Administrators group on the servers where CCM Core and CCM Document Processors are installed. Installed members of the of the Windows Local Administrator group should have all necessary rights and access privileges described later in this section. If CCM Core needs to have access to network resources, we strongly recommend that you use a domain account and grant it access to all resources on remote servers.

If security policies prohibit the creation of domain accounts or security policies have restricted the default rights assigned to the Local Administrators group, the following rights should be assigned to the CCM Core account:

- No password expiration
- Right to access the network if access to network resources is required
- Right to run as an NTService
- Right to start and stop NTServices
- All rights associated with the tasks CCM Document Processors can perform

Any account used to run CCM Core Administrator must also have the following rights:

- Right to create NTServices

- Right to start and stop NTServices
- UAC elevation rights

The CCM Core account must be authorized to activate Microsoft Word through DCOM automation. Default installations of Windows grant this right to the local Administrators group, so you can arrange this authorization by making the CCM Core account a member of this group.

If security policies prohibit the use of a local Administrator account or if the server has been locked down, the following rights must be granted explicitly to the CCM Core account.

For the Microsoft Office Word 97 - 2003 Document DCOM Component (use DCOMCNFG to set this):

- Launch and Activation Permissions: Local Launch permission and Local Activation privileges
- Access Permissions: Local Access privileges
- Configuration Permissions: Full Control

Microsoft Word 2007 and later versions require the CCM Core account to have some additional access rights when accessing documents from a remote server.

- Read rights on the Default User Temporary Internet Files directory.
- Modify rights on the Content.Word directory.

As the location of this Content.Word directory depends on a number of Windows components and their update/service pack levels, the CCM Document Processor logs the most likely locations during startup.

Some directories in this path have the Hidden attribute set. By default, these directories are not visible in the Windows Explorer.

If security policies prohibit the creation of a domain wide user with a non-expiring password, you can also create a local user on all servers where the programs require access and give all those users the same password that does not expire. Use this user to install CCM Core and replace the domain with a dot.

Chapter 8

Job recovery

Any hardware or software problem may cause CCM Core or the server running CCM Core to stop functioning. A shutdown of the server or CCM Core may also cause CCM Core to stop functioning. CCM Core is able to recover jobs that were submitted asynchronously. In this case, the client is notified of the failure and takes appropriate action.

Active jobs are never recovered. If CCM Core is shut down in a controlled way, active jobs are always processed completely before CCM Core terminates. If CCM Core is terminated by some other cause, it is impossible to determine whether or not these jobs were the cause of the termination. Therefore, the operator determines the cause of the termination and recovers the jobs manually.

Jobs submitted synchronously are not recovered. The client software notifies the user who submitted the failure to resubmit the request later or to another server.

Jobs submitted asynchronously are always recovered.

Component Object Model resources for job recovery

You can update Component Object Model (COM) resources to control printing and PDF conversion.

1. Start CCM Core Administrator.
2. In the tree view, click the **CCM Core** node.
3. In the right pane, select the **Job Recovery** tab.
   - To retry failed COM commands, in the **External COM commands** section, select **Retry failed COM commands (such as PrintDocument)**.
   - To set a default timeout for all COM commands in the Core scripting language, specify the number of seconds in the settings **Terminate COM commands after seconds when running Interactive jobs** for jobs submitted through a CCM ComposerUI Server client and **Terminate COM commands after seconds when running Batch jobs** for jobs submitted through a CCM Core client.

     When a COM command exceeds these timeouts, Microsoft Word is terminated, and the system reports a run-time error.

     Note If you set one of these settings to 0 seconds, the system disables the default timeout for the jobs associated with this setting.

     The CCM Document Processor always monitors Microsoft Word for known interactive windows. Such windows are closed automatically whenever possible to allow the job to continue. If this is not possible, the Microsoft Word process is forcibly terminated, and the operation is either retried or failed.

4. Click **Save & Apply**.

Chapter 9

Document composition

CCM Core is primarily designed as a high-grade server application that composes documents and Document Packs. CCM Core requires a configuration file and an environment to compose documents and Document Packs. For more information, see Manage the configuration file and Environments.

Environments

Environments offer the opportunity to run templates in CCM Core with a certain set of connection configuration settings. These settings are used to connect to a data source. Environments can be configured with CCM Core Administrator.

You can pass the exact environment to be used to run a template as a parameter in the `ITPRun` command. For the composition of Document Pack Templates with this command, the environment parameter is ignored. For more information, see ITPRun in the *Kofax Customer Communications Manager Core Scripting Developer's Guide*.

ITPRun command

The `ITPRun` command runs templates.

Use a rep:/ URI to create a document

You can use a rep:/ URI to compose a document.

The URLs supported to compose a document from a template or a Letter Book use the following generic format.

`rep://[`*{host}:port}*`]/{type}/{project}/[{path}/]{object}[?{key}={value}[&{key}={value}`]*]

`rep://[`*{host}:{port}*`]/letter/{project}/{letterbook}/[{path}/]{object}[?{key}={value}[&{key}={value}`]*]

host: TCP/IP host name of the system hosting the CCM Content Publication Database / CCM Repository server

port: TCP/IP port the server listens to

Note When the host and port are omitted, the settings for ContentPublicationName on the environment are used.

type: Select one of the following template types to run or `letterbook`

- `documenttemplate`
- `mastertemplate`
- `staticdocument`
- `quickdocument`
- `documentpacktemplate`

project: Project

path: Optional. Folders separated by a slash symbol

object: Template to be retrieved

key/value: Additional key/value pairs for parameters. Supported keys are the following:

- `user`=*Repository User*
- `status`=[`published|accepted|current|development`]

Examples are provided below.

```
rep:/documenttemplate/DemoProject/Letter
```

The preceding example refers to the specific Document Template `Letter` in the project `DemoProject`. The Document Template is retrieved from the CCM Repository server configured in ContentPublicationName on the default environment. The status retrieved is the status configured in RepositoryObjectStatus on the default environment.

```
rep://localhost:2587/mastertemplate/DemoProject/Letters/Documentgenerator?
status=accepted
```

The preceding example refers to the accepted status of the Master Template `Document generator` in the `Letters` folder (a subfolder of Templates) of the project `DemoProject`. The location of the CCM Repository server `host:port` is supplied as `localhost:2587`, which indicates that CCM Repository installed on the local host configured with port 2587. The status indicates that the accepted status of the Master Template is retrieved, and that the accepted status of any dynamic objects is retrieved.

```
rep://localhost:2587/letter/DemoProject/Correspondence/Welcome Letter
```

The second example refers to the Document Template defined in the `Correspondence` Letter Book under the name `Welcome Letter`. Letter Books allow an additional level of indirection in application integration.

Keys and extra parameters

Keys and extra parameters provide an interface for applications to pass information to a template. For example, you can use the key or extra parameters to identify the customer for whom you want to create a policy or an invoice. Then, the template can use the identifying information to retrieve the records for the customer.

Keys can be used in combination with Master Templates that contain Data Retrieval using a DID. If the Master Template contains DID entries that select records from a database, you can provide keys to identify which records to select. For more information, see the section "Data retrieval" in the *Kofax Customer Communications Manager Template Scripting Language Developer's Guide*.

EXTRAs can be used in combination with Master Template that contain the EXTRA keyword. EXTRA defines what additional information the Master Template requires. For more information, see the section "EXTRA" in the *Kofax Customer Communications Template Scripting Language Developer's Guide*.

Keys and extras are passed to ITPRun as strings of values divided by semicolons. The order in which they are passed must be the same as the order in which they are expected in the Master Template.

When a Master Template requires keys or extras that are not provided in the call, the template becomes interactive and prompts the user to provide the missing keys and extras. If interaction is not allowed, the Master Template reports an error.

Enable DisablePostIncludes

DisablePostIncludes needs to be enabled (set to True) to prevent the ITPRun command from post-including document into the result document (to process the __INC expression in result documents). Post-includes in CCM Core are performed lazy by default (ITPLAZYPOSTINC=Y). If not set or set to False, the ITPRun command uses the ITPPOSTINC setting in the itp.cfg to determine if post-includes have to be processed.

Specify the environment

Templates are always run in the context of an environment. You need to specify the name of the environment in which the template is run. If no environment parameter is given, the default environment is used to run the template.

During the composition of Document Pack Templates with the ITPRun command, the environment parameter is ignored. For more information, see ITPRun command in the *Kofax Customer Communications Manager Core Developer's Guide*.

IBM i connection parameters

This section lists and describes the parameters that you can use when retrieving data from an IBM i (AS/400, iSeries) host.

```
PreCMD
```

Optional.

The Pre command is executed after the library list is set.

```
OnSuccessCMD
```

Optional.

The OnSuccess command is executed if the template has completed successfully.

```
OnFailureCMD
```

Optional.

The OnFailure command is executed if the template has failed.

```
PostCMD
```

Optional.

The `Post` command is executed at the end of the run (after OnSuccess or OnFailure).

Data Backbone XML setting

This section describes the Data Backbone XML setting.

```
DBB_XMLInput
```

Optional, string.

The name of a data XML file used to fill the Data Backbone. The data XML must match the XSD of the Data Backbone.

```
DBB_XMLOutput
```

Optional, string.

The name of a file where the XML with data of the Data Backbone of a template should be stored after the template run has completed.

The file name passed to DBB_XMLInput and DBB_XMLOutput should be a valid path/file specification on the computer running CCM Core. The name can be preceded by `session:`, in which case the file is located in the session directory. The DBB parameters allow you to create templates that do not need a DID as data is loaded directly into the Data Backbone from the XML.

OutputMode setting

This section describes the `OutputMode` setting.

```
OutputMode
```

Optional, string.

The `OutputMode` specifies the type of output document produced by the `ITPRun` command.

The following formats are currently supported:

- `native` produces a document in the same format as the template.
- `utf8` produces text in UTF-8 encoding.
- `utf16` produces text in UTF-16 encoding.
- `aiadocxml` produces an XML file representing the structure of the result document. This structure is based on the Content Wizards used in the document.
- `pack` produces a Document Pack. If the template is a Document Template, it is treated as a Document Pack Template. The produced Document Pack will have a single slot containing the result of that template.

The default is `native`.

A template can query the `OutputMode` parameter by calling the function `runmodel_setting("OutputMode")`.

The following Template scripting language functions are prohibited if the `OutputMode` is set to something other than `native`:

- `add_to_output`
- `footers`
- `headers`
- `inc`
- `pagestyle`
- `paper_types`
- `put_buffer_in_document`
- `put_in_document`
- `put_in_text_file`
- `put_in_text_file2`
- `stylesheet`

In the `OutputMode XML`, the template is not allowed to produce any output. This restriction also prohibits the use of `#...#` and the `TEXTBLOCK` statement (unless it uses the `ASSIGN_TO` keyword).

The `OutputMode` setting is available in CCM Core 4.2.3 or higher.

Master Templates running in a sandbox content

By default, CCM Core 4.4 packages run Master Templates in a sandboxed environment. The sandbox prohibits any statements and functions that could interface with the server, the file system, such as `WRITE`, or the environment, such as `session_parameter`.

You can configure the sandbox in three modes:

- **Pre-flight validation**

  Any Master Template that contains prohibited statements or functions is blocked before it is started.

  To enable this mode, add the setting `ITPSANDBOX=Y` to the ITP.CFG file. This mode is the default for new installations of CCM Core 4.4 and higher.

Note The validation performed before the startup is based on a static analysis during compilation. Code that cannot be executed blocks the Master Template. Code blocked based on parameters is permitted, but will be blocked during execution.

- **Dynamic validation**

  Any Master Template is checked during execution. When the Master Template tries to execute a prohibited statement or function, it is terminated. As this check is performed dynamically during execution, some functions are blocked only when they try to execute a prohibited operation but could be permitted with a different combination of parameters. For example, `insert_image` will be blocked if the image is read from a file, but permitted if the image is read from the Data Backbone.

  To enable this mode, add the setting `ITPSANDBOX=L` to the ITP.CFG file.

- **Disabled**

  All checks are disabled. Master Templates are allowed to use all features of the Template scripting language.

To enable this mode, add the setting `ITPSANDBOX=N` to the ITP.CFG file. This mode is default for CCM Core 4.2.3 and older versions.

The sandbox configuration is applied globally to all jobs for the CCM Core instance. All CCM Document Processors must be restarted to apply a changed setting.

Closed Loop Identifier

The Closed Loop Identifier provides an identification passed to each template and included into the resulting metadata XML and in the metadata of the resulting Document Pack. The Closed Loop Identifier parameter on `ITPRun` provides the initial value. Templates can change or extend the value to provide alternative or more detailed identification for each result document in a Document Pack.

The Template scripting language can access the Closed Loop Identifier through the following built-in Field Sets:

- `_Template.ClosedLoopIdentifier`: *Read-only*
  Contains the value of the Closed Loop Identifier as provided to the `ITPRun` command.
- `_Document.ClosedLoopIdentifier`: *Writable*
  Contains the current value of the Closed Loop Identifier. The Template can change this value. The value at the end of the template is used to identify the result documents in the Document Pack.

For Document Templates executed directly, `_Document.ClosedLoopIdentifier` is initially equal to `_Template.ClosedLoopIdentifier`.

In the context of a Document Pack Template, the precursor template can override the initial value of the `_Document.ClosedLoopIdentifier` field. The resulting value of this field is used as the initial value for `_Document.ClosedLoopIdentifier` in each of the subsequent Document Templates defined in the Document Pack Template.

RunDocumentPackTemplate Service

To facilitate running of a Document Pack Template, the `RunDocumentPackTemplate` Service is provided.

The following table describes the parameters applicable to this Service. These parameters must be passed to `RunDocumentPackTemplate` as a list separated by semicolons.

Parameter	Description
DocumentPackTemplate	Required. The rep:/ URI to be executed.
Result	Optional. The result document, path and name, and extension. The path is optional. If no path is given, the requesting application receives the file and moves it to the specific destination. If no value is given, the result is written in a result.doc.

Parameter	Description
Environment	Optional. Only supported for document templates. Templates are always run in an environment. You need to specify the name of the environment. If no environment parameter is given, the default environment is used to run the template. If an environment is passed that does not exist, an error is given. During the composition of Document Pack Templates with the ITPRun command, this parameter is ignored.
DBB_XMLInput	Optional. The name of a data XML file used to fill the Data Backbone of the template.

RunMdl Service

To facilitate running of a document template, the `RunMdl` Service is provided.

The following table describes the parameters applicable to this Service. These parameters must be passed to `RunMDL` as a list separated by semicolons.

Parameter	Description
Model	Required. The rep:/ URI to be executed.
Result	Optional. The result Document Pack, path and name, and extension. The path is optional. If no path is given, the requesting application receives the file and moves it to the specific destination. If no value is given, the result is written in a result.doc.
Keys	Optional, depending on your template. Keys and extra parameters are used to pass information to a template. You can use this mechanism when you integrate CCM into a business application. For example, you can use the key or extra parameters to identify the customer for whom a policy or an invoice needs to be created. The template can then use the identifying information to retrieve the full customer name and address.
Extras	
Environment	Optional. Templates are always run in an environment. You need to specify the name of the environment. If no environment parameter is given, the default environment is used to run the template. If an environment is passed that does not exist, an error is given.
MetaData	Optional. The name of an XML file to store metadata after the template run is completed.
DBB_XMLInput	Optional. The name of a data XML file used to fill the template Data Backbone.
DBB_XMLOutput	Optional. The name of a file to store the XML file with data of the template Data Backbone after the template run is completed.

Test a template

To test a template, follows the steps below.

1. Start CCM Core Administrator.
2. On the toolbar, click **Tools** and click **Test Tool**.

   The Test Tool opens with localhost as the host and 3003 as the port number. The port number might be different in your case.
3. In the **Service** text box, enter `RunMdl` or `RunDocumentPackTemplate`.

   The RunMdl command does not support Document Pack Templates.
4. In the **Job parameters** text box, enter the following.

   ```
   <<rep:/ URI>>
   ```
5. Click **Submit**.

   The template is processed. The Test Tool informs you that data is written to result.doc or documentpack.zip, depending on the template being tested. To choose a folder where the result is stored, click **Browse** in this window, select a folder, and then click **OK**.

Chapter 10

CCM Core scripts

This chapter provides information about scripts and their usage.

Scripts help control and configure CCM Core as well as create ITP configuration files.

You can use scripts as components in other scripts. For more information on the syntax of script components, see the section "Scripts as commands" in the *Kofax Customer Communications Manager Core Scripting Language Developer's Guide*.

Scripts are text files containing CCM Core scripting language commands. CCM Core comes with a script editor to facilitate creating and editing scripts.

All scripts are located in the scripts folder of the CCM Core instance. The Scripts folder contains scripts that you create. The User Library folder contains exit point scripts that you can customized.

Note Scripts placed in the Scripts folder are not automatically exposed as CCM Core Services. To expose a script as a CCM Core Service, it must be added in CCM Core Administrator. To learn how, see Add a Service to create a script.

CCM Core internal scripts and commands take precedence over the scripts created by the user. If you create a script with the same name as an existing CCM Core Script command, your script will not be available. To avoid this issue, adopt a naming convention for scripts. For example, you can name your scripts starting with your company name.

Chapter 11

Job scheduling

CCM Core provides a number of mechanisms to control job ordering and to automate jobs. This chapter provides a description of those mechanisms.

