Custom Commerce Adapter Developer Manual

CoreMedia Content Cloud – v13

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

January 28, 2026 (Release 2512.0)

List of Figures

List of Tables

1. Preface

This manual describes how to integrate a custom Commerce System with *CoreMedia Content Cloud* using the CoreMedia Commerce Hub architecture.

1.1 Audience

This manual is intended for architects and developers who want to integrate a custom commerce system with the CoreMedia System. The reader should be familiar with *CoreMedia CMS*, *Spring*, *Maven* and the commerce system to connect with.

1.2 Typographic Conventions

CoreMedia uses different fonts and types in order to label different elements.
The following table lists typographic conventions for this documentation:

Element	Typographic format	Example
Source code Command line entries Parameter and values Class and method names Packages and modules	Courier new	`cm systeminfo start`
Menu names and entries	Bold, linked with |	Open the menu entry **Format|Normal**
Field names CoreMedia Components Applications	Italic	Enter in the field *Heading* The *CoreMedia Component* Use *Chef*
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is *not* saved
Buttons	Bold, with square brackets	Click on the **[OK]** button
Code lines in code examples which continue in the next line	\	`cm systeminfo \` `-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

Pictograph	Description
	Tip: This denotes a best practice or a recommendation.
	Warning: Please pay special attention to the text.
	Danger: The violation of these rules causes severe damage.

Table 1.2. Pictographs

1.3 Changelog

The following table lists all changes that have been applied to the manual since its first publication.

Section	Version	Description

Table 1.3. Changes

2. Overview

The *CoreMedia Commerce Hub* controls communication of CoreMedia apps with commerce systems by defining a vendor agnostic API covering the most common eCommerce features and providing a default client-server implementation of this API.

The client part of the *CoreMedia Commerce Hub* is named *generic client*. The server part is named *adapter service*. Adapter services are vendor specific extensions of the *base adapter* which itself defines the *Commerce Hub* API and serves as a runtime environment controlling the communication between generic client and commerce system.

- Section 2.1, "Commerce Hub Architecture" [7] describes the *Commerce Hub* architecture in more detail
- Section 2.2, "Commerce Hub API" [9] describes the APIs provided by the *Commerce Hub* and the request flow between *generic client*, *adapter service* and commerce system

2.1 Commerce Hub Architecture

Commerce Hub is the name for the CoreMedia concept which allows integrating different eCommerce systems against a stable API.

Figure 2.1, " Architectural overview of the Commerce Hub " [7] gives a rough overview of the architecture.

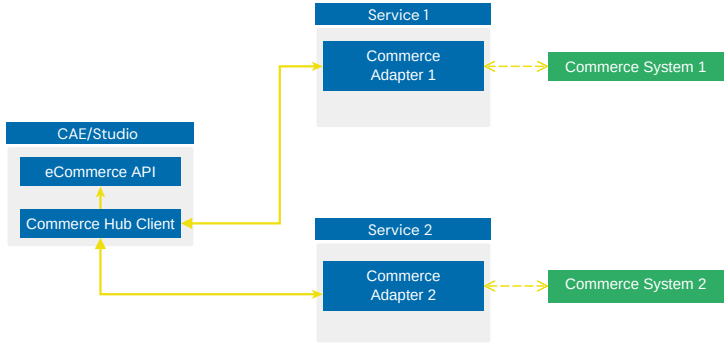

Figure 2.1. Architectural overview of the Commerce Hub

All CoreMedia components (CAE, Studio) that need access to the commerce system include a generic Commerce Hub Client. The client implements the CoreMedia eCommerce API. Therefore, you have a single, manufacturer independent API on CoreMedia side, for access to the commerce system.

The commerce system specific part exists in a service with the commerce system specific connector. The connector uses the API of the commerce system (often REST) to get the commerce data. In contrast, the generic Commerce Hub client and the Commerce Connector use gRPC for communication (see https://grpc.io/) for details.

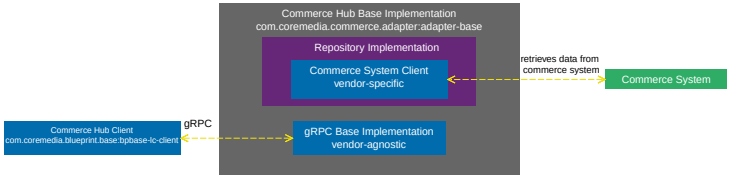

Figure 2.2. More detailed architecture view

Figure 2.2, " More detailed architecture view " [7] shows the architecture in more detail. At the Commerce Hub Client, you only have to configure the URL of the service and some other options, while at the Commerce System Client, you have to configure the commerce system endpoints, cache sizes and some more features.

2.2 Commerce Hub API

The *Commerce Hub* API consists of a gRPC API used by the *generic client*, and a Java API which consists of the Entities API as a wrapper around the gRPC messages, and a Java Feature API, used by the specific *adapter services*.

