ptc

arbortext®

Programmer's Guide to Arbortext Publishing Engine

8.1.2.0

Contents

About This Guide

Arbortext Publishing Engine is a server program that performs XML document manipulation and publishing for requests submitted by web clients, including Arbortext Editor. This documentation is intended for programmers creating custom applications to run on the Arbortext PE server. Among its capabilities are:

- Reading SGML and XML documents into memory
- Importing Microsoft Word, Adobe FrameMaker, RTF, HTML, PDF, and text documents into memory as XML documents
- Applying XSL stylesheets to XML documents, transforming them into other XML documents
- Publishing XML documents to a variety of output formats, including PostScript, PDF, HTML, HTML Help, Web, and RTF.
- Retrieving documents from and saving documents to content management repositories.

Arbortext Publishing Engine provides publishing services to Arbortext Editor users. Arbortext Editor can be configured to send documents to the Arbortext Publishing Engine, and these documents can be imported from or published to non-XML input or output formats. The publishing processing is performed on the Arbortext PE server, which returns the result to the Arbortext Editor client.

The basic capabilities of Arbortext Publishing Engine can be extended by custom applications written in Arbortext Command Language (ACL), Java, JavaScript, and VBScript. A programmer would use the documentation in this manual in combination with the documentation found in the *Programmer's Reference* and the *Arbortext Command Language Reference*.

Prerequisite Knowledge

You should have substantial experience as a programmer. This document assumes advanced skill using programming languages such as Java, JavaScript, VBScript, and Arbortext Command Language (ACL). You should be familiar with servlet containers, web servers, and HTTP protocols. You should also be familiar with the implementation at your site and with standard system administration tasks.

In a typical implementation, a client program or web browser sends an HTTP request to a web server. The web server interprets the URL and passes it to the servlet container. The servlet container knows how to call Arbortext PE Request Manager from its own configuration file, and it constructs and passes a request object and a response object to the Arbortext PE Request Manager. From the request object, the Arbortext PE Request Manager determines the client who sent it, what the request is for, what work to perform, and what data to return in the response object to the servlet container. In turn, the servlet container returns the response to the web server, which then returns it to the client making the request.

Technical Support

To contact PTC Technical Support, use the Contact Support and Customer Support Guide links on support.ptc.com.

The PTC Support pages also provide a search facility for you to browse for knowledge articles, best practices, and other information.

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC Technical Support or Customer Care Departments using the contact instructions found in your Customer Support Guide.

Documentation for PTC Products

You can access PTC product documentation using the following resources:

- Online Help

  Click **Help** from the user interface for online help available for the product.

- Reference Documentation

  PDFs of reference information are available from the Product Documentation area of support.ptc.com.

  Select the Arbortext tab to access the Arbortext Reference Documentation link.

- Help Center

  Help Centers for the most recent product releases are available from the Product Documentation area of support.ptc.com.

Select the Arbortext tab to access the Help Centers link.

You must have a Service Contract Number (SCN) before you can access the Arbortext Reference Documentation or Help Centers links. If you do not have an SCN, contact PTC Technical Support or Customer Care Departments using the contact instructions found in your Customer Support Guide.

Global Services

PTC Global Services delivers the highest quality, most efficient and most comprehensive deployments of the PTC Product Development System including Creo, Windchill, Arbortext, and PTC Mathcad. PTC's Implementation and Expansion solutions integrate the process consulting, technology implementation, education and value management activities customers need to be successful. Customers are led through Solution Design, Solution Development and Solution Deployment phases with the continuous driving objective of maximizing value from their investment.

Contact your PTC sales representative for more information on Global Services.

Comments

PTC welcomes your suggestions and comments on our documentation. You can submit your feedback to the following email address:

arbortext-documentation@ptc.com

Please include the following information in your email:

- Name
- Company
- Product
- Product Release
- Document or Online Help Topic Title
- Level of Expertise in the Product (Beginning, Intermediate, Advanced)
- Comments (including page numbers where applicable)

Documentation Conventions

This guide uses the following notational conventions:

- **Bold text** represents exact text that appears in the program's user interface. This includes items such as button text, menu selections, and dialog box elements. For example,

  Click **OK** to begin the operation.

- A right arrow represents successive menu selections. For example,

  Choose **File ▶Print** to print the document.

- `Monospaced text` represents code, command names, file paths, or other text that you would type exactly as described. For example,

  At the command line, type **`version`** to display version information.

- ***`Italicized monospaced text`*** represents variable text that you would type. For example,

  `installation-dir`**`\custom\scripts\`**

- *Italicized text* represents a reference to other published material. For example,

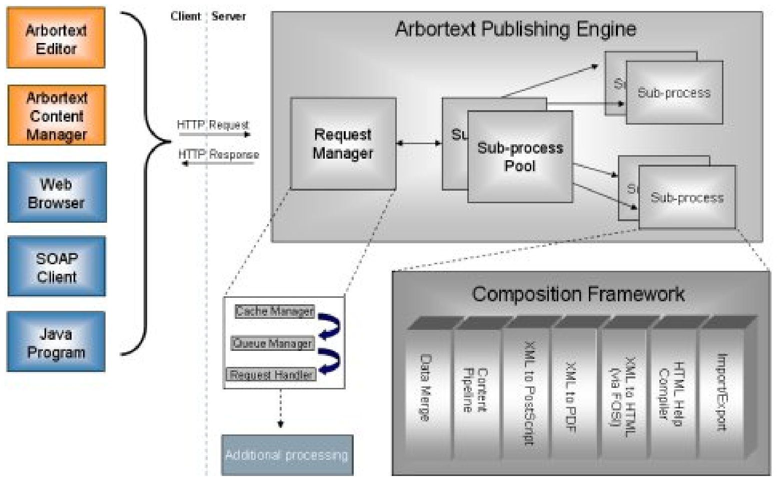
  If you are new to the product, refer to the *Getting Started Guide* for basic interface information.

I

Arbortext Publishing Engine and its Components

1
Technical Overview of Arbortext Publishing Engine

Arbortext Publishing Engine is implemented as a Java Servlet that complies with the *Java Servlet Specification, v2.4* from Sun Microsystems. The servlet is called the Arbortext PE Request Manager. This implementation makes Arbortext Publishing Engine portable across platforms and operating systems. It runs in the Tomcat servlet container (which can operate alone or in conjunction with the Microsoft Internet Information Server).



Distributed Computing and the Client/ Server Model

Arbortext Publishing Engine and its clients follow the client/server model of distributed computing. The term “distributed computing” refers to two or more programs or machines working together to solve a problem or deliver service. The client/server model is a conceptual framework for understanding distributed computing. In this model, every interaction between two computers or programs is viewed as a transaction consisting of a request and a response.

A transaction begins when a program running on one machine (the “client”) sends a request for service to a program running on another machine (the “server”). When the server receives the request from the client, the server interprets the request, performs some kind of work, and returns a response to the client. The client usually waits for the response from the server. When the client receives the response, the transaction is considered to be over. The client continues processing, and the server sleeps, awaiting a request from another client.

The term “client” can refer to either the program that transmits the request to the server (the “client program”) or to the machine on which the client program runs (the “client machine”). The term “server” can refer to either the program that receives the client request (the “server program”) or to the machine on which the server program runs (the “server machine”).

Sometimes the client and server machine are the same, and a client process transmits a request to a server process running on the same machine. To fulfill a client request, a server process may need to send a request to another server, in which case a single program can be considered both a server and a client.

A client program and a server program must share common expectations about how to exchange data in the request and response transaction. The set of rules governing how the client will construct and send a request and how the server will return a response to the client is called a “protocol”.

Web Clients and the HTTP Protocol

Arbortext Publishing Engine accepts requests and returns responses using the HTTP protocol. The HTTP protocol is a set of rules that allow an exchange of data between an HTTP client program and an HTTP server program. An HTTP client transmits an HTTP request to an HTTP server asking the server to perform some action. After sending the request, the HTTP client waits for the HTTP server to return a response.

HTTP client programs are often referred to as web clients, and HTTP server programs are often referred to as web servers. The most common web client programs are web browsers, such as Microsoft Internet Explorer, Mozilla Firefox, and Opera. While a web browser is probably the most familiar type of web client, any client program that transmits a request using HTTP protocol and receives an HTTP response is a web client.

HTTP Requests

An HTTP request consists of three parts: a request line, a set of HTTP headers, and a request body. The headers and body can be omitted if they're not needed. The request line consist of an HTTP command, a Uniform Resource Identifier (URI), and a version code.

There are many HTTP commands, but the most common are **`GET`** and **`POST`**. These are the only commands supported by Arbortext Publishing Engine.

A **`GET`** command requests that the server return the object (usually a file) specified by the URI. An HTTP **`GET`** request does not include a request body.

A **`POST`** command requests that the server return the object indicated after doing something with the request body. An HTTP **`POST`** request usually includes a body.

The URI part of an HTTP request can specify a file or a program to be run by the web server. The program can generate the file to be returned as part of the HTTP response. The URI can include “query parameters” in a ***`name`***`=`***`value`*** format with the parameters separated by ampersands (`&`); if the URI specifies a program, the query parameters are passed to the program along with the rest of the HTTP request. Each query parameter has a unique name and a string value. The HTTP standard places no restrictions on parameter names and values.

The HTTP request headers are also structured as name/value pairs. However, HTTP headers are defined by the HTTP standard specification. Generally, HTTP headers either describe the HTTP request body or inform the HTTP server of the kind of response that the client wants to receive.

The HTTP request body is only needed for **`POST`** requests. The request body is a data file that the server will process according to the request instructions before generating a response.

HTTP Responses

An HTTP server returns an HTTP response to the HTTP client that made the HTTP request. The response either confirms that the requested action was performed or returns information requested by the client.

An HTTP response consists of a status line, a set of message headers, and, possibly a message body. The status line contains a status code and a phrase indicating the high level response to the request. Codes and phrases are defined by the HTTP standard specification The most common code and phrase is `200 OK`, which mean that the request was processed successfully. Other possible values report different kinds of errors.

The message headers in an HTTP response describe the characteristics of the message body. HTTP response message headers are structured as name/value pairs and are defined by the HTTP standard specification.

The message body in an HTTP response, if one is present, provides the data requested by the HTTP client.

Web Servers

A web server is a program that accepts HTTP requests from HTTP clients. When it receives a request, the web server examines the URI part of the request and responds accordingly. A URI can specify an action to perform, a file to be returned, or a script or program that the web server should run to generate the response. After finding the file, running the program, or performing some other action, the web server generates an HTTP response and returns it to the client.

A web server is not limited to processing requests one at a time. A web server is capable of processing a large number of requests simultaneously.

Java Servlets and Servlet Containers

Java is a popular language for writing programs run by web servers. A servlet is a Java program that is implemented according to the Java Servlet Specification. Many web servers know how to run Java servlets. A Java servlet can be implemented once and then run under many different web servers.

To make a Java servlet available to service an HTTP request, a web server uses a Java servlet container. A servlet container is a program in which Java servlets can run. The servlet container provides an environment that complies with the Java Servlet Specification.

Tomcat is a standalone servlet container supported by the open-source Apache project. Apache or Microsoft IIS must use run Tomcat in a servlet container separate process.

Basic Flow of Control for the HTTP Protocol

Because Arbortext Publishing Engine is implemented as a Java servlet, a client request must use the HTTP protocol. The basic flow of control for a web client, a web server, a Java servlet container, and a Java servlet follows:

1. The client transmits an HTTP request to a web server and waits for a response.
2. The web server receives the request and examines the URI. The URI specifies that the request should be handled by a Java servlet, so the web server passes the request to its servlet container.
3. The servlet container receives the request and examines the URI to determine which servlet should process the request.
    a. If the servlet has not yet been loaded, the servlet container loads it and calls the servlet's initialization method.

   b. The servlet container passes the request to the servlet for processing.
4. The servlet processes the request, generates a response, and passes the response to the servlet container.
5. The servlet container returns the response to the web server.
6. The web server transmits the response to the client.
7. The client receives the response and makes use of the data it contains.

Arbortext Publishing Engine as a Java Servlet

Arbortext Publishing Engine is implemented as a Java servlet. The servlet is referred to as the Arbortext PE Request Manager. Like all servlets, the Arbortext PE Request Manager is written in Java; it implements the Java interface **javax.servlet.http.HttpServlet**, and behaves as required by the Java Servlet Standard.

Internally, the Arbortext PE Request Manager contains a series of components, described in the following sections, which work together to generate responses to the HTTP requests delivered by the servlet container in which the Arbortext PE Request Manager is running.

Arbortext Publishing Engine as a Web Application

According to the Java Servlet Specification, a web application is a collection of servlets, HTML documents, and other related components, organized into directories according to the standard. The Arbortext PE Request Manager is organized as a web application according to the standard.

Arbortext Publishing Engine as a Transaction Processor

Arbortext Publishing Engine groups a request it receives and the response it returns into a transaction. A transaction consists of a request, the eventual response to the request, information about the request (time submitted, time completed, client identity, and so on). A transaction can also include trace information (a configurable option) generated as the response was generated, such as an application log or intermediate files.

The Arbortext PE Request Manager creates a new transaction each time it receives an HTTP request. It assigns a unique integer called the transaction ID, provides a transaction name if one is supplied, and a temporary directory called the transaction directory, to the transaction. Then, it stores the request in the transaction directory. As Arbortext Publishing Engine processes the transaction, it adds information to the transaction directory, including, eventually, the transaction response. After Arbortext Publishing Engine has completed the transaction and returned the response to the client, it deletes the transaction directory.

There are two kinds of transactions:

- immediate transaction

  The client waits for its response until the transaction completes and the response is returned.

- queued transaction

  Arbortext PE Request Manager returns the transaction ID to the client and saves the transaction for processing at a later time. The client can submit requests, using the transaction ID, to determine whether the transaction has completed and to retrieve the transaction result.

As a transaction is received, executed, and completed, it passes through several transaction states that describe its condition.

- `initializing`

  The Arbortext PE server is receiving the request.

- `waiting`

  The transaction is waiting to be allocated to an Arbortext PE sub-process (for immediate requests).

- `queued`

  The transaction was placed in a queue and is waiting to be executed by the Queued Transaction Scheduler.

- `processing`

  The transaction is being executed by an Arbortext PE sub-process.

- `complete`

  The transaction is finished. A completed transaction can be completed successfully or completed with errors. Note that if the results is an error report rather than the expected document, the transaction is still considered complete.

  A transaction can also expire while waiting for an Arbortext PE sub-process allocation (if it's an immediate transaction).

- `cancelled`

  The transaction has been cancelled.

The transaction lifecycles are as follows:

- An immediate request transaction follows the lifecycle:

  Initializing → Waiting → Processing → Complete
- A queued request transaction follows the lifecycle:

  Initializing → Queued → Processing → Complete or Cancelled

Internal Components of Arbortext Publishing Engine

Arbortext Publishing Engine consists of two major components, the Arbortext PE Request Manager and a collection of Arbortext PE sub-processes. As described earlier, the Arbortext PE Request Manager is a Java servlet; it receives HTTP requests from the servlet container in which it runs and returns a response for each request. Internally, it consists of a number of components that cooperate in processing each request. The Arbortext PE Request Manager is designed to be extensible; users can write custom components that take part in the request-handling process. Such components are called Dynamic Components.

The Arbortext PE Request Manager and its Dynamic Components have little or no knowledge of XML technologies. To import, publish, export, and otherwise manipulate XML documents, the Arbortext PE Request Manager uses separate programs called Arbortext PE sub-processes. Each Arbortext PE sub-process runs in a separate process. The Arbortext PE Request Manager maintains a pool of Arbortext PE sub-processes that are idle, ready to work on its behalf. When, in the course of processing a request, the Arbortext PE Request Manager determines that it needs some document-manipulation function performed, it allocates an Arbortext PE sub-process to do so and waits for the Arbortext PE sub-process to complete its work.

Like the Arbortext PE Request Manager, an Arbortext PE sub-process can be customized. Users can write Arbortext Publishing Engine applications in several languages. These applications run in Arbortext PE sub-processes just as Dynamic Components run in the Arbortext PE Request Manager.

Arbortext PE Request Manager Dynamic Components

There are three major kinds of Dynamic Components: Cache Managers, Queue Managers, and Request Handlers. These objects are initialized when the Arbortext PE Request Manager is loaded and initialized by its servlet container. When the Arbortext PE Request Manager receives a request, it passes the request to each component in turn, asking the component to process the request and produce a response. If a component indicates that it cannot do so, the Arbortext PE Request Manager continues with the next component. When a component produces a response, the Arbortext PE Request Manager ends the scan and returns the response to the client.

Other dynamic Arbortext PE Request Manager components are described in Internal Components of Arbortext Publishing Engine on page 20.

Arbortext Publishing Engine Cache Managers

A cache manager is an object capable of storing a response to a request and later returning that response to service a subsequent identical request. When the Arbortext PE Request Manager presents a request to a cache manager, the cache manager can reply in one of three ways:

- It can't fulfill requests of that type.
- It has a response stored for the request and provides it to the Arbortext PE Request Manager.

  If a cache manager has a stored response, then the response is transmitted to the client.
- It doesn't have a response for that request, but it would like to store a copy of the response when it's available.

  If a cache manager wants a copy of the response to store, the Arbortext PE Request Manager remembers the cache manager when the request is fulfilled.

If no cache manager can provide a response for a request, the Arbortext PE Request Manager continues to the queue managers.

Arbortext Publishing Engine Queue Managers

A queue manager is an object that can accept a request and save it for execution at a later time. When the Arbortext PE Request Managerr presents a request, a queue manager can reply in one of two ways:

- It can't fulfill requests of that type.
- It has queued the request for execution later; it supplies a response to be returned immediately to the HTTP client.

When a queue manager saves a request, it generates a response that includes the request’s transaction ID, together with instructions on how the HTTP client can submit another request at a later time to determine whether the request has been completed and to retrieve the transaction result. This allows a client to submit a request and receive a very quick response, rather than waiting (possibly for a very long time) for a transaction to complete processing.

A Queue Manager stores requests by placing them on queues, as described in Basic Flow of Control for Queued Requests on page 50. If no queue manager can process a request, the Arbortext PE Request Manager continues to the request handlers.

Arbortext Publishing Engine Request Handlers

A request handler is an object capable of generating a response to a request. The Arbortext PE Request Manager presents a request to each loaded request handler. When the Arbortext PE Request Manager presents a request, a request handler can reply in one of two ways:

- It can't fulfill requests of that type.
- It can fulfill the request; it supplies the response to send to the client.

If a request handler produces a response, then the Arbortext PE Request Manager checks whether any cache manager wants to store that response before returning the response to the client. If so, the response is passed to the appropriate cache manager.

If no request handler can handle a request, Arbortext PE Request Manager generates an "unknown request" error response to be returned to the client.

Additional Arbortext PE Request Manager Dynamic Components

In addition to Cache Managers, Queue Managers, and Request Handlers, the Arbortext PE Request Manager supports Initializers, Notifiers, Queues, and Request Selectors. These are all Dynamic Components, so custom versions of these components can be developed and loaded into the Arbortext PE Request Manager .

Initializers

An initializer is an object that runs during Arbortext PE Request Manager startup. Initializers prime cache managers, allocate resources, and otherwise set the stage for request processing. There are two kinds of initializers, blocking and deferred. Blocking initializers run during startup, after all other components have been initialized but before the Arbortext PE Request Manager starts accepting client requests. Deferred initializers run after the Arbortext PE Request Manager starts accepting client requests.

Notifiers

A notifier is an object that is informed by the Arbortext PE Request Manager every time a transaction changes state. The notifier object can send email, communicate with the client that submitted the transaction, or take other action.

Queues

A queue is an object that stores queued transactions for execution. Transactions are placed on a queue by a queue manager. At some point determined by configuration settings, they are retrieved for execution by the Queued Transaction Scheduler. The queue is responsible for determining which transactions it will accept and the order in which its transactions will execute. According to configuration criteria, it keeps transactions from executing until a set time or until other conditions are met.

Request Selectors

A request selector is an object that can examine an HTTP request and determine whether the request meets some built-in or configured criterion. A series of request selectors can be grouped by **And** and **Or** connectors into Test Sets. Test Sets may be used by queue managers, queues, and subprocess pools to determine whether they can process a request.

Arbortext PE Request Manager Static Components

The static components are distinct parts of Arbortext PE Request Manager, but they are not Dynamic Components. They cannot be replaced or extended by custom Java code, although their behavior can be controlled by configuration parameters.

Arbortext Publishing Engine Queued Transaction Scheduler

The Queued Transaction Scheduler is a background process that runs inside the Arbortext PE Request Manager. It awakens periodically and looks for idle Arbortext PE sub-processes and queued transactions. If it finds a queued transaction that is is ready to execute and an idle Arbortext PE sub-process in a sub-process pool that can process that transaction, it starts the execution of the transaction.

Arbortext Publishing Engine Request Context

The Arbortext Publishing Engine Request Context is a global object that provides services to Dynamic Components. The Arbortext Publishing Engine Request Context offers capabilities like allocating temporary files, scheduling clean-up events for a later time, allocating Arbortext PE sub-processes (described in Arbortext PE sub-process Pools on page 24) and other utility capabilities used by Dynamic Components.

Configuration data

Almost all of the behavior of the Arbortext PE Request Manager, including loading Dynamic Components, setting global and component parameters, and setting the parameters for Arbortext PE sub-process pools, is specified by the configuration information loaded from the **`e3config.xml`** configuration file when Arbortext Publishing Engine starts. Dynamic Components obtain their parameters from the configuration file when the Arbortext PE Request Manager initializes them.

Arbortext PE sub-process Pools

The primary purpose of Arbortext Publishing Engine is the manipulation of documents using XML and related technologies. The Arbortext PE Request Manager acts as the facilitator handling requests and responses, but it has no knowledge of XML. XML processing is performed by Arbortext PE sub-processes.

Each Arbortext PE sub-process is a running instance of Arbortext's XML processing engine, with the graphic user interface disabled and additional publishing and performance capabilities enabled. Groups of Arbortext PE sub-processes are organized into Arbortext PE sub-process pools. When a dynamic Arbortext PE Request Manager component determines that it needs the services of an Arbortext PE sub-process, it invokes a method of the Arbortext Publishing Engine Request Context to obtain an Arbortext PE sub-process from a particular pool, uses it to perform one or more operations, and then releases it so that the Arbortext PE Request Manager can use it to fulfill other requests.

An Arbortext PE sub-process pool is a collection of identical Arbortext PE sub-processes that are associated by the following:

- a set of request selectors that specify criteria for determining whether the Arbortext PE sub-processes in the pool should process a given transaction
- a set of parameters that determine how many Arbortext PE sub-processes are in the pool, how often Arbortext PE sub-processes should be terminated and restarted, and other behaviors (explained in Arbortext PE sub-process Pool Management on page 57).

Having multiple Arbortext PE sub-process pools offer two major advantages:

- Pools of Arbortext PE sub-processes can be initialized differently.
- Arbortext PE sub-processes pools can be configured to process different types of requests.

When a dynamic component (most frequently, a Request Handler) determines that it needs an Arbortext PE sub-process, it calls an Arbortext Publishing Engine Request Context routine and passes a reference to the HTTP request being processed. The Arbortext Publishing Engine Request Context offers the request to each Arbortext PE sub-process pool until one replies that it's configured to handle that type of request. The last Arbortext PE sub-process pool queried is always the default pool, which can allocate an Arbortext PE sub-process to fulfill any request not already handled.

Arbortext PE Applications

An Arbortext PE Application is a program written in Arbortext Command Language (ACL), Java, JavaScript, or VBScript that runs in an Arbortext PE sub-process. The Arbortext PE Request Manager invokes an Arbortext PE sub-process to run an Arbortext

PE Application when it receives a request with HTTP query parameters specifying the application language and a function or class name. The Arbortext PE Request Manager passes the entire HTTP request to the Arbortext PE Application. The application is responsible for constructing a complete HTTP response and returning the response to the Arbortext PE Request Manager which returns it to the HTTP client.

Arbortext Publishing Engine as a Document Conversion Server

Arbortext Publishing Engine is distributed with an Arbortext PE Application written in ACL that facilitates the conversion of a document from one format to another. For example, an HTTP POST request might pass an XML document and specify that the document should be converted to PDF using an XSL stylesheet. Arbortext Publishing Engine makes no assumptions about the client of a conversion request; the client can be any program prepared to communicate using the HTTP protocol. The 11 Arbortext Publishing Engine Document Conversion on page 163 provides extensive information.

Arbortext Publishing Engine as a Publishing Server

Arbortext Publishing Engine is also distributed with an Arbortext PE Application written in Java that provides publishing services to Arbortext Editor clients. Unlike the conversion application, the publishing client must be Arbortext Editor. Arbortext Editor users can specify an Arbortext PE server, then select publishing, import, and export operations using the Arbortext Editor program. Publishing requests are fulfilled on the Arbortext PE server.

As part of its publishing service, Arbortext Publishing Engine is distributed with a Cache Manager, an Initializer, and a Java Arbortext PE Application that gathers information about the publishing configuration of the Arbortext PE server (such as installed document types, stylesheets, and so on) and reports that information to Arbortext Editor clients or Arbortext Publishing Engine administrators.

Arbortext Publishing Engine Java Client SDK

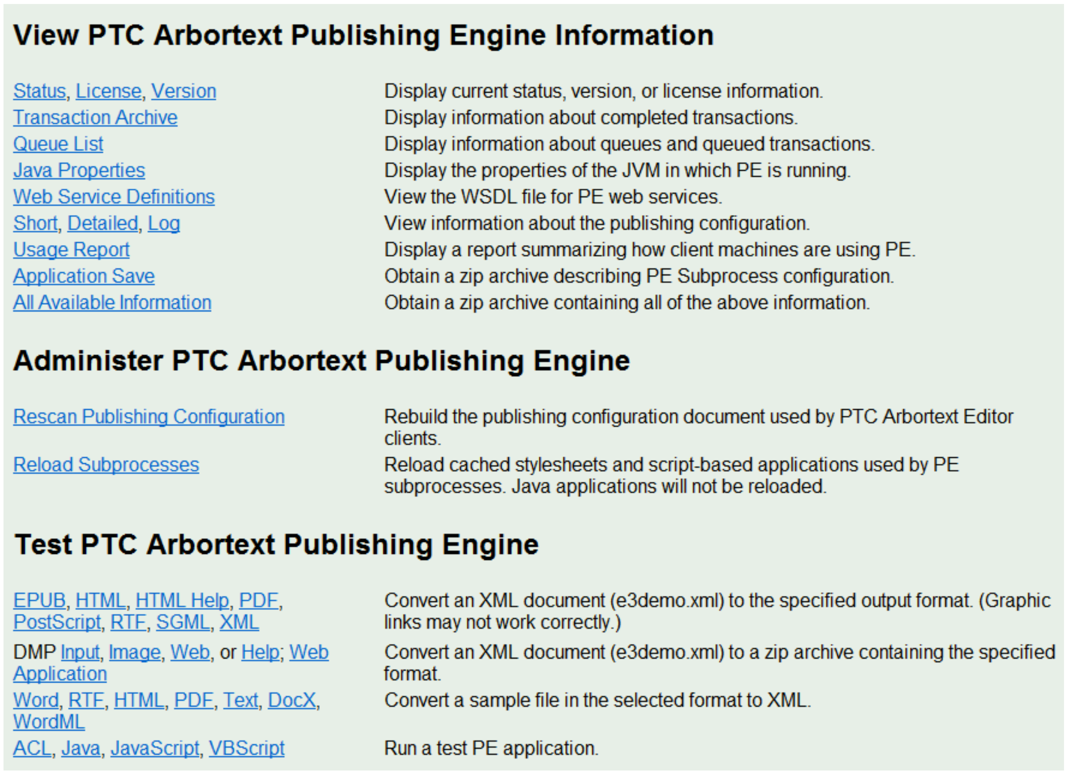
Arbortext Publishing Engine Java Client Software Developers' Kit allows Java programmers to transmit HTTP requests to and receive HTTP responses from an Arbortext PE server. Refer to 13 Using the Java Client SDK on page 199 for information.

Monitoring and Reporting Using a Web Browser

Arbortext Publishing Engine has an index page with links that return information and perform a variety of administrative actions and sample document conversions. After you've successfully installed and configured Arbortext Publishing Engine, this index page is available from a web browser. Use a URL that follows the example:

`http://servername:port/e3/`

In the URL, ***servername*** is the name of the Arbortext PE server machine, and ***port*** is the port number the servlet container or web server uses to monitor HTTP requests on its behalf.



View PTC Arbortext Publishing Engine Information

Status, License, Version	Display current status, version, or license information.
Transaction Archive	Display information about completed transactions.
Queue List	Display information about queues and queued transactions.
Java Properties	Display the properties of the JVM in which PE is running.
Web Service Definitions	View the WSDL file for PE web services.
Short, Detailed, Log	View information about the publishing configuration.
Usage Report	Display a report summarizing how client machines are using PE.
Application Save	Obtain a zip archive describing PE Subprocess configuration.
All Available Information	Obtain a zip archive containing all of the above information.

Administer PTC Arbortext Publishing Engine

Rescan Publishing Configuration	Rebuild the publishing configuration document used by PTC Arbortext Editor clients.
Reload Subprocesses	Reload cached stylesheets and script-based applications used by PE subprocesses. Java applications will not be reloaded.

Test PTC Arbortext Publishing Engine

EPUB, HTML, HTML Help, PDF, PostScript, RTF, SGML, XML	Convert an XML document (e3demo.xml) to the specified output format. (Graphic links may not work correctly.)
DMP Input, Image, Web, or Help; Web Application	Convert an XML document (e3demo.xml) to a zip archive containing the specified format.
Word, RTF, HTML, PDF, Text, DocX, WordML	Convert a sample file in the selected format to XML.
ACL, Java, JavaScript, VBScript	Run a test PE application.

View Arbortext Publishing Engine Information

- The `Status` link returns a status report. It includes:
  - basic installation, system, COM, allowed functions and global parameters information.
  - Arbortext PE sub-process pool status, including whether it's enabled or disabled, its associated configuration settings, the process IDs and allocation status of each Arbortext PE sub-process.

- all configuration settings for caches, queues, request handlers, and request selectors.
- information on the Queue List, if queues are configured. Information about individual queues is available from the Queue List page.
- system **Environment Variables**. If **PTC_D_LICENSE_FILE** is set, it will be included in the **Environment Variables** report.

- The `License` link retrieves basic information about the installation, as well as the license source, the user under which Arbortext Publishing Engine is running, and the number of processor cores and packages on the Arbortext PE server. It also lists the optional software components installed and whether they are licensed.

  A license error report will be returned with the information that **PTC_D_LICENSE_FILE** is either missing or set to an incorrect value (and the incorrect value will be reported).
- The `Version` link retrieves version information about Arbortext Publishing Engine and its Arbortext PE sub-processes.
- The `Transaction Archive` link retrieves information on the transaction archive, if one has been implemented. More information about individual archived transactions is available from the Transaction Archive page.
- You can retrieve a report on the `Queue List`, if queues are configured. Information about individual queues is available from the Queue List page.
- The `Java Properties` link returns all information about the JVM.
- The `Web Service Definitions` link returns the Arbortext Publishing Engine WSDL definitions.
- You can retrieve any of three variations of the Publishing Configuration report.
- You can retrieve a `Usage Report` with a summary of clients and transactions, and usage by client.
- You can run an Application Save to retrieve configuration about Arbortext PE sub-processes.
- All Available Information returns a zip archive containing all the information available about Arbortext Publishing Engine listed in this section, as well as the output from the sample PE applications in the testing section and the Application Save zip archive.

Administer Arbortext Publishing Engine

You can rescan the publishing configuration information and its cache for use by Arbortext Editor clients.

You can reload scripts and update cached stylesheets on Arbortext PE server for use by all clients.

Test Arbortext Publishing Engine

You can convert a sample document to any of the supported output formats listed.

You can run Arbortext Publishing Engine sample test applications which return basic information about server configuration. The source code for these sample applications is available from your installation in ***`PE_HOME`*****`\e3\samples`**, and they're described in the *Programmer's Guide to Arbortext Publishing Engine*.

None of the actions available from the Arbortext Publishing Engine index page are queued.

Logging and Tracking

There are a number of facilities that can be enabled to provide information about how Arbortext Publishing Engine processes transactions. The following sections describe the most important mechanisms.

The Servlet Log

Arbortext Publishing Engine uses the Java Apache log4j package to trace its internal operation. By default, this information is entered in the servlet log. Administrators can control the level of detail entered in the log by setting the global `debug` variable to `true` in the **`e3config.xml`** configuration file. Enabling the `debug` parameter is the first step in trying to diagnose problems with the Arbortext Publishing Engine.

With `debug` set to `true`, the Arbortext PE Request Manager will make several log entries for every transaction, specifying the start of each transaction, whether a cache manager, queue manager, or request handler generated a response, whether the transaction was queued or processed immediately, how the transaction response was transmitted, and whether the transaction was archived before it was deleted from the active transaction area.

The Transaction Archive

The transaction archive stores transactions that have completed. After a transaction's response has been transmitted to the HTTP client, the Transaction Archive examines the transaction and decides whether to store it.

By default, only transactions that generated an error response and transactions that include an application log or intermediate file are stored in the archive. By default, transactions are retained for 48 hours, and the transaction archive can occupy up to 500 MB of disk space.

However, you can configure the archive to retain all transactions processed by the Arbortext PE Request Manager, or to save no transactions at all.

Application Logs

An application log is a file containing information about how an application performed its work. The log consists of any number of entries, each marked with a particular severity (error, warning, information, etc.). The application log is stored as part of the transaction archive entry for the transaction that invoked the application.

An application makes entries in the application log by making subroutine calls that write information to the log. In addition, you must configure the Arbortext PE Request Manager to enable log entries (by default, no log entries are saved). You can do this for all applications, for all applications written in a particular programming language, or for individual applications. In each case, you can specify that only application log entries for a given severity and higher are saved, thereby controlling the amount of detail in the log. Refer to *Configuration Guide for Arbortext Publishing Engine* for information on how to configure these parameters.

Application Intermediate Files

An intermediate file is an open document or a file on disk that an application generates as part of generating a response to a request. The application can make subroutine calls to save such files and documents to the transaction archive for debugging purposes. When a subroutine call is made, the file or document is only saved if the application log setting for the application allows writing `INFO` and higher level entries to the application log.

An intermediate file can be any file used in processing the request, including the XML document created by a content pipeline as part of a publishing operation, a text file passed as input to a third-party tool, and so forth. Intermediate files are only saved if the application running on the Arbortext PE server explicitly saves them, so developers must decide when a file is worth saving and take steps to do so.

2 Supporting Documentation

For all platforms, Arbortext Publishing Engine documentation is available from Arbortext Publishing Engine Interactive or an Arbortext Editor client by choosing **Help ▸Help Center** to access Arbortext Publishing Engine online documentation. You can configure the Help Center to display just the **Publishing Engine** product documentation. See the Help Center online help to learn more about using Help Center.

The following documentation applies specifically to Arbortext Publishing Engine:

- *Arbortext Publishing Engine Release Notes*

  contains general information about what is new or changed this release.
- *Installation Guide for Arbortext Publishing Engine*

  describes installation and licensing.
- *Configuration Guide for Arbortext Publishing Engine*

  describes configuration tasks, files and parameters and values that affect the Arbortext PE server and its Arbortext PE sub-processes. It also describes the reporting, monitoring, testing, and troubleshooting operations.
- *Programmer's Guide to Arbortext Publishing Engine*

  describes how to write custom applications for Arbortext Publishing Engine in Arbortext Command Language (ACL), Java, JavaScript, and VBScript languages. A programmer would use this documentation in combination with other developer documentation to implement custom applications.
- *Test Utility User's Guide*

  contains information about how to use the Arbortext Publishing Engine Test Utility to test your custom applications that will run on your Arbortext PE server.

Javadoc for the Arbortext Editor and Arbortext Publishing Engine interfaces is delivered in the **Programming ▸Javadoc ▸Arbortext Publishing Engine** section of Help Center.

The following documentation contains information about using and customizing Arbortext features as well as information that can help you develop custom applications.

- *Arbortext Command Language Reference*

  introduction and reference to Arbortext Command Language commands, functions, hooks, callbacks, and **`set`** options. Also includes a guide to the Arbortext Repository API.

- *Programmer's Reference*

  introduction and reference to the AOM, and how to use Java, JavaScript, JScript, VBScript, COM, and C++ to access the AOM, how to perform basic document operations and work with events, and includes documentation of W3C and Arbortext interfaces (and their attributes, enumerations, and methods) supported by the AOM.

- *Customizer's Guide*

  describes how to configure and customize Arbortext features for use at your site. Examples of typical customizations are provided throughout the guide.

- *Document Types Guide*

  describes the document types delivered with Arbortext Editor and Arbortext Publishing Engine, information on working with Arbortext Architect, and details on creating and working with custom document types.

- *Content Management Guide*

  contains administrative and configuration information for the PTC Server connection..

- *Administrator's Guide*

  contains administrative and configuration information about customizing your Arbortext environment, including configuring toolbars, spelling checking, Arbortext Editor startup, fonts, and publishing. Also contains a reference to Arbortext environment variables.

- *User's Guide to Arbortext Styler*

  describes how to create and use Arbortext Styler stylesheets and describes each of the print engines: APP, FOSI, and XSL-FO.

- *FOSI Reference*

  describes how to use FOSI stylesheets, covering topics such as FOSI coding techniques, using the FOSI Editor, and testing FOSIs.

- *Content Pipeline Guide*

describes the content pipeline concepts and describes the components that make up a pipeline. It provides examples of using AOM and ACL to manipulate filters and pipelines for publishing.

- *Tutorial for Arbortext Import*

  contains step-by-step tutorials for using Arbortext Import, including the process of running conversions, building MapTemplates for parsing text files, and building MapTemplates to parse Word and HTML documents.

- *Reference Guide to Arbortext Import*

  provides instructions on how to create and edit Arbortext Import MapTemplates. It includes a complete reference to ppXML, the XML document type used by Arbortext Import to represent different source document types and word processor formats in a consistent markup language. Map development is typically the process of converting ppXML markup into user-defined XML document types.

- *Arbortext Editor, Arbortext Styler, and Arbortext Architect Release Notes*

  for Arbortext Publishing Engine, look for information about non-user interface features and fixes that would apply to the Arbortext PE sub-processes.

- *Arbortext's XSL Support*

  an annotated and excerpted version of the W3C's Extensible Stylesheet Language (XSL) Version 1.0 Recommendation as permitted by the W3C Intellectual Property policies for annotations.

II

The Arbortext PE Request Manager

3 Understanding the Internal Structure of Arbortext PE Request Manager

This section describe the internal structure of the Arbortext PE Request Manager servlet and discusses how Dynamic Components operate.

Initialization

The Arbortext PE Request Manager is implemented as a Java servlet named **com.arbortext.e3.E3servlet**. The basic components of the servlet are located in the directory ***PE_HOME*\e3\e3**, which conforms to the definition of a Web Application in the Java Servlet Specification. The **e3\e3\WEB-INF\web.xml** contains the basic mappings that tell the servlet container when to invoke the Arbortext PE Request Manager servlet. (The **web.xml** file is described in *Configuration Guide for Arbortext Publishing Engine*.) The Arbortext PE Request Manager may be loaded either when the servlet container starts executing (the default) or when the servlet container receives the first HTTP request for the Arbortext PE Request Manager.

Basic Flow of Control for Initialization

When the servlet container calls the Arbortext Publishing Engine servlet's **init** method, the **init** method begins by finding the configuration file **e3config.xml** and parsing the XML document into a set of Java objects rooted in an instance of the class **com.arbortext.e3cf.E3Config**.

The **init** method continues by creating an instance of the object **com.arbortext.e3.E3RequestContext** according to the parameters in **e3config.xml**. The **E3RequestContext** constructor performs the following steps:

1. Builds a map of the global parameters specified in **web.xml** and **e3config.xml**.
2. Examines the global parameter map. If any parameter is not specified, a default value is stored in the map. As processing proceeds, other values are calculated and stored in the map.
3. Sets the global `debug` parameter and `debug-verbose` flags so that tracing produces the desired level of detail.
4. Sets global variables for the location of temporary files and directories.
5. Determines the location of the Arbortext PE sub-process installation, either as specified by the parameter **com.arbortext.e3.epicInstallation** or relative to the location of the servlet container's web application directory.
6. Initializes the transaction archive and the directory in which transactions directories are created.
7. Initializes the **com.arbortext.e3.E3Version** object with information about the Arbortext Publishing Engine version, build number, and time and date it was launched
8. Loads, instantiates, and initializes an object for each component defined in **e3config.xml**, in the order shown in Arbortext PE Request Manager Components on page 40.
9. Runs all blocking initializers.

10. Starts a background thread to run all deferred initializers.

Components of the same type are initialized in the order they appear in the **`e3config.xml`** configuration file.

For each dynamic component, the **com.arbortext.e3.E3RequestContext** initializer calls the object's **init** method. Each object can use the services provided by the Arbortext Publishing Engine Request Context that have already been initialized. Thus, for example, a cache manager's **init** method should not try to allocate an Arbortext PE sub-process because cache managers are initialized before the Arbortext PE sub-process pools. Since the initializers run last, they can assume that all other components have initialized and are available for use.

The process of initializing the Arbortext PE sub-process pools may cause Arbortext PE sub-processes to start running, meaning an Arbortext PE sub-process will start for each pool that is not disabled and that has a `minSubprocesses` parameter greater than zero.

The process of running initializers may result in transactions being submitted to the request handlers for processing, meaning these transactions look like real transactions submitted from the network, but in fact they are pseudo-transactions submitted by the initializers.

When the thread executing the deferred initializers finishes running, it will start the Queued Transaction Scheduler. From this point, queued transactions may be submitted for execution.

When the **com.arbortext.e3.E3RequestContext** initializer returns control to the **init** method, the **init** method will return to the servlet container. From that point forward, the Arbortext PE Request Manager will be prepared to accept requests from the servlet container (from network clients on the web). The **init** returns before the deferred initializers have completed execution.

If an error occurs during this initialization process, the method **com.arbortext.e3.E3servlet.init** will throw a **ServletException** object. The object tells the servlet container not to pass any requests to the Arbortext PE Request Manager. Instead, the servlet container will call **com.arbortext.e3.E3servlet.destroy**. If another request arrives for the Arbortext PE Request Manager, the servlet container will attempt to load it again by calling the **init** method a second time.

The e3config.xml Configuration File

Every aspect of Arbortext Publishing Engine initialization and its operation is defined by its configuration file located in the install tree:

PE_HOME`\e3\e3\WEB-INF\e3config.xml`

The XML document type it complies with is located in:

PE_HOME`\doctypes\e3config\e3config.dtd`

The configuration file can be edited using either a text editor or Arbortext Publishing Engine Interactive. On Windows, you can launch Arbortext Publishing Engine Configuration program, navigate to the **Setup** tab, and click **Edit Configuration**. The *Configuration Guide for Arbortext Publishing Engine* guide describes the **`e3config.xml`** file, its configuration settings, and how to edit it. You must restart Arbortext Publishing Engine for configurations changes to take effect.

The Arbortext PE Request Manager reads **`e3config.xml`** and parses its contents into objects as defined in the Java package **com.arbortext.e3cf**. During initialization, **e3cf** objects are passed to the **init** method of each dynamic component, so that they may retrieve their parameters. The **e3cf** objects are not used after initialization. Instead, the Arbortext Publishing Engine Request Context object offers methods for locating all instances of each type of dynamic component.

Almost every element in **`e3config.xml`** can have associated parameters. Parameters can be global or local. Global parameters are specified at the top level (descendants of the **E3Config** element). Local parameters are descendants of lower-level elements. Local parameters apply only to the object within which they are defined.

Every parameter is specified by a **Parameter** element with **name** and **value** attributes. If a local parameter has the same name as a global parameter, the local parameter value overrides the global value. For example, if the global parameter **debug** has the value `false`, you could specify a local **debug** parameter with the value `true` for a specific cache manager.

Arbortext PE Request Manager Components

Arbortext PE Request Manager components are declared in the **`e3config.xml`** file by the following elements:

Arbortext PE Request Manager Components

Component	Element	Interface
Parameter	**Parameter**	not a dynamic component
Request Selectors	**RequestSelectors** and **RequestSelector**	**com.arbortext.e3.E3RequestSelector**
Cache Managers	**CacheManagers** and **CacheManager**	**com.arbortext.e3.E3CacheManager**
Queue Managers	**QueueManagers**, **QueueManager** and **TestSet**	**com.arbortext.e3.E3QueueManager**
Request Handlers	**RequestHandlers** and **RequestHandler**	**com.arbortext.e3.RequestHandler**
Arbortext PE sub-process Pools	**SubprocessPools**, **SubprocessPool**,	not a dynamic component

Component	Element	Interface
	SubprocessContext and **TestSet**	
Allowed Functions	**AllowedFunctions** and **ClientFunction**	not a dynamic component
Queues	**Queues**, **Queue**, and **TestSet**	**com.arbortext.e3. E3Queue**
Notifiers	**Notifiers**, **Notifier**, and **TestSet**	**com.arbortext.e3. E3Notifier**

The **SubprocessPools** element contains a series of **SubprocessPool** elements. Each **SubprocessPool** has **SubprocessContext** and **TestSet** child elements as well as many attributes. For information on the attributes, refer to Arbortext PE sub-process Pool Management on page 57. The **SubprocessContext** element supports **Parameter** child elements, and their values are passed to every Arbortext PE sub-process in the pool when it is initialized.

A series of tests can be defined to determine whether a request can be processed by Arbortext PE sub-processes in the pool. The **TestSet** element contains a set of **Test** elements, which can be grouped by **And** and **Or** elements. A **Test** element refers to a **RequestSelector** with an **id** attribute that matches the **name** of the test. A test evaluates to `true` if its referenced **RequestSelector** returns `true` when offered a request for evaluation. An **And** element contains **Test**, **And**, and **Or** elements, which means that it evaluates to `true` if every element it contains evaluates to `true`. An **Or** element contains **Test**, **And**, and **Or** elements, and it evaluates to `true` for any element that evaluates to `true`.

Global Arbortext PE Request Manager Parameters

You can set the global parameters described in the next several sections using **Parameter** elements (descendants of the **E3Config** element) in **`e3config.xml`**. These parameters control the behavior of the Arbortext PE Request Manager. You can specify additional global parameters, and their values can be retrieved using **E3RequestContext. getParameter**.

The debug and debug-verbose Parameters

If **debug** is set to `true`, then log messages with WARN and INFO severity levels are written to the servlet log. If it is set to `false`, only FATAL and ERROR level messages are logged.

If **debug-verbose** is set to `true`, then TRACE and DEBUG level log messages are written.

⚠ ***Caution***

*Setting **debug-verbose** to `true` includes messages from daemon threads that run every few seconds, so use this parameter with caution and only during troubleshooting.*

Creating Temporary Files

These parameters control how the Arbortext PE Request Manager creates temporary files:

- **com.arbortext.e3.tempFileDirectory**

  is the absolute path to a directory in which all temporary files and directories will be created by calls to **E3RequestContext.createTemporaryFile**. Its default is the value of the Java Virtual Machine `java.io.tmpdir` system property.

- **com.arbortext.e3.tempFilePrefix**

  specifies a string to be prepended when generating temporary file names. If not specified, the default is **`ati`**.

- **com.arbortext.e3.tempFileSuffix**

  specifies the end string for generating temporary file names. If not specified, the default is **`.tmp`**.

Managing Arbortext PE sub-process Temporary Files

The `delete-temp` parameter controls whether temporary storage for each Arbortext PE sub-process is deleted when the sub-process terminates. Setting it to `false` preserves Arbortext PE sub-process temporary file storage for debugging.

⚠ ***Caution***

Setting `delete-temp` to `false` will consume disk space quickly, so don't leave it on for very long.

Parameters that Control Application Logging

- **com.arbortext.e3.applicationLog**

  Sets the default level for tracing in all applications

- **com.arbortext.e3.applicationLog.compose**

  Sets the log level for publishing requests submitted from an Arbortext Editor client.