Scheduled jobs

CCM Document Processor Manager automatically schedules internal jobs that run every day and hour. The internal jobs are executed on the first available CCM Document Processor and provide an exit point that the administrator can modify to perform custom tasks. The tasks are guaranteed to be scheduled sequentially. The exit points are implemented as CCM Core scripts. The default implementation of these exit points resides in the User Library folder.

Exit points

The following two scripts are provided as custom exit points:

- HourlyTask.dss is called every hour.
- DailyTask.dss is called every day at midnight local time.

If all CCM Document Processors are busy processing jobs, it may take some time before a CCM Document Processor is available to run the scheduled jobs. If the delay exceeds an hour, you can run multiple hourly jobs in quick succession.

For administrative purposes, every exit point script receives the numerical parameter ScheduledTime. This parameter contains the local time (in the HHMMSS format) at which the job was scheduled for execution. You can use it as an alternative when the current local time is inappropriate.

The order in which the HourlyTask.dss and DailyTask.dss scripts are called is unspecified, but they are guaranteed not to be run at the same time on different CCM Document Processors. The default HourlyTask.dss script provided with CCM Core calls the `ExpireSessions` command to remove any sessions that have been either idle for more than four hours or that have existed for more than a week.

Downtime and clock changes

No jobs are scheduled during the time that passed since the previous shutdown. For example, if CCM Document Processor Manager is stopped every night at 23:00 and restarted at 01:00, the daily job will never be executed and the DailyTask.dss exit point will never be called. In such a situation, you can use the HourlyTask.dss script to perform any daily tasks at a more appropriate time.

If the clock on the server is set forward while CCM Core is running, internal jobs are scheduled in quick succession for each hour and midnight event in the skipped interval. It can take a couple of minutes before a clock change has been detected and the scheduled job for the skipped interval is run. If the clock on the server is set back, the scheduling is not changed and no internal jobs are scheduled until the clock passes the original time again.

Note Keep the clock on a CCM Core server synchronized with a reliable time source. Avoid unnecessary changes to the clock.

Time zones and daylight saving time

The scheduling of the internal CCM Core jobs is based on Universal Coordinated Time (UTC) and provides consistent scheduling when clock changes occur due to daylight saving time shifts. The parameter ScheduledTime passed to the exit points contains the local time at which the script was called. If a daylight saving time shift sets the clock back a full hour the HourlyTask.dss, the script is called twice with the same local time, once before the shift, and once after the shift has been applied. Tasks which use the parameter ScheduledTime or the current time for administrative purposes must account for these occurrences.

There are a few time zones, such as Newfoundland and Central Australia, which have a 30-minute offset to the UTC time. On systems configured for these time zones, the daily tasks are scheduled at 00:30 local time every day while hourly tasks are scheduled at thirty minutes past the hour.

Interactive scheduling

Jobs are distributed over the available CCM Document Processors in the order they arrive. If a CCM Core instance is used to service both interactive and batch jobs, long running batch jobs can cause a delay in the processing of interactive jobs, degrading the user experience for interactive users.

CCM Core provides two options on the Advanced tab of the CCM Core Administrator:

1. Prioritize interactive requests over Batch requests.
   This option forces CCM Core to prioritize interactive requests over batch requests whenever a CCM Document Processor becomes available. Interactive requests can still be delayed if all CCM Document Processors are busy servicing a batch request.
2. Reserve Document Processors for Interactive requests.
   This option forces CCM Core to always reserve the indicated number of CCM Document processors for interactive jobs. Batch jobs are only run if there are more CCM Document Processors available than this limit.

Jobs scheduling on all CCM Document Processors

By default, a job is run only once on the first available CCM Document Processor. You can use CCM Core clients to schedule jobs which are guaranteed to run once on every available CCM Document Processor.

These jobs are intended for maintenance purposes and have the following properties:

- The job is executed with the highest priority and preempts all other pending jobs.
- If multiple jobs are submitted to run on all CCM Document Processors, these jobs are run in order of submission.
- The job is guaranteed to run on all CCM Document Processors that were connected at the time the job was submitted, both on the local system and on remote systems. The job is not run on CCM Document Processors that connect after the job has been submitted.
- The job is guaranteed to only run on one CCM Document Processor at a time. The order in which the job is assigned to subsequent CCM Document Processors is unspecified.
- The jobs are always run asynchronously.

Currently, you can use the programs saclient.exe and swclient.exe to submit these jobs directly using the `-a` flag. Other APIs can call the sample SubmitMaintenanceJob.dss script as a wrapper to submit these jobs.

Chapter 12

Integration

This chapter provides detailed information on the integration layer of CCM Core.

APIs and Java classes

TCP/IP API for Microsoft Windows

The CCM Core distribution contains the sock_api.dll file with a TCP/IP API for the Windows platform. You can use this library to send requests to a TCP/IP interface. A separate DLL sock_api64.dll is provided for 64-bit applications.

The TCP/IP API provides the following functions for submitting requests: `SSubmitJob`, `SSubmitJobMsg`, `SSubmitJobEx4`, and `SSubmitJobEx5`.

`SSubmitJob` is simplest to use, but supports only basic functionality. The `SSubmitJobEx4` and `SSubmitJobEx5` APIs support the `SendFile`, `ReceiveFile`, and `ConvertCodepage` commands on the Windows platform. For more information on `SSubmitJobEx4` and `SSubmitJobEx5`, see SSubmitJobEx4 and SSubmitJobEx5 functions.

For use in single-threaded Windows GUI applications, the API function `SSubmitJobMsg` is provided. This function provides equivalent functionality to `SSubmitJob`, but performs the job submission in a separate thread, sending back job completion messages to a controlling window.

SSubmitJob function

The `SSubmitJob` function submits a request to CCM Core using TCP/IP sockets as a communication mechanism.

```
BOOL WINAPI SSubmitJob (
TCHAR  *Host,       // pointer to host
TCHAR  *Port,       // pointer to port
TCHAR  *JobID,      // pointer to Job Identifier
BOOL    Sync,       // wait for request to finish
TCHAR **Parameters, // parameter list, starting with
                    // the name of the requested Service
TCHAR  *Result,     // buffer for result text
int     Length,     // size of buffer
void  (*Progress) (TCHAR *)  // callback for progress messages
);
```

The `SSubmitJob` function has the following parameters:

- **Host**

Pointer to a null-terminated string that contains the name of the server running CCM Core. You can specify the name either in (IPv4) Internet Protocol dotted address notation (a.b.c.d) or as a resolvable host name.

- **Port**
  Pointer to a null-terminated string that contains the name of the port to connect to. You can specify the port either in a numerical format or as a Service name resolved through available Service databases.
- **JobID**
  Pointer to a null-terminated string that contains the Job Identifier for the job.
- **Sync**
  Specifies whether the function should wait until the job has been serviced. This parameter can have one of the following values:

Value	Meaning
FALSE	The function returns as soon as the request has been queued for processing.
TRUE	The function waits until the request has been processed by CCM Core.

  When the Sync parameter is FALSE, the `SSubmitJob` function returns when CCM Core has queued the request. When the Sync parameter is TRUE, the `SSubmitJob` function returns when CCM Core has finished servicing the request.
- **Parameters**
  Pointer to a null-terminated list of string pointers. Every string pointer in this list points to a parameter passed with the job to CCM Core. The first parameter should be the name of the requested Service. For all parameters, the meaning of the empty string ("") is that the parameter is not passed at all, and CCM Core substitutes the default value or generates an error if no default value has been specified.
  If Parameters is NULL, no parameters are passed with the job.
- **Result**
  Pointer to a buffer that receives an error message if the submission of the job or the processing of the job failed. The error message placed in the Result buffer on failure will be truncated if the buffer is too small. A size of at least 1024 characters is advised.
- **Length**
  Size of the buffer indicated by Result in characters, not bytes.
- **Progress**
  Pointer to a callback function called by `SSubmitJob` whenever the server sends back a progress message. This progress message is passed as parameter to the `Progress` function.
  `SSubmitJob` ignores NULL progress messages.

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. The buffer indicated by Result contains an error message explaining the cause of the failure.

The string passed to the `Progress` function is a static buffer and is released by the `SSubmitJob` function after the `Progress` function returns.

`SSubmitJob` sends the name of the user logged on at the workstation as the User of the job.

QuickInfo

Header: Declared in s_api.h.

Import Library: Use sock_api.lib.

Unicode: Implemented as both Unicode and ASCII.

SSubmitJobMsg function

The function `SSubmitJobMsg` submits a job to CCM Core using TCP/IP sockets as the communication mechanism. The job submission process is handled in a separate thread. After the job has been processed, a message is sent to the controlling window.

```
BOOL WINAPI SSubmitJobMsg (
TCHAR  *Host, // pointer to host
TCHAR  *Port, // pointer to port
TCHAR  *JobID, // pointer to Job Identifier
TCHAR **Parameters, // parameter list, starting with
                    // the name of the requested Service
TCHAR  *Result, // buffer for result text
int     Length, // size of buffer
HANDLE  hWindow // handle of the controlling window
);
```

The `SSubmitJobMsg` function has the following parameters:

- **Host**
  Pointer to a null-terminated string that contains the name of the server running CCM Core. You can specify the name either in (IPv4) Internet Protocol dotted address notation (a.b.c.d) or as a resolvable host name.
- **Port**
  Pointer to a null-terminated string that contains the name of the port to connect to. You can specify the port either in a numerical format or as a Service name resolved through any available service databases.
- **JobID**
  Pointer to a null-terminated string that contains the Job Identifier for the job.
- **Parameters**
  Pointer to a null-terminated list of string pointers. Every string pointer in this list points to a parameter passed with the request to CCM Core. The first parameter should be the name of the requested Service. For all parameters, the meaning of the empty string ("") is that the parameter is not passed at all, and CCM Core will substitute the default value or generate an error if no default value has been specified.
  If Parameters is NULL, no parameters are passed with the job.
- **Result**
  Pointer to a buffer that receives an error message if the submission of the job or the processing of the job failed. The error message placed in the Result buffer on failure is truncated if the buffer is too small. A size of at least 1024 characters is advised.
- **Length**
  Size of the buffer indicated by Result in characters, not bytes.
- **hWindow**
  Handle of the controlling window. `SSubmitJobMsg` sends this window a message after the job has been processed.

The SSubmitJobMsg function creates a thread to wait for the synchronous processing of the request. The SSubmitJobMsg returns TRUE if this thread has been created successfully, and FALSE if the creation of the thread failed. In case SSubmitJobMsg returns FALSE, the buffer indicated by Result contains an error message explaining why the thread could not be started.

The SSubmitJobMsg function sends an ITPDS_RESULT message to the hWindow window after the request has been processed. The wParam parameter of this message is TRUE if the request succeeded; otherwise, it returns FALSE. If the request failed, the lParam parameter points to a NULL-terminated string indicating the cause of the failure.

Use the RegisterWindowMessage function from the Microsoft Windows API to retrieve the identifier for the ITPDS_RESULT message.

The parameters passed to the function SSubmitJobMsg should not be freed until the reply message has been received.

The SSubmitJobMsg function sends the name of the user logged on at the workstation as the User of the job.

QuickInfo

Header: Declared in s_api.h.

Import Library: Use sock_api.lib.

Unicode: Implemented as both Unicode and ASCII.

SSubmitJobEx4 and SSubmitJobEx5 functions

The SSubmitJobEx4 and SSubmitJobEx5 functions submit a request to CCM Core using TCP/IP sockets as a communication mechanism. Compared to the SSubmitJob function, these functions add support for the ExchangeData and ValidateFileName callback APIs and for specifying CCM Core session IDs. The two functions provide the same functionality, except that the callback functions for SSubmitJobEx4 use the cdecl calling convention while the callback functions for SSubmitJobEx5 use the stdcall calling convention.

These functions replace the older SSubmitJobEx, SSubmitJobEx2, and SSubmitJobEx3 functions which are still available as a convenience, but deprecated. As these older functions should not be used in new installations, their functionality is not described in this guide.

```
BOOL WINAPI SSubmitJobEx4 (
TCHAR      *Host,      // pointer to host
TCHAR      *Port,      // pointer to port
TCHAR      *JobID,      // pointer to Job Identifier
BOOL        Sync,      // wait for request to finish
TCHAR     **Parameters,  // parameter list, starting with
                         // the name of the requested Service
TCHAR      *Result,      // buffer for result text
int         Length,      // size of buffer
SUBMITJOB4 *Extended     // extended parameters
);
BOOL WINAPI SSubmitJobEx5 (
TCHAR      *Host,      // pointer to host
TCHAR      *Port,      // pointer to port
TCHAR      *JobID,      // pointer to Job Identifier
BOOL        Sync,      // wait for request to finish
```

```
TCHAR     **Parameters,  // parameter list, starting with
                         // the name of the requested Service
TCHAR      *Result,      // buffer for result text
int         Length,      // size of buffer
SUBMITJOB5 *Extended     // extended parameters
);
```

The functions have the following parameters:

- **Host**
  Pointer to a null-terminated string that contains the name of the server running CCM Core. The name can be specified either in (IPv4) Internet Protocol dotted address notation (a.b.c.d) or as a resolvable host name.
- **Port**
  Pointer to a null-terminated string that contains the name of the port to connect to. The port can be specified either in a numerical format or as a Service name resolved through any available Service databases.
- **JobID**
  Pointer to a null-terminated string that contains the Job Identifier for the job.
- **Sync**
  Specifies whether or not the function should wait until the job has been serviced. This parameter can have one of the following values:

Value	Meaning
FALSE	The function returns as soon as the request has been queued for processing.
TRUE	The function waits until the request has been processed by CCM Core.

  When the Sync parameter is FALSE, the `SSubmitJobEx4` function returns when CCM Core has queued the request. When the Sync parameter is TRUE, the `SSubmitJobEx4` function returns when CCM Core has finished servicing the request.
- **Parameters**
  Pointer to a null-terminated list of string pointers. Every string pointer in this list points to a parameter passed with the job to CCM Core. The first parameter should be the name of the requested Service. For all parameters, the meaning of the empty string ("") is that the parameter is not passed at all, and CCM Core substitutes the default value or generates an error if no default value has been specified.
  If Parameters is NULL, no parameters are passed with the job.
- **Result**
  Pointer to a buffer that receives an error message if the submission of the job or the processing of the job failed. The error message placed in the Result buffer on failure will be truncated if the buffer is too small. A size of at least 1024 characters is advised.
- **Length**
  Size of the buffer indicated by Result in characters, not bytes.
- **Extended**
  Pointer to a `SUBMITJOB4` (for `SSubmitJobEx4`) or `SUBMITJOB5` (for `SSubmitJobEx5`) data structure that holds additional parameters to the function.

The string passed to the `Progress` function is a static buffer and will be released by the `SSubmitJobEx4`/ `SSubmitJobEx5` function after the `Progress` function returns.

The `SSubmitJobEx4` and `SSubmitJobEx5` functions send the name of the user logged on at the workstation as the User of the job.

QuickInfo

Header: Declared in s_api.h.

Import Library: Use sock_api.lib.

Unicode: Implemented as both Unicode and ASCII.

SUBMITJOB4 and SUBMITJOB5

The `SUBMITJOB4` and `SUBMITJOB5` data structures contain additional parameters for the extended CCM Core APIs. The `SUBMITJOB4` data structure is used with `SSubmitJobEx4`; the `SUBMITJOB5` data structure is used with `SSubmitJobEx5`. The data structures are identical except for the calling conventions used for the callback functions.

```
typedef struct {
DWORD     Version; // Version of the data structure
DWORD     Extended; // Extended attributes
void*     Context;  // Application data for callbacks

// Callback for progress messages
void    (*Progress) (void* Context, TCHAR *);

// Callback for warnings
BOOL    (*Warning) (void* Context, TCHAR *);

// Callback for filename validation
TCHAR  *(*ValidateFileName) (void* Context, TCHAR *, BOOL);

 // Callback for data exchange
TCHAR  *(*ExchangeData) (void* Context, TCHAR *, TCHAR *);


TCHAR *Reserved1;
TCHAR *SessionID;   // ITP/Server session ID
TCHAR *Reserved2;
} SUBMITJOB4;

typedef struct {
DWORD     Version; // Version of the data structure
DWORD     Extended; // Extended attributes
void* Context;      // Application data for callbacks

// Callback for progress messages
void    (__stdcall *Progress) (void* Context, TCHAR *);

// Callback for warnings
BOOL    (__stdcall *Warning) (void* Context, TCHAR *);

// Callback for filename validation
TCHAR  *(__stdcall *ValidateFileName) (void* Context, TCHAR *, BOOL);

 // Callback for data exchange
TCHAR  *(__stdcall *ExchangeData) (void* Context, TCHAR *, TCHAR *);
```

```
TCHAR *Reserved1;
TCHAR *SessionID;   // ITP/Server session ID
TCHAR *Reserved2;
} SUBMITJOB5;
```

Version field indicates which other members of the data structure are valid. Currently, the only supported value is `SUBMITJOB_4` for the `SUBMITJOB4` data structure, and `SUBMITJOB_5` for the `SUBMITJOB5` data structure.

Extended attributes for the APIs that can be a combination of any of the values described in the following table.