The gRPC API

The gRPC API defines the messages and services used for the gRPC communication between *generic client* and *adapter service*. It is not necessary to access this API from any custom code. Access should be encapsulated, using the provided Java APIs, described below. In case the existing feature set does not fulfill all needs for a custom commerce integration, the gRPC API may be extended. CoreMedia provides two sample modules, showing a gRPC API extension in the *Commerce Adapter Mock*. Please have a look at the Section 3.2, "CoreMedia Commerce Adapter Mock" [14].

> **NOTE**
>
> By Default the *base adapter* exposes the gRPC `ServerReflection` service. It is used by the *CoreMedia Commerce Hub Client* to obtain available features.

The Java API

The Java API consists of two parts. The first part defines Java Entities as a wrapper around gRPC. It is used by the *generic client* and the server in the *base adapter*.

The second part is meant for server side only. It defines the Java Interfaces, called Repositories, the *adapter services* may implement for any needed feature. This API should be used as an entry point for commerce adapter development.

Request flow

The request flow, using the above described APIs, starting from the generic client is as follows. Please have a look at Figure 2.2, " More detailed architecture view " [7] first.

1. The generic client sends a gRPC request to the vendor agnostic *base adapter*. The Entities API is used to convert the Java entity to the corresponding gRPC message.

2. The gRPC service implementation in the *base adapter* receives the gRPC request and invokes the corresponding repository methods.

   While the API definition of the repositories is placed in the *base adapter*, the implementation which is called here is part of a specific commerce adapter.

   The commerce adapter uses its vendor specific implementation to obtain the requested data from the commerce system. The data is then mapped to a CoreMedia commerce entity as defined by the base adapter.

   Finally, the service implementation in the *base adapter* converts the given entity back to a gRPC response and sends it back to the *generic client*.
3. The *generic client* receives the gRPC response and uses the Entities API to obtain and process the requested entity.

3. Integrating a Custom Commerce System

As described in Section 2.1, "Commerce Hub Architecture" [7] the CoreMedia *Commerce Hub* consists of three main parts: a *base adapter* implementation defining the API and handling the request flow, a commerce system agnostic *generic client* implementation and a commerce system specific *adapter service*. In order to integrate a custom commerce system into the CoreMedia system, an *adapter service* for that system has to be implemented, using the *base adapter*.

3.1 Developing a Custom Commerce Adapter

An *adapter service* is the link between the generic CoreMedia eCommerce client and the specific commerce system. The following chapter shows how to get started, implementing a custom *adapter service*.

As described in Section 2.2, "Commerce Hub API" [9], the CoreMedia eCommerce API is defined in the *base adapter*. It offers a rich set of commerce features which can be used by implementing the corresponding repository interfaces. In order to implement the API, the `com.coremedia.commerce.adapter:adapter-base` and `com.coremedia.commerce.adapter:adapter-api` dependencies have to be added to your project.

The `adapter-base` dependency includes the repository interfaces for all available features. They can be found in the `com.coremedia.commerce.adapter.base.repositories` package.

The `adapter-api` dependency includes the most common eCommerce entities like catalogs, categories and products. They can be found in the `com.coremedia.commerce.adapter.api.entities` package.

The minimum feature set

As mentioned before, the CoreMedia eCommerce API offers a superset of commerce features which are all implemented on the client side. The *adapter service* (server side), should of course only implement the repositories for the needed features. The client requires access to catalogs and categories for building the commerce connection. Also, products are considered a mandatory feature.

The following features are required for establishing a commerce connection:

- **Catalogs:** Implement `CatalogRepository`
- **Categories:** Implement `CategoryRepository`
- **Products:** Implement `ProductRepository`

Custom searches for arbitrary commerce entities can be implemented via `com.coremedia.commerce.adapter.base.repositories.ProductSearchSupport#search` by implementing `ProductRepository` on the adapter side.

CAUTION

Base adapter releases up to 1.5 also require a `PriceRepository` implementation

3.2 CoreMedia Commerce Adapter Mock

CoreMedia provides a dedicated mock *adapter service* implementation for customers and partners as a GitHub repository. It is meant as an example of how to implement a custom *adapter service* and provides a fully functioning Spring Boot service. The service can be build via Maven and runs either as a plain Spring Boot app or inside a Docker container.

In case the used technologies are applicable, CoreMedia recommends to use this project as a starter for building a custom *adapter service*.

Structure

The workspace can be found at https://github.com/coremedia-contributions/commerce-adapter-mock and includes a set of modules.

To get started developing a custom *adapter service* the following modules are needed.

- `adapter-mock-lib` This module holds a sample implementation of a custom *adapter service* which is used by the Spring Boot app from the `adapter-mock-app` module.