- **com.arbortext.e3.applicationLog.convert**

  Sets the log level for requests submitted using the **f=convert** function.

- **com.arbortext.e3.applicationLog.acl**

  Sets the log level for tracing in ACL applications
- **com.arbortext.e3.applicationLog.acl.*package.function***

  Sets the log level for the ACL application specified by ***package*::*function***
- **com.arbortext.e3.applicationLog.java.*class***

  Sets the log level for the Java application specified by ***class***
- **com.arbortext.e3.applicationLog.javascript.*function***

  Sets the log level for the JavaScript application specified by ***function***
- **com.arbortext.e3.applicationLog.vbscript.*function***

  Sets the log level for the VBScript application specified by ***function***

These application logging parameters can take the following values. By default, By default, the value is set to WARN, meaning messages are logged for levels FATAL, ERROR, and WARN.

- `OFF`
- `FATAL`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`
- `ALL`

In addition, an application log is always included in the set of files saved to a transaction archive. However, you can specify an additional location for sending application log messages:

- **com.arbortext.e3.applicationLog.display**

  If set to `true` (the default), application log messages are written to the Arbortext Diagnostics window. If set to `false`, messages are only logged in the transaction archive.
- **com.arbortext.e3.applicationLog.displayPatternLayout**

  Specifies the Java **`log4j`** pattern to use when constructing log entries written to Arbortext Diagnostics window on Windows.

Managing Transactions

- **com.arbortext.e3.transactionDirectory** specifies the directory where Arbortext Publishing Engine puts transaction subdirectories. For every request, Arbortext Publishing Engine will create a subdirectory in this directory and store intermediate files in it, as well as build the response to the request. The default directory is **`activeTransactions`** in the temporary directory specified by **com.arbortext.e3.tempFileDirectory**, described in *Configuration Guide for Arbortext Publishing Engine*.

- **com.arbortext.e3.transaction.maxCompletedTransactionAge**

  specifies the maximum time, in hours, that a transaction directory will be kept in the Active Transaction Directory. This parameter applies only to queued transactions that have been completed but the results have not been retrieved. The default is `168` hours (1 week).

  When Arbortext Publishing Engine finishes processing a queued transaction, it notes the time. When the interval specified by this parameter has elapsed, Arbortext Publishing Engine may copy the transaction directory into the transaction archive (according to the setting of the **com.arbortext.e3.transactionArchive.selector** parameter). Either way, the transaction directory is then deleted.

  Queued transaction results that have been retrieved are deleted according to the interval specified by **maxRetrievedTransactionAge**.

- **com.arbortext.e3.transaction.maxRetrievedTransactionAge**

  This parameter specifies the maximum time, in hours, that a transaction directory will be kept in the Active Transaction Directory. This parameter applies only to queued transactions that have been completed, and the results have been retrieved at least once. The default is `24` hours (1 day).

  After Arbortext Publishing Engine finishes processing a queued transaction, it's available for retrieval. When Arbortext Publishing Engine receives and processes the first request to retrieve a queued transaction result, it notes the time. When the interval specified by this parameter has elapsed, Arbortext Publishing Engine may copy the transaction directory into the transaction archive (according to the **com. arbortext.e3.transactionArchive.selector** parameter). Either way, the transaction directory is then deleted.

  Queued transaction results that have not been retrieved are deleted according to the interval specified by **maxCompletedTransactionAge**.

- **com.arbortext.e3.defaultTransactionName**

  This parameter specifies descriptive text for transaction names as the default for all incoming requests that do not already specify a transaction name. The specification can include a string `$t`, that will be replaced by the unique transaction ID assigned on the Arbortext PE server. The default value is an empty string.

  Arbortext Editor clients can specify a transaction name using the **Queued Transaction Names** dialog box (available from **Publishing Engine** category of **Tools ▸Preferences**) or the **Transaction Name** field on the **File ▸Publish** set of dialog boxes.

Applications can specify a transaction name using the `transaction-name` query parameter. Refer to *Configuration Guide for Arbortext Publishing Engine* for information.

Managing the Transaction Archive

- **com.arbortext.e3.transactionArchive.maxAge**

  Specifies the maximum time in hours that an entry will remain in the archive before it is automatically deleted. A value of `0` means never delete archive entries. The default is `48` hours.

- **com.arbortext.e3.transactionArchive.threadInterval**

  Specifies the number of seconds between execution of the management thread. The thread checks for transactions completed since the last run, as well as checking for transactions older that the interval specified by **com.arbortext.e3.transactionArchive.maxAge**. The default is `10` seconds.

  The thread will also check for transactions that are completed and should be archived, as well as archived completed requests that should be deleted, as configured by the **maxCompletedTransactionAge** and **maxRetrievedTransactionAge** parameters.

- **com.arbortext.e3.transactionArchive.maxSize**

  Specifies the maximum size in megabytes that the transaction archive may occupy on disk. If an entry causes the archive to exceed this size, entries will be deleted, oldest first, until the archive is less than the maximum size. A value of `0` means no deletions will be performed based upon maximum size. The default is `500` megabytes.

- **com.arbortext.e3.transactionArchive.clearOnStart**

  Specifies whether to clear the transaction archive when Arbortext Publishing Engine starts.

  – `false` (the default) keeps archive entries when Arbortext Publishing Engine starts.

    Keeping archived entries means the archive keeps all its entries each time Arbortext Publishing Engine starts.

  – `true` clears archive entries each time Arbortext Publishing Engine starts.

- **com.arbortext.e3.transactionArchive.selector**

  Specifies the filters for determining which requests are saved in the archive.

  – `none` saves no entries.

  – `error` saves error entries, including requests with a response other than 200, requests which cause an Arbortext PE sub-process to terminate abnormally, and requests for which transmission of the response to the client fails.

  - `log` (the default) saves error entries, as well as requests with log entries or intermediate files.
  - `all` saves all requests, including successful requests.

- **com.arbortext.e3.transactionArchive.enable**
  - `true` (the default) makes the transaction archive available.
  - `false` makes the transaction archive unavailable. Use this setting if you have security concerns about the availability of the archive to unauthorized users.

- **com.arbortext.e3.transactionArchiveDirectory**

  Specifies the directory where archive entries are stored. The default is the subdirectory named **`transactionArchive`** in the temporary directory specified by **com.arbortext.e3.tempFileDirectory**.

Managing Queued Transaction Processing

- **com.arbortext.e3.transaction.maxConcurrentQueuedTransactions**

  Specifies the maximum number of queued transactions that may run concurrently.

  There are two other possible values, `-1` and `0`.

  If it's set to `-1`, the default, there is no specified limit, and the Queued Transaction Scheduler will execute as many queued transactions as it can match with idle Arbortext PE sub-processes.

  If it's set to `0`, no queued transactions will be allowed whatsoever. Transactions may still be queued, but they will never be executed.

- **com.arbortext.e3.scheduler.threadInterval**

  Specifies the number of seconds between execution of the Queued Transaction Scheduler. The thread checks for queued transactions ready for processing and idle Arbortext PE sub-processes, subject to the value specified for the global **maxConcurrentQueuedTransactions** parameter and the **maxConcurrentQueuedTransactions** parameter specified for each Arbortext PE sub-process pool. The default is `1` second.

  You should not need to change this value.

In addition, the following parameters apply specifically to a site where Arbortext Editor clients are using Arbortext PE server to fulfill publishing requests. These parameters control whether the Arbortext Editor client is allowed to queue a request and to, optionally, submit information to receive notification about the completed transaction.

The settings for these parameters are included in the Publishing Configuration report that is sent to Arbortext Editor clients using Arbortext Publishing Engine for publishing. The Publishing Configuration document establishes the queuing policies for Arbortext Editor clients. Refer to *Configuration Guide for Arbortext Publishing Engine*.

- **com.arbortext.e3.queueCompositionOperations**

Specifies whether Arbortext Editor clients are allowed to submit queued publishing requests. The values are `always`, `optional`, and `never`. The value `never` means Arbortext Editor clients will never be permitted to queue a request. If **queueCompositionOperations** is not specified in **`e3config.xml`**, the default is `never`. However, **`e3config.xml`** ships with this value set to `optional`, which lets the Arbortext Editor user choose whether to queue a request.

- **com.arbortext.e3.compositionIdentificationPolicy**

  Specifies a valid HTTP query parameter name used to identify the user making a publishing request. If it's not specified, the default value is `header`. The **`e3config.xml`** file ships with this value set to `header`.

- **com.arbortext.e3.compositionEmailPolicy**

  Specifies an HTTP query parameter name that will be used to provide an email address to Arbortext Publishing Engine. This is the parameter that will specify the email address where the Arbortext PE server will send an email notification, if a notifier is configured. If the query name is not specified, the default value is an empty string `""`, which means no email will be sent.

Locating the Arbortext PE sub-process Installation

The **com.arbortext.e3.epicInstallation** parameter tells the Arbortext PE Request Manager where to find the ***`PE_HOME`*** root installation directory. From this directory, Arbortext PE Request Manager can locate the Arbortext PE sub-processes. By default, the Arbortext PE Request Manager looks three directory levels above (`../../..`) the location of the Arbortext Publishing Engine web application directory where **`e3config.xml`** is located (**`e3/e3/WEB-INF`**). If you move the Arbortext Publishing Engine web application directory from its default location in the ***`PE_HOME`*** install tree, make sure to set this parameter to reflect the new location.

The Allowed Functions List

The allowed functions list, defined in e3config.xml by the element **AllowedFunctions** contains a series of **ClientFunction** elements. Each **ClientFunction** element must have **pattern** and **type** attributes. The **pattern** attribute specifies a string, in which the wildcard `?`matches a single character and `*` matches zero or more characters. The **type** attribute must specify one of the four Arbortext PE Application types, `java`, `javascript`, `acl`, or `vbscript`.

When the Arbortext PE Request Manager receives an **f=acl**, **f=java**, **f=javascript**, or **f= vbscript** request, it consults the allowed function list to see if the function or class query parameter matches an entry in the allowed function list. If there is no match, then the operation is disallowed and the Arbortext PE Request Manager returns an error to the client.

Dynamic Component Initialization

Each dynamic component must be coded as a Java class that implements one of the interfaces listed in Arbortext PE Request Manager Components on page 40. Each object must provide an **init** method that takes as parameters the Arbortext Publishing Engine Request Context object being constructed and the **com.arbortext.e3cf** object that describes the object being initialized. Because the **init** method is called while the Arbortext Publishing Engine Request Context is being constructed, not all of the Arbortext Publishing Engine Request Context functionality is available. In particular, the Arbortext PE sub-process pools are initialized after many of the Dynamic Components have been initialized, so no dynamic component except an initializer can obtain an Arbortext PE sub-process during its initialization. For example, a cache manager that wants to use an Arbortext PE sub-process to obtain information can't do so; instead, you must write an initializer that can run after the Arbortext PE sub-process pools have been initialized.

Arbortext PE sub-process Pool Initialization

Arbortext PE sub-process pools are started during Arbortext Publishing Engine Request Context initialization. When an Arbortext PE sub-process pool starts, it reads its parameters and then launches a background work thread. The thread then launches Arbortext PE sub-processes, terminates old ones, and controls other behaviors as defined by the pool parameters in **`e3config.xml`**.

Remember that Arbortext PE sub-processes start asynchronously to the rest of initialization, so there’s no guarantee that an Arbortext PE sub-process will be initialized and ready to accept requests at any particular time. However, if an initializer calls the Arbortext Publishing Engine Request Context’s **allocateE3Subprocess** method, the call will block until an Arbortext PE sub-process has started and is ready to receive communication. This means that initializers can use an Arbortext PE sub-process if necessary.

Initializer Objects

Initializers are dynamically-loaded objects that perform initialization after all other Dynamic Components and all Arbortext PE sub-process pools have been initialized. Every initializer must implement the **com.arbortext.e3.E3Initializer** interface, which has a single required method **init(*context*, *descriptor*)**. The ***context*** is the Arbortext Publishing Engine Request Context object and ***descriptor*** is a **com.arbortext.e3cf** descriptor containing the initializer's parameters obtained from **`e3config.xml`**.

Every initializer is defined in **`e3config.xml`** as either blocking (by setting **deferred** attribute to `no`) or deferred (by setting **deferred** attribute to `yes`). After starting the Arbortext PE sub-process pools, the Arbortext PE Request Manager loads and invokes the **init** method of every blocking initializer in the order specified in **`e3config.xml`**. Just before returning to the servlet container, the Arbortext PE Request Manager starts a background thread that loads and invokes each deferred initializer in the order specified in **`e3config.xml`**. You can circumvent this serial process by implementing a single initializer that starts any number of background threads, with each thread performing initialization tasks.

Request Processing

After **com.arbortext.e3.E3servlet.init** returns, the servlet container can start sending incoming HTTP requests to the Arbortext PE Request Manager. As mandated by the Java Servlet standard specification, the servlet container calls the routines **doGet** or **doPost** (depending on the type specified in the HTTP request), and passes standard objects representing the request and response as parameters.

Basic Flow of Control for Request Processing

Each time it receives an HTTP request from its servlet container, the Arbortext PE Request Manager performs the following steps to process the request:

1. First, the Arbortext PE Request Manager assigns a transaction ID and creates a transaction directory. Then it saves the request descriptor into the transaction directory. If the request includes a body, the Arbortext PE Request Manager reads the request body from the network and saves it to disk in the transaction directory.
2. Next, the Arbortext PE Request Manager iterates through the cache managers and calls each cache manager's **com.arbortext.e3.E3CacheManager.search** method. Possible results:
    - If a cache manager returns a cached response, stop iterating and return the response to the client.
    - If a cache manager doesn't contain the response but wants to cache it, remember that and continue the iteration.

      If more than one cache manager is willing to store the response for a request, once one is generated, the response will be placed in the first cache manager willing to cache the response.
3. If no cache manager provides a response, the Arbortext PE Request Manager iterates through every queue manager and calls each queue manager's **com. arbortext.e3.E3QueueManager.service** method. Possible results:
    - If a queue manager can't queue the request, continue the iteration.
    - If a queue manager returns a response, stop the iteration and return the response to the client.

      Presumably, the response will state that the request was queued for execution later and provide the Transaction ID so that the client can check for completion and retrieve the transaction result. The Arbortext Queue Manager that ships as part Arbortext Publishing Engine does this. Custom queue managers must return something, but the content is up to the application.

Also, presumably, the queued transaction will execute at a later time, as explained in Basic Flow of Control for Queued Requests on page 50.

4. If no cache manager or queue manager provided a response, the Arbortext PE Request Manager iterates through every Request Handler and calls each request handler's **com.arbortext.e3.E3RequestHandler.service** method.

   If a request handler returns a response, the Arbortext PE Request Manager places the response in a cache manager (if one indicated an interest), then stops the iteration and returns the response to the client.

5. If no cache manager, queue manager, or request handler generated a response, the Arbortext PE Request Manager generates an error response stating that it was unable to process the request.

6. At any point in processing, when a response gets generated, the Arbortext PE Request Manager writes the response descriptor and the response body (if any) to the transaction directory.

   If an error is detected during processing, the error is translated into a response and saved to the transaction directory.

7. At the end of processing, the response descriptor and response body (if any) are transmitted back to the client and the request is considered complete.

8. The Arbortext PE Request Manager then passes the completed request to its Transaction Archive subsystem, which will optionally compress the transaction directory into a single file and save the file in the transaction archive, and then delete the transaction directory.

Basic Flow of Control for Queued Requests

This section is based on the behavior of the Arbortext Queue Manager and Arbortext Queue that are shipped as part of Arbortext Publishing Engine. A custom queue might implement an entirely different scheme for queuing and processing transactions. A queue manager's only required behavior is:

- returning a null response if it doesn't queue a transaction
- returning some kind of response if it does queue the transaction

The basic flow of control for a queued transaction is identical to that of an immediate transaction until the transaction is passed to a queue manager. The *Basic Flow of Control for Queued Requests* describes what happens in step 3 of Basic Flow of Control for Request Processing on page 49.

1. If the Queue Manager decides to queue the transaction, it passes the transaction to each **queue** in the order they're defined in **`e3config.xml`**, asking each **queue** whether it is willing to queue the transaction.

If no queue accepts the transaction, the Queue Manager returns an error response to the Arbortext PE Request Manager and it continues to the Request Handlers (picking up at step 4 of Basic Flow of Control for Request Processing on page 49).

If the Queue Manager finds a queue willing to accept the transaction, it places the transaction on the queue and sets a response to be returned to the client.

2. Once a transaction is placed on a queue, the queue works with the Queued Transaction Scheduler to determine when the transaction will be processed. The Queued Transaction Scheduler is a daemon thread that awakens periodically and asks each Arbortext PE sub-process pool to allocate an otherwise idle Arbortext PE sub-process. If the allocation request succeeds, the Queued Transaction Scheduler asks each queue whether it has a transaction ready to execute that can run in the Arbortext PE sub-process pool of the allocated Arbortext PE sub-process. If so, the Queued Transaction Scheduler starts the transaction execution. If not, the Queued Transaction Scheduler releases the Arbortext PE sub-process and continues to the next Arbortext PE sub-process pool.
3. To execute a queued transaction, the Queued Transaction Scheduler offers the transaction to each Request Handler in the order they’re defined in **`e3config.xml`**.

   If a Request Handler accepts the transaction for processing, the request Handler processes the request just as it would if it received a request for immediate processing.

   If no Request Handler is willing to take the queued request, the Queued Transaction Scheduler generates an error response for the transaction.

   After either of these events, the transaction is marked as complete.
4. Once a queued transaction has been completed, its Transaction Directory contains the same response that would have been returned to the client had the transaction not been queued (an immediate transaction). An HTTP client can retrieve the response by using the web interface or issuing an **f=qt-retrieve** request. If the response is retrieved, the Arbortext PE Request Manager notes the retrieval.

   The transaction directory is deleted from the Arbortext PE server according to which of the following occurs:

   - If a client retrieves the transaction response, the transaction is deleted after the time indicated by the global parameter **com.arbortext.e3.maxRetrievedTransactionAge** set in **`e3config.xml`**.
   - If no client retrieves the transaction response, the transaction is deleted after the time indicated by the global parameter **com.arbortext.e3.maxCompletedTransactionAge** set in **`e3config.xml`**.

More About Queues and the Queued Transaction Scheduler

The operation of the Queued Transaction Scheduler is constrained by:

- the **com.arbortext.e3.maxConcurrentQueuedTransactions** global parameter, which limits the number of queued transactions that can be executed simultaneously.
- each Arbortext PE sub-process pool's **maxConcurrentQueuedTransactions** attribute, which limits the number of subprocesses from that pool that can be allocated to executing queued transactions.

Each queue object determining whether transactions are eligible for execution and for determining the order in which their transactions execute.

The Arbortext Queue shipped with the product can be configured to :

- allow transactions to execute only on specified days as well as during specified times of day.
- require that a transaction does not start until transactions higher in the queue have started or until transactions higher in the queue have finished executing.
- limit the number of transactions it contains that can execute simultaneously.
- sort transactions into priority order, either based upon an explicit parameter when the transaction's request is transmitted to Arbortext PE Request Manager or as specified from a Queuing administration web page.

It is possible to configure several separate instances of the Arbortext Queue in **`e3config.xml`**, each accepting different transactions and having different attributes, so that different kinds of transactions are executed according to different rules. All the queuing configuration parameters are explained in *Configuration Guide for Arbortext Publishing Engine*.

Arbortext Publishing Engine Request Context Object

The Arbortext Publishing Engine Request Context is an object that implements the Java interface **com.arbortext.e3.E3RequestContext**. The Arbortext Publishing Engine Request Context object is passed to the initialization method of every dynamic component object. Dynamic Components should save the object for use during request processing. The Arbortext Publishing Engine Request Context object offers the following capabilities:

- scheduling deferred cleanup tasks
- allocating Arbortext PE sub-processes to requests
- creating mock HTTP requests for internal use

- allocating temporary files
- accessing the allowed function list
- accessing global parameters
- accessing all Dynamic Components
- accessing all Arbortext PE sub-process pools
- accessing the Java Servlet configuration and context objects
- accessing version information for both the Arbortext PE Request Manager and Arbortext PE sub-processes
- providing log status and debug information

Transaction Management

Transaction management allocates transaction IDs and transaction directories when the Arbortext PE Request Manager receives a request to process, and it archives or deletes transaction directories after a request is complete. The transaction configuration parameters are described in detail in *Configuration Guide for Arbortext Publishing Engine*.

Transaction IDs, Names, and Directories

When the Arbortext PE Request Manager receives a new request, it allocates a unique transaction ID, creates a transaction directory (for active transactions), and saves the request, including the request body, in the transaction directory. Transaction IDs are positive integers that are not reused, even when Arbortext Publishing Engine is restarted.

Each transaction can have an optional name to specify descriptive text to aid in identifying it. Arbortext Editor clients can specify a transaction name using the **Queued Transaction Names** dialog box (available from **Publishing Engine** category of **Tools ▸ Preferences**) or the **Transaction Name** field on the **File ▸Publish** set of dialog boxes. The query parameter, `transaction-name`, allows an application to provide a transaction name as part of its HTTP request. The `transaction-name` value can include a string `$t`, to be replaced by the unique transaction ID assigned on the Arbortext PE server. Names can only be unique if they include the `$t` string, so descriptive text alone is not a reliable way to make a name unique.

Each transaction has its own directory and contains only information associated with that transaction. Transaction directories are all created as subdirectories of the location specified by the parameter **com.arbortext.e3.transactionDirectory**. Each directory name follows the format **`rq_`*`n`***, where ***n*** is the transaction ID.

When the Arbortext PE Request Manager finishes processing an immediate transaction and the transaction meets the criteria specified by the parameter **com.arbortext.e3.**

transactionArchive.selector, the transaction directory is copied (and compressed) into a transaction archive entry. Then the transaction directory and its contents are deleted.

The transaction directory for a queued transaction is not deleted immediately to give the client that submitted the transaction time to retrieve the result. The transaction is deleted either upon explicit request by the user or when one of the following criteria is satisfied:

- If a client retrieves the transaction response, the transaction is deleted after the time indicated by the global parameter **com.arbortext.e3.maxRetrievedTransactionAge** set in **`e3config.xml`**. The default is 24 hours.
- If no client retrieves the transaction response, the transaction is deleted after the time indicated by the global parameter **com.arbortext.e3.maxCompletedTransactionAge** set in **`e3config.xml`**. The default is 7 days.

If the transaction is to be archived in either case, it is archived just before the transaction directory is deleted.

Transaction States

Every transaction handled by Arbortext Publishing Engine has an associated transaction state that describes its current status. The transaction lifecycles are as follows:

- An immediate request transaction follows the lifecycle:

  Initializing → Waiting → Processing → Complete
- A queued request transaction follows the lifecycle:

  Initializing → Queued → Processing → Complete or Cancelled
- `initial`

  When Arbortext PE Request Manager receives a request and is creating the transaction ID and transaction directory, it places the transaction in the `initializing` state, where it remains while the Arbortext PE Request Manager passes the transaction to each cache manager and queue manager.
- `waiting`

  The transaction is waiting to be allocated to an Arbortext PE sub-process (for immediate requests).

  While a Request Handler attempts to find and allocate an Arbortext PE sub-process willing to process the transaction, the transaction state is set to `waiting`.

  A queued transaction will never enter the `waiting` state, because an Arbortext PE sub-process is allocated before the transaction would start executing.
- `queued`

  The transaction was placed in a queue and is waiting to be executed by the Queued Transaction Scheduler.

If a Queue Manager finds a queue that accepts an incoming request, the transaction state is set to `queued`.

- `processing`

  The transaction is expected to be handled by an Arbortext PE sub-process.

  If no Cache manager or Queue Manager supplies a response, the transaction state is set to `processing` and the transaction is passed to the Request Handlers.

  While a Request Handler attempts to allocate an Arbortext PE sub-process, the transaction state is set to `waiting`. Then it's set back to `processing` when the Arbortext PE sub-process, has been allocated.

  For queued transactions, the Queued Transaction Scheduler decides when to execute a transaction. When it decides a transaction can be processed, the transaction state changes from `queued` to `processing`.

- `complete`

  The transaction is finished. A completed transaction can be completed successfully or completed with errors. Note that if the results is an error report rather than the expected document, the transaction is still considered complete.

  If a Cache Manager supplies a response to an incoming request, the transaction state is set to `complete`.

  If a Request Handler processes the transaction, or if no Request Handler is able to do so, the transaction state is set to `complete`.

  For a queued transaction, the transaction state changes to `complete` after it's processed. If a queued transaction is cancelled, either before execution starts or during execution, its state is set to `cancelled`

  A transaction can also expire while waiting for an Arbortext PE sub-process allocation (if it's an immediate transaction).

- `cancelled`

  For queued transactions, the transaction has been cancelled.

  Before or during the execution of a queued transaction, a transaction may be cancelled by an Arbortext Publishing Engine administrator from the queuing web page or by an **f=qt-cancel** request. In either case, the transaction state is set to `cancelled`. If the transaction has not yet started execution, it will never start. If it is running, then its Arbortext PE sub-process will be terminated and a new Arbortext PE sub-process started, if necessary (to satisfy the minimum Arbortext PE sub-process configured in the appropriate Arbortext PE sub-process pool) and the transaction results, if any, will be discarded.

  The Arbortext PE Request Manager will fabricate a response for the request and send an HTML page stating that the transaction was cancelled. The `cancelled` state is like the `complete` state in the sense that no further processing will be performed.

The transaction state does not indicate success or failure of a transaction.

Holding Transactions

Normally, when a transaction is placed on a queue, it is eligible for execution as soon as the Queued Transaction Scheduler is able to find an idle Arbortext PE sub-process in an Arbortext PE sub-process pool that can accept the transaction. However, a transaction may be marked `hold`, which makes it ineligible for execution until the `hold` flag is cleared. A transaction may be held or released at any time by an Arbortext Publishing Engine administrator using the **Transaction List** web page. A transaction may also be automatically held by a queue that is configured to do so. If a transaction is executing, setting its `hold` flag will have no effect unless the Arbortext Publishing Engine is restarted before the transaction finishes executing. If the Arbortext Publishing Engine is restarted, the `hold` flag will prevent the Queued Transaction Scheduler from executing the transaction. Transaction configuration and administrative actions are explained in *Configuration Guide for Arbortext Publishing Engine*.

Transaction Archive Entries

Each entry in the transaction archive is a zip archive that contains all of the files that were in the transaction directory at the time the transaction was deleted from the active transaction directory. Each entry has a name of the form **`rq_n.zip`**, where ***n*** is the transaction ID. Transaction archive entries are all located in the directory specified by the parameter **com.arbortext.e3.transactionArchiveDirectory** in **`e3config.xml`**.

Limiting the Size of the Transaction Archive

Even as a zip archive, a transaction archive entry may consume a substantial amount of disk space. There are three mechanisms available to avoid filling the file system.

- Filter which transactions are stored in the archive by setting the **com.arbortext.e3.transactionArchive.selector** parameter.
  - `none` saves no entries
  - `error` saves error entries
  - `log` (the default) saves error entries, as well as requests with log entries or intermediate files
  - `all` saves all requests, including successful requests
- Control how long a transaction will remain in the archive by setting the **com.arbortext.e3.transactionArchive.maxAge**. Transactions older than the specified duration will be deleted.
- Specify the maximum amount of disk space that the archive can occupy by setting the **com.arbortext.e3.transactionArchive.maxSize** parameter. If the archive

grows larger than the specified limit, oldest transactions are deleted to bring the spaced consumed within the maximum level.

Disabling the Transaction Archive

The transaction archive will contain request and response data for each transaction it contains. If this poses a security risk, you should disable the archive entirely by setting the **com.arbortext.e3.transactionArchive.selector** parameter to `none`.

Programming Considerations

Each time a transaction finishes executing, the transaction archive management thread may try to copy all of the files in the transaction's directory to a transaction archive entry (if configured to do so) and then tries to delete the transaction directory. If any file in the transaction directory is open for writing or reading, the transaction archive management thread will be unable to delete the transaction directory and unable to copy the file into the transaction archive. An open, locked file means the management thread will be unable to delete the transaction directory.

The most likely cause of a locked file is an application running in an Arbortext PE sub-process that opens a file and fails to close it. When you develop a custom application, you should check the Arbortext PE Request Manager's log for errors stating that files you open could not be read or deleted as part of testing your application. The management thread will keep trying to copy and delete the files in question at ten second intervals, and the messages in the log will be repeated each time it does so, so the log entries should be apparent.

Retrieving Information from the Transaction Archive

You can obtain summary information about the transaction archive using the **Transaction Archive** link from the Arbortext Publishing Engine index page. From the summary page, you can retrieve detailed information about any archived transaction, delete transactions, and retrieve one or more transactions as zip archives to review.

Arbortext PE sub-process Pool Management

A subprocess pool is a collection of identical Arbortext PE sub-processes. Each Arbortext PE sub-process pool has a unique name, a collection of attributes that describe the pool's behavior, a set of parameters that are passed to every Arbortext PE sub-process in the pool, and a collection of rules that determine which HTTP requests can be processed by Arbortext PE sub-processes in the pool.

Support for multiple Arbortext PE sub-process pools offer two major advantages:

- Pools of Arbortext PE sub-processes can have different parameters, making it possible to have groups of Arbortext PE sub-processes with different behaviors.
- Arbortext PE sub-processes pools can be configured to reserve performance capacity by ensuring that requests of a particular type don't consume all available Arbortext PE sub-processes.

Each Arbortext PE sub-process pool provides the following capabilities:

- ensuring that a configured minimum number of Arbortext PE sub-processes are running.
- starting additional Arbortext PE sub-processes (up to a configured maximum number) upon request.
- limiting the number of Arbortext PE sub-processes that can be used to process queued transactions.
- terminating any Arbortext PE sub-process that fails to respond to a request from the Arbortext PE Request Manager after a configured period of time.
- terminating Arbortext PE sub-processes that have been idle for more than a configured maximum period of time.
- terminating an Arbortext PE sub-process and starting a new one (between requests to avoid disruption of processing) if an Arbortext PE sub-process has been running for more than a configured maximum period of time.

  Terminating an Arbortext PE sub-process avoids possible consumption of resources that could accumulate over time.
- returning messages that a request has failed after a configured interval if all Arbortext PE sub-processes are busy and the maximum number of Arbortext PE sub-processes are running.

Arbortext PE sub-process Allocation

There are several ways to allocate an Arbortext PE sub-process. The most common way to allocate an Arbortext PE sub-process follows:

1. A dynamic component makes a call to the **com.arbortext.e3.E3RequestContext. allocateE3Subprocess** method; the call takes the HTTP request as a parameter. The **allocateE3Subprocess** method queries every configured Arbortext PE sub-process pool, calling each pool's **com.arbortext.e3.E3SubprocessPool. testRequest** method. The **testRequest** determines whether the Arbortext PE sub-processes in that pool are supposed to handle requests like the one specified in the HTTP request.

   If the response is `true`, the subprocess pool attempts to allocate an Arbortext PE sub-process to service the request according to the next steps.

If the answer is `false`, **allocateE3Subprocess** continues to query the next Arbortext PE sub-process pool in the list.

2. If a `true` response is returned for **testRequest**, then **allocateE3Subprocess** calls the Arbortext PE sub-process pool's **allocate** method. The **allocate** method eventually either returns an Arbortext PE sub-process object or throws an exception after processing the following:

   a. The Arbortext PE sub-process pool begins the allocation process by checking for any idle Arbortext PE sub-processes. If so, the one idle for the shortest time is allocated. If not, the Arbortext PE sub-process pool checks whether the configured maximum number of Arbortext PE sub-processes are running. If not, it starts a new one and, when it is ready to accept requests, returns the Arbortext PE sub-process object.

   b. If all Arbortext PE sub-processes in the pool are busy and the pool is configured to cascade to another pool, **allocate** returns the result of the cascaded pool's **allocate** method.

   c. If all Arbortext PE sub-processes in a pool (and its cascaded pools) are busy and no additional ones can be started, the **allocate** method waits until either an Arbortext PE sub-process is no longer busy or the waiting period expires (explained in The maxSubprocessWait Attribute on page 66). If an Arbortext PE sub-process is freed during the waiting period, **allocate** returns its object.

3. If, after this processing, no Arbortext PE sub-processes are found, **allocate** throws an exception and transmits an “All Arbortext PE sub-processes are currently busy” error.

Arbortext PE sub-processes are allocated to process queued transactions by the Queued Transaction Scheduler.

1. The Queued Transaction Scheduler begins by asking an Arbortext PE sub-process pool for an idle Arbortext PE sub-process. The Arbortext PE sub-process pool may start a new Arbortext PE sub-process if there are fewer than the maximum set by the **maxSubprocesses** parameter (see The minSubprocesses and maxSubprocesses Attributes on page 64). However, the scheduler will not wait for an Arbortext PE sub-process to become free if the maximum number of Arbortext PE sub-processes are running and none are idle.

2. When the Queued Transaction Scheduler starts the execution of a queued transaction, it associates the allocated Arbortext PE sub-process with the transaction, so that any request to allocate an Arbortext PE sub-process by the queued transaction is satisfied by the transaction that was just allocated. Any attempts to acquire an Arbortext PE sub-process by a queued transaction are ignored. The allocated Arbortext PE sub-process is released by the Queued Transaction Scheduler after the queued transaction finishes executing.

An alternate way to allocate an Arbortext PE sub-process to an immediate request is by using any dynamic component running in the Arbortext PE Request Manager to allocate an Arbortext PE sub-process by querying the list of configured pools directly. The dynamic component can call the pool's **allocate** method, which bypasses the check for whether a pool should process a particular request. Use this approach with caution.

Matching Arbortext PE sub-processes with Requests

Whenever a dynamic component needs an Arbortext PE sub-process to service a request, it calls the **com.arbortext.E3RequestContext.allocateE3Subprocess** method, passing the request as a parameter. **allocateSubprocess** in turn, calls the method **com.arbortext.e3.E3SubprocessPool.testRequest** for each Arbortext PE sub-process pool until a pool returns `true`, again passing the request as a parameter. The `default` Arbortext PE sub-process pool can handle every request; its response to **testRequest** is always `true`. Every other Arbortext PE sub-process pool evaluates the request against the Arbortext PE sub-process pool's configured **TestSet**.

A **TestSet** is the equivalent of a parenthetical boolean expression containing any number of **Test** elements grouped by **And** and **Or** operators. Each **Test** element refers to a Request Selector. A Request Selector is a dynamic component that examines a request and determines whether the request matches a predefined criteria.

Each **Test** element specification must include a **name** attribute that identifies a **RequestSelector**. Several **Test** elements may reference the same **RequestSelector**.

Each **RequestSelector** is an object implementing the **com.arbortext.e3.E3RequestSelector** interface, which has a single method called **test**. The **test** method takes an HTTP request as a parameter and returns `true` or `false`.

When configuring a **TestSet**, keep in mind:

- A **TestSet** or **And** element returns `true` if every **Test**, **And**, and **Or** element it contains returns `true`. Remember that **And** elements can be nested, so a `true` response for a lower level test may be eventually superseded by a higher level `false` response after all the **And** conditions are evaluated.
- An **Or** element returns `true` if any **Test**, **And**, or **Or** element returns `true`.

Given these assumptions, the **testRequest** method passes the current HTTP request to the test structure built from the subprocess pool's **TestSet** element. The **TestSet** structure is evaluated like a logical expression that returns the result from **testRequest** (`true` or `false`).

Arbortext PE sub-process Pool Work Thread

Each Arbortext PE sub-process pool starts a background work thread that runs periodically and attempts to perform routine maintenance. The interval between runs is configured in the **workThreadInterval** attribute of each **SubprocessPool** element in **`e3config.xml`**. The default is every `5` seconds. Each time it runs, a work thread checks for the following items. Descriptions of each item follows.

- unusable Arbortext PE sub-processes
- Arbortext PE sub-processes that should be terminated
- Arbortext PE sub-processes that have exceeded the allowed idle or lifetime period
- deferred requests to delete directories
- whether a new Arbortext PE sub-process should be started

Detecting Hung Arbortext PE sub-processes

Each time the Arbortext PE Request Manager makes a call to an Arbortext PE sub-process, the pool records the time. The work thread checks every busy Arbortext PE sub-process in its pool to see whether the time elapsed from the start of the operation has exceeded the configured **maxBusyInterval** attribute value (explained in The maxBusyInterval Attribute on page 65). If it has, the work thread sets a flag. Later, the work thread notifies the operating system to terminate these flagged Arbortext PE sub-processes. As a result, the Arbortext PE Request Manager request thread waiting for a response from an unusable Arbortext PE sub-process receives an error.

Detecting unusable Arbortext PE sub-processes avoids Arbortext PE sub-process software malfunctions that can:

- permanently consume Arbortext PE Request Manager capacity (infinite loops, indefinite waits)
- result in no response being returned to a waiting client

However, it is possible that an Arbortext PE sub-process can't respond before the **maxBusyInterval** elapses, because it is processing a request that takes a long time. During your site setup, evaluate the requests likely to be submitted, and set the **maxBusyInterval** to a value that will allow lengthy jobs to complete.

Terminating Unusable Arbortext PE sub-processes

Each Arbortext PE sub-process has a flag which is set any time it returns a fatal error. When a work thread detects an unusable Arbortext PE sub-process, it sets a flag that records the state of the Arbortext PE sub-process. At a later time, the work thread scans all running Arbortext PE sub-processes and requests the operating system to terminate any with the `hung` or `fatal` flag set to `true`.

Terminating Expired Arbortext PE sub-processes

Each time the work thread runs, it examines idle Arbortext PE sub-processes in its pool. If any idle Arbortext PE sub-process is older than the configured **maxLifetime** period (explained in The maxLifetime Attribute on page 66), the work thread terminates the Arbortext PE sub-process.

In addition, if the longest idle Arbortext PE sub-process has been idle longer than the configured **maxIdleInterval** period, the work thread terminates that one as well, provided there are more than the minimum number of Arbortext PE sub-processes running.

In either case, if the number of activated Arbortext PE sub-processes drops below the minimum configured number of Arbortext PE sub-processes (explained in The minSubprocesses and maxSubprocesses Attributes on page 64), the work thread starts a new Arbortext PE sub-process.

Deleting Temporary Directories

When a work thread terminates an Arbortext PE sub-process, the pool queues deletion of its temporary directory. Each pool maintains a list of deferred directory deletion requests. However, directory deletion is performed after the Arbortext PE sub-process has been removed from the operating system. The deletion is attempted every time the work thread runs until the deletion succeeds. Temporary directory deletion releases the disk space used for processing requests that the Arbortext PE sub-process had consumed.

Starting New Arbortext PE sub-processes

A new Arbortext PE sub-process can be started when the request load increases and there are fewer than the configured maximum number running. In addition, the pool work thread checks whether the configured minimum number of Arbortext PE sub-processes are running. If fewer than the minimum number are running, the work thread starts another Arbortext PE sub-process.

Communicating with Arbortext PE sub-processes

When **com.arbortext.e3.E3RequestContext.allocateE3Subprocess** allocates an Arbortext PE sub-process, it returns a reference to an object that implements the **com.arbortext.e3.E3Subprocess** interface. This interface has three methods.

- **deallocate** returns the Arbortext PE sub-process to its pool for allocation to another request.
- **executeCommand(String *cmd*)** method passes the ACL command ***cmd*** to the Arbortext PE sub-process for execution by the ACL command interpreter. This method does not return a value, but it will throw an exception if execution fails.
- **evaluateFunction(String *function*)** method passes the ACL expression ***function***, to the Arbortext PE sub-process for evaluation. This method returns a string containing the result of the function. The ***function*** must include a function name (including package name, if appropriate), parentheses, and parameters, all expressed as strings.

Arbortext PE sub-process Deallocation

Once it has been allocated to a dynamic component, an Arbortext PE sub-process can only be used by the code that allocated it. To release the Arbortext PE sub-process, the component that called the **allocate** method must subsequently call the **com.arbortext.e3. E3Subprocess.deallocate** method. To ensure the Arbortext PE sub-process is released, placing the**deallocate** call inside the **finally** block, which releases it regardless of any exceptions that might be thrown in the **try** block.

Deallocation is not optional. If a dynamic component fails to deallocate an Arbortext PE sub-process, the Arbortext PE sub-process will never become usable for other purposes.

Arbortext PE sub-process Pool Attributes

The attributes that define the Arbortext PE sub-process pool's behavior are explained in the following sections. The complete set of parameters and attributes for Arbortext PE sub-process pools are documented in *Configuration Guide for Arbortext Publishing Engine*.

The id Attribute

The **id** attribute of an Arbortext PE sub-process pool provides an optional unique identifier. This attribute only needs to be specified if another Arbortext PE sub-process pool refers to this one in its **cascade** attribute.

The cascade Attribute

This optional attribute specifies the value of the **id** attribute for another Arbortext PE sub-process pool that can take overflow of processing requests. The **cascade** attribute allows one pool to take the overflow from another to improve servicing requests. If the Arbortext PE sub-process pool being queried determines that it can accept an HTTP request, but no Arbortext PE sub-processes are available and no additional Arbortext PE sub-processes can be started, the pool will pass the request to the pool identified by its **cascade** attribute. You can configure a series of pools to cascade from one to another, where pool A can cascade to pool B, pool B can cascade to pool C, and so on.

The enabled Attribute

This optional attribute specifies whether the Arbortext PE sub-process is available for taking requests. The default value is `yes`. If **enabled** set to `no`, the Arbortext PE sub-process pool is ignored and it doesn't start any Arbortext PE sub-processes regardless of its parameter settings. You can set **enabled** to `no` to temporarily disable a pool without needing to delete the pool entry from the **`e3config.xml`** configuration file.

The default Attribute

The **default** attribute has a default value of `no`. If set to `yes`, the Arbortext PE sub-processes can service any HTTP request. If any **TestSet** elements are defined for this Arbortext PE sub-process pool, they are ignored.

As described in Arbortext PE sub-process Allocation on page 58, when a request handler calls **E3RequestContext.allocateE3Subprocess**, it passes the current HTTP request. The Arbortext Publishing Engine Request Context passes the HTTP request to each Arbortext PE sub-process pool defined in **`e3config.xml`**, asking whether it can fulfill the request. The Arbortext PE sub-process pool with **default** set to `yes` always responds that it can service the request.

Be sure to give only one Arbortext PE sub-process pool the `yes` value for the **default** attribute. The pools are queried in the order they're configured in **`e3config.xml`**, so make sure it's the last pool configured in the **SubprocessPools** section of **`e3config.xml`**. Because the **default** pool handles all HTTP requests, any pool configured after it will never be asked to handle a request.

The minSubprocesses and maxSubprocesses Attributes

These attributes define the number of Arbortext PE sub-processes running in a pool. The **minSubprocesses** attribute sets the minimum number of Arbortext PE sub-processes that will be started and continue to run. The **maxSubprocesses** attribute sets the maximum number of Arbortext PE sub-processes that can run in a pool when conditions controlled by other configuration settings are met.

When an Arbortext PE sub-process pool initializes, it starts Arbortext PE sub-processes until the minimum number have been started. If a pool terminates an Arbortext PE sub-process and causes the number running to drop below the minimum number configured (refer to The maxIdleInterval Attribute on page 65 and The maxLifetime Attribute on page 66’ for information on configuring terminations), the pool will immediately start a new Arbortext PE sub-process.

The maximum number of Arbortext PE sub-processes must be greater than the minimum number to start additional Arbortext PE sub-processes if needed to satisfy requests. The pool can't exceed the maximum number of Arbortext PE sub-processes, which counts both allocated and idle Arbortext PE sub-processes in the pool.

The default value for both **minSubprocesses** and **maxSubprocesses** is `1`. Specifying a value less than zero or setting **maxSubprocesses** less than **minSubprocesses** is an error.

Ordinarily, set the minimum and maximum numbers of Arbortext PE sub-processes to the same value to optimize performance. An extra Arbortext PE sub-process consumes a certain amount of memory, but even if it is seldom or never asked to service a request, it's a small penalty. If you specify starting a small number of Arbortext PE sub-processes and allocate more as needed, consider the overhead associated with startup and initialization that can delay fulfilling a request.

The maxBusyInterval Attribute

This attribute specifies the number of seconds that the Arbortext PE sub-process pool work thread (explained in Arbortext PE sub-process Pool Work Thread on page 60) will wait for an Arbortext PE sub-process response to be returned to the Arbortext PE Request Manager. The default value is `1800` seconds (30 minutes). If an Arbortext PE sub-process does not respond with a result before this interval elapses, the Arbortext PE sub-process pool assumes that the Arbortext PE sub-process is not usable and terminates it. An error will be returned that is usually transmitted to the HTTP client making the request.

If processing a request could take a significant amount of time, increase the value accordingly to be sure the Arbortext PE sub-processes can produce the result in the allotted time. For example, formatting a large document or one with many graphics could take a long time, and if the wait period elapses, the Arbortext PE sub-process could terminate before it finishes formatting and publishing the document.

If you want to disable the wait process because you know processing requests will take a very long time, you can set this attribute to zero. In this case, the work thread will never terminate an Arbortext PE sub-process for failing to respond.

The maxConcurrentQueuedTransactions Attribute

This attribute specifies the maximum number of Arbortext PE sub-processes that the Queued Transaction Scheduler can use to execute queued transactions. By default, this parameter is set to `-1`, meaning that every Arbortext PE sub-process in the pool can be allocated to the Queued Transaction Scheduler. If this attribute is set to `0`, then no Arbortext PE sub-processes in the pool will ever be allocated to the Queued Transaction Scheduler. You can set **maxConcurrentQueuedTransactions** to `0` to dedicate the Arbortext PE sub-process pool to processing only immediate transactions.

The maxIdleInterval Attribute

This attribute specifies the number of seconds after which an Arbortext PE sub-process will be terminated if it's not allocated to serving a request. Termination only occurs if more than the minimum number of Arbortext PE sub-processes are running. The default value `0` disables termination.

When the request load is sporadic, consider the delay associated with startup and initialization each time the load increases. Disabling this attribute optimizes the ability to service requests and avoids the overhead associated with repeated startups that can delay fulfilling the request. Though an Arbortext PE sub-process consumes a certain amount of memory, it's still a small penalty to have it running when it’s not in use.

The maxLifetime Attribute

This attribute specifies the duration in seconds that an Arbortext PE sub-process can run in its lifetime. After an Arbortext PE sub-process finishes servicing a request, the Arbortext PE sub-process pool checks the time elapsed since the Arbortext PE sub-process was started. After the configured time period has elapsed, an Arbortext PE sub-process will be terminated because it has exceeded this limit. If termination leaves fewer than the configured minimum number of Arbortext PE sub-processes running, a new one is started. The default value is `86400` seconds (one day).

Reaching the maximum lifetime duration won't interrupt a request being processed. Checking the Arbortext PE sub-process elapsed time only happens between requests. The purpose of having a lifetime limit is to release memory or other resources, especially if an Arbortext PE sub-process encounters a problem. Because resource usage can creep over time, setting **maxLifetime** recycles the resources and avoids having an Arbortext PE sub-process that has grown too large to function properly or efficiently.

The pool work thread checks for Arbortext PE sub-processes at regular intervals; refer to Arbortext PE sub-process Pool Work Thread on page 60 for more information. You can disable checking the duration of Arbortext PE sub-processes by specifying `0`.

The maxShutdownInterval Attribute

This attribute specifies the number of seconds that the Arbortext PE Request Manager will wait for an Arbortext PE sub-processes to terminate when the Arbortext PE Request Manager is ordered to terminate by the servlet container (for example, if Tomcat is shut down). The default value is `20` seconds.

You would not want to reduce this value. If it's too small, there might be an interval after Arbortext Publishing Engine termination when some Arbortext PE sub-processes remain running on the system.