Value	Meaning
0	No extended attributes apply.
OPTION_QUERY_CODEPAGE	The API is allowed to send its codepage to the CCM Core server.
OPTION_RECEIVE_FILE	The API is allowed to receive files from the CCM Core server.
OPTION_SEND_FILE	The API is allowed to send files to the CCM Core server.
OPTION_CONFIRM_DISCONNECT	The caller requires that CCM Core waits for confirmation after reporting a successful job. This option guarantees that the confirmation has been received by the client before CCM Core starts another job. Failure during this disconnect exchange reports the job as failed, but does not trigger an OnError statement.
OPTION_EXCHANGE_DATA	The API is allowed to exchange data with the CCM Core server.
OPTION_VERIFY_CONNECTION	The API is Verify Connection enabled.
OPTION_TCP_NODELAY	Disables the packet size optimizations done by the Nagle algorithm. This option improves responsiveness significantly at the cost of increased network traffic. This option has an effect only if the network connection is between different virtual computers. Connections over the local host/loopback interface are never buffered. Use with caution.
OPTION_ALL_DPS	The job is submitted to run once on all available CCM Document Processors.

Context may hold a pointer to arbitrary data defined by the application. It is not used by the API function. It is passed as the Context parameter to the callback functions `Progress`, `Warning`, `ValidateFileName`, and `ExchangeData`. You can use it to pass extra, application defined information to these callback functions.

Progress is a pointer to a callback function called by the API whenever the server sends back a progress message. This progress message is passed as parameter to the function `Progress`. When called, the function is passed the **Context** member field of the `SUBMITJOB4/5` structure in the Context parameter. This function is NULL if the application does not want to receive progress messages.

Warning is a pointer to a callback function called by the API whenever it encounters a recoverable error. This error is passed as a parameter to the function `Warning`. This function returns TRUE if it wants to continue, and FALSE if the call should be terminated. When called, the function is passed the Context member field of the `SUBMITJOB4/5` structure in the Context parameter.

This function can be NULL if the application wants to ignore recoverable errors. The errors are still passed to the CCM Core server but the script can choose to ignore them using the command `OnError`.

ValidateFileName is a pointer to a callback function called by the API whenever the server wants to receive or send a file from the client. For more information on `ValidateFileName`, see ValidateFileName.

When called, the function is passed the Context member field of the `SUBMITJOB4/5` structure in the Context parameter. This function is NULL if the application does not need to validate file names. In this situation, all suggested file names are accepted.

ExchangeData is a pointer to a callback function called by the API whenever the server wants to exchange data with the client. For more information, see ExchangeData.

When called, the function is passed the Context member field of the `SUBMITJOB4/5` structure in the **Context** parameter. This function is NULL if the application does need to exchange data. In this situation, all attempts to exchange data succeed and return an empty response.

QuickInfo

Header, declared in s_api.h and p_api.h.

Import Library, use sock_api.lib.

Unicode, implemented as both Unicode and ASCII.

ValidateFileName

The `ValidateFileName` callback function should be provided by an application to validate file transfers between the client and the CCM Core server. If provided, the `ValidateFileName` function is called before every transfer to allow the application to convert the passed file name and to determine whether the transfer should be allowed.

```
TCHAR * (*ValidateFileName) (
 void      *Context, // Context field of structure
TCHAR     *FileName, // suggested filename
BOOL       TransferMode // transfer modus
);
```

The function has the following parameters:

- **Context**. This is an application defined data. This parameter gets the value of the Context field of the `SUBMITJOB4/ SUBMITJOB5` structure.
- **FileName**. This is the file name for the transferred file as suggested or requested by the CCM Core server.
- **TransferMode** flag. Indicates the type of transfer requested by the CCM Core server. TransferMode is TRUE if the client should receive a file from the CCM Core server, and FALSE if the CCM Core server needs to request a file.

The `ValidateFileName` should return a pointer to the file name the API should transfer.

The `ValidateFileName` can return NULL if it needs to refuse the transfer. CCM Core terminates the transfer with the message that the client refused the transfer.

If `ValidateFileName` needs to accept the transfer without modifications in the file name, it returns the **FileName** pointer. If it needs to modify the file name, it provides memory that remains allocated until either the next call to `ValidateFileName` or until the API returns.

The calling convention of the `ValidateFileName` function is `cdecl` when it is passed through the `SUBMITJOB4` data structure to `SSubmitJobEx4`, and `std` call when it is passed through the `SUBMITJOB5` data structure to `SSubmitJobEx5`.

QuickInfo

Header, N/A.

Import Library, N/A.

Unicode, N/A.

ExchangeData

The `ExchangeDatacallback` function is provided by a business application to exchange data with the CCM Core server. If provided, the `ExchangeData` function is called for every `exchange_data` function call. The client can then process the data and send a response back.

```
TCHAR * (*ExchangeData) (
void      *Context,   // Context field of structure
TCHAR     *Key,   // The Key parameter
TCHAR     *Value   // The Value parameter
);
```

The function has the following parameters:

Context. This is an application defined data. This parameter gets the value of the Context field of the `SUBMITJOB4/ SUBMITJOB5` structure.

Key. The **k** parameter of the `exchange_data` function.

Value. The **v** parameter of the `exchange_data` function.

The `ExchangeDatafunction` function returns a pointer to the response it needs to send back to the CCM Core server. The memory this pointer refers to must remain allocated until either the next call to `ExchangeData` or until the API returns.

The `ExchangeDatafunction` function returns NULL to indicate that the data was processed successfully without sending a specific response. In this situation, the function `exchange_data` returns an empty text.

The calling convention of the `ExchangeData` function is `cdecl` when it is passed through the `SUBMITJOB4` data structure to `SSubmitJobEx4`, and `stdcall` when it is passed through the `SUBMITJOB5` data structure to `SSubmitJobEx5`.

QuickInfo

Header, N/A.

Import Library, N/A.

Unicode, N/A.

Error codes

The error messages returned by the APIs in the result buffer commonly contain a decimal error code. Windows returns these error codes if an operation fails.

For a full list of error codes, see the corresponding Windows references.

Some common error codes are described in the following table.

Code	Description	Likely cause
10054	Connection reset	Your system is not authorized to connect to the Service.
10061	Connection refused	The host or port you tried to access is not correct or CCM Core is not started.

Saclient.exe and swclient.exe

The command line programs saclient.exe and swclient.exe are provided to facilitate submitting a job to CCM Core using TCP/IP.

Saclient submits a job to CCM Core using TCP/IP sockets from an ASCII environment and swclient does the same from a Unicode environment.

Starting saclient on a command line without parameters returns all options and the syntax.

```
Usage: saclient.exe [-s|-r|-t|-c|-e|-k|-d|-i
sessionID|-a] host port jobid service [parameters]
```

The following table describes options that you can use.

-s	Sends a synchronous request
-r	Allows the client to receive files from the server [implies -s]
-t	Allows the client to send files to the server [implies -s]
-c	Confirms disconnect
-e	Enables the `exchange_data` function calls [implies -s]
-k	Enables Verify Connection (This corresponds to `OPTION_VERIFY_CONNECTION` extended attributes of the `SUBMITJOB4/ SUBMITJOB5` data structure).
-d	Disables the packet size optimizations done by the Nagle algorithm. This option improves responsiveness significantly at the cost of increased network traffic
-i	Runs the job in the CCM Core session specified by the given session ID
-a	Runs the job once on every available CCM Document Processor [disables -s]

For a complete description of these settings, see SUBMITJOB4 and SUBMITJOB5.

In addition to saclient.exe and swclient.exe, a third program called winclient.exe is provided. This is a variation of saclient.exe, created to solve specific printer switching problems with saclient.exe in a Windows environment. You should use saclient.exe instead.

.NET library

The CCM Core .NET library allows .NET applications to submit jobs to CCM Core. The API is provided in the form of the .NET assembly ITPServerDotNetApi.dll. To learn how to build and distribute applications using the CCM Core .NET library, see .NET library installation and distribution.

The main class of the .NET library is called `Aia.ITP.Server.Job`. This class provides all functionality needed for job submission. See the following sections for details on the methods, properties and events of the class `Aia.ITP.Server.Job`. The .NET library also includes the class `Aia.ITP.OnLine.Model` that implements the CCM ComposerUI Server .NET API. For more information on the CCM ComposerUI Server .NET API, see the legacy *Kofax Customer Communications Manager ComposerUI for ASP.NET Developer's Guide*.

.NET library installation and distribution

The CCM Core .NET library is designed for the .NET Framework 3.5. The assembly for the CCM Core .NET library is not a strongly named assembly and it cannot be deployed in the Global Assembly Cache. It must always be deployed as a private assembly together with the application that uses it. The library is also not exposed through COM, which means that it can only be called from .NET applications.

As the implementation of the CCM Core .NET library is based on the TCP/IP API, the .NET assembly should be deployed together with the files that comprise the CCM Core TCP/IP API. If the application may be run on 64-bit Windows platforms, it is important that both the 32-bit and the 64-bit versions of the CCM Core TCP/IP API are deployed.

To deploy the CCM Core .NET library, copy the following files to the directory of the application that uses the library:

- ITPServerDotNetApi.dll
- sock_api.dll
- sock_api64.dll

Aia.ITP.Server.Job class

The class `Aia.ITP.Server.Job` represents a job submission to CCM Core. It contains all functionality required to:

- Submit jobs to CCM Core
- Upload and download files to CCM Core
- Exchange data values with CCM Core
- Receive progress messages from CCM Core

An example of the usage is provided here.

```
Aia.ITP.Server.Job job;
job = new Aia.ITP.Server.Job("localhost",
                             "3001",
                             "MyJob_" + Guid.NewGuid().ToString(),
```

```
                                "MyService",
                                "FirstParameter",
                                "SecondParameter");
try
{
    job.Submit();
}
catch (Exception e)
{
 MessageBox.Show ("Error", "An error occurred in an ITP/Server job: " + e.Message);
}
```

This example creates an `Aia.ITP.Server.Job` object in order to call the CCM Core Service `"MyService"` with the parameters `FirstParameter` and `SecondParameter`. The job submission is destined for the CCM Core running on computer `"localhost"` (the local computer) on port 3001, with a randomly generated unique job ID based on a GUID (Globally Unique ID).

The class `Aia.ITP.Server.Job` exposes the following methods:

- `Job` (String host, String port, String jobID, String Service, String parameters) (constructor)
- `Submit ()`
- `SubmitAsync ()`

For more information on these methods, see Job method, Submit () method, and SubmitAsync () method, respectively.

Job method

The `Job` method is the constructor of the class `Aia.ITP.Server.Job`. Its parameters `host`, `port`, `jobID` and `service` represent exactly the mandatory parameters of a CCM Core job submission. The final parameter `parameters` is a variable-length list of parameters passed to the CCM Core Service. For parameters, the empty string (`""`) is interpreted as if the parameter was not passed, which causes the default value to be substituted (if any). After the object is constructed, the parameters of the constructor are stored in the properties `Host`, `Port`, `JobID`, `Service`, and `Parameters`, respectively.

This constructor does not submit the job to CCM Core, it only constructs an `Aia.ITP.Server.Job` object with some properties already set to the values passed to the constructor. After constructing the `Aia.ITP.Server.Job` object, there is an opportunity to set optional properties and register event handlers. The job can then be submitted to CCM Core using the methods `Submit()` or `SubmitAsync()`.

Submit () method

The method `Submit` of the class `Aia.ITP.Server.Job` submits the job to CCM Core running on the host and port specified by the properties `Host` and `Port`. The method does not return until the job has been completed. If the method returns normally, this means that the job has completed successfully. If an error occurs during either the submission or the processing of the job, an exception is thrown.

While the job is running, the invoked CCM Core Service may request to exchange a data value, and it may send files for download, request the upload of files or send progress messages. When such requests arrive, the object `Aia.ITP.Server.Job` fires the events `ExchangeData`, `FileDownload`, `FileUpload`, and `ProgressMessage`, respectively.

Note If the CCM Core Service requests the download or upload of a file, and the corresponding event is not handled, the CCM Core job fails immediately.

SubmitAsync () method

The method `SubmitAsync` of the class `Aia.ITP.Server.Job` submits the job to the CCM Core running on the host and port specified by the properties `Host` and `Port`. The method returns immediately after the job has been submitted, without waiting until the job has been processed. If the method returns normally, this means that the job was submitted successfully. If an error occurs during the submission of the job, an exception is thrown. If an error occurs during the processing of the job, this is not reported.

In contrast to the method `Submit`, the method `SubmitAsync` does not allow for the exchange of data values, files, or progress messages with CCM Core. If these mechanisms are needed, the method `Submit` must be used instead.

The class `Aia.ITP.Server.Job` exposes the following properties:

- Host
- Port
- JobID
- Service
- Parameters
- SessionID
- ConfirmCompletion
- KeepAlive
- NoDelay
- PrivateTransfer
- UserID
- ApplicationID

The property `Host` is a string that specifies the host name of the machine running CCM Core. You can specify the name either in (IPv4) Internet Protocol dotted address notation (a.b.c.d), or as a resolvable host name.

The property `Port` is a string that specifies the port number on which CCM Core is running. You can specify the port either in numerical format or as a Service name resolved through any available Service databases.

The property `JobID` is a string used to identify the job on CCM Core. Also, it appears in the CCM Core log files in all log lines that describe the job run.

The property `Service` is a string that specifies the name of CCM Core Service that the job should invoke.

The property `Parameters` is a read-only property of type `List<String>`. It contains the list of parameters passed to the CCM Core Service. To add or remove parameters, manipulate the object `List<String>` returned by this property, using the standard methods provided by the .NET Framework. Parameters cannot be null. The empty string is interpreted as if the parameter was not passed, which causes the default value to be substituted (if any).

The property `SessionID` is a string that specifies the CCM Core session ID that will be associated with the submitted job. CCM Core session IDs serve multiple purposes:

- Mutual exclusion. The CCM Core guarantees that multiple requests for the same session ID are not handled in parallel by multiple Document Processors. Instead, multiple simultaneous requests with the same session ID are queued and processed in a series.
- Persistent storage across jobs. CCM Core Services may use the session ID to store information across several CCM Core jobs, so that each job can use data stored by earlier jobs.

You can specify session IDs for mutual exclusion by the calling the client. Session IDs for persistent storage are always generated by CCM Core and must be sent to the calling client using the mechanism `exchange_data`. For convenience, the class `Aia.ITP.Server.Job` allows the CCM Core Service to pass a session ID using the mechanism `exchange_data` and using the key "SessionID" (case-insensitive). The data value of such a data exchange request is automatically stored in the property `SessionID` of the object `Aia.ITP.Server.Job`. This functionality works regardless of whether the `ExchangeData` event is being handled.

For more information about the use of session IDs in CCM Core, see the section "CCM Core sessions" of the *Kofax Customer Communications Manager Core Scripting Language Developer's Guide*.

The boolean property `ConfirmCompletion` specifies that CCM Core should not regard the job as completed until the client, such as the object `Aia.ITP.Server.Job`, has confirmed to CCM Core that it has received the message stating that the job has completed. This setting has no effect when the job is submitted using `SubmitAsync`. This property is set to false by default.

The boolean property `KeepAlive` specifies that the `Aia.ITP.Server.Job` object should send confirmation messages to CCM Core to indicate that it is still listening, upon request from CCM Core. This setting corresponds to the setting "Verify connection" that you can find on the Advanced tab of CCM Core Administrator. This setting has no effect when the job is submitted using `SubmitAsync`. The property is set to false by default.

The boolean property `NoDelay` specifies that the underlying TCP/IP connection to CCM Core should not use Nagle's algorithm to put as much data as possible into a single network packet. Enabling this option significantly reduces network latency, but also leads to a significant increase in network traffic. The property is set to false by default.

The boolean property `PrivateTransfer` specifies that the CCM Core load balancer should be bypassed for file transfers. Enabling this option may increase performance for large file transfers. The property is set to false by default.

The properties `UserID` and `ApplicationID` are strings used for the submission of CCM ComposerUI Server jobs. `UserID` specifies the user name of the user connected to CCM ComposerUI Server while the ApplicationID identifies the CCM ComposerUI Server application that submits the job. It is not necessary to specify these properties.

ExchangeData event

The event `ExchangeData` fires during the processing of a job when CCM Core needs to exchange a data value using the CCM Core script function `exchange_data`. Handlers of the event `ExchangeData` should be of the type `Aia.ITP.Server.Job.ExchangeDataHandler`.

```
delegate string ExchangeDataHandler (string key,string value)
```

An ExchangeData handler receives two parameters: key and value. The parameter key identifies the value being passed, and the parameter value specifies the actual value being passed. The value returned from the event handler ExchangeData is returned to the CCM Core Service.

Regardless of whether the event ExchangeData is being handled, any value passed from CCM Core with key "SessionID" (case-insensitive) is copied to the property SessionID.

FileDownload

The event FileDownload fires during the processing of a job when CCM Core needs to send a file to the client application. Handlers of the FileDownload event should be of the type Aia.ITP.Server.Job.FileDownloadHandler.

```
delegate string FileDownloadHandler(string  filename);
```

A FileDownload handler receives a parameter file name that specifies a suggestion for the name of the file being sent by CCM Core. The return value of the handler should be the name of the file as which the downloaded file will be stored. It is also possible to refuse the download by returning null or throwing an exception. This causes the CCM Core job to fail immediately. If no event handler is installed for the event FileDownload and CCM Core tries to send a file for download, the job fails as well.