  The repository implementations in this module should be adapted and serves as starting point for developing a custom commerce adapter.
- `adapter-mock-app` This module holds the ready to run Spring Boot app for the Mock Commerce adapter. The implementation sources are separated in the `adapter-mock-lib` module.

The following modules contain convenience configuration, tooling and sample code for extending the commerce API by custom gRPC services.

- `adapter-mock` This module holds the Docker setup. Using the dockerfile-maven-plugin it can be used to build a Docker image for the mock *adapter service*.
- `adapter-mock-custom` This module includes service customization samples for the mock *adapter service*. It is referenced as dependency in the `pom.xml` file of the `adapter-mock-app`.
- `adapter-mock-custom-grpc` This module holds a custom gRPC API definition which is then used by the services in the `adapter-mock-custom` module

- `workspace-config` This directory holds additional workspace configuration like the *IntelliJ IDEA* run configuration for the Spring Boot app.

Using the Commerce Adapter Mock

The CoreMedia Commerce Adapter Mock is not only a sample, showing how to implement a custom commerce adapter, but can also be used as a starter project.

If you decide to use the project as a starter, just checkout the latest revision from GitHub and rename and reorganize the modules and repositories as it suits your project.

The entry point for developing a custom commerce adapter is the `adapter-mock-lib` module. It contains the `repositories` package, holding repository implementations for a broad feature set, including the mandatory implementations for `CatalogRepository`, `CategoryRepository` and `ProductRepository`.

Beside the `repositories` package you will find some more packages, containing samples for retrieving data, configuration or dealing with preview tokens. These packages are not needed for setting up a custom *adapter service*.

NOTE

To get a better idea of how to develop an *adapter service* you can also have a look at the CoreMedia *adapter services* for Salesforce, SAP Commerce or HCL Commerce.

Useful features like caching, using the CoreMedia `Cache` or Monitoring with services like Micrometer should be considered crucial for your custom commerce adapter as well.

The latest version of the sources can be found on https://repository.coremedia.com. Usages of the CoreMedia Cache can be found in the `com.coremedia.commerce.adapter.sfcc.cache` package.

3.3 Integrating a Custom Commerce Adapter

In order to use the custom *adapter service* with the CoreMedia system. A minimum set of configuration and setup is needed.

Configuring the *adapter service* Endpoint

To enable the *generic client* to connect to a custom *adapter service* an endpoint for that service has to be added on the client side. This is done by configuring a gRPC Spring channel . Use the default channel configuration for properties that apply to all services. The specific configurations are done per named channel, where the name is an ops-friendly string of your choice, such as `fooService`. This name is used to identify the service in the site's `commerce` settings struct's `endpointName` property.

Please also refer to the Javadoc of the method `com.coremedia.blueprint.base.livecontext.client.settings.CommerceSettings#getEndpointName()`

The Vendor Name

To integrate an *adapter service* with the CoreMedia system, a vendor name for the commerce system has to be configured via `metadata.vendor` in the *adapter service*. This name is used as a prefix for all commerce IDs by the coreMedia system and should therefore never be changed.

The Commerce Settings

The *CoreMedia Commerce Hub generic client* expects a commerce system to have at least one catalog and a root category. If this is the case, no further configuration is needed to set up the commerce connection. If the commerce system provides multiple catalogs or stores, both may be configured via the site's commerce settings content item.

CAUTION

The commerce connection is an instance of the `GenericCommerceConnection` managed by the *generic client*. It is valid only if the *generic client* is able to create an instance of the `GenericStoreContext` while communicating with the custom *adapter service*.

After the commerce connection for the *adapter service* is set up correctly, the catalog along with its categories and products can be displayed in the *CoreMedia Studio* library.

NOTE

If the *CAE* is used for augmenting the commerce storefront the `LinkRepository` needs to be implemented.

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine (CAE)	The *Content Application Engine* (*CAE*) is a framework for developing content applications with *CoreMedia CMS*. While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations. The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user. It may contain any of the following modules: • *CoreMedia Master Live Server* • *CoreMedia Replication Live Server* • *CoreMedia Content Application Engine* • *CoreMedia Search Engine* • *Elastic Social* • *CoreMedia Native Personalization*

Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Native Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by

	the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.
CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.

gRPC	gRPC is an open source high performance Remote Procedure Call (RPC) framework.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the ActionScript/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.

Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
OCI (Open Container Initiative)	The Open Container Initiative (OCI) is a Linux Foundation project that defines open industry standards for container formats and runtimes. OCI specifications ensure compatibility and interoperability between container tools, engines, and orchestration platforms like Docker and Kubernetes.
ORAS (OCI Registry As Storage)	ORAS (OCI Registry As Storage) is a tool and specification that extends OCI registries to store and distribute OCI artifacts beyond container images. It provides a standardized way for developers to push and pull arbitrary content types to and from container registries, enabling these registries to function as general artifact stores.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.

ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes Application	The *User Changes Application* is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.

Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.