The maxSubprocessWait Attribute

This attribute specifies the number of seconds that a thread will wait for allocation of an Arbortext PE sub-process. When a dynamic component calls **com.arbortext.e3.E3RequestContext.allocateE3Subprocess**, it will return immediately if it finds a free Arbortext PE sub-process in a pool that is configured to handle the HTTP request in progress. If no idle Arbortext PE sub-process can be found, **allocateE3Subprocess** will attempt to start a new Arbortext PE sub-process in the appropriate pool, provided the pool has fewer Arbortext PE sub-processes running than allowed by the pool's **maxSubprocesses** attribute. If no Arbortext PE sub-process can be started, then **allocateE3Subprocess** will wait until either an Arbortext PE sub-process is idle and available from the Arbortext PE sub-process pool or until the number of seconds specified by the **maxSubprocessWait** attribute have elapsed.

If the maximum number of seconds have elapsed and no Arbortext PE sub-process is idle, **allocateE3Subprocess** will throw an exception that no Arbortext PE sub-process could be allocated to serve the request. Processing at this point depends upon the dynamic component, but the component should return an error message to the client indicating that the request failed because no Arbortext PE sub-process could be allocated.

The default value is `300` seconds. You can set the value to `0` to disable the thread. Refer to Arbortext PE sub-process Allocation on page 58 for a description of the allocation process.

SubprocessContext Parameter

There is one parameter for the **SubprocessContext** of the **SubprocessPool**, **com.arbortext.e3.initialScript**:

```
Parameter name="com.arbortext.e3.initialScript"  value="filepath"
```

initialScript allows you to set environment variables in an ACL script before loading information from the **`custom`** directory. For example, you could have different **`custom`** directories for each Sub-process pool because **initialScript** is processed before the **`custom`** directory.

You could use the ACL file specified by **initialScript** to set something like:

```
main::ENV['APTCUSTOM']='D:\special_custom'
```

The subprocessEnvironment Attribute

You can specify `subprocessEnvironment` to have Arbortext Publishing Engine set the environment variables of the Arbortext PE sub-processes. To ensure that Arbortext PE sub-processes have the expected values for environment variables, set them using the `subprocessEnvironment` parameter for each Arbortext PE sub-process pool in the **`e3config.xml`**.

If you are using the Tomcat servlet container, be aware that newer versions clear some environment variables, such as **CLASSPATH**, and then set them to values needed by Tomcat prior to starting the Tomcat servlet container.

Any other variables in the Arbortext Publishing Engine environment are passed to its Arbortext PE sub-processes unchanged.

The workThreadInterval Attribute

This attribute specifies when to run the work thread in seconds (for more information on the work thread, refer to Arbortext PE sub-process Pool Work Thread on page 60). The default value is `5` seconds, which is adequate for most sites.

Arbortext PE sub-process Pool Parameters

In addition to the Arbortext PE sub-process pool attributes, a pool may have parameters. Each pool parameter is defined by the **Parameter** element, which is a descendant of the **SubprocessPoolContext** element (which is a descendant of the **SubprocessPool** element) in **`e3config.xml`**.

When an Arbortext PE sub-process pool starts an Arbortext PE sub-process, it passes the Arbortext PE Request Manager global parameters and any Arbortext PE sub-process pool parameters and their values to the Arbortext PE sub-process, making them available to its Arbortext Publishing Engine applications.

Terminating the Arbortext PE Request Manager

The Arbortext PE Request Manager servlet terminates when the servlet container calls the method **com.arbortext.e3.E3servlet.destroy**. The Arbortext PE Request Manager passes the termination request to each Arbortext PE sub-process pool, and each pool sends a signal to every running Arbortext PE sub-process. Idle Arbortext PE sub-processes terminate silently. Allocated busy Arbortext PE sub-processes return an error to the calling thread in the Arbortext PE Request Manager, which is returned to the requesting client. If an Arbortext PE sub-processes is allocated but not yet busy, an error is returned immediately when it starts processing.

Each pool work thread waits for Arbortext PE sub-processes to terminate so it can delete their temporary directories. If all Arbortext PE sub-processes have not terminated before the time configured by the pool's **maxShutdownInterval** attribute (explained in The maxShutdownInterval Attribute on page 66), the work thread terminates, leaving some temporary storage still allocated.

4 Predefined Dynamic Components

The Arbortext PE Request Manager ships with several Dynamic Components installed, including request selectors, a cache manager, a queue manager, and request handlers. While they are distributed as part of Arbortext Publishing Engine, they are built entirely from public interfaces. They are loaded dynamically by the Arbortext PE Request Manager as specified in the **`e3config.xml`** configuration file. You can safely delete from **`e3config.xml`** any Dynamic Components that you do not need. If you do that, however, keep a backup copy of the distributed **`e3config.xml`** file in case you implement one or more of them later.

Predefined Request Selectors

Request selectors are referenced by the **TestSet** of a **SubprocessPool** to determine if an Arbortext PE sub-process pool should service a request. They can be grouped by **And** and **Or** elements within a **TestSet** to set up more complex acceptance criteria.

Individual Request Selectors are invoked when an Arbortext PE sub-process pool evaluates a call to its **testRequest** method to determine whether its Arbortext PE sub-processes should be allocated to a particular HTTP request. **testRequest** is normally called by the Arbortext Publishing Engine Request Context as part of evaluating its **allocateE3Subprocess** method.

The Test Header Match Request Selector

The Test Header Match request selector is implemented by the **com.arbortext.e3. TestHeaderMatch** class. It evaluates an HTTP request and returns `true` if the request includes an HTTP header with a specific value. It accepts two parameters, **header-name**, which specifies the name of the HTTP header to test and **header-pattern**, which specifies a pattern that the HTTP header value must match.

The Test Header Match request selector accepts standard Java patterns for its **header-pattern** parameter. So, for example patterns like `text/*` or `text/???` would both match a header value of `text/xml`. You can specify a Java pattern without wildcards, so specifying a pattern like `text/html` would test for a header value equal to `text/html`.

The Test Query Match Request Selector

The Test Query Match request selector is implemented by the **com.arbortext.e3. TestQueryMatch** class. It evaluates an HTTP request and returns `true` if the request includes an HTTP query parameter with a specific value. It accepts two parameters, **query-name**, which specifies the name of the query parameter to test, and **query-pattern**, which specifies a pattern that the HTTP query parameter value must match. Any standard Java pattern, including regular expressions, can be used for the **query-pattern** parameter value. For compatibility with earlier releases of Arbortext Publishing Engine, if the Arbortext PE Request Manager finds that the value of a **query-pattern** parameter is not a valid Java regular expression and contains “`*`” or “`?`” characters, Arbortext PE Request Manager will treat the pattern as a regular expression, converting “`*`” to “`.*`” and “`?`” to “`.`”.

The Test URI Request Selector

The Test URI Match request selector is implemented by the **com.arbortext.e3. TestURIMatch** class. It evaluates an HTTP reqeust and returns “true” if the URI has a specific value. It accepts one parameter, **uri-pattern**. The Test URI Match request selector returns `true` if the URI of the request matches the pattern specified by the value of **uri-pattern**. Any standard Java pattern can be used for the **uri-pattern** parameter value.

Predefined Cache Managers

A cache manager can store the response to an HTTP request, in anticipation of returning the same response to a later identical request.

The Publishing Configuration Cache, Initializer, and Application

The publishing configuration cache contains information about the publishing-related files installed on the Arbortext PE server. The cache stores an XML document listing the document types, stylesheets, and related configuration information retrieved during initialization. It also stores an HTML version of the XML document, a second XML document containing more detail (which is retrieved by Arbortext Editor when performing a publishing configuration comparison with the Arbortext PE server), and a text log file of the scan process by which the information was discovered. The documents are intended for debugging, and three of them can be retrieved from the Arbortext Publishing Engine Index page.

Support for the cache is implemented in three parts.

1. the cache itself, which stores the documents so that they can be returned quickly.
2. the initializer, which generates the information in the cache when the Arbortext PE Request Manager initializes by obtaining an Arbortext PE sub-process and running the next component.
3. a Java Arbortext PE Application, **com.arbortext.e3ci.Application**, which is running in the Arbortext PE sub-process.

These components provide useful functionality and offer an example of how to implement a cache manager for other purposes. The publishing configuration cache is implemented by the **com.arbortext.e3.CompConfigCache** class. It takes one parameter, **query-function-prefix**, that has a default value of `f`.

When the Arbortext PE Request Manager calls the cache's **init** method, the cache initializes itself to an empty state, and obtains and remembers the value of its **query-function-prefix** parameter (if any).

When the Arbortext PE Request Manager calls the cache's **search** method, the cache manager examines the HTTP request being processed.

- If the request has an HTTP parameter named `bypass-cache` with a value of `yes` (not case sensitive), then **search** returns `not cacheable`.
- If the request has a parameter whose name is equal to the value of the **query-function-prefix** parameter and with the value `java`, a parameter named `class`

with the value **com.arbortext.e3ci.Application**, and a parameter named `type` with a value of `xml`, `html`, or `log`, then **search** returns either `in cache` or `cacheable`, depending upon whether the requested document has been placed in the cache yet.

- Otherwise **search** returns `not cacheable`.

When the Arbortext PE Request Manager calls the cache's **search** method and receives a `cacheable` response, it calls the **cache** method to store the associated response as the XML, detailed XML, HTML, or log document, which will be return for future calls to the **search** method.

The publishing configuration initializer is implemented by the **com.arbortext.e3.CompConfigInit** class. It runs as a deferred initializer so that the rather lengthy publishing configuration scan will not unduly slow the launch of the Arbortext PE Request Manager. When its **init** method is called by the Arbortext PE Request Manager, it uses services provided by the Arbortext Publishing Engine Request Context to create and run the dummy HTTP requests. These requests generate the XML, detailed XML, HTML, and text log forms of the publishing configuration document. The three requests are:

- `f=java&class=com.arbortext.e3ci.Application&type=xml`
- `f=java&class=com.arbortext.e3ci.Application&type=html`
-
```
f=java&class=com.arbortext.e3ci.Application&
   type=log&trace-level=2
```

Each request is processed by the Arbortext Publishing Engine Request Handler as if it were received from the network. The Arbortext Publishing Engine Request Handler will follow its normal request handling procedure, which will result in each request being passed to:

- the publishing configuration cache manager (which will return `cacheable`)
- the predefined request handler (explained in Predefined Request Handlers on page 74) which will recognize each request as a request for an Arbortext PE Application and allocate an Arbortext PE sub-process to do the work
- the publishing configuration cache manager (using the **cache** method) to store each response in the cache.

The publishing configuration initializer takes one parameter, `debug`. If the `debug` parameter has any value other than `true`, the initializer adds the HTTP query parameter `console-log=no` to the `type=log` request. The `console-log=no` suppresses writing the publishing configuration scan to the Arbortext Diagnostics window.

The Java Arbortext PE Application **com.arbortext.e3ci.Application** is an Arbortext PE Application that scans the Arbortext Publishing Engine install tree for publishing information and returns a report in an XML, HTML, or log file as requested. If the application is subsequently invoked (for example, once for an XML document, then for an HTML document, and again for a log file), it does not repeat its scan, because it caches the information it discovers the first time and simply reformats it on subsequent requests.

There are a couple of potential race conditions in the operation of the publishing configuration cache, initializer, and application.

- a client might request any of the three documents before the initializer has time to complete, which can result in two copies of the application running simultaneously in separate Arbortext PE sub-processes.
- the Arbortext PE Request Manager cannot guarantee that all three of the requests generated by the initializer will be allocated to the same Arbortext PE sub-process. If each request runs in a different Arbortext PE sub-process, the application's ability to cache the results of a prior scan won't come into play, and the scan will be repeated several times, once in each Arbortext PE sub-process.

Both possibilities will result in slowing the Arbortext PE server startup but have no otherwise harmful effects.

The Stylesheet Cache

The Arbortext PE Request Manager initializes the stylesheet cache. You can re-initialize the Arbortext PE sub-process stylesheet cache to clear all previously cached stylesheets without restarting the Arbortext PE Request Manager by performing one of the following actions:

- clicking the **Reload Subprocesses** link from the Arbortext Publishing Engine web page (`http://`***`server`***`:`***`port`***`/e3/`).
- submitting an `http://`***`server`***`:`***`port`***`/e3/servlet/e3?f=init` request.

Refer to The f=init Request on page 77 for information.

Predefined Queue Managers

Arbortext Publishing Engine ships with a Queue Manager implemented by the class **com.arbortext.e3.QueueManager**. It may be configured with a **TestSet** so that it can filter queued transactions that match the test set. If no test set is configured, it queues transactions that include the HTTP query parameter `queue=yes`.

If the Queue Manager decides to queue a transaction, it iterates over each queue in the order defined in **`e3config.xml`**. It passes the transaction to each queue and asks the queue to accept the transaction. The Queue Manager continues the iteration from queue to queue until a queue accepts the transaction. If no queue is willing to store the transaction, the Queue Manager returns an error to the client; otherwise it generates a response stating that the transaction was queued.

When the Queue Manager queues a transaction, it will generate either an HTML or XML response, as determined by the HTTP request parameter `response-format`, which may be set to `html`or `xml`. An HTML response is a redirect to a web page that displays the status of the queued transaction and allows the user to retrieve the transaction response after the transaction has completed. An XML response is an XML document that lists information about the transaction and the request.

The key piece of information in the `response-format=xml` response document is the **Transaction** element, which is a descendant of the **FunctionOutput** element, which is a descendant of the root-level **PEFunctionResult** element. The **Transaction** element has the attribute **id**, which specifies the transaction ID; use this ID to submit requests to determine the status of the transaction and retrieve its response after it completes execution. Other **Transaction** attributes include **queueId**, which specifies the queue which has stored the transaction, **held** indicating whether the transaction was automatically held or allowed to execute, **queuePosition**which gives the transaction's location on the queue, and **priority**, which specifies the transaction's priority.

Predefined Request Handlers

Arbortext Publishing Engine ships with a predefined Request Handler implemented by the Java class **com.arbortext.e3.RequestHandler**. It processes HTTP requests by invoking one of a set of predefined Java objects called Request Functions. For each HTTP request, the Request Handler looks for a query parameter named `f` and invokes a Request Function corresponding to the `f` parameter value. For example, to handle a request with parameter `f=status` it invokes the request function **com.arbortext.e3.FunctionStatus**.

At initialization, the Arbortext Publishing Engine Request Handler reads parameters from the **`e3config.xml`** configuration file. It looks for two parameters, **query-function-name** and **function-prefix**. The **query-function-name** parameter tells the Request Handler the query parameter to search for when it is offered an HTTP request to process. The default value is `f`. The **function-prefix** specifies an initial substring for other initialization parameters. Its default value is `f-`. Initialization parameter names beginning with the **function-prefix** string specify the function name as the rest of the initialization parameter name and the corresponding Request Function as the parameter value.

For example, an initialization parameter named `f-status` has a value of **com.arbortext.e3.FunctionStatus**. The parameter name begins with `f-`, so the Request Handler binds the remaining part of the parameter name, `status`, to the **com.arbortext.e3c.FunctionStatus** class during request processing.

You can extend the Arbortext PE Request Manager by developing your own Request Handlers, as already discussed. However, to avoid implementing a complete Request Handler, you can implement your own Request Functions and configure the predefined Request Handler to invoke them.

Predefined Request Functions

A request function is a Java class that implements the interface **com.arbortext.e3. E3RequestFunction**. In particular, such an object must provide a method named **execute** that takes an **HttpServletRequest** object, which contains an HTTP request as a parameter, and returns an HTTP response (encoded as an **HttpRequestResponse** object).

You can extend the Arbortext PE Request Manager by writing either a request handler or a request function. Writing a request function is easier than writing a complete request handler. Consult Writing a Custom Request Handler on page 98 and Writing a custom Request Function on page 98 for more information.

In the sections that follow, the HTTP request parameters follow the format for HTTP requests described in Web Clients and the HTTP Protocol on page 15. You can use the Arbortext Publishing Engine Testing HTML page to issue most of these request functions.

The f=status Request

This request function is implemented by the Java class **com.arbortext.e3. FunctionStatus.**. It returns an HTTP response containing an HTML page describing the configuration and status of the Arbortext PE Request Manager and its Dynamic Components. Almost everything that can be configured in **`e3config.xml`** is reflected in the status page. If you make changes to **`e3config.xml`**, you should always check the Status report page to make sure that the change you made is applied as you expect.

Each of the interfaces that Dynamic Components must implement include a method named **getStatus** which is called by the **f=status** code. If you implement custom Dynamic Components, write a **getStatus** method to place information about your component in the status page.

. The **f=status** request can take one parameter, `page`. The `page` parameter can take one of several values to request a subset of information from the status report:

- `compconfig` returns an HTML version of the publishing configuration report from the publishing configuration cache.
- `compconfiglog` returns the text log of the publishing configuration report from the publishing configuration cache.
- `compconfigdetail` returns an XML version of the publishing configuration report from the publishing configuration cache.
- `java-properties` returns a list of all system properties of the JVM in which the servlet container and Arbortext PE Request Manager is running.

If `page` is omitted, then **FunctionStatus** returns the entire status page describing all components.

For example, if the request specifies `f=status&page=java-properties`, then **FunctionStatus** returns the JVM system properties page. If `f=status&page=`

`compconfiglog`, then **FunctionStatus** returns the log of the publishing configuration scan report.

The PE Publishing Configuration report lists all the document types, document type configuration files, and stylesheets available from the Arbortext PE server. You can review stylesheet names by both their paths and their names in separate entries. The report warns of any duplicate stylesheet names on the server. If a stylesheet name is not unique on the server, Arbortext Publishing Engine uses the first one it finds.

If Arbortext Publishing Engine is restarted, an Arbortext Editor client is not aware of any changes. An Arbortext Editor client can retrieve updated publishing configuration data from the Arbortext PE Request Manager. Arbortext Editor users must perform one of the following to automatically obtain the latest Arbortext Publishing Engine publishing configuration information:

- Disable and then enable **Use Publishing Engine** in the **Publishing Engine** panel of **Tools ▸Preferences**.
- Turn **`peservices`** off and then on again in **Tools ▸Preferences ▸Advanced**.
- Exit and restart Arbortext Editor.

Arbortext Editor can also compare its publishing configuration with Arbortext Publishing Engine publishing configuration using the **Tools ▸Compare Config with PE** menu item. In the **Compare Publishing Configuration with PE** dialog box, Arbortext Editor users can generate the Publishing Configuration Comparison report, which notes the differences between the publishing environment on the client and on the server. This report is helpful in troubleshooting publishing processing, and you should include it with the data you submit when reporting a problem to Technical Support.

An `f=init` request clears the stylesheet cache, allowing you to update stylesheets on the Arbortext PE server without restarting it. Refer to The f=init Request on page 77 for more information.

The f=compconfig-rescan Request

This request allocates an Arbortext PE sub-process to regenerate the publishing configuration scan report. This request refreshes the publishing configuration report and updates the cache used by Arbortext Editor clients when making publishing requests.

The HTML version of the publishing configuration report is updated in the publishing configuration cache. After the scan operation replaces the publishing cache entries, Arbortext Publishing Engine returns an HTML page stating that the operation completed successfully. You can submit a re-scan request from the **Rescan Publishing Configuration** link under the section **Administer Arbortext Publishing Engine** on the Arbortext Publishing Engine HTML page.

The HTML version of the publishing configuration report is available from the **Short** link under the section **View Arbortext Publishing Engine Information** on the Arbortext

Publishing Engine index page. Refer to Monitoring and Reporting Using a Web Browser on page 26 for information about the Arbortext Publishing Engine index page.

The f=license Request

This request function returns an HTTP response containing an HTML page with information about the session configuration and optional components being used by the first Arbortext PE sub-process started by the Arbortext PE Request Manager. It is implemented by the Java class **com.arbortext.e3.FunctionLicense**

The information is similar to what is displayed by the **Session** and **Components** tabs of the **Session** dialog box in Arbortext Publishing Engine Interactive or Arbortext Editor (**Help ▸About Arbortext Publishing Engine Interactive ▸Session** or **Help ▸About Arbortext Editor ▸Session**).

The f=version Request

This request function is implemented by the Java class **com.arbortext.e3. FunctionVersion**. It returns an HTTP response containing an HTML page that provides version information about Arbortext Publishing Engine, including build number and date, as well as information about any applications installed on the server and running in an Arbortext PE sub-process.

The f=init Request

This request function is implemented by Java class **com.arbortext.e3.FunctionInit**. It directs every Arbortext PE sub-process in every Arbortext PE sub-process pool to re-initialize and reload all ACL, JavasScript, and VBScript packages. The request sets a flag on each Arbortext PE sub-process, but the initialization takes place the next time the Arbortext PE sub-process is allocated to an Arbortext Publishing Engine dynamic component. That means that any allocated Arbortext PE sub-processes won't re-initialize until they are deallocated and subsequently allocated to another request.

You can submit an `f=init` request from the **Reload Applications** link on the Arbortext Publishing Engine index page. Refer to Monitoring and Reporting Using a Web Browser on page 26 for information.

You can load new or updated ACL, JavaScript, or VBScript scripts by using **Reload Applications** or `f=init` to instruct the Arbortext PE sub-processes to reload these types of programs, without restarting the Arbortext PE Request Manager.

You can also update Arbortext PE sub-process stylesheets using **Reload Applications** or `f=init` to clear all previously cached stylesheets without restarting the Arbortext PE Request Manager. By clearing the stylesheet cache, Arbortext PE sub-processes will use the updated stylesheet for the next request that uses it.

Note

*The **init** function does not reload Java code. If you need to update Java code in either the Arbortext PE Request Manager or in the Arbortext PE sub-processes, you must stop and restart the Arbortext Publishing Engine.*

*The **init** function does not re-initialize some other resources that Arbortext PE sub-processes might be using, such as repository connections, open documents, and cached document types.*

The f=acl Request

This request function is implemented by the Java class **com.arbortext.e3.FunctionAcl**. It allocates an Arbortext PE sub-process and invokes an Arbortext PE Application written in ACL. It returns whatever HTTP response the Arbortext PE Application generates. The ACL function must be configured in the Allowed Function List (see The Allowed Functions List on page 47) in **`e3config.xml`** or the Arbortext PE Request Manager will return an error message.

The entire HTTP request, including the request line, headers, query parameters, and request body are made available to the ACL Arbortext PE Application. The Arbortext PE Application uses routines closely modeled on those available to Java, JavaScript, and VBScript Arbortext PE Applications in retrieving request information and constructing the response. The request is structured like the following (without the line breaks):

```
f=acl&function=package-name::function-name
    &parameter1=value1
    &parameter2=value2
    &parameterN=valueN
```

The f=java Request

This request function is mapped to the Java class **com.arbortext.e3.FunctionJava**. It allocates an Arbortext PE sub-process and invokes an Arbortext PE Application written in Java, returning whatever response the Arbortext PE Application generates.

The Java Arbortext PE Application is identified by the **class** parameter, which must specify the name of the Java class to be invoked. All other query parameters, along with the rest of the HTTP request, are passed to the Arbortext PE Application for processing. The Java class must be configured in the Allowed Function List (see The Allowed Functions List on page 47) in **`e3config.xml`** or the Arbortext PE Request Manager will return an error message. The request is structured like the following:

```
f=java&class=classname&parameter1=value1&parameter2=value2
```

For more information, refer to 7 Writing Arbortext PE Applications in Java on page 115.

The f=javascript Request

This request function is mapped to the Java class **com.arbortext.e3.FunctionJavascript**. It allocates an Arbortext PE sub-process and invokes an Arbortext PE Application written in JavaScript, returning whatever response the Arbortext PE Application generates.

The JavaScript Arbortext PE Application to be run is identified by the **function** parameter, which must specify the name of the JavaScript function to be invoked. All other query parameters, along with the rest of the HTTP request, are passed to the Arbortext PE Application for processing. The JavaScript function must be listed in the Allowed Function List (see The Allowed Functions List on page 47) in **`e3config.xml`** or the Arbortext PE Request Manager will return an error message.

```
f=javascript&function=functionname&parameter1=value1&parameter2=value2
```

For more information, refer to 8 Writing Arbortext PE Applications in JavaScript on page 127.

The f=vbscript Request

This request function is mapped to the Java class **com.arbortext.e3.FunctionVbscript.** . It allocates an Arbortext PE sub-process and invokes an Arbortext PE Application written in VBScript, returning the response generated by the Arbortext PE Application. VBScript is specifically for Windows platforms.

The VBScript Arbortext PE Application to be run is identified by the **function** parameter, which must specify the name of the VBScript function to be invoked. All other query parameters, along with the rest of the HTTP request, are passed to the Arbortext PE Application for processing. The VBScript function must be listed in the Allowed Function List (see The Allowed Functions List on page 47) in **`e3config.xml`** or the Arbortext PE Request Manager will return an error message instead.

For more information, refer to 9 Writing Arbortext PE Applications in VBScript on page 135.

The f=convert Request

This request function is mapped to the Java class **com.arbortext.e3. FunctionNewConvert**. It provides built-in document conversion capabilities to HTTP clients. The conversion is performed by an Arbortext PE sub-process; HTTP query parameters specify conversion parameters: output format, stylesheet, and other inputs to the conversion. The file to be converted may be specified by a query parameter or as the body of a POST request. The request is structured like the following:

```
type=outputtype&file=pathandfilename&f=convert
```

Note

To avoid web browser problems, don't put `type=`***`outputtype`*** *at the end of the request. For example, web browsers might try to render a request ending in* `type=pdf` *or* `type=html`*.*

For more information, refer to 11 Arbortext Publishing Engine Document Conversion on page 163.

The f=app Request

This request function is mapped to the Java class **com.arbortext.e3.FunctionApp**. It allocates an Arbortext PE sub-process and allow users of Arbortext Publishing Engine to use an HTTP request to pass a job file containing a list of Arbortext Advanced Print Publisher (APP) commands for execution by the APP sub-process.

The `f=app` request takes two parameters.

- `file=`***`<path>`*** — An absolute path and file name of an APP activation file on the Arbortext PE server. If a value is omitted, Arbortext Publishing Engine will return an HTML error page stating the path is missing. The path to the activation file must be listed in the Arbortext Publishing Engine allowed functions list. If the path is not listed there, Arbortext Publishing Engine will return an HTML error page stating that the request cannot be executed. The entry in the table must specify a pattern of type "app".
- `destroy` — (Optional.) When set to `yes` (the default), specifies that the Arbortext PE sub-process that executes the `f=app` request be terminated at the end of the operation. Doing so will prevent the Arbortext PE sub-process from being left in a state that may produce unexpected results for a subsequent application. Setting `destroy` to `no` allows the Arbortext PE sub-process to remain running.

A GET request is accompanied by only parameters. A POST request may include a data (body) file. The Arbortext PE Request Manager will store the body file and pass its its location to the APP sub-process. Any other parameters on an `f=app` request will be ignored with the following exceptions.

- Some parameters, such as `queue=yes` to direct that a request be queued, are used by the Arbortext PE Request Manager.
- All parameters will be listed in the request data file passed to the APP sub-process.

The `f=app` request will always return a response containing an HTML web page with content-type of "text/html". This page may be generated by the APP sub-process or by Arbortext Publishing Engine if an error occurs.

Refer to *Automated Publishing with APP and Arbortext Publishing Engine* in the Reference Documents area of support.ptc.com and in the Arbortext Advanced Print

Publisher online help for details on configuring and using APP to publish with Arbortext Publishing Engine.

Pre-Defined Queues

Arbortext Publishing Engine ships with one Queue object, a Java object named **com. arbortext.e3.ArbortextQueue**. The Arbortext Queue implements an ordered container of transactions, sorted according to priority. It provides a number of features for managing transactions and determining the order in which they will execute, as specified in **`e3config.xml`**. It is possible to configure several distinct queues in **`e3config.xml`**, each accepting a different kind of transaction, and each having different behaviors.

The features in the following sections are implemented as part of the Arbortext Queue. If you implement your own queue object, you won't get the capabilities offered by the Arbortext Queue unless you implement them in your own code. These sections will focus on the behavior of a single instance of the Arbortext Queue object.

Selecting Transactions for Queuing

The Arbortext Queue may be configured in **`e3config.xml`** with a **TestSet** element that specifies which transactions the instance of Arbortext Queue will accept for storage. If the **TestSet** is omitted, the queue instance will accept any transaction the Queue Manager offers it. Since queue managers offer transactions to the queues in the order configured in **`e3config.xml`**, this means that each queue except the last one should be configured with a **TestSet**.

It is important to note the distinction between a Queue Manager selecting transactions and a Queue selecting transactions. The Queue Manager determines whether a given transaction should be queued. If not, the Arbortext PE Request Manager treats the transaction as an immediate transaction. Once the Queue Manager selects a transaction for queuing (in the Arbortext implementation, by detecting `queue=yes` in the request), the Queue Manager offers the transaction to each configured Queue. Each Queue then determines, according to its configured criteria in **`e3config.xml`**, whether it is willing to accept the transaction.

If a Queue Manager determines that a transaction should be queued, but no queue is configured to accept transactions of that type, the situation is treated as a configuration error. The Queue Manager will return an error page to the Arbortext Publishing Engine Request Handler, which will return it to the client, and the transaction will not execute.

Transaction Priorities

Once a Queue Manager has determined that a transaction should be queued and an instance of the Arbortext Queue has accepted the transaction, the queue instance places the transaction on its internal list of transactions in order of priority. Priority is determined by the value of the `queue-priority` query parameter, which specifies the priority of the request when placing the transaction on a queue. `1` is the highest priority and `5` is the lowest. If the parameter is not present or its value is not numeric, the priority defaults to `3`. If the priority is less than 1, it is adjusted to 1; if the priority is greater than 5, it is adjusted to 5.

The Arbortext Queue maintains its list of transactions in priority order. When a new transaction is inserted, it is placed as the last transaction of its priority. For example, a queue list contains transactions A with priority 1, B with priority 2, C with priority 3, and D with priority 4. A new transaction E with priority 3 would be inserted after C. The new order would be A, B, C, E, D. A would execute first, B second, C third, E fourth, and D last.

The Arbortext Queue allows an administrator to move a transaction up or down in the list from the **Queued Transactions List** (available from the **Queue List** page retrieved from the **Queue List** link on the Arbortext Publishing Engine HTML index page). The move actions are available under **Actions**. When a transaction is moved, its priority may be adjusted to keep the list in priority order. Refer to *Configuration Guide for Arbortext Publishing Engine* for information on queue management.

Selecting a Transaction for Execution

Like all Arbortext Publishing Engine Queues, the Arbortext Queue will periodically be called by the Queued Transaction Scheduler to identify a transaction to execute. The Arbortext Queue will offer the first transaction on the list that is not already executing and not being held, subject to the conditions described in the next sections.

Enabling and Disabling a Queue

An Arbortext Queue may be enabled or disabled. When disabled, a queue will never supply a transaction to the Queued Transaction Scheduler for execution. Whether a queue is enabled or disabled, it will always accept transactions from Queue Managers if the transaction meets its configured criteria.

A queue can be enabled or disabled from the **Queue List** page retrieved from the **Queue List** link on the Arbortext Publishing Engine HTML index page or by an **f=q-enable** request from a client. Refer to Arbortext Queue Request Functions on page 85 for more information.

When the Arbortext PE Request Manager starts, the queue parameter `initial-state`specifies the state of the queue and has the following values:

- `enabled`

  specifies that the queue should always be enabled.
- `disabled`

  specifies that the queue should always be disabled.
- `saved` (the default)

specifies that the state should be set to what it was the last time the Arbortext PE server was shut down.

If this parameter is not specified, the queue will start out enabled.

Holding and Releasing Transactions

A queued transaction may be held or released. When held, no Arbortext queue will present the transaction to the Queued Transaction Scheduler for execution. By default, a transaction is not held when it is placed on a queue. However, a transaction may be marked `hold`, which makes it ineligible for execution until the `hold` flag is cleared. A transaction may be held or released at any time by an Arbortext Publishing Engine administrator using the **Transaction List** web page.

If the queue parameter `hold-all` is set to `yes` in **`e3config.xml`**, then the Arbortext Queue it controls will hold each transaction it accepts.

Active and Inactive States and the Active Interval

An Arbortext Queue may be active or inactive. When inactive, the queue does not supply any transactions to the Queued Transaction Scheduler for execution. Whether the queue is active or inactive, it will always accept transactions from Queue Managers.

The queue parameter **active-interval** can be specified in **`e3config.xml`** to configure an Arbortext Queue to be active during certain periods; outside those periods, the queue is inactive. The **active-interval** parameter value consists of a list of time periods separated by semicolons specifying when the queue is active, following the format ***`p1;p2;p3;pn`***.

- Each period consists of the following: `sunday`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`, `sunday`, `weekday`, or `weekend`.
- You can also further specify an optional time frame by appending a comma and a time range. A time range consists of a 24–hour ***`HH`*****`:`*****`MM`*****`–`*****`HH`*****`:`*****`MM`*** specification. If the second ***`HH`*****`:`*****`MM`*** is omitted, it defaults to `23:59` of the same day. If the second ***`HH`*****`:`*****`MM`*** specification is earlier than the first, it specifies a time on the following day.

Examples of **active-interval** specifications:

- `Parameter name="active-interval" value="weekday"`

  The queue is active all day Monday through Friday but not on weekends.
- `Parameter name="active-interval" value="weekday,20:00-06:00;weekend"`

  The queue is active Monday through Friday from 8 p.m. to 6 a.m. the next morning and all day on weekends.

- `Parameter name="active-interval" value="monday,12:00-16:00;wednesday,17:00"`

  The queue is active on Mondays from 12 p.m. to 4 p.m. and on Wednesdays from 5 PM to midnight (`23:59` is assumed for the end period when no end time is specified).

If a queue does not have an **active-interval** parameter or it's set to an empty string `""`, then it is always active.

An active queue must also be enabled to process its transactions. When a queue is disabled, it can still accept transaction requests, but no transactions can be executed by the Queued Transaction Scheduler, even if the queue is active. An administrator can enable or disable a queue from the **Queue List** page retrieved from the **Queue List** link on the Arbortext Publishing Engine HTML index page. However, an administrator can't make a queue active or inactive from the **Queue List** page. Refer to *Configuration Guide for Arbortext Publishing Engine* for more information on queue management.

Limiting a Queue's Resource Consumption

You can configure queue parameter **max-concurrent-transactions** to limit the number of transactions from a single queue that can be performed simultaneously. If **max-concurrent-transactions** is omitted, it defaults to `0`, which means there is no limit. After the number of transactions set by this parameter have started executing, the Arbortext Queue will not present another transaction to the Queued Transaction Scheduler for execution until one of the executing transactions completes.

Transaction Scheduling Options

You can configure the queue parameter **scheduling-option** in **`e3config.xml`** to one of the following values

- `strict-complete`

  No transaction shall start executing until all previous transactions on the queue have finished executing, including held transactions. Note that choosing this value means that transactions from this queue will never execute in parallel.

- `strict-parallel`

  No transaction shall start executing until all previous transactions on the queue have started executing, including held transactions.

- `relaxed` (the default)

  A transaction can start execution after every previous transaction on the queue that is not held has started executing.

  This value allows a transaction to start executing even if earlier transactions on the queue are held or are unable to execute for some reason.

Waiting for Transactions on Previous Queues

You can set the queue parameter **previous-queues** in **`e3config.xml`** to specify one or more queues on which the current queue must wait before executing transactions. The Arbortext Queue will not present transactions to the Queued Transaction Scheduler for execution until the queues in the **previous-queues** list contain no transactions except those being held. Specify **previous-queues** by providing a list of queue IDs separated by commas, following the format:

```
Parameter name="previous-queues" value="queue1,queue2,queue3"
```

If this parameter is not specified, then there is no previous queue dependency.

Forcing a Transaction to Execute Next

Each of the previous sections describes circumstances that might make a transaction ineligible for execution. It is possible to bypass many of these restrictions and force a transaction to execute before any other transactions on any queue.

A web client can issue an **f=qt-execute** request to designate a transaction to be the next to execute. This transaction is then identified by a marker stored in the Queued Transaction Scheduler. Only one transaction can be designated in this way. If transaction A is marked and then transaction B is also marked, transaction B becomes the next transaction to execute. Transaction A returns to its normal course of processing without special handling.

The designation ignores the queue's enabled and active states, and it also ignores the transaction's hold status and scheduling options. However, it will obey the global **maxConcurrentQueuedTransactions** parameter setting, the **maxConcurrentQueuedTransactions** setting on any Arbortext PE sub-process pool, and the **max-concurrent-transactions** setting on the queue containing the transaction.

Arbortext Queue Request Functions

The queue management functions (**f=q-*queue-function)***) and the queued transaction management functions (**f=qt-*queued-transaction-function)***) can be used by a custom application to send requests to a Queue Manager. To request a queue action, the application would construct a request specifying parameters supported by the queue request function. For example, to disable a queue with the ID `queue23`, you would submit an **f=q-enable** request with the parameters:

```
f=q-enable&id=queue23&enable=no
```

If any function detects an error (incorrect or missing required parameters), it will return an HTML or XML response (as determined by the `response-format` parameter) describing the error. The HTTP response code will be 500 when an error is returned.

Queue Management Functions

Each of the queue management functions begins with **q-** and accepts two parameters.

- `id` (required)

  Specifies the ID of the queue as configured in **`e3config.xml`**. If a request specifies `id=*`, then the request applies to every queue defined in **`e3config.xml`**.

  If the `id` parameter is missing or does not match the `id` attribute of any queue defined in **`e3config.xml`**, this function will return an error.

- `response-format` (optional)

  Specifies whether the function returns either an HTML or an XML response. The value of `response-format` may be `xml` or `html`, and it is not case sensitive. The default value is `html`. By default, the queue functions assume that their caller is a web browser and return an HTML page suitable for display. If the function specifies `response-format=xml`, the function returns an XML document suitable for parsing by a client program (that is, not a web browser).

The f=q-enable Request

This function enables or disables a queue. Enabling a queue allows transactions on the queue to become available for execution, but the Queued Transaction Scheduler determines when the transactions can be executed. Disabling a queue has no impact on any transactions on the queue that are already executing.

In addition to `id` and `response-format`, this function takes the parameter `enable`. If the request specifies `enable=yes`, the queue is enabled. If the request specifies `enable=no`, the queue is disabled. If the request specifies `enable=yes` and the queue is already enabled, or `enable=no` and the queue is already disabled, this request returns successfully.

The f=q-holdall Request

This function sets or clears the **hold** flag on all of the transactions on a queue. A transaction with the **hold** flag is not eligible for execution. Setting the **hold** flag on a transaction that's already executing has no effect (unless Arbortext Publishing Engine stops before the transaction completes, and after it restarts, the transaction will not execute until the hold is removed).

In addition to `id` and `response-format`, this function takes the parameter `hold`. If the request specifies `hold=yes`, all transactions on the queue are held. If the request specifies `hold=no`, all transactions on the queue have their **hold** flag removed.

The f=q-list Request

This function returns information about queues. For a queue specified by `id` (or for all of them if `id=*`), the function returns the ID, whether the queue is enabled, whether the queue is active, and the number of transactions on the queue that are waiting, executing, and complete. The complete count includes transactions that have finished normally (with or without error) and transactions that have been cancelled.

Queued Transaction Management Functions

Most of the queued transaction management functions begins with **qt-** and accepts two parameters.

- `id` (required)

  Specifies the ID of the transaction as assigned by the Arbortext PE Request Manager.

- `response-format` (optional)

  Specifies whether the function returns either an HTML or an XML response. The value of `response-format` may be `xml` or `html`, and it is not case sensitive. The default value is `html`. By default, the queue functions assume that their caller is a web browser and return an HTML page suitable for display. If the function specifies `response-format=xml`, the function returns an XML document suitable for parsing by a client program (that is, not a web browser).

The **f=qt-list** request function takes a different set of parameters.

The f=qt-cancel Request

This function cancels a queued transaction. For a transaction that has not started executing or is currently executing, this function will generate an XHTML document showing that the transaction was cancelled. That document is also stored as the result of the transaction. This cancellation document will be in the same format as the report that is produced if the Arbortext PE sub-process terminates or there is some other serious error preventing a transaction to finish.

If this function is applied to an executing transaction, the Arbortext PE sub-process used by the transaction will be terminated.

If this request specifies an immediate request, a queued transaction that has completed, or does not identify an active transaction, this function will return an error.

The f=qt-discard Request

This function causes a transaction directory of a completed queued transaction to be placed in the transaction archive (according to **com.arbortext.e3.transactionArchive.enable** and **com.arbortext.e3.transactionArchive.selector** global parameters in

`e3config.xml`). Immediately after copying the transaction directory to the archive, it's deleted from the Active Transaction Directory and the transaction is no longer an active transaction. This intent of this function is to allow an application to clean up the directory after retrieving the result of a queued transaction.

Consider a user consulting a list of completed transactions using a web browser or some other client application. As transactions finish, they appear on the list. The user can then retrieves the results of each transaction. This function would prevent the list from becoming mostly transactions for which the user has already retrieved the results. Transactions are removed after a configured interval (as set by the **com.arbortext.e3.transaction.maxRetrievedTransactionAge** global parameter in **`e3config.xml`**).

This function acts on the transaction regardless whether the results from a transaction have been retrieved or not. This action allows users to clear the list of unwanted transactions without retrieving unwanted results.

If this request specifies an immediate request, a queued transaction that has not executed, or if the `id` parameter does not identify an active transaction, this function will return an error.

The f=qt-execute Request

This function forces the queued transaction indicated by the `id` parameter to be executed as soon as possible. The Queued Transaction Scheduler will ignore the enabled and active states of the queue containing the transaction, as well as the transaction's **hold** state. It will make the transaction next in line by ignoring any transactions ahead of the target transaction.

The Queued Transaction Scheduler will not ignore the global **com.arbortext.e3.transaction.maxConcurrentQueuedTransactions** parameter or the **maxConcurrentQueuedTransactions** attribute on the Arbortext PE sub-process pool that will allocate an Arbortext PE sub-process to process this transaction.

There can be only one transaction designated to be next in line. If an **f=qt-execute** request specifies transaction A and then another **f=qt-execute** request specifies Transaction B before transaction A starts executing, transaction A will lose its special next-in-line status. The next transaction to execute would be B. Transaction A returns to its prior state and continues to follow the normal course of transaction scheduling.

If this request specifies an immediate request, a queued transaction that is executing or has finished, or if the `id` parameter does not identify an active transaction, this function will return an error.

The f=qt-hold Function

This function sets or clears the **hold** flag on the specified transaction. Setting the **hold** flag on an executing transaction has no effect, with the exception that if Arbortext Publishing Engine terminates before the transaction completes, the **hold** will be applied after Arbortext Publishing Engine restarts.

In addition to `id` and `response-format`, this function takes the parameter `hold`. If the request specifies `hold=yes`, the transaction is held. If the request specifies `hold=no`, the **hold** flag is removed.

If this request specifies an immediate request or does not identify an active transaction, this function will return an error.

f=qt-move

This function moves a queued transaction up or down in its queue. The move may result in changing the transaction's priority.

In addition to `id` and `response-format`, this request requires two parameters:

- `direction` (required)

  Specifies either `up` or `down` (not case sensitive). `up` moves the transaction toward the top of the queue, and `down` moves the transaction toward the bottom of the queue.

- `places` (required)

  Specifies either `all` or a positive integer to represent the number of transactions to bypass.

This function moves the transaction past the number of transactions specified by `places` in the direction specified by `direction`.

After the transaction is moved, the transaction's priority is adjusted as follows:

- If the transaction was moved up, its priority is set to the priority of the transaction behind it.
- If the transaction was moved down, its priority is set to the priority of the transaction ahead of it.

If the value of the `places` parameter is greater than the number of transactions the specified transaction can bypass, then the transaction is moved to either the head or end of the queue as specified by `direction`.

If this request specifies an immediate request, a queued transaction that has finished, or if the `id` parameter does not identify an active transaction, this function will return an error.

The f=qt-retrieve Request

This function returns the result of a completed queued transaction.

Note

*For **f=qt-retrieve** only, the `response-format` parameter supports a different value than the other **qt-** functions, `zip`.*

If the **f=qt-retrieve** request includes `response-format=zip` , which will return a zip archive containing two entries:

- **`response.xml`** will contain the HTTP result code, message, and HTTP headers.
- **`body.dat`** will contain the response body. If processing the transaction did not generate a response body, then the archive will not contain **`body.dat`**.

If `response-format=zip` is not included, then the result will be the HTTP response that would have been returned if the response had been an immediate transaction. The response will include the HTTP status code, HTTP headers, and HTTP response body (if any).

If this request specifies an immediate request, a queued transaction that has not completed, or if the `id` parameter does not identify an active transaction, this function will return an error.

The f=qt-setpriority Request

This function changes the priority of a queued transaction and, if necessary, moves the transaction to a new location in its queue.

In addition to `id` and `response-format`, this function requires the `priority` parameter, which takes an integer value from 1 to 5 inclusive. There is no default. If its value is specified as less than 1, it is adjusted to 1. If its value is specified as greater than 5, it is adjusted to 5.

If this function raises a transaction's priority, the transaction is moved so that it becomes the last transaction of the set of transactions with its new priority. If this function lowers a transaction's priority, the transaction is moved so that it becomes the first transaction of the set of transactions with its new priority. (The principle is to move a transaction the minimum number of places necessary to keep the queue sorted in priority order.)

If this request specifies an immediate request, a queued transaction that has finished, or if the `id` parameter does not identify an active transaction, this function will return an error.

The f=qt-status Request

This function returns the status of a queued transaction. The returned XML or HTML page shows the request status as explained in Arbortext Publishing Engine as a Transaction Processor on page 18.

For an incomplete transaction, the response will include the transaction status and information about the request: HTTP headers, query parameters, body size. For a complete transaction, this function will also supply the HTTP response code, response headers, and size of response body.

The information returned will also include the name of the Arbortext PE sub-process pool and its process ID if the transaction is executing. If the transaction is executing or awaiting execution, the information returned will also include the name of the queue, the transaction priority, whether the queue is disabled, whether the transaction is held, and the number of transactions ahead of the transaction in the queue.

Other differences depend on whether XML or HTML is the format returned for a **qt-status** request:

- For HTML, additional information about whether the queue is active or inactive. For a completed transaction, we won't return the body file size or response headers; instead we'll provide a button that the user can use to retrieve the body file, which will be transmitted using the headers.
- For the XML, no information about whether a queue is enabled or active, because the XML data is intended for use by programs. An application would submit an **f=q-list** request for information about the queue.

The f=qt-list Function

This function returns information about queued transactions. By default, the report returns information about all queued transactions and about every queue, and it can filter for transactions that are queued, executing, cancelled, or complete.

Note

*This function is different from the rest of the queued transaction management functions beginning with **qt-** in the set of parameters it supports.*

It takes the following parameters:

- `hostname`

  Specifies the host name of the client that submitted the transaction
- `ip`

  Specifies the IP address of the client that submitted the transaction
- `userid`

  Specifies the ID of user who submitted the transaction
- `id`

  Specifies the name of the queue for the transaction

  You may specify all queues `id=*`or by omitting the `id` parameter.
- `state`

  Specifies the state of the transaction, `queued`, `executing`, `cancelled` or `complete`.

Each parameter may be omitted, specified once, or specified more than once. If a parameter is omitted, then transactions are not filtered with regard to that parameter. If a parameter is specified, then only transactions that match that parameter value are returned. If a parameter is specified more than once, then transactions that match either of the parameter specification are returned.

For example, if the `state` parameter is not specified, then all transactions are not filtered for their state. If `state=complete` is specified, then only completed transactions are returned. If `state=complete` and `state=queued` are both specified, then queued and completed transactions are returned.

If two different parameters are specified more than once each, then transactions matching the combinations of both parameter values are returned. For example, request specifies:

```
f=qt-list&id=queue1&id=queue2&state=complete&state=queued
```

Only transactions on either `queue1` or `queue2` and with a state of either `queued` or `complete` are returned.

Pre-Defined Notifiers

Arbortext Publishing Engine ships with one Notifier object, a Java object named **com.arbortext.e3.queue.MailNotifier**.