FileUpload

The event FileUpload fires during the processing of a job when CCM Core wants to request a file from the client application. Handlers of the event FileUpload should be of the type Aia.ITP.Server.Job.FileUploadHandler.

```
delegate string FileUploadHandler(string filename);
```

A FileUpload handler receives a parameter filename that indicates the file that is being requested by CCM Core. The return value of the handler should be the name of the actual file that will be uploaded to CCM Core. It is also possible to refuse the upload by returning null or throwing an exception. This causes the CCM Core job to fail immediately. If no event handler is installed for the event FileUpload and CCM Core tries to request an upload, the job fails as well.

ProgressMessage

The event ProgressMessage fires during the processing of a job when the CCM Core wants to send a progress message to the client application. Handlers of the event ProgressMessage should be of the type Aia.ITP.Server.Job.ProgressMessageHandler.

```
delegate void ProgressMessageHandler(string text);
```

A ProgessMessage handler receives the message from CCM Core in the parameter text.

TCP/IP for the IBM i platform

The CCM Core IBM i connection includes the IBM i command SBMITPJOB in the library ITPCOM31. You can use this command to submit a request to a remote CCM Core Service.

```
>>-SBMITPJOB-------------------------------------------------------->

>--SERVICE(--Name of the Service called--)------------------------->

>-+------------------------------------+------------------------->
  |        .-------------------------.  |
```

```
  |        V                    (3)   |  |
  '--PARS(----script-parameter------+-)'

>-HOST(--+-hostname--+--)--PORT(--+-service-name-+--)------------>
         '-ip-number-'             '-port-number--'

>--JOBID(--job-identifier--)-------+--------------------+-------->
                                   |        .-*NO--.    |    (4)
                                   '-SYNC(--+-*YES-+--)-'

>--+----------------------------+--------------------------------->
   | (1)            .-*NO--.    |
   '------CONFIRM(--+-*YES-+--)-'

>--+----------------------------+--+-------------------------+---->
   | (1)            .-*NO--.    |  | (1)          .-*NO--.    |
   '------RECEIVE(--+-*YES-+--)-'  '------SEND(--+-*YES-+--)-'

>--+--------------------+--+------------------+----------------->
   |        .-*YES-.    |  |       .-*NO--.   |
   '-MSG-(--+-*NO--+--)-'  '-LOG(--+-*YES-+--)-'

>--+-------------------------------------------------+------------->
   | (1)           .-*LIBL/-------.                  |
   '------XCHFIL(--+--------------+--file-name--)-'
                   '-library-name/'

>--+-------------------------------------------------+----------->
   | (1)           .-*LIBL/-------.                  |
   '------XCHPGM(--+--------------+--program-name--)-'
                   '-library-name/'

>--+----------------------+--+-------------------------+------->
   |           .-*DNS-.   |  |            .-*MAP---.   |
   '-RESOLVE(--+-*IP--+--)-'  '-CODEPAGE(--+-*HEX---+--)-'
                                          +-*JOB---+
                                          '-number-'
```

When using this functionality, consider the following:

- This functionality is only supported by the client if `SYNC(*YES)` is specified.
- This functionality is only supported by the client if either `RECEIVE(*YES)` or `SEND(*YES)` is specified.
- You can specify up to 32 parameters of up to 4095 characters can be specified.
- Parameters past this point are not prompted by default.

The `SBMITPJOB` has the following parameters:

- `SERVICE`
  Required. Specifies the name of the Service that the job is submitted to.
- `PARS`
  Optional. Specifies up to 32 parameters for the job. Every parameter can contain up to 4095 characters.
- `HOST`
  Required. Specifies the host name of the remote host running the CCM Core.
- `PORT`
  Required. Specifies the CCM Core port number or port name.
- `JOBID`

Required. The Job ID used to identify the job in the CCM Core queue. This parameter cannot be blank.

- SYNC
  Optional. Specifies whether the command waits until CCM Core has processed the job. *YES: Wait until the job is completed. *NO: Terminate after the server has acknowledged receipt of the job.
- CONFIRM
  Optional. Specifies whether the client must confirm termination of the job back to CCM Core before the job is considered to be completed successful. *YES: Require confirmation. *NO: Do not require confirmation.
  This option is only used if SYNC(*YES) is specified.
- RECEIVE
  Optional. Specifies whether the client is allowed to receive files from CCM Core. *YES: The client is allowed to receive files. *NO: The client is not allowed to receive files. Any attempts to use the SendFile command in a CCM Core script will fail.
  This option is only used if SYNC(*YES) is specified.
- SEND
  Optional. Specifies whether the client is allowed to send files to CCM Core. *YES: The client is allowed to send files. *NO: The client is not allowed to send files. Any attempts to use the ReceiveFile command in a CCM Core script will fail.
  This option is only used if SYNC(*YES) is specified.
- MSG
  Optional. Specifies if the command sends progress messages to its caller. *YES: Send messages to the caller. *NO: No messages are sent.
  If the MSG(*YES) parameter is specified, the SBMITPJOB command can send the following messages to the calling program:
  1. A single completion message if the command terminated successfully.
  2. A single escape message if the command failed.
  3. Informational messages if CCM Core sent progress messages to the AS/400 client.
- LOG
  Optional. Specifies if the command logs messages in the REQHST file. *YES: All messages are logged. *NO: No messages are logged.
- XCHGFIL
  Optional. Specifies to which file the client writes the data sent with the exchange_data function. If this parameter is not specified data is not written to a file.
  SBMITPJOB sends an empty response back to the CCM Core unless the XCHGPGM parameter is also specified. In that case the response of the XCHGPGM exit program is sent back.
- XCHGPGM
  Optional. Specifies which exit program the client should call if data is sent with the exchange_data function. If this parameter is not specified, no exit programs are called.
  SBMITPJOB sends an empty response back to CCM Core unless the XCHGPGM parameter is specified. In that case the response of the XCHGPGM exit program is sent back.
- RESOLVE

Optional. Specifies in which order the host name specified with the `HOST(...)` parameter is resolved.

`*DNS`: First attempt to resolve the host name through any configured DNS server, and then attempt to translate the address as a numerical IP address.

`*IP`: First attempt to translate the address as a numerical IP address and then attempt to resolve the host name through any configured DNS server.

- `CODEPAGE`
  Optional. Specifies how the client should send its code page back to the CCM Core if the script requires a client code page.

`*MAP`: Attempt to map all characters to Unicode and send this mapping to the CCM Core.

`*HEX`: Do not perform a code page translation.

`*JOB`: Send the code page of the job. If the job uses `CCSID *HEX`, the system code page is sent. If the system also uses `CCSID *HEX`, no code page translation is performed.

`number`: Send a specific code page number to the server. This option requires that the code page translation table must be available to CCM Core.

Note The `exchange_data` function call in a CCM Core script fails unless the `SBMITPJOB` command specified either a `XCHGPGM` or `XCHGFIL` parameter.

The SBMITPJOB program accesses and creates files through the Integrated File System (IFS).

- To access /folder1/folder2/document.ext in QDLS, use `/QDLS/folder1/folder2/document.ext`
- To access /folder1/folder2/file.ext in IFS, use `/folder1/folder2/file.ext`
- To access member `MYMBR` in MYLIB/MYFILE, use `/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/MYMBR.MBR`

If no path is specified, files are stored in the root of the IFS file system.

An example is provided here.

```
SBMITPJOB
 SERVICE ('a service')
 PARS('key info' 34)
 HOST('10.0.0.11')
 PORT(3001)
 JOBID('AS/400 Job')
 SYNC(*YES)
```

This command sends a request to the Service `'a service'` configured in CCM Core that runs on 10.0.0.11 with external port 3001. The job has Job ID `'AS/400 Job'` and two parameters (`'key info'` and `34`). The `SBMITPJOB` command waits until the request has been processed.

REQHST file

If the LOG(*YES) parameter is specified, the `SBMITPJOB` command logs status information in the file `REQHST`. This file must be in the library list.

This file has the following format:

```
R REQHST
  SERVICE       64          TEXT('Service')
```

```
  USER          10          TEXT('User')
  HOST          64          TEXT('Host')
  PORT          16          TEXT('Port')
  JOBID         32          TEXT('Job ID')
  DATE           8  0       TEXT('Date')
  TIME           6  0       TEXT('Time')
  SYNC           1          TEXT('Synchronous request')
  STATUS         1          TEXT('Status')
  MESSAGE     1024          TEXT('Message text')
```

The `STATUS` field indicates the type of message:

- **C** Successful completion
- **D** Received a file from CCM Core. The `MESSAGE` field contains the file name on the IBM i host.
- **F** Failure
- **P** Progress message from CCM Core
- **S** Startup message. This message is logged for every request.
- **U** Sent a file to CCM Core. The `MESSAGE` field contains the file name on the IBM i host.

XCHGFIL file

If the XCHGFIL parameter is used, `SBMITPJOB` writes all exchanged information to the specified file. This file must have the following format.

```
R EXCHANGE
   JOBID         32          TEXT('Job ID')
   KEY           64          TEXT('Key')
   VALUE       1024          TEXT('Value')
```

The record format is named `EXCHANGE`.

The caller is responsible for creating this file.

XCHGPGM exit program

If the XCHGPMG parameter is used, `SBMITPJOB` calls the specified program as an exit program whenever information is exchanged. This exit program can specify a response, which is sent back to CCM Core.

The exit program must have the following interface in CL format:

```
PGM        PARM(&JOBID &KEY &VALUE &RESPONSE)

DCL        VAR(&JOBID)    TYPE(*CHAR) LEN(  32) /* Input  */
DCL        VAR(&KEY)      TYPE(*CHAR) LEN(  64) /* Input  */
DCL        VAR(&VALUE)    TYPE(*CHAR) LEN(1024) /* Input  */
DCL        VAR(&RESPONSE) TYPE(*CHAR) LEN(1024) /* Output */

ENDPGM
```

Java submission interface

Job class

The following is a public class job that extends `java.lang.Object`. This class implements an interface on CCM Core for job submission.

```
java.lang.Object
  |
  +--com.aia_itp.itpdsapi.Job
```

The following example shows how to submit a job to a local CCM Core server to run a template. The result document is returned in PDF format for which the ITPDSDataReceiver interface is implemented. For the CCM Core Service, the following parameters are defined:

- Template to be executed.
- XML data input for the template.

The CCM Core Service is configured on port 5335.

```
  import com.aia_itp.itpdsapi.*;
  import java.io.*;

  class MyClass implements ITPDSDataReceiver {
    static String PORT_ITPSERVER= "3001";

    ....

    public void ProduceDocument(String model, String xmldata) throws Exception {
      Job j = new Job("127.0.0.1", PORT_ITPSERVER);
      j.addParameter("RunMdlPdf"); // Service name
      j.addParameter(model);
      j.addParameter(xmldata);
      j.setAdvancedCapabilities(null, this);
      if(!j.submit(true)){
        throw new Exception(j.getLastError());
      }
    }

    public OutputStream ITPDSReceiveData(String DataItem)
    {
      try
      {
        return new BufferedOutputStream(new FileOutputStream("/temp/myfile.pdf"));
      }
      catch (FileNotFoundException e)
      {
        return null;
      }
    }

    public void ITPDSReceiveDataFinished(String DataItem, OutputStream out){
      try
      {
        out.close();
      }
      catch (IOException e)
      {
      }
    }

    ....

  }
```

The following job creates a CCM Core job for a CCM Core server and instance specified by host and port.

```
public Job(java.lang.String host, java.lang.String port)
```

The job has two parameters:

1. `host`. IP address or host name of the CCM Core server
2. `port`. Port on which the CCM Core instance is configured on the server

The following job creates a CCM Core job for a CCM Core server and instance specified by `host` and `port`. The job is identified by a specified `jobID`.

```
public Job(java.lang.String host,java.lang.String port, java.lang.String jobID)
```

The job has three parameters:

1. `host`. IP address or host name of the CCM Core server
2. `port`. Port on which the CCM Core instance is configured on the server
3. `jobID`. ID identifying the job. This ID is shown in the monitor and in the log

Methods

The Java submission interface includes the following methods.

- **setProgressListener**

```
public void setProgressListener(ProgressListener audience)
```

This method enables `ProgressListener` for this CCM Core job. `ProgressListener` receives all progress events.

The method has one parameter:

1. `audience`. Object implementing the `ProgressListener` interface.

- **setAdvancedCapabilities**

```
public void setAdvancedCapabilities(ITPDSDataSender sender,ITPDSDataReceiver receiver)
```

This method tells the server that this client implements advanced capabilities.

It has two parameters:

1. `sender`. Object implementing the `ITPDSDataSender` interface.
2. `receiver`. Object implementing the `ITPDSDataReceiver`.

- **setAdvancedCapabilities**

```
public void setAdvancedCapabilities(ITPDSDataSender sender,ITPDSDataReceiver
 receiver,ITPDSExchangeData
exchange_data)
```

This method tells the server which advanced capabilities this client implements.

It has three parameters:

1. `sender`. Object implementing the `ITPDSDataSender` interface.
2. `receiver`. Object implementing the `ITPDSDataReceiver` interface.

3. `exchange_data`. Object implementing the `ITPDSExchangeData` interface.

- **setConfirmDisconnect**

```
public void setConfirmDisconnect(boolean confirm_disconnect)
```

This method configures whether the disconnect from the server is confirmed.

- **addParameter**

```
public void addParameter(java.lang.String parameter)
```

This method adds a single parameter string to the parameter list for this job.

It has one parameter:

1. `parameter`. Job parameter.

- **setParameters**

```
public void setParameters(java.lang.String[] parameters)
```

This method sets the list of job parameters.

It has one parameter:

1. `parameters`. An array of job parameters (strings).

- **clearParameters**

```
public void clearParameters()
```

This method clears the parameter list.

- **setHost**

```
public void setHost(java.lang.String host)
```

This method sets the CCM Core server for this job.

It has one parameter:

1. `host`. IP address or host name of the CCM Core server.

- **setPort**

```
public void setPort(java.lang.String port)
```

This method sets the CCM Core instance port for this job.

It has one parameter:

1. `port`. Port using which the CCM Core instance is configured on the server.

- **setServer**

```
public void setServer(java.lang.String host, java.lang.String port)
```

This method sets the CCM Core server and instance for this job.

It has two parameters:

1. `host`. IP address or host name of the CCM Core server.
2. `port`. Port using which the CCM Core instance is configured on the server.

- **setJobID**

```
public void setJobID(java.lang.String jobID)
```

This method sets the CCM Core job identifier for this job.

It has one parameter:

1. `jobID`. Job identifier.

- **setSessionID**

```
public void setSessionID(java.lang.String sessionID)
```

This method sets the CCM Core session identifier for this job.

It has one parameter:

1. `sessionID`. Session identifier.

- **getParameters**

```
public java.lang.String[] getParameters()
```

This method returns the current parameter list for this job.

- **getHost**

```
public java.lang.String getHost()
```

This method returns CCM Core server for this job.

- **getPort**

```
public java.lang.String getPort()
```

This method returns CCM Core instance port for this job.

- **getJobID**

```
public java.lang.String getJobID()
```

This method returns CCM Core job identifier for this job.

- **getSessionID**

```
public java.lang.String getSessionID()
```

This method returns CCM Core session identifier for this job.

- **getLastError**

```
public java.lang.String getLastError()
```

This method returns the last known error if there is one.

- **submit**

```
public boolean submit(boolean sync)
```

It throws `java.lang.Exception`.

You can use this method to submit a job to CCM Core using the properties and parameters specified in the job. This method returns TRUE if the job was asynchronous and the job was queued, or if the job was synchronous and successfully finished processing. If the job was synchronous and did not complete successfully, the method returns FALSE. If the job is asynchronous, the connection is closed after the job is queued, denying all possibilities for progress information and data transfers.

It has one parameter:

1. `sync`. Specifies whether the job is submitted synchronously or asynchronously.

- **submit**

```
public boolean submit(boolean sync, java.lang.String user)
```

It throws `java.lang.Exception`.

This method is used to submit a job to CCM Core using the properties and parameters specified in the job. This method returns TRUE if the job was asynchronous and the job was queued, or if the job was synchronous and successfully finished processing. If the job was synchronous and did not complete successfully, the method returns FALSE. If the job is asynchronous, the connection is closed after the job is queued, denying all possibilities for feedback and other transfers, such as that of files.

It has two parameters:

1. `sync`. Specifies whether the job is submitted synchronously or asynchronously.
2. `user`. Submits the job with this `userid`. The `userid` does not have to exist.

- **submit**

```
public boolean submit(boolean sync, int ByteCoding)
```

It throws `java.lang.Exception`.

This method is used to submit a job to CCM Core using the properties and parameters specified in the job. This method returns TRUE if the job was asynchronous and the job was queued, or if the job was synchronous and successfully finished processing. If the job was synchronous and not completed successfully, false will be returned instead. If the job is asynchronous, the connection is closed after the job is queued, denying all possibilities for feedback and other transfers, such as that of files.

It has two parameters:

1. `sync`. Specifies whether the job is submitted synchronously or asynchronously.
2. `ByteCoding`. Submits the job using this byte coding. `ByteCoding` can be either ITPDS.RQST_IN_ASCII or ITPDS.RQST_IN_UNICODE.

- **submit**

```
public boolean submit(boolean sync, java.lang.String user, int
byteCoding)
```

It throws `java.lang.Exception`.