The Mail Notifier can send email when a transaction changes state. The Mail Notifier may be configured to use a **TestSet** element in **`e3config.xml`** to identify the transactions for which it will send email. If no **TestSet** is configured, the Mail Notifier will send email for every transaction.

Refer to the *Configuration Guide for Arbortext Publishing Engine* for information on configuring a notifier.

5 Customizing the Arbortext PE Request Manager

If you want to write your own Dynamic Components to run in the Arbortext PE Request Manager, you need to:

- develop and compile your Java code. When you compile your Java code, you'll need to include the file **`e3\e3\WEB-INF\lib\e3servlet.jar`** in your class path. **`e3servlet.jar`** contains the definitions of the interfaces that you must implement.
- place the class files in ***`PE_HOME`*****`\e3\e3\WEB-INF\classes`** or place the classes in a JAR file and put the JAR in **`e3\e3\WEB-INF\lib`**.
- update **`e3config.xml`** to define your components so that Arbortext PE Request Manager will load them when Arbortext Publishing Engine initializes.

As a Java servlet, the Arbortext PE Request Manager may be called to process more than one HTTP request at a time. Each request is handled by a separate thread, so all custom code must be thread-safe. Consult a good programming resource if you need to learn more about thread-safe code before developing a dynamic component. In general:

- Avoid the use of global variables.

- Guard accesses to any global variables you do use by using synchronized blocks or routines.
- For performance, keep synchronized blocks or routines as small as possible so that one thread blocks others as briefly and infrequently as is safe.
- Allocate working storage at the start of each request.
- Make sure the name of every disk file you create or write to is unique.
- Test your dynamic component by transmitting many simultaneous requests to the server, to make sure that your code really is thread-safe.

Writing a Custom Cache Manager

To write a cache manager, you need to develop a Java class that implements the interface **com.arbortext.e3.E3CacheEntry** and update **`e3config.xml`** to define it. You need to provide **cache**, **init**, and **search** methods. You'll also need to develop one or more objects that implement the interface **com.arbortext.e3.E3CacheEntry**, because that's what the **search** method returns.

Cache Entry Object

A cache entry object encapsulates a response that was returned to a client which can also safely be returned to another client making the same request. The cache entry object also contains a locked or unlocked flag and a state code. Though a state code doesn't belong in a cache entry object, it allows the Cache Manager's **search** method to return a single result for the cache entry, rather than a cache entry and a separate state code.

There are three possible state code values, however, two of them indicate that the cache entry does not contain any content.

- `NOT_CACHEABLE`

  The cache manager is not willing to cache responses for the request being processed.

- `CACHEABLE`

  The cache manager would be willing to cache a response for the active request, but the cache manager does not currently contain a response.

- `IN_CACHE`

  The cache entry contains a valid response that could be returned to the client.

An entry may be locked by a call to **E3CacheManager.search**. If a cache entry is locked, the cache manager refrains from deallocating the cache entry object. A cache entry should always be locked during transmission to the client. The caller requesting the lock is responsible for calling **E3CacheEntry.unlock**.

Implementing a Cache Manager

There are several methods that must be implemented by a cache manager. Because the Arbortext PE Request Manager can receive simultaneous requests, every method except the **init** and **destroy** methods must be thread-safe .

The init Method

The Arbortext PE Request Manager calls the **init** method when it initializes. It takes an **E3RequestContext** object (described in Arbortext Publishing Engine Request Context

Object on page 52) and a **com.arbortext.e3cf.E3CacheManagerDescriptor** as parameters.

The **init** method performs initialization for the cache manager object. If a cache manager can't initialize, it should throw an exception (using the class **com.arbortext.e3. E3RequestException**) with an error message explaining the circumstances causing the error.

The **init** method runs before the Arbortext PE sub-processes start, so it can't retrieve data from an Arbortext PE sub-process to cache. To retrieve and cache data from an Arbortext PE sub-process, you need to develop and load an Arbortext PE sub-process Initializer (explained in Writing a Custom Initializer on page 98).

The destroy Method

The Arbortext PE Request Manager calls the **destroy** method after the servlet container has ordered the Arbortext Publishing Engine servlet to terminate. According to the Java servlet standard, the servlet container will pass no more requests to the servlet once the **destroy** method has been called. Use the **destroy** method to deallocate any resources that will not be cleaned up automatically by terminating the servlet.

The getid Method

The **getid** method must return a unique string that identifies the cache manager. It should return the value of the **id** attribute from the **E3CacheManagerDescriptor** that was passed to the **init** method.

The getStatus Method

The **f=status** function from an HTTP request calls the **getStatus** method, which takes a **PrintWriter** as a parameter. Use it to display information about your cache manager in the Arbortext Publishing Engine status HTML page (from the Arbortext Publishing Engine Testing page, find the **status** link). Structure the portion of your status report in HTML suitable to include in an HTML document **body**.

The search Method

The most important method for a cache manager is the **search** method. It takes two parameters, the HTTP request received from a client and a flag indicating whether the cache entry matching the request (if one exists) should be locked. It must return a cache entry. If there is no entry in the cache that can fulfill the request for the client, it should create a substitute cache entry with a status code of `NOT_CACHEABLE` or `CACHEABLE`. The cache manager should return `CACHEABLE` to direct the Arbortext PE Request Manager to call the **cache** method to pass in the response for the request when one is produced.

The cache manager must lock the cache entry if it contains an `IN_CACHE` status and caller set the **doLock** parameter.

The cache Method

The Arbortext PE Request Manager calls the **cache** method if these conditions are met:

- A response to a request has been produced by a request handler and the response is not an error.
- The **search** method returned a cache entry with a status of `CACHEABLE`.

The **cache** method takes an HTTP request and an HTTP response as parameters. The cache manager should copy the response and store it in its data structures, with as much data from the request as necessary, so that it can return a cache entry that contains the HTTP response and status `IN_CACHE` next time the **search** method receives a similar request.

The **cache** method should make a copy of the HTTP response object; it should not retain a reference to the response object itself, because the response object is allocated by the servlet container and might, depending upon implementation, be reused after transmission to the client. Similarly, if the response object refers to a file on disk, the **cache** method should make a copy of the file in question; there's no way for the **cache** method to be sure that the file won't be deleted or reused while the cache manager references it.

Writing a Custom Queue Manager

To write a queue manager, you must develop a Java class that implements the interface **com.arbortext.e3.E3QueueManager** and update **`e3config.xml`** to define it. The most efficient approach also includes implementing either a request handler or several request functions, as described in Writing a Custom Request Handler on page 98 and Writing a custom Request Function on page 98.

A queue manager must provide the same **init**, **destroy**, **getId**, and **getStatus** methods as described for a cache manager. In addition, your class must provide a **service** method which takes an HTTP request as a parameter. Use the **service** method to examine the HTTP request and determine whether you want to save it to fulfill at a later time. If so, you should construct an HTTP response and return it as the return value for the **service** method. If the request should not be saved for later processing, the **service** method should return null.

To queue a request, you should consider the following:

- Developing a mechanism for storing requests that guards against Arbortext PE Request Manager restarts, such as saving them to disk.
- Implementing additional requests that allow users to inquire whether a queued request has been completed.
- Obtaining the results
- Administering requests and responses in the queue.

Writing a Custom Request Handler

To write a Request Handler, you must develop a Java class that implements the interface **com.arbortext.e3.RequestHandler** and update **`e3config.xml`** to define it. The Java class must provide the same **init**, **destroy**, **getId**, and **getStatus** methods used by a cache manager or queue manager. In addition, your class must provide a **service** method that takes an HTTP request as a parameter . It should return either an HTTP response that Arbortext PE Request Manager will return to the client or null.

Arbortext Publishing Engine has a built-in Request Handler that examines and dispatches HTTP requests according to its request functions. The source code for the Arbortext Publishing Engine Request Handler uses the **com.arbortext.e3.RequestHandler** interface.

Writing a custom Request Function

Request functions are Dynamic Components loaded by the Arbortext Publishing Engine Request Handler. You can avoid the overhead of implementing a complete request handler by implementing a request function instead.

To write a request function, you need to develop a Java class that implements the interface **com.arbortext.e3.RequestFunction** and provides two public methods, **init** and **execute**. The **init** method is called by the request handler when it initializes. The **execute** method takes an HTTP request as a parameter and returns an HTTP response.

After compiling your Java class, put it in ***PE_HOME*`\e3\e3\WEB-INF\classes`** or place the classes in a JAR file and put the JAR in **`e3\e3\WEB-INF\lib`**. Remember to update **`e3config.xml`** to add a **RequestFunction** element to load the request function.

Writing a Custom Initializer

To write an initializer, you need to develop a Java class that implements the interface **com.arbortext.e3.E3Initializer**. Your Java class must provide the same **getId** and **getStatus** methods as described for cache managers. The Java class must also implement an **init** method that will be called after all other dynamic objects are loaded and initialized, and after the Arbortext PE sub-process pools are initialized. As a result, your **init** code can obtain an Arbortext PE sub-process and execute ACL commands and evaluate ACL functions. Your **init** code can also create a simulated Arbortext Publishing Engine request and send the request to a cache manager, queue manager, or request handler for processing. Make sure that your initializer is thread-safe, as it could be running at the same time as any number of client requests.

Note

*In addition, an Initializer must also provide a **destroy** method, which currently will not be called (this is a known problem).*

After compiling your Java class, put it in ***PE_HOME*\e3\e3\WEB-INF\classes** or place the classes in a JAR file and put the JAR in **e3\e3\WEB-INF\lib**. Remember to update **e3config.xml** to add an **Initializer** element to load it and determine when it should run.

You have the option of making your initializer a blocking or deferred initializer.

- If you specify it as a blocking initializer (**defer=no**), your **init** method will be called before the Arbortext PE Request Manager **init** method returns to the servlet container. That means that the Arbortext PE Request Manager will not start accepting requests from clients until after your initializer completes its operation.
- If you specify that your object is a deferred initializer (**defer=yes**), then the Arbortext PE Request Manager **init** method creates a background thread to run your **init** method (and any other deferred initializers) asynchronously after the Arbortext PE Request Manager **init** method returns to the servlet container. That means that the Arbortext PE Request Manager may start accepting client requests before your initializer is called or before it completes its operation and returns.

The drawback to waiting for a blocking initializer to completes before the Arbortext PE Request Manager starts processing client requests is that you could substantially lengthen the time it takes for the Arbortext PE Request Manager to start. This wait time is directly related to the amount of work your initializer performs. The drawback to deferring the initializer is that you must be certain the Arbortext PE Request Manager can properly process requests before your initializer starts or finishes.

Writing a Custom Request Selector

To develop a request selector, you must create a Java class that implements the interface **com.arbortext.e3.E3RequestSelector**. In addition to the **getId**, **init** , **destroy** and **getStatus** methods, a request selector must implement a **test** method. The **test** method takes an HTTP request as a parameter and returns `true` or `false` to indicate whether the request meets its configured criteria.

After compiling your request selector, place the class file in ***PE_HOME*\e3\e3\WEB-INF\classes** or place the classes in a JAR file and put the JAR in **e3\e3\WEB-INF\lib**.

Update **e3config.xml** to define your request selector. Add a **RequestSelector** element to load and initialize the request selector, and then add it to a **TestSet** associated with an Arbortext PE sub-process pool.

Writing a Custom Queue

To develop a Queue object, you must create a Java class that implements the interface **com.arbortext.e3.E3Queue**. In addition to the **getId**, **init**, **destroy**, and **getStatus** methods, a queue must implement a number of additional methods that allow the Queue object to store transactions upon request by a Queue Manager and offer transactions to be executed by the Queued Transaction Scheduler.

Fundamental Queue Methods

Every queue object must implement the following methods, which support placing a transaction on a queue, removing a transaction after it finishes executing, and determining which transactions to execute next

The dequeue Method

This method is called by the Queued Transaction Scheduler after it selects a transaction for processing. The queue should delete the transaction from its list and return `true`. If the transaction is not on its list, the queue should return `false`.

The enqueue Method

This method is called by a Queue Manager with the `afterRestart` parameter set to `false` to determine whether the queue is willing to store the indicated transaction . This method is called by the Arbortext PE Request Manager with `afterRestart` set to `true` after the Arbortext PE Request Manager restarts, to restore a transaction that was queued but not executed to completion prior to the restart.

The queue should respond whether it is willing to store the transaction. If it is, it should add the transaction to its list and return `true`. If it is not, then it should return `false`.

The queue can assume that if `afterRestart` is set to `true`, its **sort** method will be called after all transactions have been inserted.

The findTransaction Method

This method is called by the Queued Transaction Scheduler to determine whether the queue has a transaction that should be executed. If so, the method returns the transaction object. Otherwise, it returns null. The method should only return a transaction if the Arbortext PE sub-process pool is willing to execute transactions of the same type. The method can call the **testRequest** method of **com.arbortext.e3.E3SubprocessPool** to make the determination.

Queue Management Methods

The following methods set or return information about the queue itself.

The getEnabled Method

This method returns `true` or `false` to indicate whether the queue is enabled. When it returns `true`, the Queued Transaction Scheduler should call the queue's **findTransaction** method to find the next eligible transaction.

The isActive Method

This method should return `true` or `false` to indicate whether the queue is active. When it returns `true`, the Queued Transaction Scheduler should call the queue's **findTransaction** method to find the next eligible transaction.

The setEnabled Method

This method should be called to enable or disable a queue. Set the parameter `flag` to `true` for enabled or `false` for disabled. If `flag` is set to `false`, the queue can assume that the Queued Transaction Scheduler will not call the `findTransaction` method.

Transaction Management Methods

The following methods manage the transactions stored on the queue.

The contains Method

This method determines whether the transaction is on the queue's transaction list. The queue should return `true` or `false`.

The getCompletedTransactionCount and getExecutingTransactionCount Methods

These methods are called to obtain the number of completed or executing transactions that were queued by this queue. Since transactions are deleted from the queue before they start executing, the queue will need to iterate through all known transactions using an iterator obtained by the **com.arbortext.e3.RMTransaction.list** method looking for executing or completed transactions with the appropriate queue ID.

The getTransactionIndex Method

This method returns the position of the transaction specified by `target` in the queue. It specifies the number of transactions ahead of `target`. It returns `0` if the transaction is at the top of the queue, and `-1` if `target` is not on the queue.

The getWaitingTransactionCount Method

This method returns the number of transactions on the queue.

The isEmpty Method

This method should return `true`if there are no transactions on the queue, and `false` if there are transactions on the queue.

Iterator

This method should return an iterator over the transactions on the queue.

The move Method

This method moves a `transaction` in the queue. The parameter `steps` specifies the number of transactions to bypass. The parameter `isUp` should specify `true` to move the transaction toward the top of the queue or `false` to move toward the bottom of the queue.

After the transaction is moved, the transaction's priority is adjusted as follows:

- If the transaction was moved up, its priority is set to the priority of the transaction behind it.
- If the transaction was moved down, its priority is set to the priority of the transaction ahead of it.

The setTransactionPriority Method

This method should set the transaction’s priority to the value specified by `newPriority`, which takes an integer value from 1 to 5 inclusive, highest to lowest. The transaction moves to the appropriate position within the queue’s transactions, which are sorted in priority order.

The sort Method

This method is called by the Arbortext PE Request Manager after it restarts. As it restarts, it will find the transaction directories of all queued transactions that did not complete before the Arbortext PE Request Manager terminated, and place them on queues by calling each queue’s **enqueue** method. When all **enqueue** calls are done, the Arbortext PE Request Manager will call each queue’s **sort** method to put the transactions in order of priority.

The Queued Transaction Scheduler will not start until after each queue’s **sort** method has been called.

Writing a Custom Notifier

To develop a Notifier object, you must create a Java class that implements the interface **com.arbortext.e3.E3Notifier**. In addition to the **getId**, **init**, **destroy**, and **getStatus** methods, a notifier must implement the method **changeState**.

The Arbortext Publishing Engine Request Handler calls the **changeState** method of every notifier, in the order defined in **`e3config.xml`**, each time a transaction changes state. The action that the notifier takes when its **changeState** method is called is entirely up to the notifier. The **changeState** has an `oldState` parameter (the previous state of the transaction) and a `newState` parameter (the new state of the transaction).

Unlike other Dynamic Components, where the cache manager, queue manager, request handler, or queue that accepts a request ends the iteration, the **changeState** method is called for every configured notifier.

III

The Arbortext PE sub-process

6 Implementing Arbortext PE Applications

An Arbortext PE Application is custom code that runs in an Arbortext PE sub-process. An application is invoked by the Arbortext PE Request Manager in response to an HTTP request that specifies a class or function name. The purpose of the application is to process the request, generate an appropriate HTTP response, and return the response to the Arbortext PE Request Manager for transmission to the requesting client.

An Arbortext PE Application can use any of the AOM interfaces and ACL functions within an Arbortext PE sub-process to manipulate documents. Because Arbortext PE sub-processes run in batch mode, you can only use interfaces or functions that are not related to the graphical user interface.

Concurrency

An Arbortext PE Application does not need to be thread-safe. An Arbortext PE sub-process only handles one request at a time; therefore, it can only run one Arbortext PE Application at a time. However, an Arbortext PE Application does need to be safe for multi-processing, because the Arbortext PE Request Manager could ask two or more Arbortext PE sub-processes to run the same Arbortext PE Application at the same time.

For example, if an Arbortext PE Application attempts to write temporary data using an absolute path and file name (**`e:\tempdata\tempdatafile.txt`**), two instances of the Arbortext PE Application might run at the same time and overwrite each other's file. If an Arbortext PE Application stores data to the Windows clipboard, two instances of the operation might run simultaneously and interfere with each other.

Every Arbortext PE Application needs to be serially reusable. An Arbortext PE Application should not attempt to maintain any state from one call to the next. The Arbortext PE Request Manager could ask a particular Arbortext PE sub-process to execute any Arbortext PE Application repeatedly, so the Arbortext PE Application must not assume any prior state each time it is invoked. If a particular client submits several Arbortext PE Application requests to an Arbortext PE server, the Arbortext PE Request Manager might allocate a different Arbortext PE sub-process to serve each request. If the second request expects to find information left behind by the first request (for example, a value stored in a global variable), the application would only work if the Arbortext PE Request Manager happened to allocate the same Arbortext PE sub-process to serve both requests.

Installing an Arbortext PE Application

Place your custom Arbortext PE Application in the appropriate ***`PE_HOME`*****`\custom`** directory to make it accessible to Arbortext PE sub-processes (refer to Overview of Custom Programs and Scripts on page 280 for information). A Java application must be either a set of class files or a Java Archive (JAR file) containing class files. You would place the **`.class`** or **`.jar`** files in **`custom\classes`**. ACL, JavaScript, and VBScript applications (**`.acl`**, **`.js`**, and .**`vbs`** files) would be placed in **`custom\init`**.

If you use the same install tree to run Arbortext Editor and Arbortext Publishing Engine, the presence of ACL, JavaScript, and VBScript applications in **`custom\init`** can prevent Arbortext Editor from loading without error. These applications may be relying on the presence of scripts that are available to Arbortext PE sub-processes and Arbortext Publishing Engine Interactive but not to Arbortext Editor.

As a workaround, you can place your ACL, JavaScript, and VBScript applications in ***`PE_HOME`*****`\custom\scripts`**, and place a companion ACL script that loads the application files in ***`PE_HOME`*****`\custom\init`**. The ACL script specifies the condition to run only under an Arbortext PE sub-process or Arbortext Publishing Engine Interactive.

For example, you can use the ACL predefined variables, **$is_e3** and **$is_e3_ interactive**, to determine the mode in which Arbortext Publishing Engine is running.

- **$is_e3** determines if the Arbortext Publishing Engine is running in server-mode (no user interface). If true, it returns any value other than zero.
- **$is_e3_interactive** determines if the Arbortext Publishing Engine is running in Arbortext Publishing Engine Interactive mode (running the user interface). If true, it returns any value other than zero.

The following example illustrates how to use these variables in an ACL script:

```
if (main::is_e3) {
  # I'm running in PE server mode.
    }
  else if (main::is_e3_interactive) {
    # I'm running in PE Interactive.
      }
  else {
    # I'm running in Editor.
      }
```

Sample Applications

The Arbortext Publishing Engine install tree contains a number of sample Arbortext Publishing Engine applications available under ***PE_HOME*\e3\samples**. You can compile and run the Java sample, and you can install and run the other sample applications. Because Arbortext Publishing Engine ships with the sample applications enabled, you would make copies and change the class or function names of the samples to distinguish between the test applications you install and the versions that ship with the product. You would also need to add the class or function names to the allowed functions list in **e3config.xml**.

Additional samples are provided for ACL and JavaScript applications. You can copy the sample files from ***PE_HOME*\e3\samples\acl** or **e3\samples\javascript** to **\custom\init** and change the function names. You'll also need to add your functions to the allowed functions list to run the sample applications.

You can run Arbortext Publishing Engine sample test applications which return basic information about server configuration using links on the Arbortext Publishing Engine HTML web page. The source code for these sample applications is also available from **e3 \samples**. The Arbortext Publishing Engine Testing HTML page is available using a URL like the following:

http://***hostname*:*port***/e3

Click on one of the programming language links under **Test Arbortext Publishing Engine** by choosing **ACL**, **Java**, **JavaScript**, or **VBScript** next to **Run a test PE**

application. These samples report the request headers, Arbortext PE sub-process environment variables and (for Java only) JVM properties.

The samples Directory

Arbortext Publishing Engine handles log information using Log4j. Arbortext Publishing Engine developers need to be familiar with Log4j before using Arbortext PE server sample logging code. Arbortext Publishing Engine provides sample logging code in ACL, Java, VBScript and JavaScript languages that can be implemented as part of a custom application of the same type. The logging code in the application can capture and report log information.

- Java

  `e3\samples\java\com\arbortext\e3\testapp\E3Sample2.java`
- JavaScript

  `e3\samples\javascript\e3samples2.js`
- VBScript

  `e3\samples\vbscript\e3samples2.vbs`
- ACL

  `e3\samples\acl\e3samples2.acl`

In the set of Arbortext Publishing Engine samples, there is a **test_log** example function for each supported programming language that you can incorporate into your code.

General instructions for using the functions in the sample files:

1. Set the logging level using the global parameter **com.arbortext.e3.transactionArchive.selector** in **`e3config.xml`**. (Refer to *Managing the Transaction Archive* in Global Arbortext PE Request Manager Parameters on page 41 for information.)
2. Add the function to the Allowed Functions list in **`e3config.xml`**.
3. See specific instructions in the comments of the sample code files.

Refer to *Configuration Guide for Arbortext Publishing Engine* for information about application logging and how to set configuration parameters in **`e3config.xml`**.

The Allowed Functions List

The allowed function list prevents the Arbortext PE Request Manager from executing applications that are not authorized to run. Each time you create an application, you must add an entry to the **AllowedFunctions** list in **`e3config.xml`**. The entry gives Arbortext Publishing Engine permission to execute the application when it's called from a request. Insert a **ClientFunction** element under the **AllowedFunctions** element, and then specify the **pattern** and **type** attributes.

Each **ClientFunction** entry specifies a pattern that's used to check incoming application names for a match. When the Arbortext PE Request Manager receives a request containing an `f=acl`, `f=java`, `f=javascript`, or `f=vbscript` query parameter, it compares the application name specified by the application query parameter (**class** for `f=java` and **function** for `f=acl`, `f=javascript` and `f=vbscript`) against each entry of the appropriate language type in the **AllowedFunctions** list.

The **pattern** attribute can use wildcard characters to expand matching capabilities for a set of applications with similar names. Use the wildcards `*` to match zero or more characters and `?` to match one character. The **ClientFunction** entry also has a **type** to specify the application's programming language, `acl`, `java`, `javascript`, or `vbscript`. A match occurs if the application name matches any pattern for its particular function type.

The Arbortext PE sub-process Application Context

Every Arbortext PE sub-process contains a Java object called the Arbortext PE Application Context. The Arbortext PE Application Context is responsible for loading and executing Arbortext PE Applications upon request from the Arbortext PE Request Manager. The Arbortext PE Application Context starts when the Arbortext PE sub-process launches. Before the Arbortext PE Request Manager asks an Arbortext PE sub-process to run an Arbortext PE Application, the Arbortext PE Request Manager passes parameter information to the Arbortext PE Application Context. The parameter information includes all global parameters defined in **`e3config.xml`**, plus all of the parameters defined for the Arbortext PE sub-process pool in which the Arbortext PE sub-process will run.

The Arbortext PE Application Context stores the parameter information it receives from the Arbortext PE Request Manager in an object called the Application Configuration. An instance of this object, which implements the Java class **com.arbortext.e3.E3ApplicationConfig**, is passed to the **init** method of every Arbortext PE Application written in Java. The information is also available to JavaScript, VBScript, and ACL applications, as described in later sections.

Support for Custom Applications with the APP Engine

When integrating custom Arbortext Publishing Engine application code with the APP formatting engine, be aware of the following options and limitations:

- Formatting hooks

  The following hooks will not be called when formatting a document using APP:

  - **formatcompletehook**
  - **formatcontinuehook**
  - **formatpagestatushook**
  - **formaterrorhook**
- Input and configuration files
  - Layout file

    The APP formatting engine does not generate or interact with the layout file. The locations listed in the layout file represent OIDs in the document open in the Edit window. APP does not have access to that document during formatting.

    The layout file was created to support the line numbering application, and it is used in other applications to draw constructs such as gutter rules. The same functionality can be achieved in APP in an Arbortext Styler stylesheet or using custom source edits.
  - TMX files

    `.tmx` files include Tex macro customizations that can be included in your **`custom/inputs`** directory. These will not have any effect on formatting when using the APP engine.
  - EXC and PAT files

    `.exc` and **`.pat`** files provide custom hyphenation rules. These are not used by the APP engine.
  - DCF files

    APP does not interact with **`.dcf`** files.
  - PDF configuration files

    The APP and FOSI engines use separate PDF configuration files. Refer to Print and PDF Configuration Files for further information.
- Script interaction

  The FOSI engine can complete certain actions outside of the stylesheet using system functions that call custom ACL code directly during the e-i-c processing of the stylesheet. The APP engine does not use system functions.
- Command line differences

  Command line functions do not cause the APP engine to format the document. They only work with the FOSI engine.

  - Printing using startup commands

    epic -c "print noformat wait" ***`testfile.xml`***

- Formatting using startup commands (**`allpasses`**, **`onepass`**, **`wait`**)
- **no format**
- **allpasses**
- **onepass**
- **wait**
- **force**
- **format** and all its modifiers: **continue**, **onepass**, **allpasses**, **layout**, **quit**, **stop**
- The **-v** flag for starting up Arbortext Editor.

  Passes any valid window arguments to the Print Preview window.
- **linenum**

Note

linenum *will run using the FOSI engine even if APP is the effective print engine for the environment. It embellishes the document and shows line numbers in the Edit view. If you subsequently preview using menus, however, APP will run as expected and the line numbers do not show.*

7 Writing Arbortext PE Applications in Java

An Arbortext Publishing Engine Java application is an object which implements the interface **com.arbortext.e3.E3Application**. It is loaded and invoked by the Arbortext PE Application Context (described in The Arbortext PE sub-process Application Context on page 111).

The Arbortext PE Request Manager invokes a Java application when it receives a request with the `f` query parameter set to `java` and the `class` parameter specifying the full Java class name of the application. When it receives such a request, the Arbortext PE Request Manager, by way of the predefined Request Handler and Java request function, allocates an Arbortext PE sub-process and passes the request to the Arbortext PE Application Context.

Initialization

When an Arbortext PE Application Context receives a request, it first determines that the request specifies **f=java**, then examines the `class` parameter to see if this is the first time the class has been invoked since the Arbortext PE sub-process started. If so, the Arbortext PE Application Context loads the Java class into the Arbortext PE sub-process's embedded Java Virtual Machine. In order for this to happen, the Arbortext PE Application class must be present in a **`custom\classes`** directory either as a **`.class`** file or as a member of a **`.jar`** file. After loading the application, the Arbortext PE Application Context calls its **init** method and passes a parameter specifying an instance of **com. arbortext.e3.E3ApplicationConfig**, allowing the Arbortext Publishing Engine application to access configuration information from **`e3config.xml`**. The configuration parameters include the global parameters plus any other parameters defined for the Arbortext PE sub-process pool, both defined in **`e3config.xml`**.

If initialization succeeds and the **init** method returns normally, then the Arbortext PE Application Context passes the request to the Java application's **doGet** or **doPost** method.

If any error is detected, the **init** should throw an exception. In that case, the Arbortext PE Application Context will return an error to the Arbortext PE Request Manager.

If a Java Arbortext PE Application initializes successfully, the Arbortext PE ApplicationContext will retain a reference to the application object. If the Arbortext PE Request Manager sends another request for the same application, the Arbortext PE Application Context will again invoke the **doGet** or **doPost** method. The Arbortext PE Application Context will not create another instance of the application object, nor will it call the application's **init** method again.

If a Java application fails to initialize (meaning its **init** method throws an exception), the Arbortext PE Application Context will discard the application object. It will not call the object's **destroy** method. If the Arbortext PE Request Manager sends another request for the same application, the Arbortext PE Application Context will allocate another new object and call its **init** method. Depending on the condition which caused the first call to throw an exception, this attempt could succeed or throw another exception.

Request Processing

After loading and initializing the Java application, if necessary, the Arbortext PE Application Context calls the application's **doGet** or **doPost** method to handle the HTTP request. The parameters for both methods are a request object and a response object. The request object contains all available information about the request. The response object, which is empty initially, is a location in which the Java application can build the response. After **doGet** or **doPost** returns, the Arbortext PE Application Context returns the response object to the Arbortext PE Request Manager, which transmits the response to the client using the same protocol as the request, either HTTP.

The request and response objects, which implement the interfaces **com.arbortext.e3.E3ApplicationRequest** and **com.arbortext.e3.E3ApplicationResponse**, are modeled upon the Java Servlet standard request and response interfaces. Developers familiar with a Java Servlet implementation will find only minor differences. For requests, the only difference is that the **E3ApplicationRequest** object lacks the methods associated with authentication support.

The response object, **E3ApplicationResponse**, has more significant differences. In a standard Java servlet, the Java code generates a response body by obtaining an output stream from the response object and writing to it; the data is transmitted to the client as it is written. In a PE Java application, it is necessary to construct the entire response before transmitting it to the client. A PE Java application should write the response data to either a file on disk or a string variable. The application must then call **setOutputFile** or **setOutputPage** to add the response file or string to the response object for eventual return to the Arbortext PE Request Manager.

An Arbortext Publishing Engine Java application begins processing a request when the Arbortext PE Application Context calls the **doGet** or **doPost** method. Processing finishes when **doGet** or **doPost** returns or throws an exception. For a normal return, the content of the response object passed to **doGet** or **doPost** is returned to the client. If the **doGet** or **doPost** methods throw an exception, the content of the response object (if any) is discarded and an error page describing the exception is returned instead.

The Arbortext PE Application Context calls the **doGet** or **doPost** methods only after a call to the **init** method has successfully returned. Request processing within a single Arbortext PE sub-process is serial, so there's no parallel request processing within a single Arbortext PE sub-process. The Arbortext PE Application Context never calls **doGet** or **doPost** to process a second request while a previous call to **doGet** or **doPost** is in progress.

However, Arbortext Publishing Engine Java application developers do need to consider two concurrency issues:

- It's possible that several Arbortext PE sub-processes might receive simultaneous requests for the same Arbortext PE Application from the Arbortext PE Request Manager on behalf of different clients. Therefore, it's possible that multiple instances of the same Arbortext PE Application might be running simultaneously in different Arbortext PE sub-processes. To prevent interference with one another, the Arbortext PE Application must use unique files, directories, and other resources.
- An Arbortext PE Application must be serially reusable to handle requests from different clients. An Arbortext PE Application must be prepared to accept calls to **doGet** or **doPost**, one after another. Before returning a response or throwing an exception for **doGet** or **doPost**, the Arbortext PE Application must clean up files and resources, and leave its internal data in a state suitable for a subsequent call to **doGet** or **doPost**.

Note that Arbortext Publishing Engine Java developers do not need to worry about overlapping temporary file name spaces. As described earlier, each Arbortext PE sub-process has its own unique temporary directory. When an Arbortext PE sub-process starts an embedded JVM, it sets the **`java.io.tmpdir`** system property to this value. Java programmers who create temporary files and allow the JVM to pick the location are safe from having their temporary files overwritten by other JVMs.

Termination

When the Arbortext PE sub-process terminates, it will attempt to inform its embedded JVM. The JVM will then inform the Arbortext PE Application Context, which will call the **destroy** method of each loaded Arbortext Publishing Engine Java application. The **destroy** method should release resources, delete scratch files, and so forth.

An Arbortext PE sub-process can terminate without calling each Java Arbortext PE Application's **destroy** method. This might happen if the Arbortext PE sub-process crashes suddenly or is forced to terminate by the operating system. In these situations, system resources (open files, network resources, and so on) are deallocated by the system.

Creating a Java Arbortext PE Application

Your Java Arbortext PE Application must implement the **com.arbortext.e3. E3Application** interface, which is modeled on the Java Servlet 2.3 specification.

The Arbortext Publishing Engine interface is located in the install tree at ***`PE_HOME`*****`\lib\classes\pecommon.jar`**. This JAR file contains the following interface definitions:

- **com.arbortext.e3.E3Application**
- **com.arbortext.e3.E3ApplicationConfig**

  Passed to the **init** method.
- **com.arbortext.e3.E3ApplicationRequest**

  Passed to **doGet** and **doPost** methods
- **com.arbortext.e3.E3ApplicationResponse**

  Passed to the **doGet** and **doPost** methods

All Javadoc for the Arbortext Editor and Arbortext Publishing Engine interfaces is available in the **Programming ▸Javadoc ▸Arbortext Publishing Engine** section of Help Center.

A Java Arbortext PE Application must define the methods **init**, **doGet**, **doPost**, and **destroy**. The **doGet** and **doPost** methods take the request object **E3ApplicationRequest** and response object **E3ApplicationResponse** as parameters. The request and response objects are passed by the Arbortext PE Request Manager to an Arbortext PE sub-process for processing. The Arbortext PE sub-process passes the request and response objects to the Arbortext PE Application.

The **E3ApplicationConfig** interface provides configuration information from the Arbortext Publishing Engine **`e3config.xml`** file. Use this interface to retrieve the configuration parameters and their values. The configuration information is passed to the **init** method (explained in later in this section), which should retain it for possible subsequent use by **doGet** or **doPost** methods.

In Arbortext Publishing Engine, exceptions are handled in the following ways:

- Errors in **E3ApplicationRequest** and **E3ApplicationResponse** methods called by user code will be handled as exceptions. Any exceptions not handled by the user's code will be caught by Arbortext Publishing Engine.
- Arbortext Publishing Engine also catches any unhandled exceptions thrown by user-written Arbortext PE Application methods.

For any exceptions not caught by the custom application, an HTML page is returned to the client describing the error in as much detail as possible.

Note

*The **f=java** function can specify a custom application that returns a file of any type. It can also set the HTTP status code, reason phrase, and headers to any requested value.*

The init Method

The **init** method is called by the Arbortext PE sub-process to allow the Arbortext PE Application to initialize before the first call to either **doGet** or **doPost**.

The `config` parameter passes information to the Arbortext PE Application through the Arbortext PE sub-process.

The destroy Method

The **destroy** method is called by the Arbortext PE sub-process before it terminates, allowing the Arbortext PE Application to remove files and free resources. No calls to **doGet** or **doPost** will be made after the Arbortext PE sub-process calls **destroy**.

The doGet Method

The **doGet** method is called by the Arbortext PE sub-process to handle an Arbortext Publishing Engine HTTP GET request. Its *request* parameter is an object containing all available information about the HTTP request.

The **doGet** method builds an HTTP response by calling methods of the *response* object.

The doPost Method

The **doPost** method is called by the Arbortext PE sub-process to handle an Arbortext Publishing Engine HTTP POST client request. Its *request* parameter is an object containing all available information about the HTTP request.

The **doPost** method builds an HTTP response by calling methods of the *response* object.

The E3ApplicationRequest Class

This class defines an object that conveys information about an HTTP request to an Arbortext PE Application. The Arbortext PE sub-process creates an **E3ApplicationRequest** object and passes it as a parameter to the **doGet** and **doPost** methods.

An **E3ApplicationRequest** object provides parameter names and values, HTTP headers, and in HTTP POST requests, a **File** object representing the posted file to the Arbortext PE Application.

The **E3ApplicationRequest** class has a set of methods for accessing the parameters in the request object.

The E3ApplicationResponse Class

This class defines an object that collects the elements of the HTTP response that Arbortext Publishing Engine sends to the client making the request. The Arbortext PE sub-process creates an **E3ApplicationResponse** object and passes it as a parameter to the **doGet** or **doPost** method. When **doGet** or **doPost** returns, the Arbortext PE sub-process passes the information in the **E3ApplicationResponse** object to the Arbortext PE Request Manager to return to the client.

The **doGet** and **doPost** methods can use the **E3ApplicationResponse** object to control every element of an HTTP response, including the HTTP result code, any HTTP headers, and, optionally, a file to be returned to the HTTP client.

The **E3ApplicationResponse** object contains the following:

- the state code
- the HTTP status code
- a collection of HTTP headers
- a string containing either an error message, an HTML page, or a redirect target
- a file, if one was requested
- the archive flag, which determines whether the transaction will be archived

- The alternate archive flag, which determines whether the transaction will be archived in an alternate location

The state code directs the Arbortext PE Request Manager to use this data in constructing a response to the client in one of the following ways:

- return a fabricated HTML page based on the status code
- return a fabricated HTML page based on the status code and string error message
- return the status code, HTTP headers, and string HTML page
- return the status code, HTTP headers, and file
- return the status code, HTTP headers, and file, deleting the file after transmission to the client
- redirect the client to another URL

The archive flag determines whether the transaction will be archived. If the flag is set to `true`, the transaction will be archived, provided the transaction archive is enabled. If the flag is set to `false`, the transaction won't be archived unless one of the following overrides it:

- The **com.arbortext.e3.transactionArchive.selector** parameter is set to `all`
- The **com.arbortext.e3.transactionArchive.selector** parameter is set to `error` or `log` and the transaction has failed to publish.

If the transaction will be archived, the alternate archive flag determines whether the transaction will be stored in the alternate transaction archive location. If the alternate archive flag is not set, the transaction is archived to the standard transaction archive.

The standard transaction archive is accessible from the **Transaction Archive** link on the Arbortext Publishing Engine index page. The alternate transaction archive is not available from this web page, which allows transactions with sensitive data to be stored in another location which is only accessible to a user with permission to log on to the server or read the network share location.

The E3ApplicationConfig Class

This class permits Arbortext Publishing Engine to pass configuration information to an Arbortext PE Application. An instance of this class is passed to the Arbortext PE Application in conjunction with the call to its **init** method.

Note

If repository credentials or other sensitive information is stored in ***`web.xml`*** *or* ***`e3config.xml`****, you should remove permission to access the ACL, JavaScript, VBScript and Java sample applications from the Allowed Functions list in the* ***`e3config.xml`*** *configuration file. These sample applications display the global parameters, which would be a security issue if the parameters contain confidential information.*

This interface supports the following methods:

- **getInitParameter** returns the value of its initialization parameter `name` as a string. Returns null if there is no such parameter.
- **getInitParameterNames** returns an Enumeration of String objects listing the name of every initialization parameter.
- **addIntermediateFile** copies the file whose absolute path is provided by the `source` parameter into the transaction directory as an intermediate file. The `contentType` and `description` parameters are included as comments. This method allows the Arbortext PE Application to save a temporary file that contains information useful in debugging that can be retrieved from the transaction archive.
- **getApplicationLogger** returns a Java Log4j Logger object that can be used to write to the servlet log. The log level will be set as indicated by the set of **com.arbortext.e3.applicationLog** parameters in **`e3config.xml`**. (Refer to *Parameters that Control Application Logging* in Global Arbortext PE Request Manager Parameters on page 41 for more information.)
- **getTransactionDir** returns a string containing the absolute path to the transaction directory of the request currently being processed.

Calling the Conversion Processor From a Java Arbortext PE Application

You can write a Java application that can call the conversion processor (explained in 11 Arbortext Publishing Engine Document Conversion on page 163) by importing the following packages from ***`PE_HOME`*****`\lib\classes\peclient.jar`**:

```
com.arbortext.e3.DocumentConverter
com.arbortext.e3.DocumentConverterException
```

Call the conversion processor by calling the static method:

```
com.arbortext.e3.DocumentConverter.doConvert(
          String inFile,
          String outFile,
          Map     params
          );
```

The parameter *inFile* must specify the absolute path to a document to be converted. The **com.arbortext.e3.DocumentConverter.doConvert** method does not process open documents. If your application creates or modifies a document that will be converted, you must save the document to disk and close it before invoking **doConvert**. After **doConvert** returns, you can open your document again if you need to make further modifications.

The parameter *outFile* must specify the absolute path to the output file that **doConvert** will produce. If this file already exists, it will be overwritten during conversion processing.

The parameter *params* must be a Java map with **String** keys and values. Each parameter entry must correspond to a supported conversion parameter (refer to Document Conversion Parameters on page 165 for a list and descriptions).

For example, to specify a stylesheet:

```
params.put( "stylesheet", "c:\absolute\path\to\stylesheet.style" );
```

The **doConvert** method has no return value. If an error occurs during processing, it throws a **com.arbortext.e3.subprocess.DocumentConverterException**, which has two methods:

- **getReason** returns the HTML reason code (400, 500, or some other valid code)
- **getPage** returns the XHTML page describing the error (the same page an **f=convert** request would return to an HTTP client)

The following is an example of Java code that calls **com.arbortext.e3.DocumentConverter.doConvert**:

```
import com.arbortext.e3.subprocess.DocumentConverter;
import com.arbortext.e3.subprocess.DocumentConverterException;
…
String  inFile  = "c:\absolute\path\to\input\file.xml";
String  outFile = "c:\absolute\path\to\output\file.pdf";
Map      params  = new HashMap();

params.put( "type","pdf" );
params.put( "stylesheet" , "d:\absolute\path\to\stylesheet.style" );

try {
        DocumentConverter.doConvert( inFile, outFile, params );
        // Conversion succeeded
}
catch( DocumentConverterException e ) {
         String reason =  e.getReason();
         String page     = e.getPage();
         // Log the failure and exit.
}
```

Sample Java Arbortext PE Application

Basic sample applications are included in the ***PE_HOME*\e3\samples** subdirectory, one each for Java, JavaScript,VBScript, and ACL. These samples applications reside on the Arbortext PE server and are loaded into an Arbortext PE sub-process when it starts. The source code for the Java sample application is located in:

```
PE_HOME\e3\samples\java\com\arbortext\e3\testapp\E3AppTest.java
PE_HOME\e3\samples\java\com\arbortext\e3\testapp\E3Sample2.java
```

The **E3AppTest** sample is compiled internally when you test it from the Arbortext Publishing Engine index HTML page, so you don't need to compile it beforehand. The sample application reports information about the Arbortext Publishing Engine environment. Document manipulation, such as opening, closing, and changing content, is accomplished using the AOM DOM interfaces. For complete information about the Arbortext Publishing Engine Testing HTML page, see Monitoring and Reporting Using a Web Browser on page 26. For information on the AOM, refer to the *Programmer's Reference*.

For information on **E3Sample2.java**, see Sample Applications on page 109.

Troubleshooting Java Applications for Arbortext Publishing Engine

Reloading Java Applications

The **f=init** function does not affect Java applications as it only reloads JavaScript, VBScript, and ACL custom applications. After an Arbortext PE sub-process has started, its JVM can't reload a **.class** or **.jar** file from the ***PE_HOME*\custom** directory. You need to restart the servlet container to reload **.class** or **.jar** files in the Arbortext PE sub-process. For example, if you were using Tomcat, you would stop and restart it to reload your Java file.

Logging

Your application can obtain a Logger object and write to the servlet log by calling the **getApplicationLogger** method of the **E3ApplicationConfig** interface. The logger level will be set as indicated by the parameters **com.arbortext.e3.applicationLog.java.*classname*** (***classname*** is the Java class that implements the application), **com.arbortext.e3.applicationLog.java**, or **com.arbortext.e3.applicationLog** in **e3config.xml**. Refer to *Parameters that Control Application Logging* in Global Arbortext PE Request Manager Parameters on page 41 for more information.

Examining Transaction Files

After your application successfully finishes processing a request, the request, the generated response returned to the client, and any other information generated during processing could be stored in the transaction archive, if the transaction archive is configured to save all transactions.

When debugging an application, you can configure the transaction archive to save all transactions by setting **com.arbortext.e3.transactionArchive.selector** to `all` in **`e3config.xml`**. Then retrieve the transactions that your application fulfilled, and inspect the data passed to your application by the HTTP request and the response returned by your application for those inputs.

Saving Intermediate Files

If your application creates any temporary files or documents as part of generating its the response that is returned to the client, you can save those files to the transaction archive using the **addIntermediateFile** method of the **E3ApplicationConfig** interface (see The E3ApplicationConfig Class on page 121). The intermediate files will accompany the transaction if you use **addIntermediateFile**. They will be placed in the transaction directory, provided the transaction is placed in the transaction archive, and you can examine them by retrieving the transaction from the archive.

Using the Arbortext Publishing Engine Test Utility

Arbortext Publishing Engine offers an interactive testing utility called Arbortext Publishing Engine Test Utility to validate and test your Java, JavaScript, VBScript, and ACL applications as well as document conversion (**f=convert**) parameters. You can test your custom applications without having an Arbortext Publishing Engine production environment in place.

You can launch the Arbortext Publishing Engine Test Utility as a standalone program or from the Arbortext Publishing Engine Interactive **Tools** menu. You choose the test type and set all the parameters and their values for the custom application. The utility constructs the query string from your specifications and validates it. You can also run the test and report the results as though it had been handled by Arbortext Publishing Engine. If errors occur, they're included in the report. The Arbortext Publishing Engine Test Utility is documented in the *Test Utility User's Guide* manual, which you can find in the **`/docs`** on the Arbortext Publishing Engine distribution archive or CD-ROM, as well as in the ***`PE_HOME`*****`/e3/docs`** directory after you install Arbortext Publishing Engine. The Arbortext Publishing Engine Test Utility standalone executable is located in:

`PE_HOME``\e3\bin\e3test.cmd`

Avoiding Content Type Problems in the Arbortext Publishing Engine HTTP Request

You may experience problems with a returned file if you submit an HTTP or HTTPS request that ends with a file extension. The web browser can interpret the request improperly. If Arbortext Publishing Engine passes the content-type header correctly (for instance, `application/pdf`) in its response to the browser, the web browser may ignore the content-type header and try to render the response based on a file extension occurring at the end of the URL. To avoid this problem, you may want to structure an HTTP request so that file extensions do not appear at the end.

In the following example, the first request may cause a problem. By reordering the parameters in the same request, as in the second example, the request will succeed. The best practice is to place the **f=java**, **f=javascript**, **f=vbscript**, **f=acl**, or **f=convert** specification at the end of the URL.

The following HTTP **f=acl** request may cause a problem because the request ends in **`.xml`**. The web browser may try to interpret XML as the content-type, rather than the PDF content-type which is specified and is the content-type that will be returned. Ignore the line breaks in the examples:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?f=acl&function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml
```

The following HTTP request will succeed because the URL ends with the **f=acl** specification, which won't confuse the web browser:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml&f=acl
```

You may also want to take advantage of the fact that the web browser can interpret content-type from a URL that ends in a file extension. You can include a dummy parameter at the end of the URL to specify a file extension, for example, `dummy=file.pdf` for a PDF file. The dummy parameter will be ignored by Arbortext Publishing Engine, but the web browser may try to render the response based on the file extension `.pdf` occurring at the end of the URL.

8 Writing Arbortext PE Applications in JavaScript

An Arbortext Publishing Engine JavaScript application is a subroutine function which takes two parameters, a Java request object and a Java response object. The Arbortext PE sub-process uses the Rhino JavaScript package to make Java objects accessible to JavaScript programs. Essentially, this means that JavasScript Arbortext Publishing Engine applications run in the same programming environment as Arbortext Publishing Engine Java applications. For more information on making Java available to JavaScript using LiveConnect, refer to the *Calling Java from JavaSctipt* section of the *Programmer's Reference*.