This method is used to submit a job to CCM Core, using the properties and parameters specified in the job. This method returns true if the job was asynchronous and the job was queued, or if the job was synchronous and successfully finished processing. If the job was synchronous and did not complete successfully, the method returns FALSE. If the job is asynchronous, the connection is closed after the job is queued, denying all possibilities for feedback and other transfers, such as that of files.

It has three parameters:

1. `sync`. Specifies whether the job is submitted synchronously or asynchronously.
2. `user`. Submits the job with this `userid`. The `userid` does not have to exist.
3. `ByteCoding`. Submits the job using this byte coding. `ByteCoding` can be either ITPDS.RQST_IN_ASCII or ITPDS.RQST_IN_UNICODE.

ITPDS class

The ITPDS class is a public class that extends java.lang.Object. constants for using the com.aia_itp.itpdsapi package.

```
java.lang.Object
  |
  +--com.aia_itp.itpdsapi.ITPDS
```

Fields

The ITPDS class has the following fields.

- **RQST_IN_ASCII**

This is a public static final `int`. It identifies that the job is submitted in ASCII.

- **RQST_IN_UNICODE**

This is a public static final `int`. It identifies that the job is submitted in Unicode by default.

The constructor of the ITPDS class is public `ITPDS()`.

ITPDSDataReceiver interface

`ITPDSDataReceiver` is a public interface.

The `ITPDSDataReceiver` interface provides for receiving binary data from the CCM Core server. This is typically the result document of a CCM process.

This interface is used to receive data send by `Src(…) Dest(…)` of the `SendFile` command.

```
import com.aia_itp.itpdsapi.*;
  import java.io.*;

  class MyClass implements ITPDSDataReceiver
  {

    ....

    public OutputStream ITPDSReceiveData(String DataItem)
    {
      try
      {
```

```
        return new BufferedOutputStream(new FileOutputStream("/temp/myfile.pdf"));
      }
      catch(Exception e)
      {
        return null;
      }
    }

    public void ITPDSReceiveDataFinished(String DataItem, OutputStream out)
    {
      try
      {
        out.close();
      }
      catch(Exception e)
      {
      }
    }

    ....

  }
```

Methods

The ITPDSDataReceiver includes the following methods.

- **ITPDSReceiveData**

```
public java.io.OutputStream ITPDSReceiveData(java.lang.String DataItem)
```

This method is called when the CCM Core server executes the `SendFile` command. This method returns either null to indicate that it does not want to receive the data or an `OutputStream` object to which the data is written.

It has one parameter:

1. `DataItem`. The client parameter as passed in `Src(...) Dest(...)` of the `SendFile` command.

- **ITPDSReceiveDataFinished**

```
public void ITPDSReceiveDataFinished(java.lang.String DataItem,
java.io.OutputStream out)
```

This method is called when receipt of the data has been finished. When this method is called, all data has been written to `OutputStream` returned by `ITPDSReceiveData`. It is typically used to close the `OutputStream` object.

It has one parameter:

1. `DataItem`. The client parameter as passed in `Src(...) Dest(...)` of the `SendFile` command.

   `out`. The `OutputStream` object as returned by `ITPDSReceiveData`.

ITPDSDataSender interface

`ITPDSDataSender` is a public interface.

The `ITPDSDataSender` interface provides for sending binary data to the CCM Core server. This is typically the XML stream used as data input for a CCM process.

This interface is used to send data when it is requested by `Src(...) Dest(...)` of the `ReceiveFile` command.

```
import com.aia_itp.itpdsapi.*;
  import java.io.*;

  class MyClass implements ITPDSDataSender
  {

    ....

    public ITPDSInputStream ITPDSSendData(String DataItem)
    {
      try
      {
        // We're going to 'read' the file corresponding to 'DataItem'...
        java.io.File f = new java.io.File(DataItem + ".xml");
        InputStream stream =  new BufferedInputStream(new FileInputStream(f));
        return new ITPDSInputStream(stream, (int)f.length());
      }
      catch(Exception e)
      {
        return null;
      }
    }

    public void ITPDSSendDataFinished(String DataItem, ITPDSInputStream out)
    {
      try
      {
        out.getInputStream().close();
      }
      catch(Exception e)
      {
      }
    }

    ....

  }
```

Methods

The `ITPDSDataSender` interface includes the following methods.

- **ITPDSendData**

```
public ITPDSInputStream ITPDSSendData(java.lang.String DataItem)
```

This method is called when the CCM Core server executes the `ReceiveFile` command.

It returns either null to indicate that it does not want to send the data or an ITPDSInputStream object from which the data will be read.

The returned `InputStream` object must be wrapped in an `ITPDSInputStream` object because CCM Core requires the size of the data to be available before actually sending the data.

The method has one parameter:

1. `DataItem`. The client parameter as passed in `Src(...) Dest(...)` of the `ReceiveFile` command.

- **ITPDSendDataFinished**

```
public void ITPDSSendDataFinished(java.lang.String DataItem,ITPDSInputStream in)
```

This method is called when sending the data has been finished. When it is called, it means that all data has been sent to the CCM Core server. It is typically used to close the `InputStream` object.

It has one parameter:

1. `DataItem`. The client parameter as passed in `Src(...) Dest(...)` of the `ReceiveFile` command.

   `in`. The `ITPDSInputStream` object as returned by `ITPDSSendData`.

ITPDSExchangeData interface

The `ITPDSExchangeData` interface provides for receiving name-value pair data from the CCM Core server.

You can use this interface to exchange data through the `exchange_data(key, value, time-out)` function.

```
import com.aia_itp.itpdsapi.*;

class MyClass implements ITPDSExchangeData {

  ....

  public String ITPDSExchangeData(String Key, String Value){
    return "my_own_data";
  }

  ....

}
```

Methods

The `ITPDSExchangeData` interface has one method, which is `ITPDSExchangeData(java.lang.String ID, java.lang.String Data)`. This is a public `java.lang.String` method.

This method is called when the CCM Core server executes the `exchange_data(Key, Value, Time-out)` function call. This method returns an optionally empty string to the server.

It has two parameters:

1. **Key**. The k parameter as passed in the `exchange_data` function.
2. **Value**. The v parameter as passed in the `exchange_data` function.

ITPDSInputStream class

`ITPDSInputStream` is a public class that extends java.lang.Object.

```
java.lang.Object
  |
  +--com.aia_itp.itpdsapi.ITPDSInputStream
```

This class wraps a normal `InputStream` object or an object from a derived class, such as `BufferedInputStream`, and adds a specification of the size to it.

An `ITPDSInputStream` object is used when the class implements the `ITPDSDataSender` interface.

`ITPDSInputStream(java.io.InputStream in, int size)` is a public constructor that creates an `ITPDSInputStream` object and specifies the size.

It has the following parameters:

1. **in**. Any `InputStream` (or descendant from `InputStream`) object.
2. **size**. Exact size in bytes of the data that CCM Core reads from the `InputStream`.

Methods

The `ITPDSInputStream` class has the following methods:

- **getInputStream**

public `java.io.InputStream getInputStream()`

Returns the `InputStream` object specified when creating the `ITPDSInputStream` object.

- **getSize**

public `int getSize()`

Returns the size specified when creating the `ITPDSInputStream` object.

ProgressListener interface

The `ProgressListener` interface provides for receiving progress information from the CCM Core server.

This interface is used to intercept messages send by the `Progress Message(...)` command.

```
import com.aia_itp.itpdsapi.*;

  class MyClass implements ProgressListener {

    ....

    public void progressReturned(String message){
      System.out.println(message);
    }

    ....

  }
```

Methods

The `ProgressListener` interface has one method `progressReturned`.

public void `progressReturned(java.lang.String message)`

This method implements the sink for progress messages.

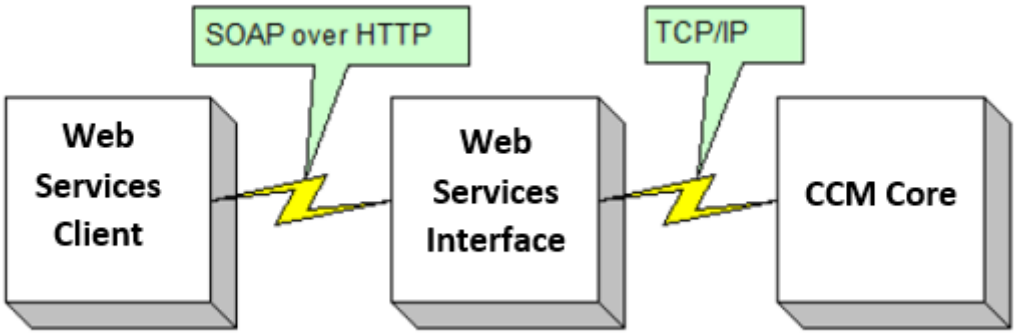
It has the following parameters:

- **message**. The progress message as passed in the command `Progress Message(...)`.

Web Services interface

You can submit jobs to CCM Core with the CCM Core Web Services interface. CCM Core Web Services clients use SOAP over HTTP to submit a job to the ASP.NET Web Services interface. This Web Services interface takes care of unpacking the SOAP message and sending it to CCM Core over a TCP/IP interface. The Web Services interface is available for both ASP.NET and J2EE.

The following figure represents the architecture of submitting jobs to CCM Core using the CCM Core Web Services interface.



ASP.NET implementation

After the installation, you can find the WSDL interface specifications by accessing the following web page: http://<machinename>:<port>/itpserver/itpserver.asmx

At this URL, a web page is presented with references to WSDL interface specifications for all supported interface variants.

J2EE implementation

After the installation, you can find the WSDL interface specifications by accessing the following web page: http://<machinename>:<port>/itpserver

At this URL, a web page is presented with references to WSDL interface specifications for all supported interface variants.

Interface variants

The CCM Core Web Services interface provides three variants:

- The interface provided through http://.../itpserver/itpserver.asmx uses bare SOAP parameters.
- The interface provided through http://.../itpserver/itpserverwrapped.asmx uses wrapped SOAP parameters, as for instance required by BizTalk.
- The interface provided through http://.../itpserver/services/ITPServer is provided for backward compatibility with an earlier J2EE implementation of the CCM Core Web Services interface. This interface is only available in the J2EE implementation.

Submit a synchronous job to the Web Services interface

This section describes parameters used to submit a synchronous job to the Web Services interface.

Submit

The job `Submit` takes the following parameters:

- **Service**
  This is the name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Input files**
  Input files to send files to CCM Core that you can collect with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **SubmitResult**
  Contains the answer of CCM Core to the Web Service request. It can contain output files to store files sent by CCM Core and a string returned by CCM Core, such as for error messages.

You can find the complete WSDL interface specification itpserverwsdl.xml and itpserverwrappedwsdl.xml in the Manuals folder of your CCM Core installation.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserver.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/Submit"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <service xmlns="http://www.aia-itp.com/itpserver/wsdl">string</service>
    <parms xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
```

```
      <string>string</string>
    </parms>
    <inDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </inDocuments>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <error xmlns="http://www.aia-itp.com/itpserver/wsdl">string</error>
    <outDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </outDocuments>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverwrapped.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/Submit"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Submit xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <service>string</service>
      <parms>
        <parm>string</parm>
        <parm>string</parm>
      </parms>
      <inDocuments>
        <Doc>
```

```
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </inDocuments>
    </Submit>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitResponse xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <error>string</error>
      <outDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </outDocuments>
    </SubmitResponse>
  </soap:Body>
</soap:Envelope>
```

Client.cs offers the function `SubmitJob(...)` that you can use in a C# environment to create and unwrap the SOAP messages as well as to post the messages. Wsclient.cs is a sample implementation of the function `SubmitJob(...)`. These functions are currently only available for the bare version.

SubmitEx

The job `SubmitEx` is an extension to the job `Submit`. It allows for communication between the caller and the CCM Core script. It allows the script to use `Progress()` to log messages to the caller, and it allows `exchange_data()` calls to get and set key/value pairs from/to the client. The job `SubmitEx` takes the same parameters as the `Submit` job plus an array with key/value pairs as an extra parameter.

The job `SubmitEx` takes the following parameters:

- **Service**
  This is the name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Key/value pairs**

Array of key/value pairs that you can collect for the function `exchange_data`.

- **Input files**
  Input files to send files to CCM Core that can be collected with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.

SubmitExResult

`SubmitExResult` contains the answer of CCM Core to the Web Service request. This parameter can contain the following objects:

- **Output files**
  Files sent by CCM Core are stored here.
- **Key/value pairs**
  The array with the key/value pairs as set in the function call and changed with the function `exchange_data`.
- **Progress messages**
  Result of calls of `Progress()` by the CCM Core script.
- A string returned by CCM Core, such as, for error messages.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserver.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitEx"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <service xmlns="http://www.aia-itp.com/itpserver/wsdl">string</service>
    <parms xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </parms>
    <inKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </inKeyValues>
    <inDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </inDocuments>
```

```
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <error xmlns="http://www.aia-itp.com/itpserver/wsdl">string</error>
    <outKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </outKeyValues>
    <outDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </outDocuments>
    <progress xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </progress>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverwrapped.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitEx"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitEx xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <service>string</service>
      <parms>
        <parm>string</parm>
        <parm>string</parm>
      </parms>
```

```
      <inKeyValues>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </inKeyValues>
      <inDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </inDocuments>
    </SubmitEx>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitExResponse xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <error>string</error>
      <outKeyValues>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </outKeyValues>
      <outDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </outDocuments>
      <progress>
        <parm>string</parm>
        <parm>string</parm>
      </progress>
    </SubmitExResponse>
```

```
  </soap:Body>
</soap:Envelope>
```

SubmitEx2

The job `SubmitEx2` is an extension to the job `SubmitEx`. It supports Job IDs and Session IDs. Therefore, the job `SubmitEx2` takes the same parameters as the `SubmitEx` job as well as the following extra parameters:

- **Service**
  This is the name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Key/value pairs**
  Array of key/value pairs used for the function `exchange_data`.
- **Input files**
  Input files to send files to CCM Core that you can collect with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **Session ID**
  Used to run a CCM Job in the context of an existing CCM session. If you pass on an invalid Session ID, CCM Core executes the job, but not in the context of the CCM session.
- **Job ID**
  Used to identify the job in the CCM log files. If you pass on an empty value, the Web Service call generates a unique Job ID and uses that when it submits the job.

SubmitEx2Result

`SubmitEx2Result` contains the answer of CCM Core to the Web Service request. It can contain the following objects:

- **Output files**
  Files sent by CCM Core are stored here.
- **Key/value pairs**
  The array with the key/value pairs as set in the function call and changed with the function `exchange_data`.
- **Progress messages**
  Result of calls of `Progress()` by the CCM Core script.
- A string returned by CCM Core, such as, for error messages.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserver.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitEx2"

<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <service xmlns="http://www.aia-itp.com/itpserver/wsdl">string</service>
    <parms xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </parms>
    <inKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </inKeyValues>
    <inDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </inDocuments>
    <sessionID xmlns="http://www.aia-itp.com/itpserver/wsdl">string</sessionID>
    <jobID xmlns="http://www.aia-itp.com/itpserver/wsdl">string</jobID>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <error xmlns="http://www.aia-itp.com/itpserver/wsdl">string</error>
    <outKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </outKeyValues>
    <outDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
```

```
        <content>base64Binary</content>
      </Doc>
    </outDocuments>
    <progress xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </progress>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverwrapped.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitEx2"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitEx2 xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <service>string</service>
      <parms>
        <parm>string</parm>
        <parm>string</parm>
      </parms>
      <inKeyValues>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </inKeyValues>
      <inDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </inDocuments>
      <sessionID>string</sessionID>
      <jobID>string</jobID>
    </SubmitEx2>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitEx2Response xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <error>string</error>
      <outKeyValues>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </outKeyValues>
      <outDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </outDocuments>
      <progress>
        <parm>string</parm>
        <parm>string</parm>
      </progress>
    </SubmitEx2Response>
  </soap:Body>
</soap:Envelope>
```

Submit an asynchronous job to the Web Services interface

This section describes parameters used to submit a asynchronous job to the Web Services interface.

SubmitAsync

The job `SubmitAsync` takes the following parameters:

- **Service**
  The name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Input files**
  Input files to send files to CCM Core that you can collect with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **SubmitResult**
  Contains the answer of CCM Core to the Web Service request. It can contain output files to store files sent by CCM Core and a string returned by CCM Core, such as for error messages.
- **CorrelationID**
  This unique identifier is returned by the Web Services API as part of the returned SOAP message when a job is finished, so the calling application knows which task has finished.
- **ReturnPath**

The URL to which the SOAP message is sent after the Web Services API finishes its task.

`SubmitAsync` responds with OK if execution of the job was scheduled successfully; otherwise, an error indication is returned. If the job is finished, the results are posted to the URL mentioned in the parameter `ResultPath`.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserver.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitAsync"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <service xmlns="http://www.aia-itp.com/itpserver/wsdl">string</service>
    <parms xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </parms>
    <inDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </inDocuments>
    <correlationId xmlns="http://www.aia-itp.com/itpserver/wsdl">string</correlationId>
    <returnPath xmlns="http://www.aia-itp.com/itpserver/wsdl">string</returnPath>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <error xmlns="http://www.aia-itp.com/itpserver/wsdl">string</error>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverwrapped.asmx HTTP/1.1
```

```
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitAsync"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitAsync xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <service>string</service>
      <parms>
        <parm>string</parm>
        <parm>string</parm>
      </parms>
      <inDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </inDocuments>
      <correlationId>string</correlationId>
      <returnPath>string</returnPath>
    </SubmitAsync>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitAsyncResponse xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <error>string</error>
    </SubmitAsyncResponse>
  </soap:Body>
</soap:Envelope>
```

SubmitAsync results

To receive `SubmitAsync` results, a Web Service must implement the function `ReplyAsync` if the request was sent using the bare variant (itpserver.asmx), or the function `ReplyAsyncWrapped` if the request was sent using the wrapped variant (itpserverwrapped.asmx).