The Arbortext PE Request Manager processes requests for Arbortext Publishing Engine JavaScript applications by passing requests with the `f` query parameter with the value `javascript` to the request function **com.arbortext.e3.FunctionJavascript**. The request function allocates an Arbortext PE sub-process and passes the request to the Arbortext PE Application Context (described in The Arbortext PE sub-process Application Context on page 111). The Arbortext PE Application Context creates the request and response objects. Then, it directs the Arbortext PE sub-process JavaScript interpreter to evaluate the function specified in the **function** parameter of the HTTP request. The function must be defined in a **`.js`** file in a **`custom\init`** directory or in a **`.js`** file in **`custom\scripts`** which is loaded from **`custom\init`**. The returned value for the JavaScript function is ignored because the data must be transmitted to the client in the response object.

A JavaScript Arbortext PE Application can access information about the request and set information to be returned as part of the response using the same interface as an Arbortext PE Application written in Java. Unlike a Java Arbortext PE Application, a JavaScript Arbortext PE Application has no initialization or termination component.

Creating a JavaScript Arbortext PE Application

An Arbortext PE Application runs within an Arbortext PE sub-process and responds to HTTP requests routed to it from the Arbortext PE Request Manager.

The **f=javascript** function has a **function=*function-name*** parameter where ***function-name*** specifies the JavaScript function to handle the request.

The Arbortext PE sub-process calls the JavaScript function specified by ***function-name*** and passes LiveConnect JavaObject references to the **E3ApplicationRequest** and **E3ApplicationResponse** objects. The JavaScript Arbortext PE Application must specify two parameters to represent the **E3ApplicationRequest** and **E3ApplicationResponse** objects. When the JavaScript function returns, the Arbortext PE sub-process transmits the response information in the **E3ApplicationResponse** object back to the Arbortext PE Request Manager.

Note

For an Arbortext Publishing Engine installation, only one type of JavaScript is supported. To be sure your JavaScript Arbortext PE Application is compatible, you need to specify the JavaScript interpreter for your `.js` file. At the top of the script, place the following statement:

```
//<script type="text/javascript">
```

The JScript language is not supported for Arbortext Publishing Engine.

Note

*The **f=javascript** function can specify a custom application that returns a file of any type. It can also set the HTTP status code, reason phrase, and headers to any requested value.*

Testing JavaScript Syntax in Arbortext Publishing Engine Interactive

You can launch Arbortext Publishing Engine Interactive to test the syntax of the code in your custom JavaScript applications. When you source your JavaScript file, be aware that Arbortext Publishing Engine Interactive will not actually run the application.

To test JavaScript applications using Arbortext Publishing Engine Interactive

1. Your JavaScript file should be in the ***PE_HOME*\custom\init** directory.
2. Launch Arbortext Publishing Engine Interactive by choosing it from its shortcut on your Arbortext Publishing Engine program group.

   Any scripts in ***PE_HOME*\custom\init** are automatically sourced at startup. If the JavaScript application contains syntax or other errors, you'll automatically receive a message explaining the nature of the error.
3. If you wish to leave Arbortext Publishing Engine Interactive running, you can make a change to the JavaScript file and source it manually using the Arbortext Publishing Engine Interactive command line prompt. You would use the **source** command and specify the path, like the following example:

   source ***path-and-script-name***.js

Note

If the command line is not enabled at the bottom of the interface, choose ***Options ▸ Preferences****. If you want to change the default setting, select the* ***Preferences*** *button. If you want to change the setting for the current session only, select the* ***Current Settings*** *button. Then, on the* ***Window*** *tab, select the* ***Command Line*** *option. You can exit by choosing the save option of your choice.*

Calling the Conversion Processor from a JavaScript Arbortext PE Application

You can write a JavaScript application that can call the conversion processor (explained in 11 Arbortext Publishing Engine Document Conversion on page 163). The Arbortext PE sub-processes use the Rhino package to allow JavaScript to access Java objects. Refer to the Calling the Conversion Processor From a Java Arbortext PE Application on page 122 for an explanation of the Java objects used to invoke the conversion processor. Each parameter entry must correspond to a valid conversion parameter (see Document Conversion Parameters on page 165 for a list and descriptions).

The following is an example of JavaScript code that calls **com.arbortext.e3. DocumentConverter.doConvert**:

```
var   inFile    = "c:\absolute\path\to\input\file.xml";
var   outFile  = "c:\absolute\path\to\output\file.pdf";
var   params    = new java.util.HashMap();

params.put( "type", "pdf" );
params.put( "stylesheet" , "d:\absolute\path\to\stylessheet.style" );

try {
       com.arbortext.e3.subprocess.DocumentConverter.doConvert(
        inFile, outFile, params );
    // Conversion succeeded
}
catch( e ) {
    var reason = e.getReason();
    var page   = e.getPage();
    // Log the failure and exit.
}
```

Sample JavaScript Arbortext PE Application

The sample JavaScript applications are included in Arbortext Publishing Engine installation. These applications are on the server and loaded into an Arbortext PE sub-process when it starts. The JavaScript sample applications are:

PE_HOME`\e3\samples\javascript\E3AppTest.js`
PE_HOME`\e3\samples\javascript\e3samples2.js`

The **`E3AppTest.js`** sample is available for testing from the Arbortext Publishing Engine HTML web page (for information, see Monitoring and Reporting Using a Web Browser on page 26). It is handled by the Arbortext PE Request Manager, so you don't need to place it in the ***PE_HOME*****`\custom\init`** directory. The **`E3AppTest.js`** sample application reports information about the Arbortext Publishing Engine environment. Document manipulation, such as opening, closing, and changing content, is accomplished using the AOM and DOM interfaces (refer to the *Programmer's Reference*). For information on **`e3samples2.js`**, see Sample Applications on page 109.

Troubleshooting JavaScript Arbortext PE Applications

Reloading JavaScript Applications

During the development and testing phase, you can make changes to JavaScript applications and then reload them from **`custom\init`**. You can issue an Arbortext Publishing Engine HTTP request specifying **f=init** to reload your JavaScript application. It’s not necessary to stop and restart the servlet container.

If there is a syntax error in your custom application, the error is returned in an HTML page in response to the first Arbortext PE sub-process request to perform work (usually from a request containing a **f=java**, **f=javascript**, **f=vbscript**, or **f=acl** function).

Logging

Your application can obtain a Logger object and write to the servlet log by calling the **getApplicationLogger** method of the **E3ApplicationConfig** interface. The logger level will be set as indicated by the parameters **com.arbortext.e3.applicationLog.javascript.*functionname*** (***functionname*** is the JavaScript function that implements the application), **com.arbortext.e3.applicationLog.javascript**, or **com.arbortext.e3.applicationLog** in **`e3config.xml`**. Refer to *Parameters that Control Application Logging* in Global Arbortext PE Request Manager Parameters on page 41 for more information.

Examining Transaction Files

After your application successfully finishes processing a request, the request, the generated response returned to the client, and any other information generated during processing could be stored in the transaction archive, if the transaction archive is configured to save all transactions.

When debugging an application, you can configure the transaction archive to save all transactions by setting **com.arbortext.e3.transactionArchive.selector** to `all` in **`e3config.xml`**. Then retrieve the transactions that your application fulfilled, and inspect the data passed to your application by the HTTP request and the response returned by your application for those inputs.

Saving Intermediate Files

If your application creates any temporary files or documents as part of generating its the response that is returned to the client, you can save those files to the transaction archive using the **addIntermediateFile** method of the **E3ApplicationConfig** interface (see The E3ApplicationConfig Class on page 121). The intermediate files will accompany the transaction if you use **addIntermediateFile**. They will be placed in the transaction directory, provided the transaction is placed in the transaction archive, and you can examine them by retrieving the transaction from the archive.

Using the Arbortext Publishing Engine Test Utility

Arbortext Publishing Engine offers an interactive testing utility called Arbortext Publishing Engine Test Utility to validate and test your Java, JavaScript, VBScript, and ACL applications as well as document conversion (**f=convert**) parameters. You can test your custom applications without having an Arbortext Publishing Engine production environment in place.

You can launch the Arbortext Publishing Engine Test Utility as a standalone program or from the Arbortext Publishing Engine Interactive **Tools** menu. You choose the test type and set all the parameters and their values for the custom application. The utility constructs the query string from your specifications and validates it. You can also run the test and report the results as though it had been handled by Arbortext Publishing Engine.

If errors occur, they're included in the report. The Arbortext Publishing Engine Test Utility is documented in the *Test Utility User's Guide* manual, which you can find in the **`/docs`** on the Arbortext Publishing Engine distribution archive or CD-ROM, as well as in the ***`PE_HOME`*****`/e3/docs`** directory after you install Arbortext Publishing Engine. The Arbortext Publishing Engine Test Utility standalone executable is located in:

`PE_HOME``\e3\bin\e3test.cmd`

Avoiding Content Type Problems in the Arbortext Publishing Engine HTTP Request

You may experience problems with a returned file if you submit an HTTP or HTTPS request that ends with a file extension. The web browser can interpret the request improperly. If Arbortext Publishing Engine passes the content-type header correctly (for instance, `application/pdf`) in its response to the browser, the web browser may ignore the content-type header and try to render the response based on a file extension occurring at the end of the URL. To avoid this problem, you may want to structure an HTTP request so that file extensions do not appear at the end.

In the following example, the first request may cause a problem. By reordering the parameters in the same request, as in the second example, the request will succeed. The best practice is to place the **f=java**, **f=javascript**, **f=vbscript**, **f=acl**, or **f=convert** specification at the end of the URL.

The following HTTP **f=acl** request may cause a problem because the request ends in **`.xml`**. The web browser may try to interpret XML as the content-type, rather than the PDF content-type which is specified and is the content-type that will be returned. Ignore the line breaks in the examples:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?f=acl&function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml
```

The following HTTP request will succeed because the URL ends with the **f=acl** specification, which won't confuse the web browser:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml&f=acl
```

You may also want to take advantage of the fact that the web browser can interpret content-type from a URL that ends in a file extension. You can include a dummy parameter at the end of the URL to specify a file extension, for example, `dummy=file.pdf` for a PDF file. The dummy parameter will be ignored by Arbortext Publishing Engine, but the web browser may try to render the response based on the file extension `.pdf` occurring at the end of the URL.

9

Writing Arbortext PE Applications in VBScript

An Arbortext PE Application written in VBScript is a VBScript subroutine which takes no parameters. The Arbortext PE Request Manager processes requests for VBScript applications by passing requests whose `f` parameters have the value `vbscript` to the Request Function **com.arbortext.e3.FunctionVbscript**. The request function allocates an Arbortext PE sub-process and passes the request to the Arbortext PE Application Context (described in The Arbortext PE sub-process Application Context on page 111). The Arbortext PE Application Context allocates request and response objects, as for a Java application, and stores references to them in global variables. Then it calls the VBScript function indicated by the request's `function` parameter. The function must have been defined in a **`.vbs`** file in a **`custom\init`** directory or in a **`.vbs`** file in **`custom\scripts`** which is loaded from **`custom\init`**.

VBScript Arbortext PE Applications are specifically for Windows systems. The VBScript Arbortext PE Application is called without any parameters. It obtains request and sets response information by invoking methods in the packages **PEAppRequest** and **PEAppResponse**, which access the request and response objects allocated by the Arbortext PE Application Context.

The return value of the VBScript Arbortext PE Application function is ignored. Whether it succeeds or produces an error response, everything to be returned to the client must be stored in the response object.

Unlike a Java Arbortext PE Application, a VBScript application has no initialization or termination components.

Passing Parameters

The entire HTTP request, including all information about the request provided by the Java Servlet interface, is passed to the Arbortext PE sub-process and made available to the VBScript Arbortext PE Application. This includes the HTTP request headers, query parameters, and request body. The client can pass an arbitrary number of HTTP query parameters to control the behavior of the application.

The VBScript Arbortext PE Application can call the following VBScript functions to retrieve information about the request.

VBScript Functions for Accessing the Request

Function	Purpose
PEAppRequest::getAuthType ( )	Returns the name of the authentication scheme used to protect the Arbortext PE Request Manager servlet; for example, `BASIC`, `SSL`, or an empty string if the servlet is not protected.
PEAppRequest:: getCharacterEncoding( )	Returns the name of the character encoding used in the body of this request or an empty string if the request does not specify a character encoding.
PEAppRequest:: getContentLength( )	Returns the length, in bytes, of the request body or `-1` if the length is not known.
PEAppRequest:: getContentType( )	Returns the MIME type of the body of the request, or an empty string if the type is not known.
PEAppRequest:: getContextPath( )	Returns the portion of the request URI that indicates the context of the request.
PEAppRequest:: getDateHeader(name)	Returns the value of the specified request header as the number of milliseconds since 0:00 January 1, 1970.
PEAppRequest::getHeader(*name*)	Returns the value of the specified request header or an empty string if the header was not specified on the request. If the header has more than one value, only the first is returned.
PEAppRequest:: getHeaderNames( )	Returns an array of the names of each request header.
PEAppRequest::getHeaders(*name*)	Returns an array of all values of header *name*.
PEAppRequest::getInputFile ( )	Returns the absolute path to the disk file containing the message body of the HTTP POST request; returns an empty string if there is no message body

Function	Purpose
	(HTTP GET request, HTTP POST request with a null body).
PEAppRequest:: getIntHeader(*name*)	Included for completeness; identical to **PEAppRequest::getHeader()**.
PEAppRequest::getLocale( )	Returns the preferred locale for the content being sent to the client, based on the request's Accept-Language header
PEAppRequest::getLocales(array[])	Returns an array of the name of each locale the client will accept.
PEAppRequest::getMethod( )	Returns the name of the HTTP method for this request, either `GET` or `POST`.
PEAppRequest:: getParameter(*name*)	Returns the value of a request parameter as a String, or an empty string if the parameter does not exist. If the parameter has more than one value, the first is returned.
PEAppRequest:: getParameterNames(array[])	Returns an array of the name of each request parameter specified on the request.
PEAppRequest:: getParameterValues(*name*)	Returns an array of each value of request parameter *name*
PEAppRequest::getPathInfo( )	Returns any extra path information associated with the URL the client sent when it made the request.
PEAppRequest:: getPathTranslated( )	Returns the extra path information after the servlet name but before the query string, translated to a real path.
PEAppRequest::getProtocol( )	Returns the name and version of the protocol the request uses in the form `protocol/major version.minor version`, for example `HTTP/1.1`.
PEAppRequest:: getQueryString( )	Returns the query string that is contained in the request URL after the path, or an empty string if the URL does not have a query string.
PEAppRequest:: getRemoteAddr( )	Returns the Internet Protocol (IP) address of the client that sent the request.
PEAppRequest:: getRemoteHost( )	Returns the fully-qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.

Function	Purpose
PEAppRequest:: getRemoteUser( )	Returns the login of the user making the request, if the user has been authenticated. Returns an empty string otherwise.
PEAppRequest:: getRequestURI( )	Returns the part of the request's URL from the protocol name to the query string.
PEAppRequest::getScheme( )	Returns the name of the scheme used to make this request, for example `http` or `https`.
PEAppRequest:: getServerName( )	Returns the host name of the server that is processing the request.
PEAppRequest:: getServerPort( )	Returns the port number on which this request was received.
PEAppRequest:: getServletPath( )	Returns the part of this request's URL that resulted in the Arbortext PE Request Manager being invoked.
PEAppRequest::isSecure( )	Returns `true` or `false` showing whether this request was made using a secure channel, such as HTTPS.

Constructing a Response

The VBScript Arbortext PE Application can call the following VBScript subroutines to build the HTTP response that will be returned to the client.

VBScript Functions for Building a Response

Function	Purpose
PEAppResponse:: addDateHeader(*name*, date)	Adds a response header with the given ***name***. The value must be the number of milliseconds since 0:00, January 1, 1970 or a time/date string that the Java layer can convert.
PEAppResponse::addHeader(*name*, *value*)	Adds a response header with the given name and value.
PEAppResponse:: addIntHeader(*name*, *value*)	Included for completeness; identical to **PEAppResponse::addHeader()**.
PEAppResponse:: containsHeader(*name*)	Returns `true` or `false` showing whether the named response header has already been set.
PEAppResponse:: getArchiveFlag	Returns the value of the archive flag, which determines whether the transaction will be archived. When set to `true`, the transaction will be archived.

Function	Purpose
	If it's set to `false`, the transaction will not be archived.
PEAppResponse:: getAlternateArchiveFlag	Returns the value of the alternate archive flag, which determines whether the transaction will be archived in an alternate location. When set to `1`, the transaction will be archived, provided the alternate location is set up in **`e3config.xml`**. If it's set to `0`, the transaction will not be archived in an alternate location.
PEAppResponse:: getCharacterEncoding( )	Returns the name of the character encoding used for the MIME body to be sent in this response.
PEAppResponse::getCode( )	Returns the HTTP response code that will be transmitted to the client.
PEAppResponse:: getHeaderNames(array[])	Returns an array of a list of all response headers that have been set.
PEAppResponse:: getHeaderValues(*name*)	Returns an array of a list of all header values stored for header *name*.
PEAppResponse::getLocale( )	Returns the name of the locale assigned to this response.
PEAppResponse:: getOutputFile( )	Returns the absolute path to the file that's currently defined for transmission to the client. It returns an empty string if **PEAppResponse::setOutputFile()** hasn't been called.
PEAppResponse::getState( )	Returns a code indicating what will be returned to the client. States are as follows: 1: Returns a fabricated HTML page based upon the status code 2: Returns a fabricated HTML page based upon the status code and error message 3: Returns the status code, HTTP headers, and HTML page set using **PEAppRequest:: setOutputPage()** 4: Returns the status code, HTTP headers, and output file specified using **PEAppRequest:: setOutputFile()**. The output file will not be deleted after transmission to the client. 5: Same as state 4, but after transmitting the output file, it's deleted.

Function	**Purpose**
	6: Sends a temporary redirect to the client.
PEAppResponse::getString( )	Returns the response string buffer, or an empty string if the buffer is empty.
PEAppResponse:: hasResultFile( )	Returns `true` or `false` showing whether this response will or will not return a file (i.e., state is 4 or 5).
PEAppResponse:: hasStringResult( )	Returns `true` or `false` showing whether this response has or does not have a string to return (i.e., state is 3).
PEAppResponse::reset( )	Clears the string buffer and output file, and sets the state to 1.
PEAppResponse::sendError(*code*)	Resets the response, then stores status code *code* and sets state to 1.
PEAppResponse:: sendErrorMsg(*code*, *message*)	Resets the response, then stores the status code and error message, and sets state to 2.
PEAppResponse :: sendRedirect(*location*)	Resets the response, sets state to 6, and then saves *location* as the URL to which the client should be redirected.
PEAppResponse:: setArchiveFlag	Returns the value of the archive flag. When set to `true`, the transaction can be archived. If it's set to `false`, the transaction will not be archived.
PEAppResponse:: setAlternateArchiveFlag	Returns the value of the alternate archive flag. When set to `1`, the transaction will be archived, provided the alternate location is set up in **`e3config.xml`**. If it's set to `0`, the transaction will not be archived in an alternate location.
PEAppResponse:: setContentLength(*length*)	Sets the HTTP Content-Length response header.
PEAppResponse:: setContentType(*type*)	Sets the HTTP Content-Type response header.
PEAppResponse:: setDateheader(*name*, *value*)	Sets the HTTP response header *name* to *value*, which should be a time/date expressed as the number of milliseconds since 0:00, January 1, 1970 or a time/date string the Java layer can convert.
PEAppResponse::setHeader(*name*, *value*)	Sets the HTTP response header *name* to *value*.

Function	Purpose
PEAppResponse:: setIntHeader(*name*, *value*)	Included for completeness; same as **PEAppResponse::setHeader**
PEAppResponse::setLocale(*name*)	Sets the response locale, updating HTTP headers as appropriate.
PEAppResponse:: setOutputFile(*path*, *delete*)	Configures the response to return the file *path* (which should be absolute). If *delete* is 0, the file is not deleted after transmission; otherwise the file is deleted after transmission. Sets the response state to 4 or 5, depending upon the value of *delete*.
PEAppResponse:: setOutputPage(*page*)	Configures the response to return the string *page* as the response body. Sets the response state to 3.
PEAppResponse::setStatus(*value*)	Sets the status to be returned with the response to *value*.

Retrieving the Configuration Parameters

The VBScript Arbortext PE Application can call the following routines to retrieve the names and values of the configuration parameters maintained by the Arbortext PE sub-process Application Context.

VBScript Functions for Obtaining Configuration Parameters

Function	Purpose
PEAppConfig::addIntermediateFile(*fileName*, *contentType*, *description*)	copies the file whose absolute path is provided by the *fileName* parameter into the transaction directory as an intermediate file. The *contentType* and *description* parameters are included as comments.
PEAppConfig :: getInitParameter(*name*)	Returns the value of parameter ***name*** or the null string if there is no such parameter.
PEAppConfig :: getInitParameterNames(names[])	Places the name of each defined parameter in the ACL array *names* and returns the number of parameters defined.
PEAppConfig::debug(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.

Function	Purpose
PEAppConfig::error(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::fatal(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::info(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::trace(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::isDebugEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.
PEAppConfig::isInfoEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.
PEAppConfig::.isTraceEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.

Testing VBScript Syntax in Arbortext Publishing Engine Interactive

You can launch Arbortext Publishing Engine Interactive to test the syntax of the code in your custom VBScript applications. When you source your VBScript file, be aware that Arbortext Publishing Engine Interactive will not actually run the application.

To test VBScript applications using Arbortext Publishing Engine Interactive

1. Your VBScript file should be in the ***PE_HOME*\custom\init** directory.
2. Launch Arbortext Publishing Engine Interactive by choosing it from its shortcut on your Arbortext Publishing Engine program group.

   Any scripts in ***PE_HOME*\custom\init** are automatically sourced at startup. If the VBScript application contains syntax or other errors, you'll automatically receive a message explaining the nature of the error.
3. If you wish to leave Arbortext Publishing Engine Interactive running, you can make a change to the VBScript file and source it manually using the Arbortext

Publishing Engine Interactive command line prompt. You would use the **`source`** command and specify the path, like the following example:

```
source path-and-script-name.vbs
```

Note

If the command line is not enabled at the bottom of the interface, choose **Options ▸ Preferences**. *If you want to change the default setting, select the* **Preferences** *button. If you want to change the setting for the current session only, select the* **Current Settings** *button. Then, on the* **Window** *tab, select the* **Command Line** *option. You can exit by choosing the save option of your choice.*

Calling the Conversion Processor from a VBScript Arbortext PE Application

Note

Writing a VBScript application to call the conversion processor is not currently supported.

Sample VBScript Arbortext PE Applications

The sample VBScript applications are included in the Arbortext Publishing Engine installation. These applications are on the server and loaded into an Arbortext PE sub-process when it starts. The VBScript sample applications are:

```
PE_HOME\e3\samples\vbscript\E3AppTest.vbs
PE_HOME\e3\samples\vbscript\e3samples2.vbs
```

The **`E3AppTest.vbs`** sample is available for testing from the Arbortext Publishing Engine HTML web page (for information, see Monitoring and Reporting Using a Web Browser on page 26). It is handled by the Arbortext PE Request Manager, so you don't need to place it in the ***`PE_HOME`*****`\custom\init`** directory. The **`E3AppTest.vbs`** sample application reports information about the Arbortext Publishing Engine environment. Document manipulation, such as opening, closing, and changing content, is accomplished using the AOM and DOM interfaces (refer to the *Programmer's Reference*). For information about **`e3samples2.vbs`**, see Sample Applications on page 109.

Troubleshooting VBScript Arbortext PE Applications

Reloading VBScript Applications

During the development and testing phase, you can make changes to VBScript applications and then reload them from `custom\init`. You can issue an Arbortext Publishing Engine HTTP request specifying **f=init** to reload your VBScript application; it's not necessary to stop and restart the servlet container.

If there is a syntax error in your custom application, the error is returned in an HTML page in response to the first Arbortext PE sub-process request to perform work (usually from a request containing a **f=java**, **f=javascript**, **f=vbscript**, or **f=acl** function).

Logging

Your application can obtain a Logger object and write to the servlet log by calling the **getApplicationLogger** method of the **E3ApplicationConfig** interface. The logger level will be set as indicated by the parameters **com.arbortext.e3.applicationLog.vbscript.*functionname*** (***functionname*** is the VBScript function that implements the application), **com.arbortext.e3.applicationLog.vbscript**, or **com.arbortext.e3.applicationLog** in **`e3config.xml`**. Refer to *Parameters that Control Application Logging* in Global Arbortext PE Request Manager Parameters on page 41 for more information.

Examining Transaction Files

After your application successfully finishes processing a request, the request, the generated response returned to the client, and any other information generated during processing could be stored in the transaction archive, if the transaction archive is configured to save all transactions.

When debugging an application, you can configure the transaction archive to save all transactions by setting **com.arbortext.e3.transactionArchive.selector** to `all` in **`e3config.xml`**. Then retrieve the transactions that your application fulfilled, and inspect the data passed to your application by the HTTP request and the response returned by your application for those inputs.

Saving Intermediate Files

If your application creates any temporary files or documents as part of generating its the response that is returned to the client, you can save those files to the transaction archive using the **PEAppConfig::addIntermediateFile** function. The intermediate files will accompany the transaction if you use **addIntermediateFile**. They will be placed in the transaction directory, provided the transaction is placed in the transaction archive, and you can examine them by retrieving the transaction from the archive.

Using the Arbortext Publishing Engine Test Utility

Arbortext Publishing Engine offers an interactive testing utility called Arbortext Publishing Engine Test Utility to validate and test your Java, JavaScript, VBScript, and ACL applications as well as document conversion (**f=convert**) parameters. You can test your custom applications without having an Arbortext Publishing Engine production environment in place.

You can launch the Arbortext Publishing Engine Test Utility as a standalone program or from the Arbortext Publishing Engine Interactive **Tools** menu. You choose the test type and set all the parameters and their values for the custom application. The utility constructs the query string from your specifications and validates it. You can also run the test and report the results as though it had been handled by Arbortext Publishing Engine. If errors occur, they're included in the report. The Arbortext Publishing Engine Test Utility is documented in the *Test Utility User's Guide* manual, which you can find in the **`/docs`** on the Arbortext Publishing Engine distribution archive or CD-ROM, as well as in the ***`PE_HOME`*****`/e3/docs`** directory after you install Arbortext Publishing Engine. The Arbortext Publishing Engine Test Utility standalone executable is located in:

`PE_HOME``\e3\bin\e3test.cmd`

Avoiding Content Type Problems in the Arbortext Publishing Engine HTTP Request

You may experience problems with a returned file if you submit an HTTP or HTTPS request that ends with a file extension. The web browser can interpret the request improperly. If Arbortext Publishing Engine passes the content-type header correctly (for instance, `application/pdf`) in its response to the browser, the web browser may ignore the content-type header and try to render the response based on a file extension occurring at the end of the URL. To avoid this problem, you may want to structure an HTTP request so that file extensions do not appear at the end.

In the following example, the first request may cause a problem. By reordering the parameters in the same request, as in the second example, the request will succeed. The best practice is to place the **f=java**, **f=javascript**, **f=vbscript**, **f=acl**, or **f=convert** specification at the end of the URL.

The following HTTP **f=acl** request may cause a problem because the request ends in **`.xml`**. The web browser may try to interpret XML as the content-type, rather than the PDF content-type which is specified and is the content-type that will be returned. Ignore the line breaks in the examples:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?f=acl&function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml
```

The following HTTP request will succeed because the URL ends with the **f=acl** specification, which won't confuse the web browser:

```
http://www.myserver.com:8000/e3/servlet/e3
     ?function=e3apps::myapp
     &mime-type=application/pdf
     &file=d:\scripts\mydoc.xml&f=acl
```

You may also want to take advantage of the fact that the web browser can interpret content-type from a URL that ends in a file extension. You can include a dummy parameter at the end of the URL to specify a file extension, for example, `dummy=file.pdf` for a PDF file. The dummy parameter will be ignored by Arbortext Publishing Engine, but the web browser may try to render the response based on the file extension `.pdf` occurring at the end of the URL.

10 Writing Arbortext PE Applications in ACL

An Arbortext PE Application written in ACL is an ACL subroutine which takes no parameters. The Arbortext PE Request Manager processes requests for ACL applications by passing requests whose `f` parameters have the value `acl` to the Request Function **com.arbortext.e3.FunctionAcl**. The request function allocates an Arbortext PE sub-process and passes the request to the Arbortext PE Application Context The Arbortext PE sub-process Application Context on page 111). The Arbortext PE Application context allocates request and response objects, as for a Java application, and stores references to them in global variables. Then it calls the ACL function indicated by the query's **function** parameter. The function must have been defined in a **`.acl`** file in a **`custom\init`** directory or in an **`.acl`** file in a **`custom\scripts`** directory which is loaded from **`custom\init`**. For example, to invoke the ACL function **test::abc**, the HTTP query would include the parameter **function=test::abc**.

The ACL Arbortext PE Application is called without any parameters. It obtains request and sets response information by invoking methods in the packages **PEAppRequest** and **PEAppResponse**, which access the request and response objects allocated by the Arbortext PE Application Context.

The return value of the ACL Arbortext PE Application function is ignored. Whether it succeeds or produces an error response, everything to be returned to the client must be stored in the response object.

Unlike Java Arbortext PE Applications, ACL applications have no initialization or termination components.

Passing Parameters

The entire HTTP request, including all information about the request provided by the Java Servlet interface, is passed to the Arbortext PE sub-process and made available to the ACL Arbortext PE Application. This includes the HTTP request headers, query parameters, and request body. The client can pass an arbitrary number of HTTP query parameters to control the behavior of the application.

The ACL Arbortext PE Application can call the following ACL functions to retrieve information about the request.

ACL Functions for Accessing the Request

Function	Purpose
PEAppRequest::getAuthType ( )	Returns the name of the authentication scheme used to protect the Arbortext PE Request Manager servlet; for example, `BASIC`, `SSL`, or an empty string if the servlet is not protected.
PEAppRequest:: getCharacterEncoding( )	Returns the name of the character encoding used in the body of this request or an empty string if the request does not specify a character encoding.
PEAppRequest:: getContentLength( )	Returns the length, in bytes, of the request body or `-1` if the length is not known.
PEAppRequest:: getContentType( )	Returns the MIME type of the body of the request, or an empty string if the type is not known.
PEAppRequest:: getContextPath( )	Returns the portion of the request URI that indicates the context of the request.
PEAppRequest:: getDateHeader(name)	Returns the value of the specified request header as the number of milliseconds since 0:00 January 1, 1970.
PEAppRequest::getHeader(name)	Returns the value of the specified request header or an empty string if the header was not specified on the request. If the header has more than one value, only the first is returned.
PEAppRequest:: getHeaderNames(array[])	Returns an array of the names of each request header.
PEAppRequest::getHeaders(name, array[])	Returns an array of all values of header *name*.
PEAppRequest::getInputFile ( )	Returns the absolute path to the file on disk containing the message body of the HTTP POST request. It returns an empty string if there is no

Function	Purpose
	message body (HTTP GET request, HTTP POST request with a null body).
PEAppRequest:: getIntHeader(name)	Included for completeness; identical to **PEAppRequest::getHeader()**.
PEAppRequest::getLocale( )	Returns the preferred locale for the content being sent to the client, based on the request's Accept-Language header
PEAppRequest::getLocales(array[])	Returns an array of the name of each locale the client will accept..
PEAppRequest::getMethod( )	Returns the name of the HTTP method for this request, either `GET` or `POST`.
PEAppRequest:: getParameter(name)	Returns the value of a request parameter as a String, or an empty string if the parameter does not exist. If the parameter has more than one value, the first is returned.
PEAppRequest:: getParameterNames(array[])	Returns an array of the name of each request parameter specified on the request, and returns the number of names specified.
PEAppRequest:: getParameterValues(*name*, array[])	Returns an array of each value of request parameter *name*
PEAppRequest::getPathInfo( )	Returns any extra path information associated with the URL the client sent when it made the request.
PEAppRequest:: getPathTranslated( )	Returns the extra path information after the servlet name but before the query string, translated to a real path.
PEAppRequest::getProtocol( )	Returns the name and version of the protocol used by the request, in the form `protocol/major version.minor version`, for example `HTTP/1.1`.
PEAppRequest:: getQueryString( )	Returns the query string that is contained in the request URL after the path, or an empty string if the URL does not have a query string.
PEAppRequest:: getRemoteAddr( )	Returns the Internet Protocol (IP) address of the client that sent the request.
PEAppRequest:: getRemoteHost( )	Returns the fully-qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.

Function	Purpose
PEAppRequest:: getRemoteUser( )	Returns the login of the user making the request, if the user has been authenticated. Returns an empty string otherwise.
PEAppRequest:: getRequestURI( )	Returns the part of this request's URL from the protocol name up to the query string.
PEAppRequest::getScheme( )	Returns the name of the scheme used to make this request, for example `http` or `https`.
PEAppRequest:: getServerName( )	Returns the host name of the server that is processing the request.
PEAppRequest:: getServerPort( )	Returns the port number on which this request was received.
PEAppRequest:: getServletPath( )	Returns the part of this request's URL that resulted in the Arbortext PE Request Manager being invoked.
PEAppRequest::isSecure( )	Returns `0` for false or `1` for true indicating whether this request was made using a secure channel, such as HTTPS.

Constructing a Response

The ACL Arbortext PE Application can call the following subroutines to build the HTTP response that will be returned to the client.

ACL Functions for Building a Response

Function	Purpose
PEAppResponse:: addDateHeader(*name*, date)	Adds a response header with the given ***name***. The value must be the number of milliseconds since 0:00, January 1, 1970 or a time/date string that the Java layer can convert.
PEAppResponse::addHeader(*name*, *value*)	Adds a response header with the given *name* and *value*.
PEAppResponse:: addIntHeader(*name*, *value*)	Included for completeness; identical to **PEAppResponse::addHeader()**.
PEAppResponse:: containsHeader(*name*)	Returns `0` for false or `1` for true showing whether the named response header has already been set.
PEAppResponse:: getArchiveFlag	Returns the value of the archive flag, which determines whether the transaction can be archived. When set to `true`, the transaction can be archived.

Function	Purpose
	If it's set to `false`, the transaction will not be archived.
PEAppResponse:: getAlternateArchiveFlag	Returns the value of the alternate archive flag, which determines whether the transaction can be archived in an alternate location. When set to `1`, the transaction will be archived, provided the alternate location is set up in **`e3config.xml`**. If it's set to `0`, the transaction will not be archived in an alternate location.
PEAppResponse:: getCharacterEncoding( )	Returns the name of the character encoding used for the MIME body to be sent in this response.
PEAppResponse::getCode( )	Returns the HTTP response code that will be transmitted to the client.
PEAppResponse:: getHeaderNames(array[])	Returns an array of a list of all response headers that have been set.
PEAppResponse:: getHeaderValues(***name*****, array [])**	Returns an array of a list of all header values stored for header *name*.
PEAppResponse::getLocale( )	Returns the name of the locale assigned to this response.
PEAppResponse:: getOutputFile( )	Returns the absolute path to the file that's currently defined for transmission to the client. It returns an empty string if **PEAppResponse::setOutputFile()** hasn't been called.
PEAppResponse::getState( )	Returns a code indicating what will be returned to the client. States are as follows: 1: Returns a fabricated HTML page based upon the status code 2: Returns a fabricated HTML page based upon the status code and error message 3: Returns the status code, HTTP headers, and HTML page set using **PEAppRequest:: setOutputPage()** 4: Returns the status code, HTTP headers, and output file specified using **PEAppRequest:: setOutputFile()**. The output file will not be deleted after transmission to the client. 5: Same as state 4, but after transmitting the output file, it's deleted.

Function	Purpose
	6: Sends a temporary redirect to the client.
PEAppResponse::getString( )	Returns the response string buffer, or an empty string if the buffer is empty.
PEAppResponse:: hasResultFile( )	Returns `0` if this response will not return a file (meaning the state is not 4 or 5). Returns `1` if this response will return a file (meaning state is 4 or 5).
PEAppResponse:: hasStringResult( )	Returns `0` if this response does not have a string to return (i.e., state is not 3). Returns `1` if this response has a string to return (i.e., state is 3).
PEAppResponse::reset( )	Clears the string buffer and output file, and sets the state to 1.
PEAppResponse::sendError(*code*)	Resets the response, then stores status code *code* and sets state to 1.
PEAppResponse:: sendErrorMsg(*code*, *message*)	Resets the response, then stores the status code and error message, and sets state to 2.
PEAppResponse :: sendRedirect(*location*)	Resets the response, sets state to 6, and then saves *location* as the URL to which the client should be redirected.
PEAppResponse:: setArchiveFlag	Returns the value of the archive flag. When set to `true`, the transaction can be archived. If it's set to `false`, the transaction will not be archived.
PEAppResponse:: setAlternateArchiveFlag	Returns the value of the alternate archive flag. When set to `1`, the transaction can be archived, provided the alternate location is set up in **`e3config.xml`**. If it's set to `0`, the transaction will not be archived in an alternate location.
PEAppResponse:: setContentLength(*length*)	Sets the HTTP Content-Length response header.
PEAppResponse:: setContentType(*type*)	Sets the HTTP Content-Type response header.
PEAppResponse:: setDateheader(*name*, *value*)	Sets the HTTP response header *name* to *value*, which should be a time/date expressed as the number of milliseconds since 0:00, January 1, 1970 or a time/date string the Java layer can convert.
PEAppResponse::setHeader(*name*, *value*)	Sets the HTTP response header*name* to *value*.

Function	Purpose
PEAppResponse:: setIntHeader(*name*, *value*)	Included for completeness; same as **PEAppResponse::setHeader**
PEAppResponse::setLocale(*name*)	Sets the response locale, updating HTTP headers as appropriate.
PEAppResponse:: setOutputFile(*path*, *delete*)	Configures the response to return the file *path* (which should be absolute). If *delete* is 0, the file is not deleted after transmission; otherwise the file is deleted after transmission. Sets the response state to 4 or 5, depending upon the value of *delete*.
PEAppResponse:: setOutputPage(*page*)	Configures the response to return the string *page* as the response body. Sets the response state to 3.
PEAppResponse::setStatus(*value*)	Sets the status to be returned with the response to *value*.

Retrieving the Configuration Parameters

The ACL Arbortext PE Application can call the following routines to retrieve the names and values of the configuration parameters maintained by the Arbortext PE sub-process Application Context.

ACL Functions for Obtaining Configuration Parameters

Function	Purpose
PEAppConfig::addIntermediateFile(*fileName*, *contentType*, *description*)	copies the file whose absolute path is provided by the *fileName* parameter into the transaction directory as an intermediate file. The *contentType* and *description* parameters are included as comments.
PEAppConfig :: getInitParameter(*name*)	Returns the value of parameter ***name*** or the null string if there is no such parameter.
PEAppConfig :: getInitParameterNames(names[])	Places the name of each defined parameter in the ACL array *names* and returns the number of parameters defined.
PEAppConfig::debug(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.

Function	Purpose
PEAppConfig::error(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::fatal(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::info(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::trace(*message*)	Places *messages* in the servlet log if the application log level is set to display messages of this severity.
PEAppConfig::isDebugEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.
PEAppConfig::isInfoEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.
PEAppConfig::.isTraceEnabled()	Returns`1` if the specified log level is enabled, `0` if it's not.

Testing ACL Syntax with Arbortext Publishing Engine Interactive

If you create custom ACL scripts in the **`custom\init`** or **`custom\scripts`** directories, you can test them using an Arbortext Publishing Engine Interactive session.

To test ACL applications using Arbortext Publishing Engine Interactive

1. Your ACL file should be in the ***`PE_HOME`*****`\custom\init`** directory.
2. Launch Arbortext Publishing Engine Interactive from its shortcut on your Arbortext Publishing Engine program group.

   Any scripts in ***`PE_HOME`*****`\custom\init`** are automatically sourced at startup. If the ACL application contains syntax or other errors, you'll automatically receive a message explaining the nature of the error.
3. If you wish to leave Arbortext Publishing Engine Interactive running, you can make a change to the ACL file and source it manually using the Arbortext Publishing Engine Interactive command line prompt. You would use the **`source`** command and specify the path, like the following example:

```
source path-and-script-name.acl
```

Note

If the command line is not enabled at the bottom of the interface, choose **Tools ▸ Preferences**. *Then, on the* **Window** *panel, select the* **Command Line** *option.*

Calling the Conversion Processor from an ACL Arbortext PE Application

You can write an ACL application that can call the conversion processor (explained in 11 Arbortext Publishing Engine Document Conversion on page 163) by invoking the following function:

e3::convert2(*inFile*, *outFile*, *parameter*[])

- *inFile* must specify the absolute path to a document to be converted. The **e3::convert2** function can not process open documents. If your application creates or modifies a document that will be converted, you must save the document to disk and close it before invoking **e3::convert2**. After **e3::convert2** returns, you can open your document again if you need to make further modifications.
- *outFile* must specify the absolute path to the output file that **e3::convert2** will produce. If this file already exists, it will be overwritten during conversion processing.
- *parameter* must be an ACL associative array. Each array entry must correspond to a valid conversion parameter (see Document Conversion Parameters on page 165 for a list and descriptions).

  For example, to specify a stylesheet:

  ```
  parameter[ "stylesheet" ] = "d:\absolute\path\to\stylesheet.style";
  ```

The **e3::convert2** function returns an associative array encoded as a string. Extract the array content by invoking the function **e3::string_to_array(** ***inString***, ***outArray*** **)**. The resulting array will contain three elements:

- `result`: will be either `ok` or `error`
- `reason`: HTTP reason code (400, 500, or some other valid code)
- `page`: an XHTML page describing the error (the same page an **f=convert** request would return to an HTTP client)

Example of ACL code that calls the **e3::convert2** function:

```
local   inFile   = "c:\absolute\path\to\input\file.xml";
local   outFile = "c:\absolute\path\to\output\file.pdf";
local   parameters[];
```

```
local   resultArray[];

parameters[ "type" ] = "pdf";
parameters[ "stylesheet" ] = "d:\absolute\path\to\stylesheet.style";

local resultString = e3::convert2( inFile, outFile, parameters );

e3::stringToArray( resultString, resultArray );

if ( resultArray[ "result" ] == "ok" ) {
    # conversion succeeded:  continue processing
    }
    else {
    # conversion failed:  error page is in resultArray[ "page" ]
    }
```

Sample ACL Arbortext PE Applications

The sample ACL applications are included in the Arbortext Publishing Engine installation. These applications are on the server and loaded into an Arbortext PE sub-process when it starts. The sample ACL applications are:

```
PE_HOME\e3\samples\acl\E3AppTest2.acl
PE_HOME\e3\samples\acl\e3samples2.acl
```

The **E3AppTest2.acl** sample is available for testing from the Arbortext Publishing Engine HTML web page (for information, see Monitoring and Reporting Using a Web Browser on page 26). It is handled by the Arbortext PE Request Manager, so you don't need to place it in the ***PE_HOME*\custom\init** directory. The **E3AppTest2.acl** sample application reports information about the Arbortext Publishing Engine environment. Document manipulation, such as opening, closing, and changing content, is accomplished using ACL commands and functions (refer to the *Programmer's Reference*). For information about **e3samples2.acl**, see Sample Applications on page 109.

To be able to use the functions in **e3samples2.acl**, you need to copy it the ***PE_HOME*\custom\init** directory. Be sure that you add the ACL package and its functions (**e3samples2::***) to the allowed ACL function list (refer to The Allowed Functions List on page 110 for information). Then issue an **f=init** request, go to the Arbortext Publishing Engine web page and click **Reload Subprocesses**, or stop and restart the servlet container for Arbortext Publishing Engine to make them available.

Troubleshooting ACL Arbortext PE Applications

Reloading ACL Applications

During the development and testing phase, you can make changes to ACL applications and then issue an Arbortext Publishing Engine HTTP request specifying **f=init** or click **Reload Subprocesses** on the Arbortext Publishing Engine web page to reload them from the ***PE_HOME*\custom\init** directory.

If there is a syntax error in your custom application, the error is returned in an HTML page in response to the first Arbortext PE sub-process request to perform work (usually from a request containing a **f=java**, **f=javascript**, **f=vbscript**, or **f=acl** function).

Logging

Your application can obtain a Logger object and write to the servlet log by calling the **getApplicationLogger** method of the **E3ApplicationConfig** interface. The logger level will be set as indicated by the parameters **com.arbortext.e3.applicationLog.acl.*functionname*** (***functionname*** is the ACL function that implements the application), **com.arbortext.e3.applicationLog.acl**, or **com.arbortext.e3.applicationLog** in **e3config.xml**. Refer to *Parameters that Control Application Logging* in Global Arbortext PE Request Manager Parameters on page 41 for more information.

Examining Transaction Files

After your application successfully finishes processing a request, the request, the generated response returned to the client, and any other information generated during processing could be stored in the transaction archive, if the transaction archive is configured to save all transactions.

When debugging an application, you can configure the transaction archive to save all transactions by setting **com.arbortext.e3.transactionArchive.selector** to `all` in **e3config.xml**. Then retrieve the transactions that your application fulfilled, and inspect the data passed to your application by the HTTP request and the response returned by your application for those inputs.

Saving Intermediate Files

If your application creates any temporary files or documents as part of generating its the response that is returned to the client, you can save those files to the transaction archive using the **PEAppConfig::addIntermediateFile** function. The intermediate files will accompany the transaction if you use **addIntermediateFile**. They will be placed in the transaction directory, provided the transaction is placed in the transaction archive, and you can examine them by retrieving the transaction from the archive.

Using the Arbortext Publishing Engine Test Utility

Arbortext Publishing Engine offers an interactive testing utility called Arbortext Publishing Engine Test Utility to validate and test your Java, JavaScript, VBScript, and ACL applications as well as document conversion (**f=convert**) parameters. You can test your custom applications without having an Arbortext Publishing Engine production environment in place.

You can launch the Arbortext Publishing Engine Test Utility as a standalone program or from the Arbortext Publishing Engine Interactive **Tools** menu. You choose the test type and set all the parameters and their values for the custom application. The utility constructs the query string from your specifications and validates it. You can also run the test and report the results as though it had been handled by Arbortext Publishing Engine. If errors occur, they're included in the report. The Arbortext Publishing Engine Test Utility is documented in the *Test Utility User's Guide* manual, which you can find in the **`/docs`** on the Arbortext Publishing Engine distribution archive or CD-ROM, as well as in the ***`PE_HOME`*`/e3/docs`** directory after you install Arbortext Publishing Engine. The Arbortext Publishing Engine Test Utility standalone executable is located in:

`PE_HOME``\e3\bin\e3test.cmd`

Avoiding Content Type Problems in the Arbortext Publishing Engine HTTP Request

You may experience problems with a returned file if you submit an HTTP or HTTPS request that ends with a file extension. The web browser can interpret the request improperly. If Arbortext Publishing Engine passes the content-type header correctly (for instance, `application/pdf`) in its response to the browser, the web browser may ignore the content-type header and try to render the response based on a file extension occurring at the end of the URL. To avoid this problem, you may want to structure an HTTP request so that file extensions do not appear at the end.

In the following example, the first request may cause a problem. By reordering the parameters in the same request, as in the second example, the request will succeed. The best practice is to place the **f=java**, **f=javascript**, **f=vbscript**, **f=acl**, or **f=convert** specification at the end of the URL.

The following HTTP **f=acl** request may cause a problem because the request ends in **`.xml`**. The web browser may try to interpret XML as the content-type, rather than the PDF content-type which is specified and is the content-type that will be returned. Ignore the line breaks in the examples:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?f=acl&function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml
```

The following HTTP request will succeed because the URL ends with the **f=acl** specification, which won't confuse the web browser:

```
http://www.myserver.com:8000/e3/servlet/e3
     ?function=e3apps::myapp
     &mime-type=application/pdf
     &file=d:\scripts\mydoc.xml&f=acl
```

You may also want to take advantage of the fact that the web browser can interpret content-type from a URL that ends in a file extension. You can include a dummy parameter at the end of the URL to specify a file extension, for example, `dummy=file.pdf` for a PDF file. The dummy parameter will be ignored by Arbortext Publishing Engine, but the web browser may try to render the response based on the file extension `.pdf` occurring at the end of the URL.