These functions receive the following parameters:

- **jobResult**
  The result/error indication of running the job in the string format.
- **OutDocuments**
  An array of files generated by the CCM Core service.

- **CorrelationId**
  The call identification specified when submitting the job. The function `ReplyAsync` (wrapped) should return a result/error string that, except for optional logging, is discarded.

itpserverreply.dll

The itpserverreply.dll file provides a reference implementation that serves to specify the interface. To use it, do the following:

1. Install the Web Services interface.
2. Browse to the following URLs for sample request and reply messages:

http://<host:port>/<folder>/itpserverreply.asmx?WSDL for the WSDL specification

http://<host:port>/<folder>/itpserverreply.asmx?op=ReplyAsync

http://<host:port>/<folder>/itpserverreply.asmx?op=ReplyAsyncWrapped

<host:port>/<folder> is the machine/folder in which the Web Services interface is installed, such as localhost:8080/itpserver.

The Web Service that you implement must adhere to this interface specification.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserverreply.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/ReplyAsync"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <jobResult xmlns="http://www.aia-itp.com/itpserver/wsdl">string</jobResult>
    <outDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </outDocuments>
    <correlationId xmlns="http://www.aia-itp.com/itpserver/wsdl">string</correlationId>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
```

```
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <result xmlns="http://www.aia-itp.com/itpserver/wsdl">string</result>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverreply.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/ReplyAsyncWrapped"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReplyAsyncWrapped xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <jobResult>string</jobResult>
      <outDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </outDocuments>
      <correlationId>string</correlationId>
    </ReplyAsyncWrapped>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReplyAsyncWrappedResponse xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <result>string</result>
    </ReplyAsyncWrappedResponse>
  </soap:Body>
</soap:Envelope>
```

SubmitAsyncEx2

The `SubmitAsyncEx2` job takes the same parameters as the `SubmitEx2` job except the job is submitted asynchronously. As the job is submitted asynchronously, the call also needs the `CorrelationID` parameter and the `ReturnPath` parameter.

The job takes the following parameters:

- **Service**
  This is the name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Key/value pairs**
  Array of key/value pairs that are used for the function `exchange_data`.
- **Input files**
  Input files to send files to CCM Core that you can collect with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **JobID**
  The job runs on CCM Core with this ID. If no job ID is passed on, the Web Service API uses a unique ID, so you can track your job in the CCM Core logs.
- **SessionID**
  The job runs on CCM Core within the context of the CCM session indicated by `SessionID`. If no valid `SessionID` is passed on, CCM Core executes the job.
- **CorrelationID**
  This unique identifier is returned by the Web Services API as part of the returned SOAP message when a job is finished, so the calling application knows which task has finished.
- **ReturnPath**
  The URL to which the SOAP message is sent after the Web Services API finishes its task.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserver.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitAsyncEx"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <service xmlns="http://www.aia-itp.com/itpserver/wsdl">string</service>
    <parms xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <string>string</string>
      <string>string</string>
    </parms>
    <inKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
```

```
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </inKeyValues>
    <inDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </inDocuments>
    <sessionID xmlns="http://www.aia-itp.com/itpserver/wsdl">string</sessionID>
    <jobID xmlns="http://www.aia-itp.com/itpserver/wsdl">string</jobID>
    <correlationId xmlns="http://www.aia-itp.com/itpserver/wsdl">string</correlationId>
    <returnPath xmlns="http://www.aia-itp.com/itpserver/wsdl">string</returnPath>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <error xmlns="http://www.aia-itp.com/itpserver/wsdl">string</error>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverwrapped.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/SubmitAsyncEx2"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitAsyncEx2 xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <service>string</service>
      <parms>
        <parm>string</parm>
        <parm>string</parm>
      </parms>
      <inKeyValues>
        <KeyValue>
          <key>string</key>
```

```
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </inKeyValues>
      <inDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </inDocuments>
      <sessionID>string</sessionID>
      <jobID>string</jobID>
      <correlationId>string</correlationId>
      <returnPath>string</returnPath>
    </SubmitAsyncEx2>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitAsyncEx2Response xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <error>string</error>
    </SubmitAsyncEx2Response>
  </soap:Body>
</soap:Envelope>
```

SubmitAsyncEx2 results

To receive SubmitAsyncEx2 results, a web service must implement the function `ReplyAsyncEx2` if the request was sent using the bare variant (itpserver.asmx), or the function `ReplyAsyncEx2Wrapped` if the request was sent using the wrapped variant (itpserverwrapped.asmx).

These functions receive the following parameters:

- **jobResult**
  The result/error indication of running the job (in string format).
- **outKeyValues**
  An array of key/values sent by the CCM Core Service.
- **outDocuments**
  An array of files generated by the CCM Core Service.
- **progress**
  An array of progress messages sent by the CCM Core Service.

- **CorrelationId**
  The call identification that was specified when submitting the job.

The function `ReplyAsyncEx2(Wrapped)` should return a result/error string that, except for optional logging, is discarded.

itpserverreply.dll

The itpserverreply.dll file provides a reference implementation that serves to specify the interface. To use it, do the following:

1. Install the Web Services interface.
2. Browse to the following URLs for sample request and reply messages:
   http://<host:port>/<folder>/itpserverreply.asmx?WSDL for the WSDL specification
   http://<host:port>/<folder>/itpserverreply.asmx?op=ReplyAsync
   http://<host:port>/<folder>/itpserverreply.asmx?op=ReplyAsyncWrapped
   <host:port>/<folder> is the machine/folder in which the Web Services interface is installed, such as localhost:8080/itpserver.

The Web Service that you implement must adhere to this interface specification.

Sample request (bare version)

The following code is a sample bare request.

```
POST /itpserver/itpserverreply.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/ReplyAsyncEx2"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <jobResult xmlns="http://www.aia-itp.com/itpserver/wsdl">string</jobResult>
    <outKeyValues xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
      <KeyValue>
        <key>string</key>
        <value>string</value>
      </KeyValue>
    </outKeyValues>
    <outDocuments xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
      <Doc>
        <id>string</id>
        <content>base64Binary</content>
      </Doc>
    </outDocuments>
    <progress xmlns="http://www.aia-itp.com/itpserver/wsdl">
```

```
      <string>string</string>
      <string>string</string>
    </progress>
    <correlationId xmlns="http://www.aia-itp.com/itpserver/wsdl">string</correlationId>
  </soap:Body>
</soap:Envelope>
```

Sample reply (bare version)

The following code is a sample bare reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <result xmlns="http://www.aia-itp.com/itpserver/wsdl">string</result>
  </soap:Body>
</soap:Envelope>
```

Sample request (wrapped version)

The following code is a sample wrapped request.

```
POST /itpserver/itpserverreply.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.aia-itp.com/itpserver/wsdl/ReplyAsyncEx2Wrapped"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReplyAsyncEx2Wrapped xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <jobResult>string</jobResult>
      <outKeyValues>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
        <KeyValue>
          <key>string</key>
          <value>string</value>
        </KeyValue>
      </outKeyValues>
      <outDocuments>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
        <Doc>
          <id>string</id>
          <content>base64Binary</content>
        </Doc>
      </outDocuments>
      <progress>
        <string>string</string>
        <string>string</string>
      </progress>
```

```
      <correlationId>string</correlationId>
    </ReplyAsyncEx2Wrapped>
  </soap:Body>
</soap:Envelope>
```

Sample reply (wrapped version)

The following code is a sample wrapped reply.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReplyAsyncEx2WrappedResponse xmlns="http://www.aia-itp.com/itpserver/wsdl">
      <result>string</result>
    </ReplyAsyncEx2WrappedResponse>
  </soap:Body>
</soap:Envelope>
```

Sample clients for synchronous and asynchronous jobs

Sample clients for a synchronous job and asynchronous job (wsasyncclient) submission are provided in the client folder that resides in: `{path}\APIs\Web Services\Sample Client`. These clients demonstrate how to call CCM Core with the Web Services interface using the bare variant.

If you want to use a sample client, you need to install the .NET Framework on the machine used to run the sample client.

To use a sample client, the following parameters need to be provided:

- Services address. This is the address of the Web Services server.
- Service. The name of the Service called.
- Parameters. Parameters needed for the particular Service.
- Upload file. Optional. File that needs to be sent from the client to CCM Core.
- File ID. Required when the Upload file parameter is used. ID for the file sent from the client. This ID can be used by the Service to collect the file sent with the Upload file parameter.

Compatibility interfaces

This section provides information on the interfaces provided for backward compatibility with an earlier J2EE implementation of the CCM Core Web Services interface. These interfaces are only available in the J2EE implementation of the CCM Core Web Services interface with the URL http://.../itpserver/services/ITPServer.

Note The interfaces described in the following sections are for reference purpose only and should not be used in new implementations.

SubmitEx

The job `SubmitEx` takes the following parameters:

- **Service**
  The name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Key/value pairs**
  Array of key/value pairs used for the function `exchange_data`.
- **Input files**
  Input files to send files to CCM Core that can be collected with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **Job ID**
  The job runs on CCM Core with this ID. If no job ID is passed on, the Web Service API uses a unique ID, so you can track your job in the CCM Core logs.

SubmitExResult

`SubmitExResult` contains the answer of CCM Core to the Web Service request. It can contain the following objects:

- **Output files**
  Files sent by CCM Core are stored here.
- **Key/value pairs**
  The array with the key/value pairs as set in the function call and changed with the function `exchange_data`.
- **Progress messages**
  Result of calls of `Progress()` by the CCM Core script.
- A string returned by CCM Core, such as, for error messages.

Sample request for SubmitEx

The following code is a sample `SubmitEx` request.

```
POST /itpserver/services/ITPServer HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: ""

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsdl="http://wsdl.aia_itp.com">
  <soapenv:Header/>
  <soapenv:Body>
    <wsdl:SubmitEx>
      <wsdl:service>string</wsdl:service>
      <wsdl:parms>
        <wsdl:string>string</wsdl:string>
        <wsdl:string>string</wsdl:string>
```

```
      </wsdl:parms>
      <wsdl:inKeyValues>
        <wsdl:KeyValue>
          <wsdl:key>string</wsdl:key>
          <wsdl:value>string</wsdl:value>
        </wsdl:KeyValue>
        <wsdl:KeyValue>
          <wsdl:key>string</wsdl:key>
          <wsdl:value>string</wsdl:value>
        </wsdl:KeyValue>
      </wsdl:inKeyValues>
      <wsdl:inDocuments>
        <wsdl:Doc>
          <wsdl:content>base64Binary</wsdl:content>
          <wsdl:id>string</wsdl:id>
        </wsdl:Doc>
      </wsdl:inDocuments>
      <wsdl:jobID>string</wsdl:jobID>
    </wsdl:SubmitEx>
  </soapenv:Body>
</soapenv:Envelope>
```

Sample reply for SubmitEx

The following code is a sample `SubmitEx` reply.

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <SubmitExResponse xmlns="http://wsdl.aia_itp.com">
      <SubmitExResult>
        <error>OK</error>
        <outDocuments>
          <Doc>
            <content>base64Binary</content>
            <id>string</id>
          </Doc>
        </outDocuments>
        <outKeyValues>
          <KeyValue>
            <key>string</key>
            <value>string</value>
          </KeyValue>
          <KeyValue>
            <key>string</key>
            <value>string</value>
          </KeyValue>
        </outKeyValues>
        <progress>
          <string>string</string>
          <string>string</string>
        </progress>
      </ns1:SubmitExResult>
    </ns1:SubmitExResponse>
  </soap:Body>
</soap:Envelope>
```

SubmitEx2

The `SubmitEx2` job takes the same parameters as the `SubmitEx` job as well as an extra string parameter to indicate in which session the job should run.

The job `SubmitEx2` takes the following parameters:

- **Service**
  This is the name of the Service the job is submitted to such as `RunMdl`.
- **Parameters**
  The parameters for the job. They are passed as a string to the Web Services interface.
- **Key/value pairs**
  Array of key/value pairs that are used for the function `exchange_data`.
- **Input files**
  Input files to send files to CCM Core that can be collected with the `ReceiveFile` command. The content of the file as well as an ID are passed. CCM Core can use this ID to collect the file.
- **Session ID**
  Used to run a CCM Job in the context of an existing CCM session. If you pass on an invalid Session ID, CCM Core executes the job, but not in the context of the CCM session.
- **Job ID**
  Used to identify the job in the CCM log files. If you pass on an empty value, the Web Service call generates a unique Job ID to use when it submits the job.

SubmitEx2Result

`SubmitEx2Result` contains the answer of CCM Core to the Web Service request. This parameter can contain the following objects:

- **Output files**
  Files sent by CCM Core are stored here.
- **Key/value pairs**
  The array with the key/value pairs as set in the function call and changed with the function `exchange_data`.
- **Progress messages**
  Result of calls of `Progress()` by the CCM Core script.
- A string returned by CCM Core, such as for error messages.

Sample request for SubmitEx2

The following code is a sample `SubmitEx2` request.

```
POST /itpserver/services/ITPServer HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: ""

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsdl="http://wsdl.aia_itp.com">
```

```
  <soapenv:Header/>
  <soapenv:Body>
    <wsdl:SubmitEx2>
      <wsdl:service>string</wsdl:service>
      <wsdl:parms>
        <wsdl:string>string</wsdl:string>
        <wsdl:string>string</wsdl:string>
      </wsdl:parms>
      <wsdl:inKeyValues>
        <wsdl:KeyValue>
          <wsdl:key>string</wsdl:key>
          <wsdl:value>string</wsdl:value>
        </wsdl:KeyValue>
        <wsdl:KeyValue>
          <wsdl:key>string</wsdl:key>
          <wsdl:value>string</wsdl:value>
        </wsdl:KeyValue>
      </wsdl:inKeyValues>
      <wsdl:inDocuments>
        <wsdl:Doc>
          <wsdl:content>base64Binary</wsdl:content>
          <wsdl:id>string</wsdl:id>
        </wsdl:Doc>
      </wsdl:inDocuments>
      <wsdl:sessionID>string</wsdl:sessionID>
      <wsdl:jobID>string</wsdl:jobID>
    </wsdl:SubmitEx2>
  </soapenv:Body>
</soapenv:Envelope>
```

Sample reply for SubmitEx2

The following code is a sample `SubmitEx2` reply.

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <ns1:SubmitExResponse xmlns="http://wsdl.aia_itp.com">
      <SubmitExResult>
        <error>OK</error>
        <outDocuments>
          <Doc>
            <content>base64Binary</content>
            <id>string</id>
          </Doc>
        </outDocuments>
        <outKeyValues>
          <KeyValue>
            <key>string</key>
            <value>string</value>
          </KeyValue>
          <KeyValue>
            <key>string</key>
            <value>string</value>
          </KeyValue>
        </outKeyValues>
        <progress>
          <string>string</string>
          <string>string</string>
        </progress>
      </ns1:SubmitExResult>
```

```
    </ns1:SubmitExResponse>
  </soap:Body>
</soap:Envelope>
```

Directory Watch interface client

You can instruct CCM Core to watch a specific folder for jobs to be processed. You submit jobs by copying files to, or writing files in, this folder. CCM Core must be allowed to access the watched folder. You can use this interface if your application can copy files to the watched folder or create files in the watched folder in an NT domain.

Configure the Directory Watch interface

The CCM Core Directory Watch interface is implemented as a Windows Service that monitors one or more folders and submits synchronous requests to CCM Core based on these files. You can specify a separate folder for every CCM Core Service in the CCM Core Directory Watch interface.

Starting and stopping the Service

Currently no user interface is provided for the CCM Core Directory Watch interface. You can start and/or stop the Windows Service through the Windows Service Manager.

The Service is configured to start automatically when the server is rebooted.

Connecting to CCM Core

The CCM Core Directory Watch interface retrieves the connection configuration from the CCM Core configuration and does not require any additional setup.

Handling connection failures

If the CCM Core Directory Watch interface fails to connect to CCM Core, it logs an error and retries submitting the job until it is accepted. If the job cannot be submitted due to a configuration error, the job is rejected as a failure.

Number of connections

If there are sufficient jobs queued, the CCM Core Directory Watch interface submits two jobs per licensed CCM Document Processor simultaneously. If the license is issued for an unlimited number of CCM Document Processors, the number of simultaneous requests is limited to two jobs per CPU core on CCM Core.

You can explicitly set the number of simultaneous requests to 2 * n by adding the setting `ScalingMaxDP=n` to the `[Configuration]` section of the DP.INI file.

Configure watched directories

The CCM Core Directory Watch interface can watch one folder for every CCM Core Service. This configuration is stored in the ITPDirWatch.ini file. This file must be located in the configuration folder of the CCM Core instance.

Each Service is described in a separate section in the ITPDirWatch.ini file.

```
[<service>:Service]
ServiceType=<type>
WatchedDirectory=<folder>
```

`<service>` Name of the CCM Core Service that the request is submitted to.