11 Arbortext Publishing Engine Document Conversion

Arbortext Publishing Engine ships with an ACL Arbortext PE Application for transforming one document into another, possibly of a different format. The application is invoked when the Arbortext PE Request Manager receives an HTTP request with an **f=convert** query parameter. For a **`GET`** request, Arbortext Publishing Engine converts a document on or accessible to the Arbortext PE server. For an HTTP **`POST`** request, Arbortext Publishing Engine converts the document passed as the HTTP message body.

Document conversion supports:

- SGML and XML input documents
- Importing Microsoft Word, RTF, and WordML; Adobe FrameMaker MIF; XML of a different document type; HTML; PDF; and text document formats to XML
- Extracting an input document of the previously listed types from a zip archive
- Applying profiling, stylesheets, publishing rules, data merging, and graphic reference mapping to the input document
- Transforming to HTML, PDF, PostScript, HTML Help, EPUB, RTF, SGML, and XML output formats
- Producing PDF and PostScript using Arbortext Advanced Print Publisher
- Support for returning a zip archive containing the output file and related documents

An example of an **f=convert** request within the HTTP request looks like:

```
type=html&file=D:\Documents\bigdoc.xml&f=convert
```

This request specifies loading the file **`/usr/pedocs/bigdoc.xml`**, and then transforming it to HTML using the default stylesheet for the document type used by **`bigdoc.xml`**. The resulting HTML document will be returned to the web client as the body of the HTTP response.

Every **f=convert** request must specify an output document format. Other parameters specify the input file to be converted for **`GET`** requests and additional instructions on how to read the input document and transform it into the desired output.

The basic flow of control for a conversion operation is to load a document into memory, operate on the in-memory document, then transform the in-memory document to the desired output format on disk. The details vary depending upon the format of the input document, transformation parameters specified, and the desired format for the output document.

You can also call the document conversion processor from an ACL, Java, or JavaScript application. Refer to Customizing Document Conversion on page 188 for information.

Document Conversion Parameters

The following table describes the parameters accepted by the **f=convert** function for processing by the Arbortext PE Request Manager. If a parameter is only used in producing a particular type of output, it is ignored if another output type is specified. Note that because APP produces PDF and PostScript, parameters for other output types don't apply when the print engine is APP.

Note

The FOSI and XSL-FO print engines are on sustained support and do not receive enhancements or maintenance fixes. APP is the recommended engine for print output.

Conversion Parameters

Parameter	Explanation
app-config-file=*pathname*	Specifies the name of the PDF configuration file when using the APP print engine to convert documents to PDF or PostScript only. If it's not specified, the default configuration file for the document type is used. This parameter is ignored for other output types or if the APP print engine is not being used.
app-snapshot=`yes|no`	Specifies whether to generate an APP snapshot zip archive when using the APP print engine to convert documents to PDF or PostScript. Set to `yes` to produce the snapshot. As a result, the transaction will also be archived on the Arbortext PE server (ignoring **archive-transaction**). For more information on using the snapshot zip for troubleshooting, refer to Troubleshooting APP Publishing. If **app-snapshot** is not specified, then the behavior defaults to the value of the ACL **`set appsnapshot`** option. Specifying **app-snapshot** on a request overrides the value of the **`set appsnapshot`** option. This parameter is ignored if the APP print engine is not being used.
archive-transaction=`yes|no`	Specify whether to archive the transaction produced by the request. If set to `yes`, the transaction will be archived if **com. arbortext.e3.transactionArchive.enable** is set to `true` in the **`e3config.xml`**

Parameter	Explanation
	configuration file. If **app-snapshot** is set to `yes`, this parameter is ignored.
change-tracking= **`original|changes|latest`**	If the document contains change tracking markup, this parameter specifies which document version should be published. • `original` publishes the original document with no changes. • `changes` publishes the document showing proposed changes. • `latest` publishes the document with all changes applied (the default and is assumed if the parameter is omitted) This parameter is ignored if output type is `xml`, `sgml`, or `subset=toc`. This parameter has no effect on documents without change tracking.
cropmarks=`yes|no`	Determines whether crop marks appear in PDF output. This parameter is ignored for other output types or if the APP print engine is not being used. The default is `no`.
encoding=*specification*	Specifies the character encoding to use for EPUB, HTML, and Web publishing output. If omitted, the publishing operation uses the document type default. Consult the online help topic *Character sets and encoding* for encoding specification values.
export-stylesheet=*pathname*	Specifies the absolute path to a stylesheet to be used in producing RTF output. When producing RTF output, if **export-stylesheet** is not specified, **stylesheet** will be used instead if **stylesheet** specifies a **`.style`** file. Note that you can specify both a **stylesheet** parameter to transform an XML document to another XML document, and an **export-stylesheet** to produce RTF on the same request. If not specified, the default stylesheet for the document type is used.

Parameter	Explanation
failure-mode= **`linkcheck|localgraphics`**	Specifies method to use to check graphics for the input document. The `linkcheck` method verifies graphic and file entity references of the input file. The `localgraphics` setting collects graphic entity references specified by a URL and copies the graphic entities to a local temporary **`http`** cache subdirectory. The actions performed by `localgraphics` include the actions for `linkcheck`. By default, neither method is applied.
file=*pathname*	Specifies the absolute path, logical ID, or URL for the input file to be converted. Required for **`GET`** requests; optional for **`POST`** requests.
file-type=*type*	Specifies the format of the data in the input file. Refer to Loading a Document for Conversion on page 173 for supported values. **file-type** is optional except when importing WordML documents. If omitted, Arbortext Publishing Engine will try to determine the input file type from the HTTP content-type header for a **`POST`** request. Arbortext Publishing Engine will examine the file extension of the input file for a **`GET`** request.
flatten-entities=`yes|no`	Specifies whether all entities and XML inclusion references should be resolved when producing an XML or SGML document. The default is `no`.
frameset=*directorypath*	Specifies the absolute path to a directory containing a frameset to apply to web publishing output. If omitted, the publishing operation uses the document type default.
graphic-transform=*specification*	Specifies rules for transforming graphics when converting a document to the specified output format. ***specification*** consists of one or more rules separated by vertical bars: ***`rule1`*** | ***`rule2`*** | ***`ruleN`***. A rule consists of a list of input types separated by commas, followed a colon, and then an output type. For example:

Parameter	Explanation

	`svg,cgm:png|bmp:gif`
	specifies two rules, the first converts SVG and CGM graphics to PNG, and the second converts BMP to GIF.
	For example:
	`tif,drw,eps,iso,idr:svg`
	specifies a single rule that converts TIF, DRW, EPS, ISO and IDR graphics to SVG.
	Note
	Conversion to SVG is only supported for IDR and ISO graphics.
	The supported graphic input and output types depend on the type of output format. For all HTML output formats, the types are the same as those specified by **`set graphicwebtransform`** preference. For RTF output, the types are the same as those in the **`set graphicrtftransform`** preference. For print or PDF output using APP, the types are the same as those in the **`set graphicapptransform`** preference. If this parameter is not specified, the values of these preferences is used instead for the associated output types.
	This parameter is ignored if there are no graphics of the type specified, or if the output format is `postscript`, `sgml`, `xml`, or **`subset`**`=toc`.

graphic-web=`yes|no`	This parameter is deprecated. Use the **graphic-transform** parameter instead. Specifies whether to convert embedded intelligent (such as 3D and ISO) graphics to web-friendly graphics (GIF, JPEG, or PNG) when producing HTML or Web output. The default is `no`.
import-inline-map-comments= `yes|no`	Specifies whether to include Import MapTemplate comments in the imported XML file. Setting **import-inline-map-comments** to `yes` includes the comments. The default is `no`.

Parameter	Explanation
import-map=*path*	Specifies the absolute path to the MapTemplate file used when importing a Microsoft Word, Adobe FrameMaker, HTML, PDF, RTF, text, XML, or WordML document to XML. There is no default; this parameter is required if the input document is in any of the supported formats.
input-entry=*filename*	Specifies the file name of the entry in a zip archive to be used as the input document. There is no default; this parameter is required if the input file is a zip archive.
input-entry-type=*type*	Specifies the format of the data in the zip archive entry specified by **input-entry=*filename***. ***type*** can be any supported input file type other than `zip`. **input-entry-type** is optional and if omitted, Arbortext Publishing Engine will examine the file extension of the file name **input-entry** parameter value.
locale=`none`|*coountry code*	Valid when publishing to web via HTTP request Specifies the locale for the publish action, setting the UI language and displaying the correct content. Enter `none` to use the default (current) locale.
pdf-config-file=*pathname*	Specifies the absolute path to a PDF configuration file (**`.pdfcf`**), which controls PDF output using FOSI engine. **pdf-config-file** is optional and if omitted, Arbortext Publishing Engine will use the default configuration file for the document type of the document being converted (usually **`standard.pdfcf`**). For information about PDF configuration files, please refer to Print and PDF Configuration Files in Arbortext Editor Help.
print-engine-override=`app|fosi|xslfo`	Specifies whether to use APP, FOSI, or XSL-FO engine for PDF or PostScript output. The Arbortext PE sub-process saves the current value of

Parameter	Explanation
	`printengineoverride` and then applies this setting before performing the requested document conversion. After the conversion is complete, the Arbortext PE sub-process restores the saved value for **`printengineoverride`**. This parameter is ignored for any other output type or if APP is not being used on the Arbortext PE server.
print-options=*option-list*	Specifies a list of options that are used in producing PostScript output. This parameter is ignored when producing PDF output direct from XML. Refer to Producing PostScript Output on page 182 for supported values. **print-options** is optional and if omitted, no options are passed to the **`print`** command. This parameter is ignored for any other output type or if APP is being used.
profile=*specification*	Specifies how to apply profiling to the input file before converting it to the output format. Refer to Applying a Profile on page 177 for a description of profiling expressions. By default, no profiling is performed.
regmarks=`yes|no`	Determines whether registration marks appear in PDF output. This parameter is ignored for other output types or if the APP print engine is not being used. The default is `no`.
return-composer-errors=`no|error|warning`	Specifies whether to return the event log in place of the output document. This option applies to producing PostScript, HTML, and PDF. The default value is `no`, meaning publishing errors are ignored and the output document is returned (if one is generated).
return-formatter-errors=`yes|no`	Specifies whether, if formatting errors are detected, the errors should be returned in an XML document instead of the formatted output document. The default value `no` returns the output document if one is generated. The value `yes` returns the event log instead of the

Parameter	**Explanation**
	output document if the log contains error messages.
return-parser-errors=`yes|no`	Specifies whether, if parser errors occur when opening an XML or SGML document, the errors should be returned in an XML document instead of the formatted output document. The default value is `no`, meaning parser errors are ignored.
rule-file=*rulefileUniqueID*	Specifies the unique ID of the publishing rule file containing the publishing rule set or publishing rule to execute. The **Unique ID** is assigned when the rule file is created. If ***rulefileUniqueID*** does not specify a valid rule file on the Arbortext PE server, an error is returned. This parameter must be specified along with the **rule=*rulename*** parameter or an error is returned. There is no default value.
rule=*rulename*	Specifies the name of the publishing rule set or publishing rule to be used for publishing the document. If ***rulename*** does not specify a valid rule or rule set in the rule file specified by **rule-file=*rulefileUniqueID***, an error is returned. This parameter must be specified along with the **rule-file=*rulefileUniqueID*** parameter or an error is returned. There is no default value.
use-ruleset-parameters= **`yes|no`**	Specifies whether a parameter value specified in the rule set definition in a rule file will be used or ignored. Ordinarily, the rule set output process specifies its own values for parameters such as **absoluteManifestPaths**, **generateManifest**, **generateRuleLogs**, **outputMode**, **outputModePattern**, **ruleTargetOverride**, and **ruleTargetPattern** rule set parameters. If **use-ruleset-parameters** is set to `yes`, a parameter value specified in the rule set definition in a rule file will be used. For any parameter not specified in the rule file, the default value is used. (Publishing

Parameter	Explanation
	rule set parameters and their defaults are documented in *Customizer's Guide*.) If this parameter is specified,**rule-file=*rulefileUniqueID*** and **rule=*rulename*** must also be specified, or an error is returned. The default is `no` and may be omitted.
stylesheet=*pathname*	Specifies the absolute path, URL, or logical ID of the stylesheet to apply during document conversion. (a **`.style`**, **`.fos.xsl`** or **`.3f`** file). For Web, specify an **`.xsl`** stylesheet or a **`.style`** file that supports XSL. For EPUB, specify a **`.style`** stylesheet. This parameter is optional; the default is the default stylesheet for the document type.
subset=`toc`	Specifies whether to extract a subset of a loaded, profiled document when publishing HTML output. If this parameter is omitted, the entire document is returned. If **subset=toc** is specified, only the table of contents is returned.
type=*target*	Specifies the format of the output document. This parameter is required, except when specifying a publishing rule or rule set (which produces an error). Refer to Conversion Processing on page 176 for a list of output types. If **zip-output=yes** is specified, the output document will be placed in a zip archive and the zip archive will be returned.
xml-char-ent= **`same|char|entref|numref`**	Specifies how non-ASCII characters are handled in XML output. • `same` keeps the characters the same as the source • `char` converts to characters in the target encoding • `entref` converts to character entity references • `numref` (the default) converts to numeric character references

Parameter	Explanation
	This parameter is supported only for the output **type** `xml`, otherwise it is ignored.
xml-header=`yes|no`	Specifies whether to include the XML header when producing XML output. The default is `no`.
zip-graph-dir=*directory*	Specifies the name of a directory that will contain graphic files produced by the conversion process when **zip-output=yes** is specified. Refer to Generating a Zip Archive on page 184 for a description of the default used if this parameter is omitted.
zip-include-composerlog=`yes|no`	Specifies whether to return a zip archive containing a manifest file, the result of the publishing operation, XML and HTML versions of the event log, and an error page if one was generated. The default is `no`. .This parameter is ignored unless **zip-output=`yes`** is specified or the output **type=*target*** specifies `web` (implying **zip-output=`yes`**). Refer to Generating a Zip Archive on page 184.
zip-output=`yes|no`	Specifies whether the conversion output should be placed in a zip archive, and the zip archive returned instead of the output file. The default is `no`. Refer to Generating a Zip Archive on page 184.
zip-root=*filename*	Specifies the name for the published output file if **zip-output=yes** is specified. Refer to Generating a Zip Archive on page 184 for a description of the default used if this parameter is omitted.

Loading a Document for Conversion

The conversion process begins by finding and loading an XML document into the Arbortext PE sub-process. If the input document is not XML, it is imported from its native format to XML, and the resulting XML document is loaded.

Specifying the Input File

For a `GET` request, the HTTP query must include a **file** parameter specifying the input file, otherwise an error is returned.

For a `POST` request, the input file can be included as the HTTP request body or specified in a **file** parameter. If the **file=*pathname*** parameter is specified, the submitted request body will be ignored, as though a `GET` request had been submitted. A `POST` request with neither a **file** parameter or a request body is an error.

The HTTP query parameter **file=*pathname*** must specify one of the following:

- the absolute path to the input file
- a URL that the conversion processor can use to retrieve the input file from the network
- a Logical ID that the conversion processor can use to fetch the input file from a document repository

Determining the Input File Type

For either a `GET` or a `POST` request, the input file type can be specified by the **file-type=*type*** parameter. If a **file** parameter is specified, but no **file-type** parameter is specified, the conversion processor attempts to determine the input file type by examining the file extension. The supported input file types, the **file-type=*type*** parameter values, and the associated file extension mapping are described in the following table.

Input file types

Input file type	file-type parameter values	File extension
SGML	sgml	`.sgm`, `.sgml`
XML	xml	`.xml`, `.dita`, `.ditamap`
HTML	html	`.htm`, `.html`
Microsoft Word	word	`.doc`, `.docx`
WordML	xml	`.xml`
Adobe FrameMaker	frame	`.mif`
Rich Text Format	rtf	`.rtf`
Text	text	`.txt`
PDF	pdf	`.pdf`
Zip archive	zip	`.zip`

For a **POST** request passing the input file as the HTTP request body (no **file** or **file-type** parameter specified), the conversion processor determines the input file type according to the HTTP header parameter `content-type` as follows:

Mapping content-type header to input file type

Header value	Input file type
application/msword	Microsoft Word
application/pdf	PDF
application/rtf text/rtf	Rich Text Format
application/vnd.framemaker application/x-framemaker application/x-frame	Adobe FrameMaker
application/zip	zip archive
text/html	HTML
text/plain	Text
text/sgml	SGML
text/xml	XML. When importing WordML files, the **file-type** parameter must also be set to `wordml`.

Converting a Zip Archive Entry

If the input file is a zip archive, you must specify the **input-entry=*entry*** parameter. If the archive member name specified by **input-entry** does not end in one of the recognized file extensions, you must also specify the **input-entry-type=*type*** parameter.

When the input file is a zip archive, the conversion process create a temporary directory and extracts the archive content to that directory. Then it proceeds as if the values of the **file** and **file-type** functions had been specified as the values of the **input-entry** and **input-entry-type** parameters.

Logical File Identifiers (Logical IDs)

The **file** parameter can specify an input document using a logical file identifier. A logical ID is a URL that specifies an object in a document repository.

To use logical IDs, you must install a custom script to establish a connection to the repository when the Arbortext PE sub-process initializes. For information on the sample scripts included with your Arbortext Publishing Engine installation, refer to Connecting to a Repository Adapter on page 196.

Returning Parser Errors

When the conversion process loads an XML or SGML document, it uses an XML or SGML parser to convert text into a usable document. Some documents may contain parser errors, text patterns not allowed by the XML and SGML standards. By default, the conversion processor ignores parser errors and does its best to process each document as if it were correctly formed.

If you specify **return-parser-errors=`yes`** and the input SGML or XML document contains errors, the conversion processor will return an XHTML document listing the errors.

Checking Graphic References

After the input file has been opened or imported, you can apply optional processing to check for graphic reference errors or to improve graphic handling by setting the **failure-mode** parameter to one of the following values:

- `linkcheck` value directs the conversion processor to scan the loaded document looking for graphic and entity references that point to unknown files. If any are found, the conversion process returns an XHTML document listing the errors. If no errors are found, processing continues normally.
- The `localgraphics` value includes `linkcheck` processing. Then it instructs the conversion processor to copy all remote graphics to a local directory and adjust the graphic references to point to the local files. The local copies will be discarded at the end of the conversion operation.

Specifying **failure-mode=`localgraphics`** can improve performance by reducing the number of times a graphic is retrieved from an external server during the publishing process. This improvement is only realized during the course of a single publishing operation; there is no caching of graphic files between conversion requests.

Conversion Processing

After it loads the input document, the conversion processor translates the loaded document to the specified output format. The operations it performs depend upon the output format selected and the other options specified.

Specifying the Output Format

The output format is specified by the **type** parameter. This parameter both controls the processing performed and specifies the HTTP content-type header value to be used in returning the output file to the HTTP client that submitted the **f=convert** request. The following table lists the supported output types and the corresponding header value.

Output File Types

Type parameter value	content-type header value
epub	application/epub+zip
html	text/html
htmlhelp	application/octet-stream
pdf	application/pdf
postscript	application/postscript
rtf	text/rtf
sgml	text/sgml
web	application/zip
xml	text/xml

There is no defined MIME type for the HTML Help format. Therefore the conversion processor returns a content-type of `application/octet-stream`.

For Web output, the conversion process return a zip archive containing the directory produced by the web publishing process and a number of subdirectories.

In addition, the conversion processor returns a zip archive (content-type `application/zip`) if the parameter **zip-output=yes** is specified on the request. Lastly, the conversion process may return an XHTML page (content-type `text/xml`) with a return code of 500 if it detects an error that prevents it from successfully completing the requested conversion.

Applying a Profile

Profiling is an optional process for deleting specified portions of the input document. For example, a document for a training class might have some content appropriate to the instructor, other content appropriate to students, and content common to both. To produce a student document, the profiling feature would delete content designated only for the instructor. The conversion process applies profiling in one of two ways:

- For the XML and SGML output formats, the conversion process applies profiling as the next step after the document is opened.
- For other output formats, profiling is applied as part of the process of converting to the output format.

Profiling is performed the same way for all outputs, and all the **profile** parameter values are supported. Profiling options can be specified as:

- a logical expression.

  `profile=`***`logicalexpression`***`=`***`markup`***

The markup must match the content model of a **SetProfileGroup** in the profiling DTD.

- a name, which refers to a profile specification that must be defined in the **`.pcf`** profiling configuration file for the document type. Same as **SetProfileGroup**.

  `profile=`***`resolutiongroupname`***`=`***`name`***

- a name, which refers to a profile specification that must be defined in the **`.pcf`** profiling configuration file for the document type. Same as **resolutiongroupname**.

  `profile=`***`setprofilegroup`***`=`***`name`***

When specifying **profile=*logicalexpression*=*markup***, ***markup*** must be either a **ProfileRef** or a **LogicalExpression** element as defined in the **`profiling.dtd`**.

- A **ProfileRef** element specifies a profile selector and value.
- A **LogicaExpression** element specifies either a **LogicalGroup** or a **LogicalNot** element.
  - A **LogicalGroup** element must contain two descendant elements, each of which can be a **LogicalGroup**, a **LogicalNot**, or a **ProfileRef** elements.
  - A **LogicalNot** element must contain one child element, either a **LogicalGroup** or a **ProfileRef** element.

Example of a ProfileRef

```
profile=logicalexpression=
  <ProfileRef alias="User Level" value="Expert"/>
```

Example of two ProfileRefs

```
profile=logicalexpression=
<LogicalExpression>
  <LogicalGroup operator="OR">
    <ProfileRef alias="User Level" value="Expert"/>
    <ProfileRef alias="User Level" value="Typical"/>
  </LogicalGroup>
</LogicalExpression>
```

Applying a Stylesheet

A stylesheet provides instructions on how to convert the input document to the output document. There are two stylesheet parameters, **`stylesheet`**, which can be specified with every output type, and **`export-stylesheet`**, which is only used when producing RTF output.

There are several kinds of stylesheets:

- Arbortext Styler stylesheets (**`.style`** files) can contain definitions for several APP, XSL, or FOSI stylesheets, each for a different output format. The conversion processor extracts and uses the appropriate APP, XSL, or FOSI stylesheet for the target output type.
- FOSI stylesheets (**`.fos`** files) provide instructions for transforming an XML or SGML document to HTML, PDF, or PostScript.
- XSL stylesheets (**`.xsl`** files) provide instructions for transforming one XML or SGML document to another XML or SGML document. In some cases, the resulting document is the output of the conversion operation. In other cases, the XML document produced by the XSL transformation is the input to a further processing step.
- **`.3f`** APP template files provide instructions for transforming an XML or SGML document to PostScript or PDF.

Some conversion operations do not use stylesheet processing at all.

You can specify a stylesheet in the following ways:

- as an absolute path and file name on the Arbortext PE server. To avoid exposing the path to the Arbortext Publishing Engine installation tree, a stylesheet path can be specified using a server-side variable, for example:

  PE_HOME`\doctypes\axdocbook\axdocbook.style`
- as a URL

You can update Arbortext PE sub-process stylesheets and then use `f=init` to clear all previously cached stylesheets without restarting the Arbortext PE Request Manager. By clearing the stylesheet cache, Arbortext PE sub-processes will use the updated stylesheet for the next request that uses it.

Specifying Encoding

If you are converting to EPUB, HTML, HTML Help, or Web output, you can specify a character encoding method using **encoding=*specification***. A character encoding is a rule for representing multi-byte characters as sequences of ASCII characters.

Note

The online help topic Character sets and encoding lists the values for encoding specifications.

The Arbortext PE Request Manager surrounds parameters with double quotes and encodes the characters`\n`, `\f`, `\r`, `\t`, `'`, `"`, `\`, and `$` to alleviate confusion about escaped characters in a request.

Returning Formatter Errors

If the output type is HTML, PostScript, or PDF, one stage in the conversion process runs the Arbortext Publishing Engine formatter. Normally, minor errors in the formatting process are ignored, as Arbortext Publishing Engine attempts to correct minor errors and continue processing.

If you specify **return-formatter-errors=`yes`** and formatter warning or error messages are issued, then the conversion process stops processing and returns an XHTML document with a return code of 500 and a list of the errors.

Errors and warning from Arbortext Advanced Print Publisher (APP) are sent to the event log, which is returned to the client during publishing on the Arbortext PE server.

Returning Composer Errors

When an Arbortext PE sub-process publishes a document, minor errors are normally ignored.

- If you specify **return-composer-errors=`error`** and publishing error messages are issued, then the conversion process discards the output document and returns an XHTML document containing a list of the errors, along with a result code of 500.
- If you specify **return-composer-errors=`warning`** and warning or error messages are issued, then the conversion process discards the output document and returns an XHTML document containing a list of the errors and warnings, along with a result code of 500.
- If you specify **return-composer-errors=`no`** (the default), PE will attempt to correct minor errors and continue processing. Even if warnings or errors are detected, they will be discarded as long as PE is able to produce an output document.

Errors and warning from APP are sent to the event log, which is returned to the client during publishing on the Arbortext PE server.

Producing HTML Output

You can specify any kind of stylesheet (**`.style`**, **`.fos`**, or **`.xsl`**) for producing an HTML output document. If the document being converted contains graphics, and if **zip-ouptut=yes** is specified as a parameter, then the referenced graphics will be copied to a graphic subdirectory and converted to GIF format.

You can customize graphics processing when publishing to HTML, described in Customizing Document Conversion on page 188.

Producing HTML Help Output

You can specify a **`.style`** or **`.xsl`** stylesheet when converting to HTML Help. The XML document produced by the stylesheet is passed to the Microsoft HTML Help compiler, which must be installed on the Arbortext PE server. The compiler can produce a number of output files, but only the **`.chm`** file is returned as the result of the conversion operation. Because the HTML Help compiler is a Windows program, you can produce HTML Help only on Arbortext Publishing Engine running on a Windows platform.

Refer to Publishing a Document for HTML Help and Setup Considerations for Arbortext Publishing Engine for further information.

Producing EPUB Output

You can specify a **`.style`** stylesheet and encoding when converting to EPUB. The Arbortext PE server must have Calibre installed. The **`set epubinstalldir`** ACL option must be specified on the Arbortext PE server to the location where Calibre 0.8.0 or later is installed. A valid Calibre install directory must contain the files **`ebook-convert.exe`** and **`ebook-viewer.exe`**. You can place the **`set epubinstalldir`** statement in an ACL script in the ***`Arbortext-path`*`/custom/init`** directory to declare the location of the installation.

You must also install Calibre on the Arbortext Editor client and implement **`set epubinstalldir`** to enable the EPUB viewer.

Producing PDF Output

The conversion processor can produce PDF using the Arbortext Publishing Engine with the FOSI print engine or the APP print engine, or it can produce PostScript output.

For information about PDF configuration files, please refer to Print and PDF Configuration Files in Arbortext Editor Help.

Note

The FOSI and XSL-FO print engines are on sustained support and do not receive enhancements or maintenance fixes. APP is the recommended engine for print output.

Producing PDF Using FOSI

You can control many details of PDF publishing by specifying a PDF configuration file. The **pdf-config-file** parameter specifies the absolute path to the **`.pdfcf`** file you want. *Installation Guide for Arbortext Publishing Engine* provides information on configuring Arbortext Publishing Engine to produce PDF. Also, refer to the *Customizer's Guide* for information on PDF configuration files.

Producing PDF Using APP

When producing PDF using Arbortext Advanced Print Publisher, the conversion processor uses APP to publish documents based on an Arbortext Styler stylesheet that uses APP for print or PDF or on a native `.3f` stylesheet. Creating Arbortext Styler stylesheets for APP is documented in the Arbortext Styler documentation. These stylesheets must be on the Arbortext PE server to be available to Arbortext Editor and other client applications. APP extends the functionality available from an Arbortext Styler stylesheet, particularly support for CJK languages and Hebrew, Arabic, and Thai.

Producing PostScript Output

You can specify any type of stylesheet to produce PostScript. Windows users need to select a PostScript printer as their default printer using the **Arbortext Publishing Engine Configuration** program, available from a shortcut on the Arbortext Publishing Engine program group.

You can control the behavior of the PostScript generator by specifying the **print-options** parameter. The **print-options=*option-list*** parameter can take some of the arguments that are supported for the **`print composed`** command of Arbortext Editor. The Arbortext Publishing Engine Interactive online help **`print`** command topic provides information about the supported **`print composed`** command arguments that are not specifically excluded in the following list.

Caution

*The conversion processor issues a **`print composed`** command and specifies some command parameters to produce the output. To be sure you don't override the values Arbortext Publishing Engine uses, do **not** specify any of the following print options:*

`[all | current | page_range]`

`[onepass | allpasses | noformat]`

`[force | auto]`

`[wait | nowait]`

`[printer=printer_name]`

`[file=path_name]`

`[color | monochrome]`

`[panel]`

`stylesheet=path`

The following options are safe for use:

`All |` ***`page_range`***

`portrait | landscape`

`papersize=uslegal | usletter | a4 | b5`

`paperheight=`***n***

`paperwidth=`***n***

`options=`***"pubps_options"***

`copies=`***n***

To submit a list of values for **print-options=*option-list*** that contains spaces and quotation marks, you need to replace them in the HTTP request with hexadecimal notation. For instance, a space is noted as `%20`, and a quotation mark is noted as `%22`. The following example sets `landscape` orientation with `datemark on`:

```
f=convert&print-options=landscape%20options=%22-datemark%20on%22
```

If you are having problems producing PostScript, refer to the *Installation Guide for Arbortext Publishing Engine* section on printer setup considerations.

Producing RTF Output

You may specify two stylesheets when converting to RTF output, one using the **stylesheet** parameter and one using the **export-stylesheet** parameter. The **stylesheet** parameter is an XSL stylesheet (**`.xsl`**) that specifies an in-memory transformation that is performed before the document is translated to RTF. If omitted, no in-memory translation is performed. The **export-stylesheet** parameter must specify an Arbortext Styler **`.style`** file. If omitted, the default stylesheet for the document type of the input document is used to produce the RTF output. If the RTF document contains graphics, Arbortext Publishing Engine will return a zip archive.

Producing Web Output

Specifying Web output always returns a zip archive containing the output file. This means that processing is conducted as though **zip-root=`on`**, and the parameters **zip-root** and **zip-graph-dir** are ignored. The **frameset** parameter controls the output structure inside the zip archive. The **encoding** parameter can specify the character encoding.

Producing XML and SGML Output

You can specify an XSL (**`.xsl`**) stylesheet to transform the input document to these output types. The stylesheet may be omitted, in which case no transformation is performed.

Controlling the XML Header in XML Output

When the output type is XML, you can choose whether to write or omit the XML header when the output document is generated by specifying the parameter **xml-header=`yes|no`**. The XML header typically includes the DOCTYPE declaration and any internal ENTITY declarations that normally appear at the top of a standard DOCTYPE header. By default, the XML header is not generated.

Flattening Entities in the Output

When the output type is XML or SGML, you can choose whether to flatten all entities and XML inclusion references in the output document by specifying the parameter **flatten-entities=`yes|no`**.

Generating a Zip Archive

The HTTP protocol only allows one file to be returned as the body of the HTTP response. Output formats such as HTML, XML, SGML, and RTF can produce a file accompanied by a directory of graphic files which are referenced by the output file. If a single output file is returned, the client can't access its graphic files.

Specifying **zip-output=yes** returns the result in a zip archive, which is a single file that can contain any number of files and subdirectories. For HTML, the archive contains the HTML document and the graphics to which it refers.

When this option is specified, all output, regardless of output type, is written to a single target directory. For output types such as HTML, XML, SGML, and RTF, which support external graphic references, the conversion process allocates a graphic subdirectory in the output directory and copies all graphic files to the graphic subdirectory.

The conversion processor copies all graphics referenced by the input document to a subdirectory of the output directory. Like profiling and stylesheet application, the graphic copying process can be part of the output conversion operation (for output types HTML, RTF, and Web) or immediately after stylesheet application (for the other output types). Then the conversion processor adjusts the graphic references in the input document to reference the graphics using relative path names within the archive.

If the output type is PostScript, PDF, EPUB, or HTML Help, this step is skipped because the conversion processor embeds the graphics inside the output file.

By default, the name of the output file is **`e3out.`*`ext`***, where ***ext*** is the file extension appropriate to the type of document being converted (**`.htm`**, **`.pdf`**, **`.rtf`**, and so on). You can specify the parameter **zip-root** to specify another file name.

If a graphic subdirectory is required, the default name will be **`external.graphics`** if no **zip-root** parameter is specified. If **zip-root** is specified, the default graphic subdirectory name will be ***root.graphics*** where ***root*** is the value of the **zip-root** parameter. You can specify the **zip-graph-dir** parameter to specify any graphic subdirectory name you with.

If the output type is Web, then the **zip-output**, **zip-root**, and **zip-graph-dir** parameters are ignored. The publishing capability always produces an output directory, and it's automatically placed in a zip archive. The content of the directory is controlled by the conversion process.

Example

```
type=html
zip-output=yes
```

The output file will be named **`e3out.htm`** and the graphic directory will be named **`external.graphics`**.

Example

```
type=html
zip=output=yes
zip-root=testroot.htm
```

The output file will be named **`testroot.htm`** and the graphic directory will be named **`testroot.htm.graphics`**.

Example

```
type=html
zip=output=yes
zip-root=testroot.htm
zip-graph-dir=mygraphics
```

The output file will be named **`testroot.htm`** and the graphic directory will be named **`mygraphics`**.

Requesting an Event Log

A client specifying **zip-include-composerlog=`yes`** needs to anticipate the possible file structure returned, as it can vary based on what happens during publishing. When a zip archive is requested and **zip-include-composerlog=`yes`**, the returned result can be one of the following:

- A zip archive of a specific format, which contains a manifest file listing the other files returned, the result of the publishing operation (whether it succeeds or generates errors), XML and HTML versions of the event log, and an error page if one was generated.

  The manifest file provides an index to the zip archive contents. Arbortext PE Request Manager returns an HTTP result code of 200 if it returns a zip archive, regardless of whether the publishing operation is successful or has errors.

- A text/xml document containing an XHTML error page, if processing fails without being able to construct a zip archive.

In the case of an error generating only a text/xml type XHTML error page, there is no zip archive. The client should be able to handle the return of a text/xml document as a fatal error. Arbortext PE Request Manager returns an HTTP result code of 500 if it returns an XHTML error page.

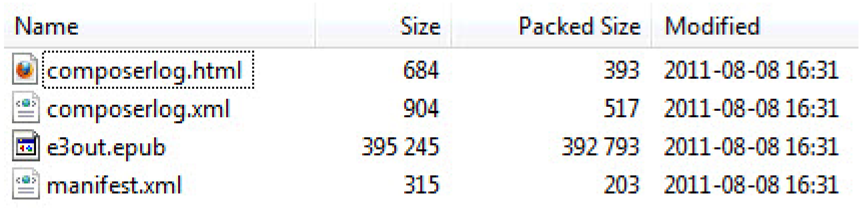
The **`manifest.xml`** file is a short XML document of the following type:

```
<!ELEMENT convert-result EMPTY >
<!ATTLIST convert-result
  success            (yes|no)  #REQUIRED
  transaction-id     #CDATA    #IMPLIED
  subprocess-id      #CDATA    #IMPLIED
  status-code        #CDATA    #REQUIRED
  fatal-count        #CDATA    #IMPLIED
  output-name        #CDATA    #IMPLIED
  reason-phrase      #CDATA    #REQUIRED
  html-composer-log  #CDATA    #REQUIRED
  error-count        #CDATA    #IMPLIED
  warning-count      #CDATA    #IMPLIED
  error-name         #CDATA    #IMPLIED
  graphic-name       #CDATA    #IMPLIED
  composer-log       #CDATA    #REQUIRED
</convert-result>
```

Using the EPUB sample as an example, you could specify:

```
http://pe-server:8080/e3/servlet/e3?file=$aptpath/e3/e3/e3demo.xml
     &type=epub&f=convert&zipoutput=yes&zip-include-composerlog=yes
```

The zip archive would contain the following:



Name	Size	Packed Size	Modified
composerlog.html	684	393	2011-08-08 16:31
composerlog.xml	904	517	2011-08-08 16:31
e3out.epub	395 245	392 793	2011-08-08 16:31
manifest.xml	315	203	2011-08-08 16:31

The returned manifest would be something like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<convert-result
success="yes"
transaction-id="74949"
subprocess-id="2832"
status-code="200"
fatal-count="0"
output-name="e3out.epub"
reason-phrase="OK"
html-composer-log="composerlog.html"
```

```
error-count="0"
warning-count="0"
composer-log="composerlog.xml">
</convert-result>
```

Producing Output Using Publishing Rules

The following parameters are used when constructing a request that uses a publishing rule or rule set:

- **file=*pathname*** (Required for **`GET`** requests)
- **file-type=*type*** (Optional)
- **input-entry=*filename*** (Required if input file is a **`.zip`**)
- **input-entry-type=*type*** (Optional)
- **rule-file=*rulefileUniqueID*** (Required)

  Must specify a valid Unique ID for the rule file. The **Unique ID** is assigned when the rule file is created and is stored in the rule file as the **uuid**. Rule file IDs must be unique on the Arbortext PE server system. If rule files with the same Unique IDs exist on the client system, they are effectively ignored when publishing with Arbortext Publishing Engine.
- **rule=*rulename*** (Required)

  Must specify a valid name for a publishing rule or rule set in the specified rule file. The rule's document type must be the same as that of the document being published.
- **use-ruleset-parameters=`yes|no`** (Optional)

If the request specifies **zip-output=`yes|no`** and **zip-include-composerlog= `yes|no`**, they will be ignored. All other **f=convert** parameters will return an error.

Conversion Result

The result of a document conversion (**f=convert**) request is an HTTP response, which is transmitted to the client that submitted the HTTP request. Like all HTTP responses, the result consists of a result code, a series of HTTP headers, and a body file.

The Conversion Response Code

If the conversion succeeds, the HTTP result code is 200 (`OK`). If conversion fails, the HTTP result code is usually 500 (`Internal Server Error`).

The Conversion Result Body

If conversion succeeds, the body returned as part of the HTTP response is the output of the conversion operation. If conversion fails, the response body is ordinarily an XHTML document describing the failure in more detail. In either case, as required by the HTTP standard, the response's **content-type** HTTP header specifies the type of data returned.

The Conversion Result Header

The conversion response returns a single HTTP header, **content-type**, to indicate the type of file encoded in the result body. Refer to Conversion Processing on page 176 for the values used. Because it's possible that a **content-type** can be ignored by a web browser, be sure to review Troubleshooting Conversion Processing on page 190.

Customizing Document Conversion

The Arbortext Publishing Engine conversion processor performs a number of consecutive processing steps; each step could be customized in a number of ways to produce different results. However, all the functionality provided by the Arbortext PE sub-process can't be wrapped into a single routine.

Approach customizing the conversion process in the following ways:

- You can use a built-in customization specifically for HTML output with graphics.
- The conversion processor is an ACL subroutine. You can write a Java, JavaScript, or ACL custom application that calls the conversion routine from your code.

Mapping Graphic Paths for HTML

Mapping graphics paths addresses the problem of returning an HTML document that contains graphics without needing to bundle the graphics with the output file. If you specify **zip-output**=`yes`, Arbortext Publishing Engine returns the converted HTML document and the graphic files it references. With this customization, you can change the output HTML document to refer to all graphics using a URL (`http://...` references), provided your output document type is HTML and you do not specify **zip-output**=`yes`. This approach works if you can place all of your graphics on a web server.

Create an ACL script that will be placed in a ***`PE_HOME`*****`\custom\init`** directory in the Arbortext Publishing Engine install tree on the Arbortext PE server. This script must require the ACL package ***`PE_HOME`*****`\e3\packages\e3.acl`** by issuing the command **`require e3`** (the path isn't needed). Invoke the ACL function **e3::add_html_image_ map(** ***path***, ***url*** **)** for each graphics path you want to map by specifying the initial path for a graphic reference as a string and a corresponding URL as a string. When the document is converted to HTML, each graphic reference path that matches the specified initial path string will be changed to the initial URL string in the output document.

Example ACL script to map graphics paths

```
require e3; e3::add_html_image_map( "c:\\graphics",
"http://myserver/graphics" ); e3::add_html_image_map( "d:\\graphics",
"http://anotherserver/graphics" );
```

In the path string, note that the backslash must be escaped by another backslash. If the converted HTML document contains the following graphic references:

- **`c:\graphics\sample.gif`**
- **`d:\graphics\test.jpg`**
- **`e:\graphics\keep.cgm`**

The graphic references in the output HTML document will be mapped to:

- `http://myserver/graphics/sample.gif`
- `http://anotherserver/graphics/test.jpg`
- `e:\graphics\keep.cgm`

The last graphic reference is not changed because there is no match for its path (**`e:\graphics`**) in a **e3::add_html_image_map** function call.

Calling the Conversion Processor from a Custom Application

You can write an ACL, Java, or JavaScript Arbortext Publishing Engine application that performs pre-processing, calls the conversion processor, and then performs post-conversion processing. Calling the conversion processor varies by application language. For information on calling the conversion processor from each supported programming language, refer to:

- Calling the Conversion Processor from an ACL Arbortext PE Application on page 158
- Calling the Conversion Processor From a Java Arbortext PE Application on page 122
- Calling the Conversion Processor from a JavaScript Arbortext PE Application on page 130

Calling the conversion processor from a VBScript application is not supported.

Note

For all application languages, the conversion processor cannot process a document that has already been opened by an Arbortext PE sub-process. If your pre-conversion processing involves creating or modifying the document that is to be converted, you must save the document to disk and then close it before invoking the conversion processor.

Troubleshooting Conversion Processing

The following sections provide some troubleshooting suggestions when using the built-in conversion processor.

Initializing the Server

Make sure Arbortext PE sub-process is using the most recent version of a custom ACL script, you can force an initialization using one of the following methods:

- Stop and restart the web server.
- Issue an Arbortext Publishing Engine **f=init** function request.

You should also check that the ACL script is configured in the allowed function list in **`e3config.xml`**. A **ClientFunction** element under the **AllowedFunctions** element must provide permission to run the script, as described in The Allowed Functions List on page 110.

Returning Errors

If the Arbortext PE sub-process encounters an error condition, Arbortext Publishing Engine will return an HTTP `content-type` of `text/html` with an XHTML document describing the error. The error returns an HTTP status code of `500` (internal server error).

Disabling Friendly Error Messages

If you're using Microsoft Internet Explorer, it has an option to control how an error message is displayed by the browser. The web browser setting can substitute a generic HTML error page in place of the error page actually transmitted. To ensure error messages are passed through, choose **Tools ▸Internet Options ▸Advanced** and clear the box labeled **Show friendly HTTP error messages**.

Tracing Conversion Progress

You can trace every conversion operation by placing the following ACL script in a file in your **`custom\init`** directory:

```
require e3;
$e3::g_debug=2
```

The trace output is written to the Arbortext Diagnostics window on Windows. You can launch it from the shortcut called **Arbortext Publishing Engine Diagnostics** in your PTC program group.

Trace output will be logged to standard output in the file ***TOMCAT_HOME*\logs \catalina.out**, where *TOMCAT_HOME* is the absolute path to the directory where you installed the Apache Tomcat servlet container.

Using the Arbortext Publishing Engine Test Utility

Arbortext Publishing Engine offers an interactive testing utility called Arbortext Publishing Engine Test Utility to validate and test your Java, JavaScript, VBScript, and ACL applications as well as document conversion (**f=convert**) parameters. You can test your custom applications without having an Arbortext Publishing Engine production environment in place.

You can launch the Arbortext Publishing Engine Test Utility as a standalone program or from the Arbortext Publishing Engine Interactive **Tools** menu. You choose the test type and set all the parameters and their values for the custom application. The utility constructs the query string from your specifications and validates it. You can also run the test and report the results as though it had been handled by Arbortext Publishing Engine. If errors occur, they're included in the report. The Arbortext Publishing Engine Test Utility is documented in the *Test Utility User's Guide* manual, which you can find in the **/docs** on the Arbortext Publishing Engine distribution archive or CD-ROM, as well as in the ***PE_HOME*/e3/docs** directory after you install Arbortext Publishing Engine. The Arbortext Publishing Engine Test Utility standalone executable is located in:

PE_HOME`\e3\bin\e3test.cmd`

Avoiding Content Type Problems in the Arbortext Publishing Engine HTTP Request

You may experience problems with a returned file if you submit an HTTP or HTTPS request that ends with a file extension. The web browser can interpret the request improperly. If Arbortext Publishing Engine passes the content-type header correctly (for instance, `application/pdf`) in its response to the browser, the web browser may ignore the content-type header and try to render the response based on a file extension occurring at the end of the URL. To avoid this problem, you may want to structure an HTTP request so that file extensions do not appear at the end.

In the following example, the first request may cause a problem. By reordering the parameters in the same request, as in the second example, the request will succeed. The best practice is to place the **f=java**, **f=javascript**, **f=vbscript**, **f=acl**, or **f=convert** specification at the end of the URL.

The following HTTP **f=acl** request may cause a problem because the request ends in **.xml**. The web browser may try to interpret XML as the content-type, rather than the PDF content-type which is specified and is the content-type that will be returned. Ignore the line breaks in the examples:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?f=acl&function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml
```

The following HTTP request will succeed because the URL ends with the **f=acl** specification, which won't confuse the web browser:

```
http://www.myserver.com:8000/e3/servlet/e3
    ?function=e3apps::myapp
    &mime-type=application/pdf
    &file=d:\scripts\mydoc.xml&f=acl
```

You may also want to take advantage of the fact that the web browser can interpret content-type from a URL that ends in a file extension. You can include a dummy parameter at the end of the URL to specify a file extension, for example, `dummy=file.pdf` for a PDF file. The dummy parameter will be ignored by Arbortext Publishing Engine, but the web browser may try to render the response based on the file extension `.pdf` occurring at the end of the URL.

IV

Arbortext Publishing Engine Clients

12 Using Adapters with Arbortext Publishing Engine

There are several approaches for using an adapter with Arbortext Publishing Engine.

- A content management repository may be an Arbortext Publishing Engine client. That is, it can ask Arbortext Publishing Engine to publish documents based upon workflow events. For instance, if you check in a new version of a document, the repository could ask Arbortext Publishing Engine to produce a PDF.
- An Arbortext PE sub-process may use an adapter to retrieve information from a repository, or to store data directory into a repository.
- The two previous approaches can work together. The repository might ask Arbortext Publishing Engine to produce a PDF and pass a POID, rather than a document. In such a case, the Arbortext PE sub-process would have to be connected to the repository to retrieve the document, publish it, and (possibly) store the result directly into the repository.

Implementing these approaches require custom code for Java, JavaScript, VBScript, or ACL. The **f=convert** function can accept a POID specifying the input file, but it doesn't store a file to a repository object. It returns the document to the HTTP client (which could be the repository).

Connecting to a Repository Adapter

JavaScript sample functions provide a starting point for implementing repository connections using Arbortext Publishing Engine. A sample JavaScript function in ***PE_HOME*\e3\samples\javascript\e3samples.js** shows how to establish a repository connection. The function name is specified in an **f=javascript** HTTP request to Arbortext Publishing Engine. You must update the list of `AllowedFunctions` in the **e3config.xml** configuration file to add the JavaScript function names you want to use. Place your modified JavaScript file in ***PE_HOME*\custom\init**. The functions in it are automatically loaded when the Arbortext PE sub-process starts.