`<type>` Type of interface. Allowed values are Document or Control Data.

`<folder>` Folder monitored for jobs.

The CCM Core Directory Watch interface shuts down if the CCM Core instance does not provide the configured Service.

ServiceType parameter values

```
ServiceType=Document
```

In a `Document` Service, the file to be processed is placed directly into the watched folder. The CCM Core Directory Watch interface passes the name of this file directly to CCM Core. CCM Core scripts have access to this parameter as the $0 parameter or through the _jobid variable.

```
ServiceType= Control Data
```

In a `Control Data` service, a file with parameters is placed into the watched folder. The CCM Core Directory Watch interface reads the parameters from this file and passes them directly to CCM Core. CCM Core scripts can access these parameters through the $1 … $n parameters. The name of the file is passed as the $0 parameter or through the _jobid variable.

A parameter file has the following format.

```
nnnnn
parameter 1
parameter 2
parameter 3
```

The first line contains the number of parameters, padded on the left with zero ("0") characters to 5 positions. The following lines contain each one parameter, up to the number of parameters specified on the first line.

Example parameter file is as follows.

```
00002
First parameter
Second parameter
```

This format is compatible with the Control Data interface used by CCM Core.

Sample ITPDirWatch.ini file

The following examples define two watched folders.

```
[ConvertDocument:Service]
ServiceType=Control Data
WatchedDirectory = d:\DirWatch\ConvertDocument
```

The folder `d:\DirWatch\ConvertDocument` is monitored for parameter files. If such a file is placed in this folder, the file is read and its contents are submitted in a job to the `ConvertDocument` Service of CCM Core.

```
[Archive:Service]
ServiceType = Document
Watched Directory = d:\DirWatch\Archive
```

The folder `d:\DirWatch\Archive` is monitored for documents, which are then submitted to the `Archive` Service of CCM Core. The name of the file is passed to the Service.

Locate log files

The CCM Core Directory Watch interface uses the logging settings of the CCM Core installation it is connecting to.

Logs are written to the shared folder of CCM Core and stored in the ITPDirWatch [[server]] folder.

Uninstall the CCM Core Directory Watch interface

To uninstall the CCM Core Directory Watch Interface, proceed with the following steps.

1. Stop the CCM Core Directory Watch interface client from the Windows Service Manager.
2. Remove the Service from Windows.
   To do so, run the following command line from the installation folder.

   ```
   ITPDirWatch -remove <server>
   ```

   `<server>` is the name of the CCM Core instance where the CCM Core folder Watch interface must submit its requests to.
3. After the Service is removed from Windows, remove the installation folder.

MQSeries interface

The CCM Core MQSeries interface is implemented as a Windows Service that monitors an MQSeries queue and submits synchronous requests to CCM Core through that MQSeries Queue.

For information on MQSeries, see the MQSeries documentation available on the Internet.

Configure the MQSeries interface

Currently, no user interface is provided for the CCM Core MQSeries interface. You can start and/or stop the Windows Service through the Windows Service Manager.

The Service is configured to start automatically when the server is rebooted, but it must be started by hand when first created.

MQSeries interface functionality

The MQSeries interface is implemented as a Windows Service that reads requests from an MQSeries queue and forwards them to CCM Core over its TCP/IP interface. Once installed for a particular CCM Core instance, it automatically detects the TCP/IP interface. You only need to configure MQSeries specific settings.

The MQSeries interface shuts down automatically if it is unable to access the request queue because of a permanent error. If the MQSeries interface is unable to access the request queue because of a temporary error, the MQSeries interface attempt to reestablish its connection to the request queue. In this case, you may receive a "queue is unavailable" error. The interval between reconnection attempts starts at 5 seconds and increases up to 60 seconds when the request queue is unavailable for a longer amount of time.

You can install multiple MQSeries interface Services. Each Service can read from different MQSeries queues on different queue managers.

Concurrent requests

The MQSeries interface can have as many requests active at any point in time as the number of licensed CCM Document Processors. Provided that all CCM Document Processors are running, this ensures that as many requests as possible are handled concurrently. If fewer processors are running, some of the active MQSeries jobs are queued at CCM Core for processing. The latter has consequences for the MQSeries priority mechanism, so you should always start all available CCM Document Processors.

If the license is issued for an unlimited number of CCM Document Processors, the number of simultaneous requests is limited to one job per CPU core on the CCM Core. You can explicitly set the number of simultaneous requests to `n` by adding the setting `ScalingMaxDP=n` to the `[Configuration]` section of the DP.INI file.

Priorities

The CCM Core MQSeries interface supports priority queues. In order for CCM Core to get priority messages as soon as possible, you should have as many Document Processors running as the CCM license allows, or at least as many as the number of CPU cores on the system if you have a license for an unlimited number of Document Processors.

High priority messages cannot interrupt running CCM Core jobs, so there may be some delay in the MQSeries queue.

Each CCM Core MQSeries interface only monitors a single queue. If multiple CCM Core MQSeries interfaces are configured to monitor priority queues, the priorities are only observed between jobs from the same queue. Jobs from different priority queues are not ordered relative to each other.

Syncpoint control

CCM Core requests are handled under MQSeries syncpoint control. As a result, requests are handled as an atomic transaction for as far as MQSeries communication is concerned. Furthermore, if the interface cannot deliver a request to CCM Core, it is backed out so that it can be processed later.

In all but one case, if a job does get delivered to CCM Core and it fails, it will not be backed out and a failure will be reported. A job only gets backed out after being delivered to CCM Core when the MQSeries interface ends abnormally.

Note If CCM Core itself ends abnormally during execution of a job, the MQSeries interface reports an error.

If the MQSeries interface is stopped in a regular way, it does not accept new jobs and waits for all running submitted jobs queued by CCM Core to complete. If all CCM Document Processors are down, it waits indefinitely until either a least one CCM Document Processor gets started and handles the jobs or the CCM Core is shut down. In the latter case, all jobs are backed out.

MQSeries configuration

CCM Core Administrator does not support MQSeries configuration. The MQSeries interface has its own mq.ini file in the config folder of the corresponding CCM Core installation. All settings are stored in the `[Configuration]` section. You can edit this file with a text editor such as Notepad.

You can install multiple instances of the CCM Core MQSeries interface where each instance has its own configuration file. The settings in these configuration files are identical to those in the mq.ini file.

The following settings exist:

- **Queue**
  The MQSeries queue that the interface is listening to for requests.
  Type. Required setting.
  Value. The name of an MQSeries queue.
- **Queue manager**
  The MQSeries queue manager that manages all queues used by the MQSeries interface.
  Type. Optional setting.
  Value. The name of an MQSeries queue manager.
  Default value. The default queue manager. You can use the MQSeries administration tools to set the default queue manager.
- **Timeout**
  This setting specifies the timeout value in milliseconds that the interface uses to retrieve jobs from the specified MQSeries queue. If you set a timeout value to n, CCM Core may only detect special circumstances, such as a shutdown request, after n milliseconds. You should set an adequate timeout value.
  Type. Optional setting.
  Value. Any positive timeout value in milliseconds.
  Default value. 1000, which is 1 second.
- **Client**
  If this setting is set to Y, it forces the interface to link to the MQSeries client DLL at start-up. If it is set to N, it first tries to link to the server DLL, and if this fails, it tries to link to the client DLL.
  Type. Optional setting.

Value. Y or N.

Default value. N.

- **Connection**
  This setting specifies the TCP/IP connection name of a MQSeries server. This is either the hostname or the network address of the remote machine. This setting is only considered if the interface links to the MQSeries client DLL and a client connection channel name has been specified. If so, these settings specify to which remote MQSeries server this Service will connect to retrieve jobs. If the interface links to the MQSeries server DLL, it always connects to the local MQSeries server.

  Type. Optional setting. This setting controls the connection settings for the MQSeries interface of this CCM Core installation.

  Value. Any TCP/IP connection name.

  Default value. Empty.

- **Channel**
  This setting is only considered if the MQSeries interface links to the MQSeries client DLL and a connection name has been specified. If so, these settings specify to which remote MQSeries server the interface connects to retrieve jobs. If the interface links to the MQSeries server DLL, it always connects to the local MQSeries server.

  Type. Optional setting. This setting controls the connection settings for the MQSeries interface of this CCM Core installation.

  Value. Any client connection channel name.

  Default value. Empty.

- **Request version**
  You can use this setting to control the version level for reading messages from a queue. Level 2 is the default. You can specify version 1 here for backward compatibility.

  Type. Optional setting.

  Value. 1 or 2.

  Default value. 2.

MQSeries protocol

This section describes how you can submit requests to CCM Core over MQSeries message queues.

Submit jobs

For every job that you want to submit, you should at least perform the following steps:

1. Connect to the queue manager that manages the request queue.
2. Open the request queue for output.
3. Put one or more request messages on the request queue. Each message specifies a single request.
4. Close the request queue.
5. Disconnect from the queue manager.

This protocol allows you to submit several jobs over a single queue. The same queue can be used simultaneously by a number of clients. The steps above only describe the most basic form of interaction with CCM Core through MQSeries. Any client that has the MQSeries middleware installed can send requests to CCM Core, provided that it has sufficient access rights.

MQSeries queues and requests

An MQSeries queue has to be created on the server. This queue has to be configured so that clients can send requests over it.

Each MQSeries request consists of a single message containing a character string. This string encodes all request parameters and may use any code page for which MQSeries for NT supports conversion to Unicode (CCM Core uses Unicode internally). This usually means that the local code page of the sending application can be used. The only requirements are as follows:

- The CodedCharSetId attribute in the MQSeries message descriptor must be set to the used code page. If the local code page is used, the default MQSeries descriptor settings automatically indicate the correct code page.
- The Format attribute in the MQSeries message descriptor must be set to MQFMT_STRING to enable automatic conversion of the request string by MQSeries.

Request format

The MQSeries interface for CCM Core supports two different formats for job submission: a coded string or an XML message.

- **Coded string**

Each request is specified as a single string of characters that has the following format.

```
<s><JobIdentifier><s><Service><s>[<parameter1><s>...]
```

Where `<s>` is a single character that does not occur in the parameter data. Both the job identifier and the Service name may not be empty and the request must end with the separator. The parameters are optional. The separator can be any character except a NULL character ("\0"), because the MQSeries code page translation treats a NULL character as the end of the string.

Example

```
#ITP Job#runmdl#order#
```

In this example, the `#` character is used as the separator.

- **XML message**

Each request is put in an XML message. The content of the message is mostly free, however, the interface looks for an `<itp:job>` element containing the job submission information. This element has the following format.

```
<?xml version="1.0" ?>
<itp:job version="1" id="jobid">
    <itp:service>service name</itp:service>
    <itp:parameters>
        <itp:parameter>parameter 1 data</itp:parameter>
        <itp:parameter>parameter 2 data</itp:parameter>
        …
```

```
    </itp:parameters>
</itp:job>
```

`jobid` with the identification of the job.

`service name` with the name of the service to call.

`parameter n data` with the value of the nth parameter.

Note The XML message should start with `<?xml`. `<?xml` has to be the first five characters in the request string; otherwise, the request fails.

Request parameters

For MQSeries each request contains the following parameters. Unlike the TCP/IP protocols, there is no need to include the user id as a parameter as MQSeries already includes this information in each message.

- Job identifier

Every job that you submit must have a non-empty identifier. It is used for progress messages, but not internally for identification of request messages. This means that job identifiers do not necessarily have to be unique.

- Service

The second parameter is also required and contains the name of the Service that you want to run.

- Optional parameters

Every job that you submit can have zero or more additional parameters. In the CCM Core administration program, you define how these parameters are mapped to script parameters. When the mapping contains more parameters than those specified in a job, these parameters are assumed to be undefined.

Invalid requests

If a request uses a code page that MQSeries cannot convert to Unicode, or if it does not conform to the request format, CCM Core rejects the request. It logs the error and does one of the following:

- By default, CCM Core moves the request to the dead-letter queue of the queue manager that manages the request queue. If no dead-letter queue has been specified, the request is removed from the request queue.
- If the MQRO_DISCARD_MSG option is used in the Report field of the request message descriptor, the message is removed from the request queue.

In both cases, a negative action notification is sent to the client if it requested this. This notification contains the reason that the message was rejected.

Action notifications

Although the MQSeries interface is asynchronous, the client can receive action notifications back from CCM Core over a reply queue. In order to do this, the client has to set the ReplyToQMgr and ReplyToQ fields in the descriptor of the request message and set the type of the message to MQMT_REQUEST.

Action notifications have type MQMT_REPORT, format MQFMT_STRING, and one of the following feedback codes:

1. MQFB_PAN. A Positive Action Notification. These messages are sent after the request has been processed successfully, but only if the MQRO_PAN option was used in the Report field of the request message descriptor. If sent, no more report messages are sent for this particular request. Positive action notifications contain a string that indicates that the job has completed successfully.
2. MQFB_NAN. A Negative Action Notification. These messages are sent after a failure, but only if the MQRO_NAN option was used in the Report field of the request message descriptor. If sent, no more report messages are sent for this particular request. Negative action notifications contain a Unicode message string that explains what went wrong.

Apart from these CCM Core related reports, the options in the Report field of the request message descriptor may be used to control the following:

- Report messages generated by MQSeries such as confirmation of delivery. These messages are only delivered after the job is finished and committed.
- The contents of MsgId and CorrelId fields in progress messages. You can use these identifiers to relate report messages to the original request.

Note No negative action notifications are sent if CCM Core shuts down in a normal way. It waits for running jobs to complete. Jobs that have not yet begun executing are backed out to the request queue, so that they can be processed at another time.

Progress messages

The MQSeries interface only logs progress messages. Unlike CCM Core Services, it does not return them to the client, because they are incompatible with syncpoint control. With syncpoint control, progress messages can only be delivered to the client at the end of a job.

File transfers

Whenever CCM Core encounters a SendFile or ReceiveFile instruction, it either sends the contents of a file to the specified queue or retrieves a message from the specified queue and stores it in a file. For the description of these commands, see *Kofax Customer Communications Manager Core Scripting Language Developer's Guide*.

The instruction SendFile recognizes the special queue *REPLYQ. If this queue is used as destination in the script, the message is put on the queue that was specified in the ReplyQ/ReplyQMgr attributes of the job.

CCM Core does not request the client for particular file transfers. The client must ensure that it puts all data on one or more queues at the start of the job. Only after the job has ended and committed, the client will be able to read back result data.

If intermediate interaction with the client is required, you should split up a job in to several transactions.

Uninstall a Service

To uninstall a Service, proceed with the following steps.

1. Start the Windows Service Manager.

2. Stop the CCM Core MQSeries interface Service.
3. To remove the Service, execute the following command line from the installation folder.

```
ITPMQSeries -remove <server> [-cfg <config>]
```

`<server>` is the name of the CCM Core installation where the CCM Core MQSeries interface must submit its requests to. `<config>` is the alternative configuration file. The `-cfg` parameter is required if this CCM Core MQSeries interface is configured to use an alternative configuration file.

4. After all Services are removed, remove the installation folder.

XML metadata from template runs

Apart from the result document, CCM Core is able to generate an XML file containing metadata about the template run. You can use this data for post-processing.

You cannot generate an XML file with metadata for Document Pack Templates and for Document Templates with `OutputMode` set to "pack".

For more information, see the section on the ITPRun command in the *Kofax Customer Communications Manger Core Scripting Language Developer's Guide*.

XML metadata content

The XML file contains data concerning the template run itself. This data is added automatically to the file. It includes the following items:

- The template used
- Time the document is generated
- Keys and extras
- CCM version used to run the template
- The full name (including path) of the result document
- The user who requested creation of the document, and the user account that actually ran CCM

These items are added to the element `<ccm:meta>` in the document.

Additionally, information on the Forms encountered in the template is listed in the element `<itp:forms>`. This element contains the answers that the user provided on all questions posed by the template in Forms. Forms that were not executed, or questions not answered, are not included.

When one of the Form questions is the TEXTBLOCK type, the answer to this question is represented as a Text Block reference ID. The actual XML of the Text Block is represented using a top-level element `<ccm:user-textblock>`, which specifies the same reference ID as used in the Form.

Additionally, information on the Content Wizards encountered in the template is listed in the element `<ccm:wizard>`. This element contains the mandatory Sections and Text Blocks defined in a Content Wizard. It also contains the Sections and Text Blocks that were selected in the Content Wizard selection dialog during execution of the Content Wizard. Content Wizards that were not executed are not included.

Apart from the information stored in the XML file by CCM itself, the template is also able to add custom information. The Field Set _Document is available for this purpose. Values added to this Field Set are

listed in a special Document section in the metadata XML. The function `add_user_xml` that was used for the same purpose has been deprecated.