Note

If repository credentials or other sensitive information is stored in **web.xml** *or* **e3config.xml***, you should remove permission to access the ACL, JavaScript, VBScript and Java sample applications from the Allowed Functions list in the* **e3config.xml** *configuration file. These sample applications display the global parameters, which would be a security issue if the parameters contain confidential information.*

You can give a specific Arbortext Publishing Engine user account exclusive permission to read a file containing user credentials. These sample functions show how to read such a file on the server and pass the credentials. The function can retrieve and pass a valid ***username*** and ***password*** to establish the repository connection.

The sample function **repository_connect_windchill** establishes a connection to Windchill PDMLink or Arbortext Content Manager with the PTC Server connection.

If you will be using these samples to initiate a permanent connection to the repository so that Arbortext Publishing Engine operations such as **f=convert** will have access to the repository's objects, the `session.disconnect();` line in the script will need to be removed or commented out in the function.

By default, Arbortext Publishing Engine runs on Windows under a local account called `SYSTEM`. You can create a different user account for Arbortext Publishing Engine (see *Installation Guide for Arbortext Publishing Engine* for instructions). Access to files on a Windows server machine is controlled by NTFS security. You can give this specific Arbortext Publishing Engine user account exclusive permission to read from a particular file.

After configuring an Arbortext Publishing Engine user account, set the permissions on your credentials text file to give exclusive read access to the Arbortext Publishing Engine user account. All other accounts should have no access. To test the Arbortext Publishing Engine user account's access to the secure file, log in to Windows as the Arbortext Publishing Engine user ID and try to access the file. After you've excluded other users with accounts on your system, you can log in using one of those accounts and make certain the file is not accessible.

Using an ASCII text file for the password file prevents someone from trying to obtain access to the file using a HTTP request containing the **f=convert** function. An Arbortext Publishing Engine request to convert and return a text file will fail, even if the request specifies the correct path and file name for the credentials file.

13 Using the Java Client SDK

The Arbortext Publishing Engine Java Client SDK allows client applications written in Java to access the Arbortext Publishing Engine.

Use of the Java Client SDK is optional. The Arbortext Publishing Engine is an HTTP server, so you could also write your own client logic using the networking support in the Java library.

Installing the Java Client SDK

You can find the Java Client SDK on the Arbortext Publishing Engine distribution (either download archive or CD-ROM) in the following directories:

`\server\e3\client\javaclientsdk.zip`

Extract the archive for your platform to a convenient location. Add the following paths to your **CLASSPATH** environment variable; ***install-path*** is the directory where you installed the Java Client SDK:

- ***`install-path`*****`\e3client.jar`** — Java Client SDK.
- ***`install-path`*****`\samples\e3samples.jar`** — Java Client SDK example and test code.

The JVM requirements for Java Client SDK are JVM 1.7 or higher. The Microsoft JVM is not supported.

Overview of the Java Client SDK

The Arbortext Publishing Engine Java Client SDK consists of the following files and folders:

- An **`e3client.jar`** file containing the `com.arbortext.e3.client` compiled package.
- A **`javadoc`** folder containing the HTML Javadoc files for the `com.arbortext.e3.client` package.
- A **`readme.txt`** file describing the files included in this package.
- A **`samples`** folder containing:
  - The **`e3samples.jar`** file containing the `com.arbortext.e3.test.script` package and the `com.arbortext.e3.test.SimpleTest` class.
  - Shell script **`e3script.bat`** for starting the sample `com.arbortext.e3.test.script` program.
  - A **`samplescript.txt`** file for testing the `com.arbortext.e3.test.script` program.
  - A **`src`** directory containing Java source files for the **`e3samples.jar`** script engine example and the **`SimpleTest`** example. These source files show how to use the Java Client SDK. You can use the script engine example to perform simple tests.
  - A **`readme.txt`** file describing the files in the **`samples`** folder.

The Java Client SDK Package

The Java package name for the Java Client SDK is `com.arbortext.e3.client`. All of the classes and methods within this package will function correctly in a multi-threaded environment.

See the `com.arbortext.e3.client` Javadoc for detailed information about the classes and interfaces. The start page for the `com.arbortext.e3.client` Javadoc is ***`install-path`*****`\javadoc\index.html`**, where ***install-path*** is the directory where you installed the Java Client SDK.

Sample Java Client SDK Code

The **`e3samples.jar`** file includes one or more classes from the `com.arbortext.e3.test.script` package and the `com.arbortext.e3.test.SimpleTest` class. The source code for both samples is in the ***`install-path`*****`\samples\src`** directory.

The `SimpleTest` class is designed as an introduction to the Java Client SDK. The `SimpleTest` class is an entry level view of how the Java Client SDK works.

The script source code shows how the classes in the **`e3client.jar`** file were used to construct the script class in the `com.arbortext.e3.test.script` package. The script source code also demonstrates how to perform multi-threaded execution using the Java Client SDK.

Note that the sample script demonstrates the Java Client SDK and is not intended to be used as a production solution. Refer to the script code as you create your own application using the native Java classes in the **`e3client.jar`** file.

Testing the Java Client SDK

The `com.arbortext.e3.test.script` is a working sample of an application which uses the Java Client SDK. Use this code to verify that the Java Client SDK can communicate with Arbortext Publishing Engine. The test files are located in the ***`install-path`*****`\samples`** folder.

Use the sample `com.arbortext.e3.test.script` to test the Java Client SDK. This package runs a script for passing commands to Arbortext Publishing Engine. A demonstration script file named **`samplescript.txt`** exercises the Java Client SDK.

You will need to edit to **`samplescript.txt`** before you can test the Java Client SDK:

- Set the `e3` variable to the URL that specifies Arbortext Publishing Engine for your servlet container.

- By default, the **`samplescript.txt`** test script uses input and output files relative to the current directory. If you need to change this setting, edit the `basedir` variable.

To run the **`samplescript.txt`** test, execute the following at the command line:

e3script.bat samplescript.txt

Note

*The **`samplescript.txt`** test intentionally contains a bad command to exercise error handling capability.*

14 Troubleshooting Tips

Be sure to also consult the troubleshooting sections of *Configuration Guide for Arbortext Publishing Engine*.

Checking the Publishing Configuration Report

The server publishing configuration report is available in the following ways:

- from the Arbortext Editor client, choose **Help ▸About Arbortext Editor ▸PE Configuration**
- from the Arbortext Publishing Engine Testing page (`http://`***`server`***`:`***`port`***`/e3/`), click the **Short** link.

You can update the report by clicking the **Rescan Publishing Configuration** link from the Arbortext Publishing Engine index page

For more information, refer to The f=compconfig-rescan Request on page 76.

The returned report provides a summary of all document types installed on the Arbortext PE server, publishing configuration path and file names (**`.ccf`** and related files used by the content pipeline), paths to framesets for web publishing, Arbortext Import/Export templates, applications installed in the **`application`** directory used by the ***`PE_HOME`*** installation, and a list of Arbortext Editor client versions supported by Arbortext Publishing Engine.

If you are looking for a file that isn't included in the publishing configuration report, you can check the log file tracing the process that produced the document from the publishing configuration **log** link on the Arbortext Publishing Engine index page.

Enabling Tracing in compose.acl

On the client machine, you can enable tracing for ***`PE_HOME`*****`\packages\tools\compose.acl`** and the **comp_*output-type*.acl** routines. From the command line, issue the following commands:

```
require compose
$compose::debug=2
$compose::verbose=1
```

These commands force the publishing modules to display the contents of the global associative parameter array at each stage of processing a publishing request.

Enabling Publishing Debugging

On the client machine, you can enable publishing debugging by issuing the ACL command:

```
set debugcomposition=on
```

After you issue this command, the data associated with each publishing operation you initiate, including the data transmitted to the Arbortext PE server and the response returned from the server, is saved in a temporary directory. To retrieve this information, choose **Tools ▸Save Application** from Arbortext Editor. In the application save directory, the publishing information will appear as a file named **`compose.zip`**.

V

Arbortext Publishing

15 Introduction

This chapter provide an overview of the publishing capabilities provided by Arbortext Editor and Arbortext Publishing Engine. They explain terminology and each component in the publishing system. Subsequent chapters explore each component in more detail.

Publishing is the process by which Arbortext Editor transforms an XML or SGML document to some other representation, such as HTML or PDF (also sometimes known as "Composition"). Most often, the input to the publishing process is a document in memory and the output of the process is stored on disk. The process is flexible and general, and almost any aspect of the operation can vary from case to case.

Some publishing operations take place entirely within Arbortext Editor. Others involve running external applications, such as the HTML Help compiler. Some publishing operations may run on a separate machine, the Arbortext Publishing Engine. Publishing operations often run smaller, simpler publishing operations as part of their processing. Publishing operations can produce transformed XML or SGML. The result of a publishing operation may reside in memory or be stored to disk.

Starting a Publishing Operation

You can start a publishing operation in either of two ways:

- From the Arbortext Editor user interface by choosing one of the following from the **File** menu:
  - **Print Preview**
  - **Print**
  - **Publish** submenu options **For Web**, **For HTML Help**, **For EPUB**, **HTML File**, **PDF File**, **RTF File**, **Using Rule**, or **Using XSL**
  - **Import** submenu option **Import a Document**

  Note

  The availability of these operations depends on products and licensing for specific features installed at your site.

- From ACL (Arbortext Command Language) code that invokes subroutines specific to the type of publishing. The only way to invoke a publishing operation from Java, JavaScript, or VBScript is by calling one of the ACL subroutines. Each publishing type supported by Arbortext Editor can also be performed by a corresponding subroutine. The subroutines follow the form **compose_for_*type***, where you could specify ***type*** as **dvi**, **epub**, **export**, **html**, **htmlhelp**, **import**, **pdf**, and **web**.

Publishing Operation Components

Each publishing operation is made up of a number of smaller operations performed in sequence. Most of the complex processing is handled by two components called "content pipelines" and "content compilers". Most publishing operations involve setting up the operation, running a content pipeline, and then running a content compiler on the pipeline output.

content pipeline

a mechanism that transforms an XML or SGML document into another XML or SGML document. The result is retained in memory or written to disk. Some of the operations that a content pipeline performs include:

- profiling (removing portions of the input document being published)
- transforming the document as specified by an XSL stylesheet

When using Arbortext PE server to fulfill publishing requests, the content pipeline always runs on the server. Refer to 16 Content Pipelines on page 213 for a more detailed description.

filter

a component of a content pipeline. Filters are written in Java, and developers can write their own custom filters. Developers can create a new content pipeline by combining existing and custom filters in new patterns.

Information about filters is included in 16 Content Pipelines on page 213.

content compiler

a mechanism that transforms an XML or SGML document into some other format (not XML or SGML). Some content compilers are integrated with Arbortext Editor and can read their input documents from memory. Others are separate programs and read their input documents from disk files. All content compilers place their output in disk files.

A content compiler is a large, monolithic, and complex program. Although it is less likely that developers want to create their own content compilers, they are more likely to use commercial third-party software as content compilers. For example; Arbortext Publishing Engine uses the third party compilers in HTML Help Workshop.

Refer to 17 Content Compilers on page 219 for a more detailed description.

publishing framework

a software procedure that combines a content pipeline and one or more content compilers to form a complete publishing operation. The publishing framework, also sometimes known as the composition framework, is implemented in ACL. It's accessible to the Arbortext Editor user interface or to ACL subroutine calls. The framework collects publishing parameters, initializes, runs a content pipeline, runs one or more content compilers, and detects and reports errors. The publishing framework also can request services from the Arbortext Publishing Engine if required.

Refer to 18 The Publishing Framework on page 223 for a more detailed description.

PE Client Composer

a layer of software that handles the Arbortext Editor client side operation that is performed when using Arbortext Publishing Engine publishing.

Every publishing operation starts on the client, either from a graphical user interface or from ACL code. When Arbortext Editor uses the Arbortext PE server, the publishing framework calls the PE Client Composer. Usually the framework calls the Client Composer just at the point where the framework is ready to run the content pipeline. The Client Composer gathers the data, transmits it to the Arbortext PE server, and waits for the server's response. When the Arbortext PE server returns a response, the Client Composer extracts the returned data and places it in a predetermined location. Then the Client Composer returns control to the publishing framework, which may determine if any other processing is needed.

Refer to 19 Arbortext Publishing Engine Client Composer on page 243 for a more detailed description.

PE Server Composer

an Arbortext PE Application that handles the server side operation for fulfilling the request from the PE Client Composer. It receives data transmitted by the Client Composer, runs the appropriate content pipeline and/or content compiler, and then returns the result to the Client Composer.

Refer to 20 Arbortext Publishing Engine Server Composer on page 261 for a more detailed description.

Publishing Parameter Array

is an ACL associative array of name and value parameters used by almost all stages of the publishing process. The array is initialized by the publishing framework. As processing continues, various components of the publishing process retrieve values from the array, and then they store new values in the array to control publishing behavior downstream. The parameter array is passed to the content pipeline and controls every aspect of pipeline execution. If Arbortext PE server is used for publishing, the parameter array is passed by the PE Client Composer to the PE Server Composer.

Parameters are explained in the context where they are used.

Some content compilers always run on the server, while others run on the client machine. Some publishing operations can only be performed on the client, while others require Arbortext PE server. Some operations run in a mixed mode; for example, a content pipeline followed by a content compiler that run on the server, and then another content compiler that runs on the client to finish the publishing operation. Some operations may be performed locally or with server assistance, depending upon the products and licenses you have.

16 Content Pipelines

A content pipeline is a mechanism for translating an XML or SGML document into another XML or SGML document. It consists of a sequence of Java objects called **filters**.

A content pipeline begins with a special filter called a **generator** that translates an XML or SGML document into a series of SAX events. It ends with a special filter called a **serializer** that translates a series of SAX events back into an XML or SGML document, placing it in-memory or on disk. Each filter between the **generator** and **serializer** accepts a stream of SAX events from the previous filter in the pipeline and passes a stream of SAX events to the next filter in the pipeline.

A filter can remove content from the XML document passing through it by omitting some SAX events from its output stream. It can insert content by adding SAX events. Some filters make only minor modifications to the document; some may replace the document entirely by applying complex transformation rules to the input stream.

The content pipeline and filters used in Arbortext software are an implementation of a freely-available technology called SAX2, the second version of the Simple API for XML. You can find information about SAX2, the SAX2 parser, and SAX filters on the web. Be sure to consult the *Content Pipeline Guide* for more information on the Arbortext implementation; however, the following sections summarize its information.

When Arbortext Editor uses Arbortext PE server for publishing, most content pipelines run on the Arbortext PE server rather than on the client. However, some auxiliary pipelines run on the client if their primary function is to prepare data for transmission to the server.

Developing and Configuring Content Pipelines

Implementing New Filters

Basically, create a Java object that implements one or more of the seven standard SAX2 interfaces. Then compile your code and place it in a JAR file where the Arbortext PE sub-processes can find it. Custom JAR files are usually placed in the **`classes`** subdirectory of the ***`PE_HOME`*****`\custom`** directory, (refer to Overview of Custom Programs and Scripts on page 280). Consult the *Content Pipeline Guide* for information on developing custom filters.

Using Existing Filters

Arbortext Editor and Arbortext Publishing Engine install a number of defined filters, and they are combined into content pipelines. The definitions can be found in the ***`Arbortext-path`*****`\composer`**directory. You can use filters shipped with Arbortext Editor or develop your own filters. Each filter is defined in an **`.ent`** (XML Entity) file in the **`composer`** directory in the Arbortext installation. Any filter can be used in a new content pipeline. Put your custom filters in the **`custom\composer`** directory to make them automatically available.

Defining a Content Pipeline

Filters and content pipelines are normally defined using XML files. In Arbortext software, the **`.ccf`** (publishing configuration file) XML document defines a content pipeline by listing a series of filters, starting with a **generator** and ending with a **serializer**. A filter may be defined in the **`.ccf`** file, but usually a filter is defined in an **`.ent`** (XML or SGML entity) file, so that its definition can be reused in several **`.ccf`** files without the need to repeat the definition in each **`.ccf`** file.

A filter definition consists of an XML **FilterDef** element which defines the Java classes that implement and invoke the filter. It also contains a series of **Parameter** elements which define the parameters that are passed to the filter to control its operation.

A pipeline definition consists an XML **Interface** element which contains **Parameter** elements that define pipeline parameters, a **Resource** element that contains filter definitions (usually entity references to the **`.ent`** files), and a **Pipeline** element that defines the initial filter of the pipeline, the order in which the filters are chained together, and the mapping of pipeline parameters to filter parameters.

The most complex part of a pipeline definition is mapping pipeline parameters to filter parameters. Essentially, the pipeline parameters are a series of string ***name*** and ***value*** pairs that are passed to the content pipeline at runtime to control its operation. To avoid namespace collisions among filter parameters, the content pipeline definition maps pipeline parameter names to filter parameter names. A single pipeline parameter might be used by two different filters, yet each filter might have its own name for the parameter.

As an example of a content pipeline definition, consider the files **`epicGenerator.ent`** and **`pdf.ccf`**, both distributed in the ***`Arbortext-path`*****`\composer`** directory. The **`epicGenerator.ent`** entity contains a definition of the **epicGenerator** filter. The definition specifies the filter's ID, the filter's type, and the names of Java classes that implement the filter and provide an interface to it. In its parameters list, each parameter has an ID (so that the parameter definition can be referenced), a name (so that the parameter can be mapped to a pipeline parameter) and attributes like parameter type, whether the parameter is required, and a default value.

The file **`pdf.ccf`** defines a content pipeline that uses the **epicGenerator** filter. The pipeline definition first defines the pipeline parameters by:

- a **name** attribute, which is used to identify the parameter value in the global parameter array when the pipeline runs.
- an **idref** attribute that references the ID of the **Parameter** element that defines the parameter in the filter definition.

The first **Parameter** in the **Interface** section of **`pdf.ccf`** has a **name** of `document` and an **idref** of `epicGenerator.docId`. This means the **Parameter** element with the ID `epicGenerator.docId` provides the parameter definition (to obtain its type, default value, whether it's required or optional, and so on). At run time, when an Arbortext PE sub-process is given an associative array of parameters, the parameter with a name of **document** will have to match this parameter definition.

The **`pdf.ccf`** continues with a **Resource** section that defines the filters it needs. The filter definitions are entity references, to the entity file **`epicGenerator.ent`** as well as others. It ends with a **Pipeline** element that lists the filters that define the pipeline.

The first **Filter** has the **id** of `epic_generator`. But it is the value of the **filterDefRef** attribute, `epicGenerator`, which maps to the filter defined in **`epicGenerator.ent`**. `epicGenerator` matches the **id** attribute of the **FilterDef** element in **`epicGenerator.ent`**.

In **`pdf.ccf`** the `epic_generator` filter's first **FilterParameter** configures the filter parameter named `docId` to be initialized from the pipeline parameter named `document`. There is a double reference in the file: the **Parameter** element in the **`epicGenerator`** filter named `docId` is referenced twice in **`pdf.ccf`**.

1. the first **Parameter** element's **idref** attribute (in the **Interface** section)

   This first reference retrieves the parameter definition. Several pipeline parameters could be defined with different names but all using the same **idref** value to reference the same definition. The pipeline would then have several parameters with different names, but with the same parameter type, default value, and so on.

2. the **name** of the first **FilterParameter** element in the `epic_generator` filter (in the **Pipeline** section).

The second reference assigns a pipeline parameter value to the filter parameter at run time. The pipeline could contain multiple occurrences of the same filter, each having the same filter parameter name but initialized by a different pipeline parameter. Conversely, the same pipeline parameter may be used to initialize the filter parameters of several unrelated filters.

Switch Filters and Conditional Execution

The `switch` filter allows you to define two or more paths through a content pipeline. For example, you might want two content pipelines that are identical except for the final filter that hands off the document. In one, the final filter might be the **fileSerializer**, which places the processed document on disk. In the other, the final filter might be the **epicSerializer**, which places the processed document in memory for use by Arbortext Editor. You could write two separate **`.ccf`** files, or you could use a switch filter to route the document to the appropriate serializer based on the value of a pipeline parameter.

The **`pdf.ccf`** contains two examples of using a switch filter. For more information, refer to the ***`Arbortext-path`*****`composer\switch.ent`**.

Creating Content Pipelines with ACL

To transform a document in a content pipeline, an ACL routine must create the pipeline, set up an array of parameters for the pipeline, and then execute the pipeline on a document. The behavior of a content pipeline is controlled by the parameter array. ACL supports associative arrays in which each entry has a string name and a string value.

To run a content pipeline, an ACL routine must create an empty array and add one entry for each parameter described as **required** by the **Parameter** element (in the **Interface** section of the **`.ccf`** file. The name of the array entry must match the **name** attribute of the **Parameter** element. The corresponding value must be compatible with the filter's **type** attribute. Optional parameters may be specified or omitted.

The pipeline will ignore any parameters in the array that are not defined in the **Interface** element section of the **`.ccf`** file. Be aware that parameter names are case sensitive, and most names use the mixed case convention made popular by Java and similar languages.

A pipeline can only transform an input XML document to another XML document. That means that a content pipeline by itself can not transform an XML document to another output type, such as PostScript, PDF, or RTF. To perform transformations from XML to another format, the document must be passed to a content compiler, as described in 17 Content Compilers on page 219.

Creating a Content Pipeline

A content pipeline is created by an ACL function call:

handle **= get_composer(*arr*, *name* [, *doc*])**

The arguments are specified as follows:

- ***arr***

  specifies the ACL associative array which will be filled with the name and value of each parameter defined for the content pipeline as specified in the **`.ccf`**file. If a parameter has a default value not otherwise specified, the default is used. If a parameter has no default value, a null string is used. Parameter values can be specified in the **`.dcf`** file associated with the document type, and the value specified in the **`.dcf`** is used.

- ***name***

  specifies the composer to create as well as the base name for the **`.ccf`**. If the value `pdf` is passed, then the search looks for a file named **`pdf.ccf`** in each directory for the composer path (including the **`custom\composer`** directory, **`set composerpath`** option, and the **`append_composer_path`** function).

- ***doc***

  (optional) specifies the document to be published. If it is omitted or specified as `0`, the current document is used.

The return value is either a handle or a null string. The handle can be used to reference the composer in later calls. If no **`.ccf`** file matches the ***name*** parameter, then a null string is returned.

Running a Content Pipeline

After a content pipeline has been created, an ACL function call runs it:

result **= run_composer(*handle*, *arr*)**

The arguments are specified as follows:

- ***handle***

  is the handle returned by **get_composer**.

- ***arr***

  is the parameter array.

If the content pipeline succeeds, the function returns `1`. If an error is detected, the function returns `0`.

The pipeline may leave open documents in memory or files on disk, depending on the values passed in the parameter array and the nature of the pipeline, Developers need to remember that management of these files is the responsibility of the code that created and ran the pipeline.

17 Content Compilers

A content compiler is a program that translates a document (not required to be XML) to a non-XML format, such as PDF, PostScript or HTML Help. Unlike content pipelines, which have a uniform interface, each content compiler is configured and invoked in a different way. Two content compilers are built into Arbortext Editor, the Arbortext Formatting Engine and the Arbortext Advanced Print Publisher. Other content compilers are separate binaries that run under the control of Arbortext Editor. Still others are programs supplied and licensed by third parties.

Content compilers are invoked by the publishing framework after the content pipeline finishes running. This post-process callback is explained in The Outer Layer of the Publishing Framework on page 225.

The Arbortext Formatting Engine and Arbortext Advanced Print Publisher

The most commonly used content compilers are the Arbortext Formatting Engine and the Arbortext Advanced Print Publisher. Both provide a mechanism for transforming an XML or SGML document for previewing or to be published to PDF or PostScript. Each offers different capabilities.

Arbortext Formatting Engine

The Arbortext Formatting Engine is invoked by the ACL **`format`** command. It uses a FOSI specification to translate an open XML or SGML document to an internal DVI (device independent) format stored on disk. The DVI file may be displayed directly in a **Preview** window or used as input to a content compiler which produces PostScript or PDF as explained in Producing PostScript and PDF from DVI on page 221. When Arbortext Editor is using the Arbortext PE server for publishing, the formatter always runs on the Arbortext PE server.

Internally the formatter has two parts, one built into Arbortext Editor and the Arbortext PE sub-processes, and another in a separate program named **`PubTeX`**.

The Windows **Task Manager** allows you to check formatting progress. This action is useful if you suspect that **PubTeX** is hung, or that Arbortext Editor or the Arbortext PE sub-process is hung waiting for **PubTeX** to finish.

Arbortext Advanced Print Publisher

The Arbortext Advanced Print Publisher is invoked internally by the publishing framework. There are no ACL commands to run it directly. It uses a **`.3f`** template specification to translate an open XML or SGML document directly to Postscript or PDF. To support previewing, Arbortext Editor may display the PDF output using a PDF viewer such as Adobe Reader. When Arbortext Editor is using the Arbortext PE server for publishing, Arbortext Advanced Print Publisher always runs on the Arbortext PE server.

Arbortext Styler Files

To transform an XML or SGML document, both the Arbortext Formatting Engine and Arbortext Advanced Print Publisher require a formatting specification. Both engines can read a formatting specification from an Arbortext Styler (**`.style`**) file. The Arbortext Formatting Engine also supports a FOSI stylesheet, which it can read directly from a **`.fos`** stylesheet. The Arbortext Advanced Print Publisher also supports an APP Template, which it can read directly from a **`.3f`** file. A style file is an independent way of representing a formatting specification for either engine. Style files are created by the Arbortext Styler (consult the *User's Guide to Arbortext Styler* for extensive information).

Note that a **`.style`** file may also contain other types of formatting specifications not used by either the Arbortext Formatting Engine or the Arbortext Advanced Print Publisher:

- When a **`.style`** file contains an XSL stylesheet, the publishing process uses a content pipeline to apply the stylesheet and transform the document being published to the standard XSL-FO format. Afterwards, the publishing process uses the Arbortext Formatting Engine to translate the XSL-FO document to DVI.
- When a **`.style`** file contains specifications for transforming a document into output formats other than PDF and Postscript, such specifications are ignored when producing a PDF or PostScript transformation.

Producing PostScript and PDF from DVI

There are two content compilers that translate a DVI file to PostScript or PDF. On Windows, the DVI-to-PostScript compiler is named **`pubview.exe`** and the DVI-to-PDF compiler is named **`pubpdf.exe`**.

When Arbortext Editor is using the Arbortext PE server for publishing, and the publishing operation is producing PostScript for printing or for saving to a disk file, **`pubview.exe`** runs on the Arbortext Editor client machine. If a publishing operation is producing PDF, the PubPDF content compiler always runs on the server.

Producing HTML

There are two ways to produce HTML:

- Producing HTML directly through a content pipeline. Because HTML is a simplified form of XML, it can be generated directly by a content pipeline by applying an XSL stylesheet or a **`.style`** file containing XSL.
- Producing HTML using the HTML formatter and applying a FOSI stylesheet or a **`.style`** file containing a FOSI.

When Arbortext Editor is using Arbortext PE server, the content pipeline and the HTML formatter run on the Arbortext PE server.

Producing HTML Help

The HTML Help Workshop contains the content compiler used to produce HTML Help **`.chm`** (Compiled Help Module) files. The process of generating HTML Help involves running a content pipeline, which uses an XSL stylesheet (or an XSL stylesheet in a **`.style`** file) to transform a document into the format that the HTML Help compiler accepts. Then HTML Help compiler runs as a content compiler to translate the XML document to a **`.chm`** file.

When Arbortext Editor is using Arbortext PE server, the HTML Help Workshop must be installed on the Arbortext PE server.

Refer to Publishing a Document for HTML Help and Setup Considerations for Arbortext Publishing Engine for further information.

18 The Publishing Framework

The “publishing framework” (sometimes also known as the “composition framework”) is the layer of ACL code that surrounds the content pipelines and content compilers. The publishing framework is responsible for:

- receiving control from Arbortext Editor or from custom ACL code in an application
- prompting for or gathering publishing parameters
- creating the content pipeline
- setting up the parameter array for the content pipeline
- running the content pipeline
- running content compiler(s)
- cleaning up afterwards (closing documents, deleting temporary files, and so on)
- launching a viewer for the output if needed
- detecting and reporting errors at each stage of processing
- calling customer-provided code at each state of processing.

The publishing framework is divided into two layers of code. The “outer layer” contains code and information specific to producing particular kinds of output (such as PDF, HTML, PostScript, and so on). It’s explained in the next section.

The “inner layer” of the publishing framework is operation-independent, and it's explained in The Inner Layer of the Publishing Framework on page 229.

The Outer Layer of the Publishing Framework

Arbortext installations contain a number of outer layer publishing packages, located in `Arbortext-path\packages\tools`. These outer layer ACL packages are named **comp_*type*.acl**, where ***type*** corresponds to the type of processing it performs. For example, HTML publishing is handled by **comp_html.acl**, while publishing to PDF is handled by **comp_pdf.acl**. Each outer layer package contains three entry points and four callback routines.

The Entry Point Functions

The entry points functions are explained in the next sections. Note that their names vary slightly.

The compose_*type* Function

The entry point for the Arbortext Editor user interface is:

main::compose_*type*(*doc*)

doc is normally omitted by Arbortext Editor as it defaults to the current document.

This routine calls its associated **compose::compose_for_*type*** (where the processing type is the same) with an empty parameter array.

The compose_for_*type*_java Function

The entry point for Java programs is:

compose::compose_for_*type*_java(*doc*, *string*, *delim*)

- ***doc***

  is the document to be published
- ***string***

  is a string containing an encoded version of the associative parameter array
- ***delim***

  is the character used to separate name and value pairs in the string

This function calls the ACL routine **epicutil::UnserializeKeyed** to translate the ***string*** into an ACL associative array, and then calls the **compose_for_*type*** function. This function is primarily used by the Arbortext Publishing Engine Server Composer, described in 20 Arbortext Publishing Engine Server Composer on page 261.

The compose_for_*type* Function

The primary entry point for each publishing ***type*** is:

compose::compose_for_*type*(*doc*, *params*)

- ***doc***

  is the document being published

- ***params***

  is the parameter array

This routine usually loads parameters and then calls the inner layer of the publishing framework.

The function implements persistent parameters for interactive operations; these parameters persist from operation to operation during an Arbortext Editor session. For example, if a user selects a particular stylesheet for publishing a document to PDF, that stylesheet will become the new default choice when the user publishes a PDF a second time.

It creates persistent parameters by determining whether the ***params*** array is empty. If it is, the function loads the parameters used in the last interactive operation for this document and publishing type into ***params***. If this is the first interactive publishing operation of its type since the document was opened, nothing is loaded.

After loading parameters, **compose_for_*type*** calls the inner layer of the publishing framework, **compose::compose_for_type** (note, in this case, **type** is literal in the function name). The **compose_for_*type*** function passes the name of the content pipeline, the parameter array, and the names of four callback functions.

When **compose::compose_for_type** returns, the publishing operation is usually complete and most **compose_for_*type*** functions simply return at that point.

The Publishing Framework Callback Functions

Most of the real work in the outer layer of the publishing framework is performed by its callback routines. The basic idea is that **compose_for_type** is passed the names of four outer layer subroutines, which it calls at predetermined points during execution. Each callback routine is passed the parameter array, and callbacks are expected to modify the name and value parameters in the array to control downstream processing.

Names of the callback routines can vary substantially between **comp_*type*** modules. The actual callback function names are passed as strings to **compose_for_type**. A callback name may be passed as an empty string, in which case **compose_for_type** skips the callback and proceeds as if it returned successfully.

The preprocess Callback

This routine is called by the inner layer before it tackles any serious work. This callback is expected to initialize type-specific data in the parameter array, to perform type-specific error checks, and to perform any other necessary setup.

comp_*type*::*preprocess_cb*(*doc*, *params*[], *interactive*)

- ***doc***

  is the document being published
- ***params***

  is the parameter array
- ***interactive***

  is set to `1` if the operation was invoked from the Arbortext Editor user interface

The callback can return:

- `-2`

  error detected, error message was logged
- `0`

  error detected, no message was logged
- `1`

  success, **compose_for_type** should continue processing
- `2`

  processing should continue, but **compose_for_type** should skip loading the default content pipeline parameters into the parameter array

The return code `-1` is not used.

The dialog Callback

The dialog callback is called only if the publishing operation is interactive (meaning it was invoked from Arbortext Editor and not from custom code, as noted in The Outer Layer of the Publishing Framework on page 225). It displays the dialog box to the user and prompts for parameters, such as which stylesheet to use, where to put the output file, and so forth. The callback records the responses in the parameter array.

comp_*type*::*dialog_cb*(*doc*, *params*)

- ***doc***

  is the document being published
- ***params***

  is the parameter array

The callback can return:

- `-1`

  error detected, error message was logged
- `0`

  error detected, no message was logged
- `1`

  success, **compose_for_type** should continue processing

The fixup Callback

The fix-up callback is always called just before **compose_for_type** runs the content pipeline. This is the final opportunity to get operation-specific parameters set correctly before the content pipeline runs.

comp_*type*::*fixup_cb*(*doc*, params)

- ***doc***

  is the document being published
- ***params***

  is the parameter array
- `-1`

  error detected, error message was logged
- `0`

  error detected, no message was logged
- `1`

  success
- `2`

  processing should continue, but **compose_for_type** should skip running the content pipeline

The postprocess Callback

The post-process callback is called after the content pipeline runs and it runs the content compiler(s). It also displays the results of the operation to the user (such as launching Adobe Acrobat Reader to view PDF output), and it also performs clean-up.

comp_*type*::*postprocess_cb*(*doc*, *params*[], *interactive*)

- ***doc***

  is the document being published

- ***params***

  is the parameter array
- ***interactive***

  is set to `1` if the operation was invoked from the Arbortext Editor user interface

The callback can return:

- `-1`

  error detected, error message was logged
- `0`

  error detected, no message was logged
- `1`

  success

The Inner Layer of the Publishing Framework

The inner layer of the publishing framework is operation-independent, which means that it doesn't care what kind of output the operation is producing. Conceptually, its operation is intended to run the content pipeline, with calls to outer layer callbacks that provide type-specific processing before and after the content pipeline runs. In practice, this simple picture is complicated in a number of ways, some arising from the needs of various publishing types, and some from the requirements imposed by using Arbortext PE server.

The inner framework is invoked when the outer framework calls:

compose::compose_for_type(*doc*, *comp_type*, *interactive*, *params*[], *preproc_cb*, *dialog_cb*, *fixup_cb*, *postproc_cb*)

- ***doc***

  is the document being published
- ***comp_type***

  is the name of the content pipeline to run
- ***interactive***

  is set to `1` if the operation was invoked from the Arbortext Editor user interface
- ***params***

  is the parameter array
- ***preproc_cb***, ***dialog_cb***, ***fixup_cb***, ***postproc_cb***

are each strings containing the names of the four outer layer callback routines. An empty string bypasses the callback.

The callback can return:

- `-1`

  if the user aborts the operation, such as by choosing **Cancel** from the dialog box
- `0`

  if the operation fails
- `1`

  if the operation succeeds

compose_for_type logs its operation in the event log, a separate document that can be displayed after the operation. It reports any warnings, errors, or a fatal error, and Arbortext Editor can display the event log.

The **compose_for_type** function starts a new publishing job in the event log to collect messages, calls **compose_for_type_int** to gather the messages, and then terminate the event log job afterward.

Note

*The function **compose_for_type_int** has the same parameters as **compose_for_type**, and it returns the same results. Future discussion will refer to **compose_for_type** for brevity.*

compose_for_type Processing

Note

*In every operation performed by **compose_for_type**, a failure writes an error to the event log and returns `0`.*

1. The **compose_for_type** operation begins by determining whether the request is being sent to Arbortext PE server. If **`set peservices`** is `on` and the parameter **compose.suppress-e3-composition** is not present in the parameter array, then the Arbortext PE server will perform the publishing operation.
2. Then **compose_for_type** checks to make sure that information about the Arbortext PE server is available, and that the Arbortext PE server supports the version of Arbortext Editor that is running.
3. Next, **compose_for_type** calls the pre-processing callback, if one was provided.
   - If the callback returns `-2`, **compose_for_type** returns `0`.

- If the callback returns `-1`, **compose_for_type** logs a generic failure message, and returns `0`.
- If the callback returns `1` or `2`, **compose_for_type** continues processing.

4. Next, **compose_for_type** either creates the content pipeline or initializes for Arbortext PE server publishing. Both operations load the default parameters for the operation type into the parameter array. Then **compose_for_type** calls **get_ composer** to create the content pipeline, passing its ***comp_type*** parameter as the name of the **`.ccf`** file to load. If the Arbortext PE server is performing the publishing, **compose_for_type** calls **compose::initialize_e3_composition** to retrieve the default Arbortext PE server publishing parameters, again passing ***comp_type*** as the pipeline name.

   For example, if the output type is `pdf`, both routines look for a file named **`pdf. ccf`** in the composer path on the client or server machine.

   If the pre-processing callback returned `2` (meaning processing should continue, but **compose_for_type** should skip loading the default content pipeline parameters), then the call to **get_composer** or **initialize_e3_composition** is omitted, and no default parameters are loaded.
5. **compose_for_type** determines if the publishing operation is interactive and a **dialog** callback was passed. When **compose_for_type** calls the dialog callback, a return value of `-1` causes **compose_for_type** to return `0`. If the callback returns `0`, **compose_for_type** logs an error message and returns. Otherwise processing continues. If the parameter array contains values controlling profiling, the values are modified to work with the profiling filters in the content pipeline.
6. If the document being published contains Arbortext data merge queries, **compose_ for_type** calls the data merge facility to update the document with the query results.
7. If the publishing parameters specify profiling, **compose_for_type** makes sure that the values in the publishing parameter array are properly initialized.
8. If the document being published is a DITA topic, **compose_for_type** sets up some parameters to either set up some extra parameters to publish the topic as is or sets parameters to wrap the topic in a temporary DITA map.
9. **compose_for_type** calls the parameter **fixup** callback, if one was specified. If the callback returns `-1`, **compose_for_type** returns `0`. If the **fixup** callback returns `0`, **compose_for_type** logs a message and then returns. The **fixup** callback can return `1` for success or `2` for success but suppress the content pipeline run. If `2` is returned, that value is saved for later. (In the second step of *Processing if not using Arbortext PE server* later in this section, if the **fixup** callback returns `2`, **compose_for_type** skips running the content pipeline.)
10. **compose_for_type** calls the publishing preprocessor hook to see if there is custom code that was registered using the ACL **add_hook** function. The **hook** function code can modify the parameter array as needed. If it returns a value less than zero,

the publishing process terminates. If not, operation continues. The custom code must follow this signature:

***hook_function*(*doc*, *type*, *interactive*, *params*[])**

- ***doc***

  is the document being published

- ***type***

  is the name of the composer

- ***interactive***

  is set to `1` if the operation was invoked from the Arbortext Editor user interface

- ***params***

  is the parameter array

11. **compose_for_type** checks to see whether the document being published is a DITA map. If so, it recursively calls an outer-framework routine, **compose_for_createprds**, to transform the DITA map into a preliminary resolved data set, which is a document that includes the topics and other content referenced by the DITA map. Publishing continues with the PRDS in place of the original document.

Processing divides at this point. If Arbortext PE server is enabled, **compose_for_type** follows one code path. If Arbortext PE server is not enabled, processing follows another.

Processing if using Arbortext PE server

1. First, **compose_for_type** sets some additional parameters that control execution of the Arbortext Publishing Engine Client Composer.
2. **compose_for_type** calls the Arbortext Publishing Engine Client Composer, then returns to its caller. The Arbortext Publishing Engine Client Composer handles communication with the Arbortext PE server and calls the **post-process** callback when the Arbortext PE server responds.
3. If the publishing operation is queued on the Arbortext PE server, then the **post-process** callback does not run. Instead, when the user retrieves the result of the queued publishing operation from the Arbortext PE server, Arbortext Editor runs the **post-process** process as if the operation had just completed.

Processing if not using Arbortext PE server

1. **compose_for_type** runs the content pipeline. It skips the pipeline run if instructed to by the **fixup** callback and the document being published is not a DITA map. If the pipeline run succeeds and **compose_for_type** called the DITA preprocessor, it now calls **compose::postprocessForDITA** to delete the temporary DITA PRDS document. If the pipeline returns an error, processing terminates.

2. **compose_for_type** calls the **post-process** callback function, then returns to its caller.

The Publishing Framework Hook

The publishing framework hook allows user code to run at several points of publishing processing that uses **compose_for_*type*()** functions described in The Outer Layer of the Publishing Framework on page 225.

You can create an ACL **compositionframeworkhook** hook function and register it using the ACL **add_hook** function. The publishing framework will invoke the hook function before **compose_for_*type*()** starts executing, after each operation **compose_for_*type*()** runs, and before **compose_for_*type*()** returns to its caller. Your hook function can return `0` to allow **compose_for_*type*()** to continue executing or return a negative value to terminate the publishing operation. It can modify any or all values in the publishing parameter array, thereby controlling the future flow of execution in **compose_for_*type*()**.

The publishing framework hook function has the following prototype:

hook(*doc*, *type*, *where*, *params[]*)

- ***doc*** is the document identifier of the document being published.
- ***type*** is the name of the pipeline or type of published output as represented by its **`.ccf`** base file name.

  The complete list of pipeline **`.ccf`** files is located in ***`Arbortext-path`*****`\composer`**.
- ***where*** is the point in publishing processing where the hook is called. It is a literal string value that is passed by the publishing framework to the hook function each time the framework calls it. For each string, there is a corresponding ACL variable that specifies the point of operation.

  `Arbortext-path`**`\packages\tools\composer.acl`** contains the set of `HK_`***`variable-name`*** ACL variables with defined values you can use for a ***where*** string specification. You can determine the point at which the hook function is called by finding the variable associated with one of the ***where*** values and searching for it in **`compose.acl`**.
- ***params[]*** is the parameter array that holds the parameters and values to be used by the pipeline.

Refer to the *Arbortext Command Language Reference* entry for**compositionframeworkhook**, which describes its syntax and use, including how to specify the arguments.

Before writing a publishing framework hook, be sure to examine **`compose.acl`** and experiment with the example ACL script that follows to understand how the **compose_ for_*type*()** functions work.

Note

It's possible the behavior of this hook can change from release to release. You should check your publishing framework hook code with each release of Arbortext software.

The following example ACL function prints the location from which it is called, and sets that function up as a publishing framework hook:

```
package testhook;
function compFrameHook( doc, type, where, params[] ) {
  local is_e3 = ( main::is_e3 ? "e3" : "local" );
  eval "chf:  $is_e3 doc=$doc type=$type where=$where" output=*debug;
}
set debug==extwin;
add_hook( "compositionframeworkhook", "testhook::compFrameHook" );
eval "installed composition framework hook" output=*debug;
```

If you place this ACL script in a **`custom\init`** directory, it will echo its parameters to the Arbortext Diagnostics window as you publish a document. You might find it a useful starting point for implementing a hook function of your own, because it will illustrate the parameters your hook function should expect.

Print and Print Preview

Arbortext Editor users can choose to preview or print a document. Unlike the other publishing types, the top levels of these commands are built in to Arbortext Editor itself, rather than implemented in ACL.

Print Preview

The **`preview`** command displays a document in a window in the format that it will appear in when it prints. The preview command has a number of options that are implemented entirely in Arbortext Editor; these options stop the Arbortext Formatting Engine, launch the **Preview** window on a previously-formatted document, and perform other minor housekeeping functions.

If you invoke the **`preview`** command for real processing (i.e., not for one of the housekeeping functions) then Arbortext Editor will invoke the ACL routine **compose::compose_for_preview** in the module **`comp_dvi.acl`**. It will invoke the ACL layer if any of the following conditions are true:

- PE publishing is enabled
- Arbortext Editor is running from a compact install tree
- The document to be previewed is a DITA map or topic

- the document type of the document to be previewed is configured to allow the user to choose the stylesheet to be used
- the stylesheet being used is an XSL stylesheet, an APP template, or a `.style` file that contains an XSL stylesheet or APP template
- `set` options that use of APP rather than the Arbortext Formatting Engine

If Arbortext Editor is running from a compact install tree, then **compose::compose_for_preview** will call **compose_pdf_preview()** in the module **comp_pdf.acl**, which will translate the document into PDF and launch a PDF viewer. Otherwise `compose::compose_for_preview` will set some parameters and then call `compose::compose_for_dvi()`, is part of the outer layer of the publishing framework; this routine will either use the Arbortext Formatting Engine to translate the document to the DVI format and then use the `preview noformat` to launch the **Preview** window or use the Arbortext Advanced Print Publisher to translate the document to PDF and then launch a PDF viewer.

Print Command

The `print composed` command translates a document to Postscript and either saves the Postscript to disk or prints it. Processing may include any of the following cases:

- compact install tree: translate the document to PDF and print the PDF
- full install tree: Arbortext Formatting Engine translates the document to DVI and use the printer driver to either print the DVI file or save it to disk as PostScript
- full install tree: Arbortext Formatting Engine translates the document to DVI, translate the DVI to Postscript, then print the Postscript file or save it to disk;
- full install tree: Arbortext Advanced Print Publisher translates the document to PostScript, then either print the PostScript file or save it to disk.

Like the `preview` command, the `print composed` command has a number of minor functions that are entirely implemented within Arbortext Editor. If none of these functions are invoked, then `print composed` checks to see whether it is running in the compact install tree. If so, it calls the ACL routine **comp_print::printUsingPDF**, in ACL module **comp_print.acl**, which will translate the document to PDF and print the PDF file. If it's not running in the compact install tree, `print composed` displays the print dialog box, then calls **comp_print::composeAndPrint** to finalize processing.

comp_print::composeAndPrint determines whether to use APP or the Arbortext Formatting Engine and then calls either **comp_print::printUsingApp** or **comp_print::printUsingTex**. **comp_print::printUsingApp** sets parameters and calls the outer publishing framework layer subroutine **comp_pdf::compose_for_pdf**; **comp_print::printUsingTex** sets parameters and calls the publishing outer framework subroutine **comp_dvi::compose_for_dvi**.

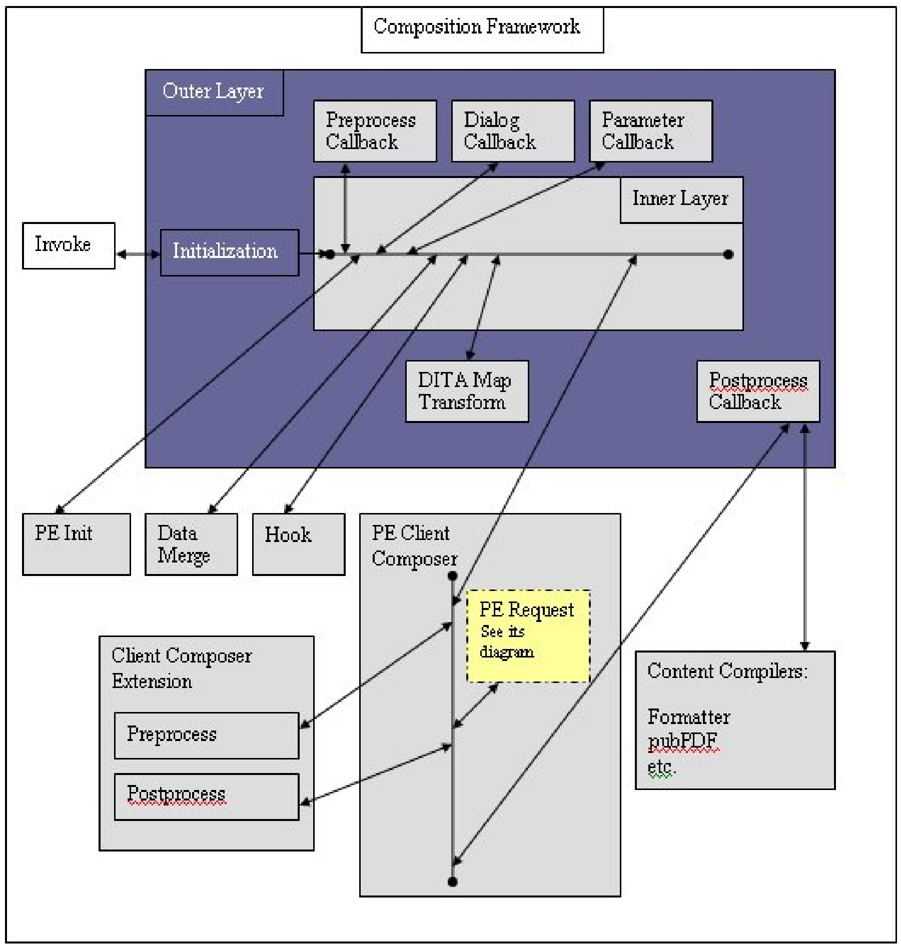
Note that the core `print composed` support is responsible for displaying the print dialog. For this reason, all of the ACL routines called by `print composed` support operate in batch mode, with no dialog boxes offered to the user. The core **`print_composed`** support calls routines in the **comp_print.acl** module to populate some of the fields in the publishing dialog boxes; these subroutines call the same subroutines that the dialog box support in the publishing framework call.