An example of a resulting XML file is provided below. Line breaks and layout are added for clarity and not necessarily part of an actual XML metadata file.

```
<?xml version="1.0" encoding="UTF-8"?>
<ccm:data xmlns:itp="http://www.aia-itp.com/4.2/formsData/">
  <ccm:meta>
    <ccm:date>2017-03-01T11:33:26</ccm:date>
    <ccm:result>result document</ccm:result>
    <ccm:model>template</ccm:model>
    <ccm:producer>user account running CCM Core (DP)</ccm:producer>
    <ccm:itp-version>5.1.1</ccm:itp-version>
    <ccm:user>user for who the template was executed</ccm:user>
    <ccm:job-id>job id</ccm:job-id>
    <ccm:keys>
      <ccm:key>first key</ccm:key>
      <ccm:key>second key</ccm:key>
    </ccm:keys>
    <ccm:extras>
      <ccm:extra>first extra</ccm:extra>
      <ccm:extra>second extra</ccm:extra>
    </ccm:extras>
    <ccm:Template>
      <ccm:label name="Wizard">A Content Wizard</ccm:label>
      <ccm:label name="Category">Letter</ccm:label>
    </ccm:Template>
    <ccm:Document>
      <ccm:label name="CustomerId">100042/ccm:label>
      <ccm:label name="OutputFormat">PDF/A</ccm:label>
    </ccm:Document>
  </ccm:meta>
  <ccm:forms>
    <ccm:form>
      <ccm:name>name</ccm:name>
      <ccm:group shown="Y">
        <ccm:name>group name</ccm:name>
      </ccm:group>
      <ccm:element>
        <ccm:key>title</ccm:key>
        <ccm:value>TRUE</ccm:value>
      </ccm:element>
      <ccm:element>
        <ccm:key>title</ccm:key>
        <ccm:value>314</ccm:value>
      </ccm:element>
      <ccm:element>
        <ccm:key>title</ccm:key>
        <ccm:value>some text</ccm:value>
      </ccm:element>
      <ccm:element>
        <ccm:key>Clause A001</ccm:key>
        <ccm:id>A001</ccm:id>
        <ccm:value>user:01c86735f934361b00000e70.1</ccm:value>
      </ccm:element>
      <ccm:recordset>
        <ccm:record>
          <ccm:element>
            <ccm:key>title</ccm:key>
            <ccm:value>some text</ccm:value>
          </ccm:element>
        </ccm:record>
```

```
      </ccm:recordset>
      <ccm:button>Ok</ccm:button>
    </ccm:form>
    <ccm:user-textblock id="user:01c86735f934361b00000e70.1">
      <ccm:content>&lt;textblock&gt;&lt;content&gt;&lt;tbk xsv="2.0.1"&gt;&lt;par
 font="header" indentation="0" hanging-indentation="false"&gt;&lt;txt
 bold="false" italic="false" underline="false" id=""&gt;&lt;![CDATA[Clause
 A001]]&gt;&lt;/txt&gt;&lt;/par&gt;&lt;par font="normal" indentation="0" hanging-
indentation="false"&gt;&lt;txt bold="false" italic="false" underline="false"
 id=""&gt;&lt;![CDATA[Any damages that are the consequence of terrorism
 are explicitly excluded and will not be covered by this policy.]]&gt;&lt;/
txt&gt;&lt;/par&gt;&lt;/tbk&gt;&lt;/content&gt;&lt;fieldset&gt;Customer&lt;/
fieldset&gt;&lt;fieldset&gt;Policy&lt;/fieldset&gt;&lt;/textblock&gt;</ccm:content>
    </ccm:user-textblock>
  </ccm:forms>
  <ccm:wizards>
    <ccm:wizard>
      <ccm:name>Content Wizard name</ccm:name>
      <ccm:section>
        <ccm:name>Section name</ccm:name>
        <ccm:section>
          <ccm:name>Section name</ccm:name>
          <ccm:section>
            <ccm:name>Section name</ccm:name>
            <ccm:textblock>
              <ccm:name>Text Block name</ccm:name>
            </ccm:textblock>
          </ccm:section>
          <ccm:textblock>
            <ccm:name>Text Block name</ccm:name>
          </ccm:textblock>
        </ccm:section>
        <ccm:textblock>
          <ccm:name>Text Block name</ccm:name>
        </ccm:textblock>
      </ccm:section>
    </ccm:wizard>
  </ccm:wizards>
</ccm:data>
```

The schema of the XML generated is provided in the file history.xsd in the Manuals directory of the CCM Core installation.

Produce XML metadata

By default, CCM Core does not produce an XML metadata file for a template run. To generate an XML metadata file, specify a file name for the parameter `MetaData` of the script command `ITPRun`. Likewise, the Service `RunMdl` has a parameter `MetaData` with the same meaning. When using CCM ComposerUI Server, the XML metadata file is always generated. The path to the generated XML metadata file is passed to the exit points `ProcessResult` and `ModelRunCompleted`. For more information about these exit points, see the section "CCM Core: OnLine exit points" in the legacy *Kofax Customer Communications Manager ComposerUI for ASP.NET Developer's Guide*.

Identify Forms and questions

The IDs of Forms and questions are included in the metadata XML. The CCM Repository Form Editor enables you to change both Form and question IDs. It is also possible to define Form and Question IDs in

the FORM statement of the Template scripting language. For more information, see the *Kofax Customer Communications Manager Template Scripting Language Developer's Guide*.

A business application that integrates CCM can identify Forms and questions that have been filled with answers during a template run in the metadata XML by their IDs. The Form Editor generates unique random IDs initially when saving a new Form. It also does this when saving existing Forms that have no IDs yet. Manually entered IDs are easier to locate in the metadata XML and allow for easier integration. For more information, see the next section.

Form and question IDs

The Form ID offers a unique reference to a Form. You can use question IDs in a similar way to give unique references to questions. Although the initially generated Form and question IDs are unique, CCM Repository does not check this after manually changing IDs.

These IDs are related to the Suspend and Resume functionality, because modifications to questions and Forms have implications for the persisted data of suspended sessions. Form IDs can also be used to identify a Form in the Metadata, similar to question IDs for questions.

Until a Form ID is changed, questions in that Form are matched based on their ID. Changes to templates and Forms while a template run is suspended does not necessarily require answering previously answered Forms. Previously entered answers are restored when a template run is resumed by mapping answers to questions with the same ID. For more information on Suspend and Resume functions, see the "Suspend and Resume" section in the legacy *Kofax Customer Communications Manager ComposerUI for ASP.NET Developer's Guide*. For more information on when questions and Forms must be answered again, see the "Changing Forms during suspension" section in the legacy *Kofax Customer Communications Manager ComposerUI for ASP.NET Developer's Guide*.

Automatically generated and manually entered IDs are not automatically changed if a question in a Form is changed. To ensure that an already answered Form is presented again after a resume, the Form or question ID has to be changed manually in the QForm Editor.

CCM Core Text Block XML format

The CCM Core Text Block XML format defines custom Text Blocks. These Text Blocks are intended to be authored by external applications. You can retrieve them from a database and insert into documents using the `read_text_block_from_file` and `import_text_block_base64` functions.

XML reference

CCM Core Text Block XML objects are only allowed to use a subset of the XML specification:

- The XML must be encoded in either UTF-8, UTF-16 or iso-8859-1. Other encodings are not supported.
- Namespace prefixes are not allowed. A default xmlns namespace declaration is ignored.
- Boolean values must be specified as either true or false. Other xsd:boolean values (0, 1, True, False) are not supported.

CCM Core Text Block XML elements

Header

```
<tbk xsv='2.2.0'>
    <par ...> ... </par>
    <lst ...> ... </lst>
    <tbl ...> ... </tbl>
    ...
</tbk>
```

The required xsv attribute specifies the version of the CCM Core Text Block XML format. This must be version 2.2.0.

The content of the Text Block is a sequence of one or more paragraphs, lists, and tables.

Note CCM Repository and CCM Core can expose Text Block XML files where the xsv attribute refers to older versions of the Text Block XML format. The 2.0.x and 2.1.x versions are a subset of the 2.2.0 format, and you can treat these versions as if they were saved as version 2.2.0. Version 1 is completely incompatible and should be rejected.

Paragraphs

The paragraph element represents a logical paragraph and the attributes that apply to this paragraph. The attributes are mapped to styles in Microsoft Word.

```
<par font='normal' indentation='0' hanging-indentation='false'>
    <txt ...> ... </txt>
    <chr .../>
    <fld ...> ... </fld>
    ...
</par>
```

All of the following attributes are required:

- `font` defines the style in which the paragraph is represented. Allowed values are normal for regular text and header for highlighted text.
- `indentation` defines the indentation level of the paragraph. Every level shifts the paragraph one tab stop to the right.
- `hanging-indentation` indicates whether or not the first line of the paragraph should start one tab stop to the left of the other lines. Allowed values are `true` or `false`.

The content of the paragraph is a sequence of zero or more text runs, special characters and CCM Core data fields.

All elements in the paragraph require the boolean attributes `bold`, `italic`, and `underline`. These attributes control the font attributes of the rendered text and expect the values true and false.

The paragraph attributes only provide context hints to the intended layout. The style sheet defines the actual visualization of the styles.

Text elements

A text element represents a simple run of text.

Note All white space between the <txt> and </txt> tags is significant.

```
<txt bold='false'
     italic='true'
```

```
     underline='false'><![CDATA[This is some formatted text.]]></txt>
```

Character elements

A character element represents a control character.

```
<chr type='NBSP' bold='false' italic='true' underline='true'/>
```

All attributes are required.

The `type` attribute expresses the type of control character. Supported values are the following:

- TAB is a Tab character.
- LBR is a soft line break (Shift+Enter in Microsoft Word).
- NBSP is a non-breakable space.
- NBHH is a non-breakable hyphen.

Field elements

The field element is used to insert a value from the Data Backbone into the result document.

The specific Field Set that provides the value is resolved at run time when the Content Wizard determines in which context the Text Block is used.

```
<fld bold='false' italic='true' underline='false' set='Customer'>Name</fld>
```

All attributes are required.

The `set` attribute defines the Field Set and the content of the tag defines the field that will be retrieved. Both references are case-sensitive and must match the definition in the CCM Repository exactly.

List elements

The list element represents a list of paragraphs and lists where every element is shown as an item.

```
<lst style='ordered'>
    <par ...> ... </par>
    <lst ...> ... </lst>
    ...
</lst>
```

All attributes are required.

The `style` attribute defines one of the following types of list that will be rendered:

- `ordered` defines a numbered list.
- `unordered` defines a bulleted list.

The `style` attributes only provide context hints to the intended layout. The style sheet defines the actual visualization of the list.

Lists are allowed to be nested. Every nested list automatically increases the nesting level by one.

Table elements

Note Table elements are currently only supported in the context of Microsoft Word DOCX projects.

The table element represents a simple table in the Text Block. This table is restricted to a fixed number of rows and columns.

```
<tbl>
    <prp ... > ... </prp>
    <row>
        <col>
        </col>
        ...
    </row>
    ...
</tbl>
```

Each cell in the table may contain a sequence of paragraphs and lists. Tables cannot be nested.

Table properties

The property element defines the format and layout of the table.

```
<prp nrows='10'
     ncols='3'
     header-row='false' footer-row='true'
     first-column='false' last-column='false'
     autosize='false'>
    <col width='1440'/><!-- one inch -->
    <col width='567'/><!-- one centimeter -->
    ...
</prp>
```

The following attributes are required:

- `nrows` defines the number of rows in the table.
- `ncols` defines the number of columns in every row.

The following attributes are optional:

- `header-row`
- `footer-row`
- `first-column`
- `last-column`

  These four attributes determine whether special formatting rules from the Microsoft Word style must be applied to the indicated row or column. If one of these values is set to true, the exception is applied to that row or column.
- `autosize` instructs Microsoft Word to enable or disable the autosizing of the table. If `autosize` is set to true, Microsoft Word automatically sizes the table based on the content of the cells and any `<col .../>` column definitions are ignored.

The `<col .../>` elements define the width of the columns if the autosize attribute is not present or set to false. If the table is manually sized there must be a `<col width='...' />` element for every column. The width is specified in units of 1440th of an inch (twips).

Rows

The row element defines a row of cells in the table. The number of `<row>` elements must exactly match the `nrows` attribute of the table.

Cells

The cell element defines a cell in a row. The number of `<col>` elements in every row must exactly match the `ncols` attribute of the table.

```
<col>
    <par ...> ... </par>
    <lst ...> ... </lst>
    ...
</col>
```

A cell contains a sequence of one or more paragraphs and lists. It is not possible put a table within another table cell.

Chapter 13

Information for system administrators

This chapter provides information necessary for system administrators.

Assign the Log on as a Service right to a user

The user profile that runs the CCM Core NT Services needs to have the rights to log on as a Service. To assign the "Log on as a Service" right to a user profile, use the Local or Domain Security settings in Windows.

The administrator can assign this privilege directly to a user profile or to a group where it transfers to the members of the group.

Ensure that the assigned right does not conflict with the "Deny login as a service" setting.

Manage the configuration file

CCM Core creates an ITP configuration file. We recommend that you do not make any changes to the file. However, if required, you can modify the file.

CCM Core uses a default configuration file itp.cfg to run and create templates. This configuration file is located in the Config folder for each CCM Core instance.

This section describes the most important CCM Core configuration settings. If you need to change any setting, proceed to the steps below.

1. Navigate to `<deploy root>\CCM\Work\<version>\Instance_<number>\core\Config` and open the itp.cfg text file.
2. Add a setting and a value to the following setting.

```
ITPLAZYPOSTINC=Y
```

Note A few defaults that CCM Core uses cannot be overwritten. CCM Core always use its own defaults for these settings. Changing these settings breaks CCM Core. Do **not** change the following settings:

- `ITPDIR`
- `ITPPROGDIR`
- `ITPDIDDIR`
- `ITPTMPDIR`
- `ITPERRORDIR`
- `ITPDMINIFILE`
- `ITPDATAMAN_SERVER_NAME`
- `ITPMODDOCDIR`
- `ITPMODDIR`
- `ITPRESDIR`

ITPVALIDATEFIELDSET=N

You can configure CCM Core to validate at run time that all Fields in a Field Set variable are filled before using this Field Set. If a Field is not filled, the process fails. This validation is disabled by default.

ITPLANGMODDOC=ENG

With this setting you can specify the language used for the Template Scripting language keywords and standard functions. You can choose between English (ENG), German (DEU), and Dutch (NLD).

ITPINCLUDEPATH

By default, CCM Core searches pre-includes in the templates/Includes folder in the ITPWORK folder. You can change this default.

ITPLAZYPOSTINC=Y and **ITPPOSTINC=Y**

By default, post-includes are enabled in lazy mode. This mode switches post-includes on when the `@(inc(…))` method is used. Under most circumstances, these values produce the desired result. If the `__INC` expressions in the result document are not produced using the `@(inc…))` function, the configuration file must be adapted to include the documents; ITPLAZYPOSTINC must be set to N.

Note ITPPOSTINC=Y is necessary for ITPLAZYPOSTINC to function. If you do not use post-includes in your Template script, you can set ITPPOSTINC to N.

ITPEURO

If this setting is set to Y, the `amount_in_words` function uses the euro as currency. If this setting is set to N or not present, the `amount_in_words` function uses a local currency. This setting can be overridden by the use of the `euro` function. Default is N.

ITPALLOWMISSINGINC

Controls whether a missing `__INC` document causes an error when translating a template. If this setting is set to Y, CCM removes the `__INC()` statement and continues translating. If this setting is set to N,

CCM reports an error and stops the translation. This behavior is in compliance with the behavior of Office Vision. Default is N.

ITPALLOWMISSINGPOSTINC

Controls whether a missing `__INC` document causes an error when executing the post-include on a result document. If this setting is set to Y, CCM silently removes the `__INC()` statement and continues to operate. If this setting is set to N, CCM reports an error. This behavior is in compliance with the behavior of Office Vision. Default is N.

ITPUNICODEEURO

Enables the functionality to translate single byte euro character to Unicode euro character. Default is N.

ITPCURRENCYEURO

Enables the functionality to translate the International Currency Symbol to Unicode Euro characters.

Default is N.

ITPDMRESTARTTIMEOUT

This is an ITP/(OnLine)Server-specific setting.

CCM Core restarts the Data Manager process if the credentials or environment change. By default, CCM Core waits 1000 msec until the Data Manager terminates and reports the EVL7102 message if after the timeout Data Manager is still busy. This setting allows you to set the timeout before the restart of the Data Manager. Default is 1000 msec.

ITPREPRETRY

Controls how long CCM Core attempts to connect to an unavailable Editorial Repository.

By default, CCM Core attempts for approximately 150 seconds to reconnect if the Editorial Repository is shut down or when an existing connection is disrupted. This setting has the format `ITPREPRETRY=ia,id,na,nc`.

`ia` is a number of connection attempts for interactive template runs. Set this value to 0 to disable reconnection attempts.

`id` is a delay (in seconds) between connection attempts for interactive template runs.

`na` is a number of connection attempts for non-interactive model runs. Set this value to 0 to disable reconnection attempts.

`nd` is a delay (in seconds) between connection attempts for non-interactive model runs.

CCM Core adds an additional small random delay between connection attempts to desynchronize multiple CCM Document Processors. The built-in connection retry feature in Windows also introduces additional delays, which depend on Windows TCP/IP configuration parameters and networking performance.

You should combine this setting with the job timeout feature of CCM Core. The default value is 15,10,15,10.

An example is provided here.

```
ITPRETRY=0,0,15,10
```

The preceding example setting disables reconnection attempts for interactive model runs and keeps the default for non-interactive model runs. This setting is available from CCM Core 4.4.

ITPDISABLEPROJECTCONSTRAINT

This setting controls whether a CCM Core model or session can use dynamic content from a single project or from multiple projects.

As of CCM Core 4.4, dynamic content is restricted to a single project by default. Default is N.

This setting is available from CCM Core 4.4.