How Arbortext PE server uses the Publishing Framework

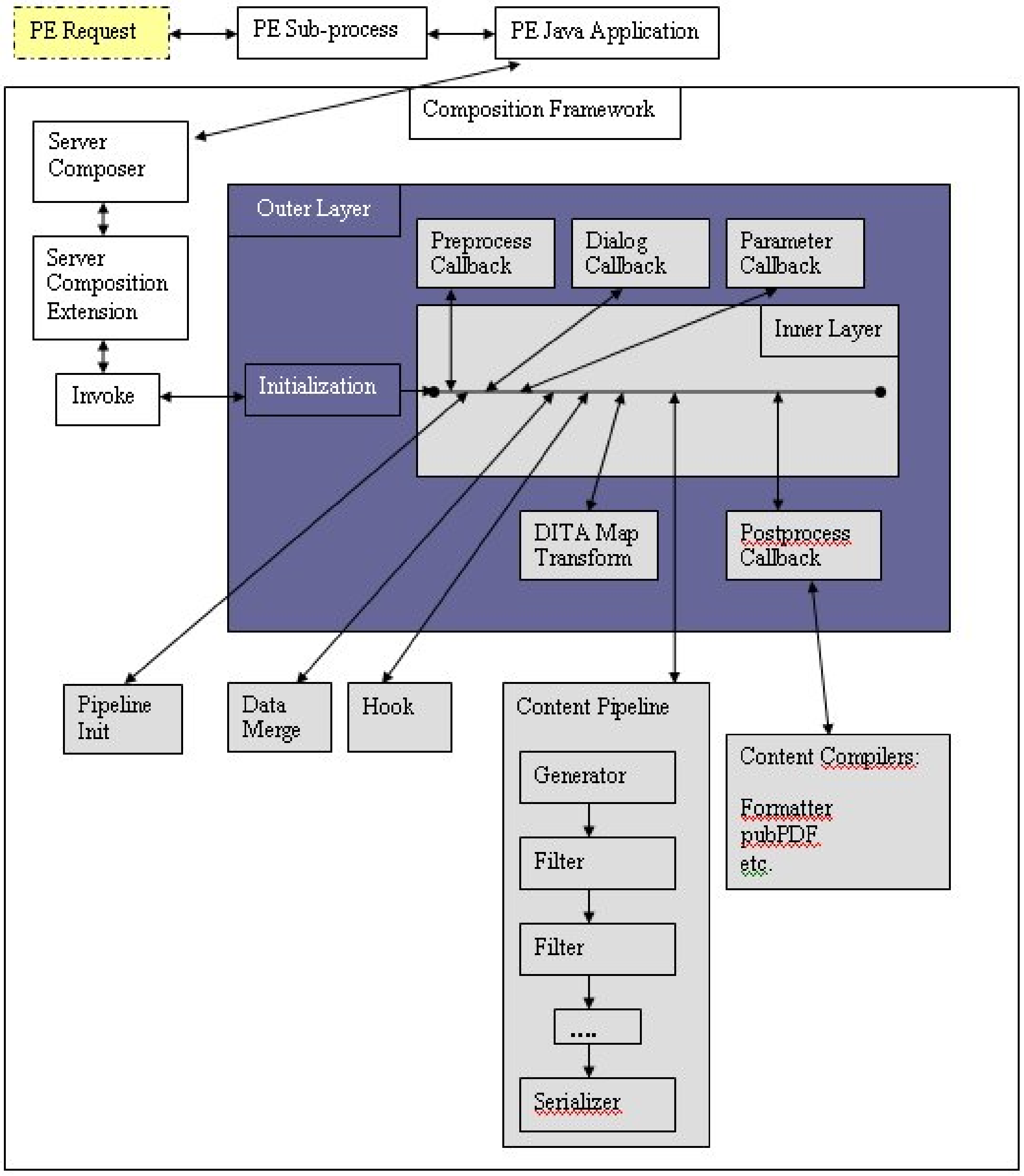
When Arbortext PE server publishing is enabled on a client, either Arbortext Editor or Arbortext Publishing Engine Interactive, the publishing framework actually runs twice: once on the client and once on the server. On the client, the framework performs processing that results in a request being transmitted to the server. On the server, the framework performs the publishing operation and produces results that are transmitted back to the client.

This approach means that the specialized processing that follows running the content pipeline, such as file clean-up and execution of one or more content compilers, only needs to be coded in the **post-process** callback of the outer layer of the framework. The code can easily determine whether it's running on the client or the server by checking the global parameter array entry `e3.serverComposition`, which is undefined (`0`,`''`, `false`) on the client and `1` on the server.

The following diagram illustrates the publishing framework processing performed on the client in preparation for sending a publishing request to the Arbortext PE server.



The following diagram illustrates the publishing framework processing of an Arbortext Publishing Engine request on the server.



Writing your own Outer Layer Module

The only way to implement an outer layer module is by writing an ACL routine. You can write your own outer layer module by providing the entry points and callbacks needed, and then by calling **compose_for_type** from your **compose_for_*type*** entry point. Look at these files as models:

`Arbortext-path`**`\packages\tools\`****comp_*type*.acl**

Modifying the Inner Layer

You should not modify the inner layer itself. Any changes you make to the code in **`compose.acl`** is likely to fail or produce unexpected results in subsequent releases of the software.

You should, however, be able to control the behavior of **compose_for_type()**, which is the core function in **`compose.acl`**, by using the The Publishing Framework Hook on page 233.

Debugging the Publishing Framework

There are several ways to debug the publishing framework.

The Event Log

Content pipelines write information to the event log as they initialize and, later, as they perform their operation. The event log is an XML document divided into sections, each covering a publishing operation. When the publishing framework is invoked, it starts a new section in the event log. After the operation is complete, the framework terminates the section. If you are using Arbortext Editor and the sections contains any error messages, the framework launches a window to display the event log.

You can elect to filter results in the Event Log by thread or context. Use the **View ▸ Display ▸Filter** menu option to open the **Filter Event Log Entries** dialog box.

You can display the event log by entering the following ACL command from the Arbortext Editor or Arbortext Publishing Engine Interactive command line:

```
show_composer_log()
```

On the server, the event log records the process ID and the transaction ID as an INFO level message, and the information can be passed to the Arbortext Editor client or the calling application. The event log entry would be something like:

```
<Level> MESSAGE</Level>
<Message>Publishing engine process ID = '372';
transaction ID = '74935'.</Message>
```

Composer Debug Flags

You can trace the execution of the publishing framework by entering the following three commands on the Arbortext Editor or Arbortext Publishing Engine Interactive command line:

```
source compose.acl;
$compose::debug=2;
$compose::verbose=1;
```

You can skip the initial **`source`** command if you've already performed a publishing operation.

Setting these two ACL global variables result in:

- The parameters `debug` and `verbose` will be passed to the content pipelines that run for future publishing operations. Some pipeline filters will heed these values and write additional information to the event log.
- The publishing framework will display dialog boxes showing the current values of its parameter array as it does its work. As each dialog box is displayed, execution will halt until you press the **OK** button.

If you don't want to dismiss each of the dialog boxes for each publishing operation, but you still want to inspect the results from the parameter arrays, you can set `compose::debug` to `1`. The publishing framework will not display any dialog boxes, but it will place an entry in the event log for each point where a dialog box would have been displayed.

Setting debugcomposition

The publishing process often creates a number of internal files.

- When publishing with Arbortext Publishing Engine, Arbortext Editor generates a log of the communication process, generates a zip archive for transmission to the Arbortext PE server, and receives a zip archive back from the Arbortext PE server.
- When publishing to PDF using Arbortext Editor with Arbortext Styler, Arbortext Editor might successively produce a revised XML document, a DVI file, a PostScript file, and a PDF output file..

If you set the ACL option **`set debugcomposition`** to `on`, the intermediate files and logs are saved for debugging. You can retrieve these files by selecting **Tools ▸Save Application**. The **Save Application** utility generates a directory containing a substantial amount of information, including publishing information. The publishing information is contained in the file **`compose.zip`** in the resulting **`appsave`** directory.

Enabling Application Logging on the Server

You can set the global Arbortext Publishing Engine parameter **com.arbortext.e3.applicationLog** or **com.arbortext.e3.applicationLog.Compose** to `INFO`, `DEBUG`, `TRACE` in or `ALL` in **`e3config.xml`**. Then, the server side activity of each publishing operation that is implemented in Java will log information to the servlet log.

If you set the global PE parameter **com.arbortext.e3.applicationLog.display** to `true` (the default), application log messages are written to the Arbortext Diagnostics window.

You can launch Arbortext Diagnostics from the shortcut called **Arbortext Publishing Engine Diagnostics** in your PTC program group.

19 Arbortext Publishing Engine Client Composer

The client side of an Arbortext Publishing Engine publishing operation is handled by Java code called the Arbortext Publishing Engine Client Composer. The Client Composer collects all of the information that the Arbortext Publishing Engine Server Composer needs to perform some portion of a publishing operation, transmits the data to the server, awaits a response, and processes the data returned by the server. The server operations are explained in 20 Arbortext Publishing Engine Server Composer on page 261.

The overall architecture of Arbortext Publishing Engine publishing—on both the client and the server—is designed to prevent content pipelines and content compilers from being aware of where they're running. Some content pipeline filters and some content compilers contain open source or commercial code that could not be modified to understand Arbortext Publishing Engine functionality.

Synchronous and Asynchronous Operations

Arbortext Publishing Engine publishing supports both synchronous and asynchronous publishing operations. Most operations are synchronous, meaning the inner layer of the publishing framework calls the Client Composer, which in turn does its job, awaits a response from the Arbortext PE server, and returns to the publishing framework in a single thread of execution. While awaiting a response from the server, Arbortext Editor is blocked; the user is prevented from doing work.

However, `preview` and `print composed` operations are asynchronous, unless the user specifies `print composed wait` or `preview wait` (then the operation is synchronous). The Client Composer creates a background task to perform the publishing operation, then returns to the publishing framework instantly. For an asynchronous operation, Arbortext Editor does not wait for the Arbortext PE server to respond. Instead, the user can continue working as if the operation succeeded. After some time passes, the **Preview** window will open or the printer will start printing.

Immediate and Queued Operations

Arbortext Publishing Engine publishing supports both immediate and queued operations. Immediate operations do not use the queuing feature on the Arbortext PE server, and their execution follows the synchronous operation format. Queued requests use the queueing feature on the Arbortext PE server. The Client Composer transmits a request to the Arbortext PE server to queue the operation. The server returns a transaction ID, and the user can use Arbortext Editor's Queued Transaction Viewer to determine when the queued request completes and to retrieve the queued transaction's results.

A queued transaction is not an asynchronous operation in the sense it was described earlier. Asynchronous transactions do not persist past the termination of Arbortext Editor, but a queued transaction persists for as long as it's configured to do so. You can submit a queued transaction request, exit from Arbortext Editor, and start Arbortext Editor at a later time and check the status of any queued transactions that you submitted using the Queued Transaction Viewer.

Only **Print Preview** and **Print** requests may be asynchronous, but **Print Preview** and **Print** operations, whether synchronous or asynchronous, may never be queued. If you submit an asynchronous **Print Preview** or **Print** request and then exit from Arbortext Editor, you'll never get anything back from Arbortext Publishing Engine. The transaction ID which Arbortext Publishing Engine assigns to immediate synchronous transactions, is not returned to Arbortext Editor, so synchronous publishing operations are not visible in the Queued Transaction Viewer.

Arbortext Publishing Engine Client Composer Operation

The Arbortext Publishing Engine Client Composer performs only processing that is common to all publishing types. To perform processing specific to the output type, the Client Composer loads a Client Composer Extension object and calls methods defined by the extension just before the request is transmitted to the Arbortext PE server, and does it again after the response has been received.

The Client Composer is a Java object named **com.arbortext.e3c.ClientComposer**. It defines one public method called **compose(*Map-parameters*, *Map-types*)**.

The method returns either the string `ok` or an error message. The two Map arguments are parallel associative arrays. For each map key ***name***, **parameters[*name*]** is the parameter value and **types[*name*]** is the parameter type.

If a parameter appears in **parameters[*name*]** but not in **types[*name*]** the default parameter type is `string`. If a parameter appears in **types[*name*]** but not in **parameters [*name*]**, it is ignored.

For example, to pass a parameter named `test1` of type `string` and with value `abcdef`, store the values in the **parameters[*name*]** and **types[*name*]** maps as follows:

```
parameters[ "test1" ] = "abcdef";
types[ "test1" ] = "string";
```

The Client Composer starts running when the publishing framework routine **compose_ for_type** calls the ACL routine **compose::e3_compose**. Then **e3_compose** creates the **parameters** and **types** arrays and invokes the Java method **com.arbortext.e3c. ClientComposer.compose**.

ClientComposer.compose starts by searching the **parameters[*name*]** array for a parameter named `e3.asynchronous`. If the parameter is not present or it has a value other than `1`, then **ClientComposer.compose** simply calls the private method **doCompose** and returns its result. If `e3.asynchronous` is present and equal to `1`, **ClientComposer.compose** creates a new thread which calls **doCompose**, and then **ClientComposer.compose** returns `ok` to its caller immediately. All of the real work is performed by **doCompose**, which has the same parameters and result signature as **ClientComposer.compose**.

The **doCompose** method begins by creating an input directory. The content of this input directory will be transmitted to the Arbortext PE server. Next, it processes the list of parameters. It looks for a few parameters by name, but, for most parameters, processing is driven by the parameter type. For each parameter, **doCompose** performs some action based upon the parameter name or type. Some parameters are saved for transmission to the Arbortext PE server, possibly with a modified value or type.

Next, **doCompose** loads the Client Composition Extension and calls its **preProcess** method. It writes key information about the publishing operation, including the modified parameter list, to the input directory file **`index.xml`**. Then it compresses the input directory into a single JAR file.

After creating the JAR file, the **ClientComposer.doCompose** method obtains the server URL from the Arbortext Editor (from the **Tools ▸Preferences ▸Publishing Engine** or the value of the **`set peserverurl`** option), then uses the Java Client SDK to construct a request and transmit the request to the Arbortext PE server. The request includes the query parameters **f=java** and **class=com.arbortext.e3c.Application**. The JAR file containing the input directory is sent as the body of the request. Then the Client Composer waits for the Arbortext PE server to respond.

After it finishes processing, the Arbortext PE server returns a response that contains a JAR file as the response body. The Arbortext Publishing Engine Client Composer saves the response JAR to disk and extracts two files. One is an XML document named **`response.xml`** which contains basic information about whether the request succeeded. The other file is the server composer log from the JAR which contains information generated by the server during the publishing process. The Client Composer analyzes **`response.xml`** to determine if the request succeeded or failed, and then copies information from the server event log into the client event log. The Arbortext Publishing Engine Client Composer then calls the Client Composition Extension method **postProcess**. Lastly, the Client Composer calls the **postprocess** callback of the publishing framework (described in the *postprocess Callback* section of The Outer Layer of the Publishing Framework on page 225).

Significant Parameters for the Arbortext Publishing Engine Client Composer

The publishing framework must define all Arbortext Publishing Engine Client Composer parameters before calling **ClientComposer.compose**. The Client Composer looks for the parameters listed in the following table by name, and it takes special actions based upon these parameter values. The processing for all other parameters is determined by the type code for each parameter. If a parameter is required but not present, the Arbortext Publishing Engine Client Composer will issue a fatal error message. After processing the parameters in the table, the Client Composer may take one of three actions, shown in the **Disposition** column.

- Parameters with `local` disposition are not transmitted to the Arbortext PE server.
- Parameters with `string` disposition are transmitted to the Arbortext PE server with no processing.
- Parameters with `markup` disposition are transmitted as attributes on the **Compose** element in the **`index.xml`** document written by the Client Composer to the input directory.

Significant Parameters for the Client Composer

Parameter Name	Disposition	Description
compose.document (required)	`local`	This is the ACL ID of the document being published. The Client Composer passes it to the publishing framework **postprocess** callback function.
compose.interactive (required)	`local`	If set to `1`, the publishing operation was invoked by the Arbortext Editor user interface. The Client Composer uses this variable to determine whether to display progress messages and a dialog box (with a **Cancel** button) while awaiting a response from the Arbortext PE server. This value is passed to the publishing framework **postprocess** callback function.
compose.postProcessor (required)	`local`	The name of the ACL post-process function to be called by the publishing framework after the Client Composer receives its response from the Arbortext PE server. If the value of this parameter is null, the Client Composer does not call a post-process function.
compose.suppressLogDisplay (optional)	`string`	if `1`, the event log is not displayed after the operation completes, even if it contains errors or fatal errors.
ditaDoctype (optional)	`string`	an empty string, `topic`, `map`, or omitted. If **ditaProcessing** is set to `true`, and **ditaDoctype** is set to `map`, then the Client Composer looks in the directory named by **ditaFilePath** for files named **`maplinks.unordered`** and **`metadatacollection.xml`**. The Client Composer copies them into a subdirectory named **`ditaData`** in the input directory (the directory transmitted to the server in a JAR file).

Parameter Name	**Disposition**	**Description**
ditaFilePath (optional)	`local`	Required if **ditaProcessing** is `true` and **ditaDoctype** is set to `map`. It must contain the absolute path to a directory containing files as described in **ditaDoctype** above.
ditaLinksHrefAlternate	`string`	(see *ditaLinksHrefPrefix*)
ditaLinksHrefPrefix (optional)	`local`	Specifies the relative path by which the DITA Map being published references the files in the DITA links directory. The Client Composer creates a subdirectory of the input directory using this path and copies the DITA link files to the new subdirectory. For HTML Help and Web publishing, set this parameter to something other than the value `.\`. In those cases, the Client Composer uses the value of the *ditaLinksHrefAlternate* parameter instead, to avoid copying DITA link files to the top level of the input directory.
ditaLinksDir (optional)	`local`	Specifies the absolute path to a directory of files on the client that are referenced by a DITA map but which are not XML documents. The Client Composer copies this directory to the input directory, using the name indicated by the parameters *ditaLinksHrefPrefix* and *ditaLinksHrefAlternate*.
ditaProcessing (optional)	`string`	If `true`, the document being published is a DITA topic or map document, as indicated by the **ditaDoctype** parameter.
e3.asynchronous (required)	`local`	If set to `1`, the publishing operation is performed asynchronously. Otherwise the Client Composer performs a synchronous operation.
e3.busyMessage (optional)	`local`	If specified, this parameter is displayed during a synchronous

Parameter Name	Disposition	Description
		publishing operation instead of the localized version of the message **Composing on PE Server**.
e3.busyPanel (optional)	`local`	If specified, this parameter gives the name of the dialog box displayed during a synchronous operation. If not specified, the dialog box defined in **`e3compose.xml`** is displayed.
e3.busyPanelMessage (optional)	`local`	If specified, this parameter gives the name of the message displayed in the busy panel. The default is **Composing on PE Server {*url*}**, where ***url*** is replaced by the Arbortext PE server URL. If **e3. busyPanel** is specified and **e3. busyPanelMessage** is not specified, then no message is loaded into the **e3.busyPanel** dialog box.
e3. clientCompositionExtension (required)	`local`	This parameter must specify the fully-qualified name of the Java class to be instantiated and called as part of the publishing operation. The Java class must implement the interface **com.arbortext.e3c. E3ClientCompositionExtension**.
e3.clientOptions (optional)	`string`	This must be a list of the names of ACL set options separated by `|` characters. The Client Composer retrieves the value of each option and passes the value to the Arbortext PE server.
e3.composerPath (required)	`markup`	This must be the absolute path to the composer **`.ccf`** file that will run on the Arbortext PE server, or an empty string if no composer will be used.
e3.compositionType (optional)	`local`	This is a string that is passed as the value of the *ati-operation-type* parameter when submitting a request to Arbortext Publishing

Parameter Name	Disposition	Description
		Engine. It allows the Arbortext PE server to determine what kind of publishing operation a request will perform. The administrator can configure Arbortext PE server to allocate different Arbortext PE sub-process pools, queues, notifiers, and so on to handle different kinds of publishing requests.
e3.excludedDirectories (optional)	`local`	This must be a colon-separated list of directory names. Each even numbered entry specifies the name of a subdirectory in the input directory; each even-numbered entry specifies the name of a subdirectory of the previous even-numbered entry that should not be transmitted to the Arbortext PE server.
e3.sourceDocument (optional)	`markup`	This must be the ACL ID of the document that will be passed to the Arbortext PE server. It may be omitted if no document is required, such as when importing a Word document. It must be accompanied by a type of either `local` or `opendoc`. If `local`, no document is transmitted to the Arbortext PE server. `opendoc` is described in Handling document and opendoc Parameters on page 251.
queue (optional)	`local`	If set to `yes`, the request is queued, not processed immediately.

Client Composer Parameter Types

Apart from the significant parameters, as described in the previous section, the Client Composer determines how to handle most parameters based upon the supported type codes:

- **document**

- **entry**
- **file**
- **http**
- **http-header**
- **local**
- **opendoc**
- **string**

Handling document and opendoc Parameters

The **document** and **opendoc** (open document) parameter types allow the Client Composer to package an XML document for transmission to the Arbortext PE server. The parameter type codes **document** and **opendoc** may be specified alone or followed by suffixes requesting certain kinds of optional processing. For example, a parameter with a type of `opendoc.location` would be considered an **opendoc** parameter with a suffix requesting `location` processing (described in the *Optional Parameter Suffixes* table).

For a parameter of type **document**, the parameter value must be the absolute path to an XML or SGML document. The Arbortext Publishing Engine Client Composer will ask Arbortext Editor to open the document, process the document as if it was an **opendoc** document, and then close it again. The document will be opened using the following flags:

- read-only
- no locking
- no context checking
- no parser error messages
- no stylesheet
- no prompt if the DTD or schema is unknown

The Client Composer calls Arbortext Editor and directs it to open a document by calling the Java method **Application.openDocument(*path*, *flags*)**. It species the absolute path to the document for ***path*** and sets a ***flags*** parameter to flags corresponding to the conditions described in the previous list.

For a parameter of type **opendoc**, the parameter value must be the ACL ID of a document that is already open in Arbortext Editor. The document is left open, and no changes are made to it.

For both **document** and **opendoc** parameter types, the Client Composer will copy the document in memory, modify the copy by processing it, and then write the document using a temporary name to the input directory. The name will begin with the string `x` followed by a number needed to make the name unique. The name will end with the file extension **`.xml`**, **`.html`**, or **`.sgml`**.

The Client Composer will adjust the parameter's value to the name of the file in the input directory, and the parameter's type will be adjusted to **entry**.

If the Client Composer is given a **document** parameter named `testfile` with a value of **`c:\test\inputfile.xml`**, the Arbortext Publishing Engine Server Composer will receive an **entry** parameter named `testfile` with a value of something like **`x10.xml`**. The **`x10.xml`** file in the input directory would be the modified version of the file **`c:\test\inputfile.xml`** on the client.

The Client Composer always begins processing a **document** or **opendoc** parameter by cloning the document in question, making a temporary copy in memory, and replacing entity references with the actual data (sometimes referred to as a "flattened" document). Flattening entity references ensures that the document will have no references to other XML or SGML documents when it is written to disk, which is important because the referenced documents might not be accessible to the Arbortext PE server.

After the document has been opened, copied, and flattened, subsequent processing is controlled by any suffixes included on the **document** or **opendoc** type code, as described in the following table.

Optional Suffix Parameters

Suffix Option	Description
`.gl`	Don't copy graphics
`.location`	insert location PIs
`.3d`	Convert 3D graphics to 2D versions
`.iso-eps`	Convert ISO graphics to EPS format
`.iso-jpg`	Convert ISO graphics to JPG format
`.iso-png`	Convert ISO graphics to PNG format

Graphics are copied unless `.gl` are specified.

To convert ISO graphics, only one of `.iso-eps`, `.iso-jpg`, or `.iso-png` may be specified. If none of these are present, then ISO graphics are not converted.

If the `.location` suffix is present on the type code, the Client Composer optionally inserts location processing instructions. The PIs allow a DVI file returned by the Arbortext PE server to reference locations in the ACL document, which would occur when publishing PDF, PostScript (**`print composed`**), and **`preview`**. The `.location` suffix supports the **Edit ▸Synchronize Editor Position** feature in Arbortext Editor **Preview** window.

Next, the Client Composer processes graphics referenced by the flattened document by copying all of the graphics referenced by the document to a subdirectory of the input directory. After graphic handling is processes, the Client Composer will write the flattened document to the input directory for transmission to the Arbortext PE server.

Copying Graphics

For any publishing process that returns a DVI file to Arbortext Editor, graphic copying is a solution to the problem of transmitting a document with no external references that might be invalid on the Arbortext PE server. Graphic copying is straightforward: for each graphic referenced by an XML or SGML document, as the Client Composer copies the graphic to a subdirectory of the input directory. The graphic reference is adjusted to a relative path reference. The subdirectory name is derived from the base name for the temporary XML or SGML document written in the input directory. If the document is called **`x10.xml`**, its graphics are placed in the subdirectory **`x10-graphics`**. If the document contained reference to the graphic **`c:\test\graphicfile.gif`**, the file would be copied to **`x10-graphics\graphicfile.gif`**. Its reference would be modified to specify the same value. Graphic file names are also modified, if necessary, to prevent name collisions in the destination subdirectory (which might occur for graphics collected from several locations).

Handling Entry Parameters

An **entry** parameter provides a mechanism for transmitting an arbitrary file to the Arbortext PE server as part of the publishing request. The parameter value must be the absolute path to the file. Unlike a file referenced by the **document** parameter, the file designated by an **entry** parameter is not opened as an XML or SGML document by Arbortext Editor. Instead, the file is copied to the input directory (i.e. to the directory that will be transmitted to the Arbortext PE server as a JAR file). An **entry** parameter transmits a file located on the client, without opening the file, expanding entities, copying graphics, or other processing.

The file designated by an **entry** parameter will be transmitted to the Arbortext PE server using only its name. If **entry** parameter `p1` has the value **`c:\test\data.txt`**, it will be copied into the input directory as **`data.txt`**. To avoid name collisions, no **entry** parameter should reference a file named:

- **`index.xml`**
- **`x`*`integer`*`.xml`**
- **`x`*`integer`*`.sgml`**
- **`x`*`integer`*`.html`**
- **`x`*`integer`*`-graphics`**

An entry parameter must specify a path and file, not just a directory.

Arbortext PE server retrieves the file by looking for the entry by name (such as **`data.txt`** from the example) in the JAR file it receives from the client.

Handling File Parameters

A file parameter must have the value of the absolute path to a file located on the Arbortext PE server. When it receives a publishing request from the Client Composer, the Arbortext Publishing Engine Server Composer will examine each file parameter and report an error if the specified file is not found. If a file parameter value starts with `e3:` or `pe:`, the prefix is removed from the parameter value before it is transmitted to the server.

Handling HTTP parameters

An **http** parameter is passed to the Arbortext PE server as a query parameter on the HTTP request.

Handling HTTP-Header Parameters

An **http-header** parameter is passed to the Arbortext PE server as an HTTP header on the HTTP request.

Handling local parameters

A **local** parameter is not transmitted to the Arbortext PE server. Its value is available only to the Client Composer and Client Composition Extension.

Handling string parameters

A **string** parameter is transmitted to the Arbortext PE server with no processing. The string value is simply passed to the server.

The Client Composition Extension

The Client Composition Extension is a Java class that allows operation-specific code to run in the context of the PE Client Composer. The Client Composition Extension is specified by the parameter **e3.clientCompositionExtension**. The parameter must be of type **local**, and its value should be a fully-qualified Java class name. The class must implement the interface **com.arbortext.e3.ClientCompositionExtension**.

The PE Client Composer will instantiate an object of this class once, the first time it is referenced. After the initial use, the Client Composer will save the object in a map and reuse the same object for subsequent publishing operations that specify the same Composition Extension name. A Composition Extension must be serially reusable. It does not need to be thread safe, however, as Arbortext Editor can not execute simultaneous publishing operations.

The com.arbortext.e3c.ClientCompositionExtension Interface

This interface requires the following methods for a Client Composition Extension.

The constructor Method

A Client Composition Extension must define a constructor which takes no parameters. Each client publising object is effectively a singleton; the constructor will be called only once, for the first publishing operation using a particular Client Composition Extension after Arbortext Editor is launched.

The preProcess Method

The **preProcess** method is called by the Client Composition Extension just before it zips the input directory to a JAR file and transmits it to the Arbortext PE server as part of the publishing request. It has the following signature:

```
public void preProcess(
    File inputDir,
    E3ServerCompositionRequest request,
    Map localParameters,
    E3Tracer tracer
);
```

The ***inputDir*** gives the location of the directory that will be zipped into a JAR file and transmitted to the Arbortext PE server. The Client Composition Extension can create files and subdirectories in it, and they will be transmitted to the server along with everything else in the directory.

To avoid naming collisions, recall that the Arbortext Publishing Engine Client Composer may have files beginning with the string **`x`** followed by an integer and a file extension of **`.xml`**, **`.html`**, or **`.sgml`**, along with subdirectories beginning with **`x`** followed by an integer and ending in **`-graphics`**. The Client Composer also writes a file named **`index.xml`** before transmitting the directory to the Arbortext PE server.

The parameter ***request*** is the request that is being transmitted to the Arbortext PE server. The Client Composer will serialize it to the input directory under the name **`index.xml`** before zipping the input directory into a JAR file. The Client Composition Extension can modify the request by calling its **addParameter**, **setDoc**, and **setPath** methods. Refer to the Javadoc for information on these methods.

Adding a parameter simply manipulates string values. If you add a parameter, only the content of **`index.xml`** changes. No work happens because of it, such as opening documents, flattening entities, copying graphics, and so on.

To obtain the processing that the Client Composer provides for **document**, **entry**, **file**, **local**, **opendoc**, and **string**, you must code that behavior directly.

The ***localParameters*** parameter is a map containing the **local** parameters passed to the Client Composer. It contains string parameter names and string parameter values. These parameters are not transmitted to the Arbortext PE server.

The parameter ***tracer*** is a tracing object that makes entries in the Arbortext Publishing Engine debugging log, described in Debugging the Client Composer on page 259.

The postProcess Method

The **postProcess** method is called by the Client Composer after it has received a JAR file containing the results from the Arbortext PE server. This method retrieves and uses the information returned by the Arbortext PE server.

Many publishing methods do little processing in their **preProcess** methods because the Client Composer can do some of the work. However, **postProcess** performs any processing of the publishing results returned, because there's little common processing for the Client Composer to do. The **postProcess** method has the following signature:

```
public void postProcess(
    E3ServerCompositionResult result,
    JarFile responseJar,
    E3Tracer tracer
);
```

The parameter ***result*** is an instance of the class **com.arbortexte3c. E3ServerCompositionResult**, which describes the result of the publishing operation. The result is the **`result.xml`** returned by the Arbortext PE server in the returned JAR file.

The parameter ***responseJar*** is a JAR file descriptor for the entire file returned by the Arbortext PE server. The file may contain one or more files produced by the publishing operation on the Arbortext PE server, including graphics and subdirectories containing result files. The primary job of the **postProcess** method is to extract the files in this archive and relocate them to their proper local locations rather than on the Arbortext PE server.

The parameter ***tracer*** is a tracing object that makes entries in the Arbortext Publishing Engine debugging log, described in Debugging the Client Composer on page 259.

The com.arbortext.e3c. E3ServerCompositionParameter Interface

This interface describes a parameter transmitted to the Arbortext PE server as part of a publishing request. Parameters are logically contained in an object that implements the interface **com.arbortext.E3ServerCompositionRequest**, which is written to a file named **`index.xml`** in the input directory that is transmitted to the Arbortext PE server as part of the JAR file. A parameter consists of three strings specifying the parameter name, value, and type. Parameter names must be unique. Valid types are `string`, `file`, and `entry`.

A `string` parameter has a simple string value, and Arbortext PE server does not perform any special processing. A `file` parameter represents a file on the Arbortext PE server, and its value must be an absolute path. The Arbortext PE server will check that the file exists as part of setup before running the publishing request. An `entry` parameter represents a file being transmitted to the Arbortext PE server, and its value must be the name of a file in the input directory, which becomes an entry in the JAR archive transmitted to the server.

These parameter types are a subset of the parameter types that may be passed to the Client Composer. As part of processing the parameters it receives, the Client Composer processes **local** parameters by not encoding them for transmission to the Arbortext PE server. The Client Composer processes the **document** and **opendoc** parameters by converting them to **entry** parameters and copying the underlying documents into the input directory.

You can create parameters of your own by creating objects that implement the **E3ServerCompositionParameter** interface and adding them to the server request by calling the **E3ServerCompositionRequest.addParameter** method. You can use the class **com.arbortext.e3.ServerCompositionParameter** (though intended for internal use, custom code can use it).

Simply creating a parameter and adding it to a request does not perform the processing. You must place a file in the input directory before creating an **entry** parameter, and verify that a file really does exist on the Arbortext PE server before creating a **file** parameter.

The com.arbortext.e3c.E3ServerCompositionRequest Interface

The **com.arbortext.e3c.E3ServerCompositionRequest** interface describes a request that will be transmitted to the Arbortext PE server for processing by the Arbortext Publishing Engine Server Composer. The request will be written to a file named **`input.xml`** in the input directory, which will be zipped into a JAR file for transmission to the Arbortext PE server.

Logically, a request consists of a document, a composer path, and a collection of parameters. The document is simply an **entry** parameter indicating which entry in the input JAR file is to be published. The composer path is a **file** parameter with a value of the absolute path to the content pipeline **`.ccf`** file that will be run on the server. Both the document and composer path are optional. For example, an import publishing operation does not require an input document, and some publishing operations proceed without using a content pipeline.

This object provides accessors and modifiers for the document, the content path, and the parameters.

The com.arbortext.e3c.E3ServerCompositionResult Interface

The **com.arbortext.e3c.E3ServerCompositionResult** interface describes the response to a publishing request returned by the Arbortext PE server. It indicates whether the operation succeeded and tells where to find the information returned by the server in the JAR file transmitted from the server.

The getSuccess Method

This method returns `true` if the publishing operation succeeded and `false` if an error prevented the operation from running correctly.

The getOutputFileEntry Method

This method returns the name of the entry in the output JAR file that contains the output of the publishing operation. It returns null if the operation doesn't return a single output file.

The getOutputDirPrefix Method

This method returns a string that is a prefix for all files in the output JAR file that are in the publishing operation's output directory. It returns null for those operations that don't return a directory of files. Some operations may have both an output file and an output directory. For example, if the output file was an HTML file, the output directory might contain the graphics it references. Other operations might return only a file or only an output directory. No operation intentionally returns no output.

The getErrorFileEntry Method

This method returns the name of the entry in the output JAR file that contains an error message explaining why the publishing operation failed on the Arbortext PE server. This method returns null if no error occurred.

The com.arbortext.e3c.E3Tracer Interface

The **com.arbortext.e3c.E3Tracer** interface provides a way for the Client Composer and Client Composition Extensions to write information and error messages to an output log for debugging purposes. Messages passed to the interface’s **info** method are discarded unless the Client Composer is configured to write them.

Messages sent to the tracer are written to the Application Save log and to the Arbortext Diagnostics window (launch it from the shortcut called **Arbortext Publishing Engine Diagnostics** in your PTC program group) when the Client Composer runs on Windows.

Queuing Support

For an immediate request, after transmitting the request to the Arbortext PE server, waiting for a response, and receiving the response, the Client Composer calls the internal method **handleResponse**. This method examines the response, which will be either an XHTML error message (in case of severe error) or a JAR file containing the result of the publishing operation. **handleResponse** extracts data from the JAR file and places the files it contains in the locations to which they would have been written had the publishing operation been performed locally. Then **handleResponse** calls the post-process function in the outer layer of the publishing framework, described in The Outer Layer of the Publishing Framework on page 225.

For a queued request, the Client Composer receives a nearly instant response from the Arbortext PE server. The Client Composer immediately returns `ok` to the inner layer of the publishing framework, which returns to the outer layer, and the operation is finished.

At a later time, the user may launch the Queued Transaction Viewer, which asks the Arbortext PE server for the status of all queued transactions submitted by the user. If a publishing request has completed processing, the user can select the transaction and clicks the **Download** button. Arbortext Editor retrieves the JAR archive from the Arbortext PE server, then calls the Client Composer's **handleResponse** routine, which proceeds as if the JAR file was the response to an immediate request.

Debugging the Client Composer

To obtain a log containing the Client Composer's execution information, together with the files sent to and received from the Arbortext PE server, set the ACL option **`set debugcomposition`** to `on` on the Arbortext Editor client and perform one or more publishing operations. Then select **Tools ▸Save Application**. Publishing data will be placed in the file **`compose.zip`** in the **`appsave`** directory produced by the Save Application utility.

20 Arbortext Publishing Engine Server Composer

Support for Arbortext Publishing Engine publishing on the Arbortext PE server is called the PE Server Composer.

The PE Server Composer is simply an Arbortext PE Application implemented in Java. It runs in an Arbortext PE sub-process like any other Arbortext PE Application.

The Server Composer application is invoked by an Arbortext PE sub-process at the request of the Arbortext PE Request Manager. It receives the request from the Arbortext Editor client, which includes the input directory zipped as a JAR file. It opens the JAR file, which contains information about what operation to perform along with the data required by the operation. It performs the requested publishing operation, and places the results in a temporary directory known as the output directory. Then it zips the output directory into a JAR file and returns the JAR file to the Arbortext PE Request Manager, which returns it to the Arbortext Editor client.

The PE Server Composer application consists of three components:

- The Composition Application, which is a standard Arbortext PE Application. It receives control from the Arbortext PE Request Manager and invokes the next layer, which is the PE Server Composer.
- The PE Server Composer performs operation-independent setup processing, then loads and calls an operation-specific PE Server Composition Extension.

- The PE Server Composition Extension does the real work by invoking code (usually the outer layer of the publishing framework) that knows how to run the content pipelines and content composers.

Publishing Applications

The Arbortext PE server Publishing Application is a standard Arbortext PE Application implemented in Java. It runs in an Arbortext PE sub-process and it's implemented by the class **com.arbortext.e3c.Application**. It expects to be invoked when the Arbortext PE Request Manager receives a POST request accompanied by an HTTP request body with a content-type of `application/x-java-archive` and the following request query parameters:

- **f = java**
- **class = com.arbortext.e3c.Application**
- **composer-class = com.arbortext.e3c.ServerComposer**

The Publishing Application will not be called by the Arbortext PE Request Manager if the **class** parameter has another value. It also issues an error message if there is no request body or the content-type header indicates another type.

The **composer-class** query parameter could have another value; but the code that runs on the Arbortext Editor client always specifies this value. If another value is specified, it must be a Java class implementing the interface **com.arbortext.e3c.E3ServerComposer**.

The Publishing Application begins by retrieving and validating the content-type header. It will ignore an optional `; charset=` specification as allowed by the HTTP protocol. It retrieves the POST request body and saves it to disk. Then it searches for, loads, and instantiates an object of the class indicated by the **composer-class** parameter, which is the PE Server Composer. Then it calls the Server Composer's **doCompose** method.

The Publishing Application maintains a cache of instantiated Server Composer objects, and only loads and instantiates an object for a given class once. This means that Server Composers should be serially reusable. An Arbortext PE sub-process can only process one request at a time, so a Server Composer doesn't need to handle simultaneous requests.

The Server Composer's **doCompose** method takes the POST request body file as a parameter, and it is expected to return a Java File object referencing a JAR file. The Publishing Application will return the JAR file to the Arbortext PE Request Manager (who returns it to the client) with a content-type of `application/x-java-archive`.

If the Publishing Application detects any errors, it returns an XHTML document with content-type `text/xml` instead of a JAR file.

Arbortext Publishing Engine Server Composer

The PE Server Composer is a Java object of a class implementing the interface **com.arbortext.e3c.E3ServerComposer**. A Server Composer object is instantiated the first time the Publishing Application processes a request with a **composer-class** parameter that specifies the Server Composer's class name. The Server Composer is essentially a filter intended to transform a JAR file of input data to a JAR file of output data.

The PE Client Composer code always specifies the Server Composer class name as **`com.arbortext.e3c.ServerComposer`**.

The Arbortext Publishing Engine Server Composer's primary method is:

```
File doCompose( File inputJar );
```

The ***inputJar*** parameter specifies a JAR file transmitted from the Arbortext Editor client. The return value is a **File** object naming a JAR file to be returned to the client.

A Server Composer also exports a default constructor and a **destroy** method, and neither takes any arguments. The distributed Server Composer allocates a working directory in its constructor and deletes the directory and its content in its **destroy** method.

In its **doCompose** method, the PE Server Composer begins by searching its input JAR file for an entry named **`index.xml`**, which it parses into a **com.arbortext.e3c.E3ServerCompositionRequest** object. Before continuing, it looks for the name of the PE Server Composition Extension it will call later (by retrieving the value of the parameter **e3.serverCompositionExtension**) and checks that the underlying object has been instantiated.

The PE Server Composer creates directories named **`source`** and **`target`** in the transaction directory allocated by the Arbortext PE Request Manager to handle the publishing request. Then PE Server Composer restores its input JAR archive to the **`source`** directory.

Next, the PE Server Composer creates an empty property map (see the AOM interface **com.arbortext.epic.PropertyMap** in the *Programmer's Reference*) and copies parameter names and values from the **E3ServerCompositionRequest** object to the property map. Each parameter has a type of either `string`, `file`, or `entry`. The Server Composer copies the value of `string` parameters. For `file` parameters, it checks whether the parameter value specifies the absolute path to an existing file, and fails with an error message if the file isn't found. For `entry` parameters, it checks to make sure that the JAR file entry specified by the parameter value has been restored to a file in the **`source`** directory. If so, it replaces the parameter value with the absolute path to the restored entry. If not, it issues an error message and fails.

Next, the PE Server Composer allocates an empty result object (see the Javadoc for the interface **com.arbortext.e3c.E3ServerCompositionResult**). Then it calls the Server Composition Extension specified by the client, which does the real publishing work.

When the Server Composition Extension returns, the PE Server Composer serializes the result object to the file **`result.xml`** in the output directory. Then it compresses the output directory into a JAR archive and writes the archive to the file that will be returned as the HTTP response body. The PE Server Composer then returns to its caller, which returns to the Arbortext PE Request Manager, which returns the response to the Client Composer on the client machine.

Note that the Server Composer does not need to delete its **`source`** and **`target`** directories. They are created in the transaction directory managed by the Arbortext PE Request Manager. After the Arbortext PE Request Manager returns the response to the client, it may be configured to save the transaction directory as a zip archive in the transaction archive and then delete the transaction directory, including the **`source`** and **`target`** directories. Consult the *Configuration Guide for Arbortext Publishing Engine* for information.

Writing a Custom Server Application

If you write custom code that runs as part of the Arbortext Publishing Engine Server Composer (your own Server Composition Extension, for example) be sure that your code close any files that it opens in the transaction directory, **`source`** directory, or **`target`** directory. If your code leaves a file open, the Arbortext PE Request Manager will be unable to delete the transaction directory, and the Arbortext Publishing Engine will begin to leak disk space.

The Server Composition Extension

A Server Composition Extension is a Java object that implements the interface **com.arbortext.e3c.E3ServerCompositionExtension**. The class is loaded, and an object is instantiated, by the PE Server Composer when the Server Composition Extension's class name is specified in the parameter **e3.serverCompositionExtension**.

A Server Composition Extension must be serially reusable. After it's instantiated, it can perform any number of publishing operations. It is not required to be thread safe, as a single Arbortext PE sub-process can only perform one publishing operation at a time. However, because several Arbortext PE sub-processes could run separate instantiations of the same Server Composition Extension at the same time, developers should produce programs safe for multi-processing, following good programming practices such as using unique file names. Refer to Concurrency on page 108 for more information.

The **com.arbortext.e3c.E3ServerCompositionExtension** interface specifies a single method, **doCompose**:

```
void doCompose(
    E3ServerCompositionRequest request,
    E3ServerCompositionResult result,
    File sourceDir,
    File targetDir,
    Map jarMap,
    PropertyMap propMap,
    E3Tracer tracer
);
```

- ***request***

is an object into which the file **`index.xml`** from the source directory was loaded.

- ***result***

  is the object to be returned to the client as **`result.xml`**.

- ***sourceDir***

  is the directory where the input JAR file was restored.

- ***targetDir***

  is the directory which will be compressed into a JAR file and returned to the client.

- ***jarMap***

  is a map that translates JAR entry names in the input JAR to the absolute paths of the files in the **`source`** directory where each entry was restored.

- ***propMap***

  is a map that contains the publishing parameters

- ***tracer***

  used to write trace information

Server Composition Extension Processing

Most Server Composition Extensions follow the same basic flow of operation. However, the processing of each Server Composition Extension will depend on the publishing type.

1. The extension creates a content pipeline for the publishing type.
2. Then it copies the content pipeline's default parameters into a Java map.
3. The Server Composition Extension copies the parameters sent by the client into the same map, so that parameters from the client override the defaults.
4. Next, the Server Composition Extension opens the document to be published and calls the appropriate **compose_for_*type*** function to invoke the publishing framework. Since the **`set peservices`** setting is always `off` on the Arbortext PE server, the publishing framework will perform the publishing operation.
5. Once the publishing framework returns, the Server Composition Extension closes the document and returns.

Debugging the Server Composer

The **`source`** and **`target`** directories that the Server Composer creates are located in the transaction directory allocated by the Arbortext PE Request Manager when it processed the **f=java** request that ran the Server Composition Application.

To obtain the transaction directories, configure Arbortext Publishing Engine so that the transaction archive retains all transactions (explained in *Configuration Guide for Arbortext Publishing Engine*), then use the Arbortext Publishing Engine web page (explained in Monitoring and Reporting Using a Web Browser on page 26) to locate the transaction archive directory and the archive entry (zip archive) for the transaction. You can restore the zip archive to a temporary directory.

The Server Composition Application, Server Composer, and Server Composition Extensions all log their behavior using the same **`log4j`** mechanisms as other Arbortext PE Applications. To obtain the log information, set the level of logging you desire, as described in *Configuration Guide for Arbortext Publishing Engine*.

21 Server Configuration for Publishing

Overview

When Arbortext Editor transmits a publishing request to the Arbortext PE server, it only includes the document to be published and any graphic files that the document references. It does not include files describing the document type, the stylesheet to be used in publishing the document, or any custom code that might be needed.

The Publishing Configuration subsystem allows the Arbortext PE server to provide Arbortext Editor clients with a list of the document types, stylesheets, framesets, and other information stored on the Arbortext PE server. The server generates a Publishing Configuration document, and then the client retrieves this document and consults it to determine what is installed on the server.

There are actually four versions of the Publishing Configuration document:

- an XML document that lists everything available for publishing operations on the Arbortext PE server
- an expanded, detailed version of the XML document, which contains MD5 checksums for everything installed on the server. This version enables Arbortext Editor to perform an exhaustive comparison of its configuration with that of the Arbortext PE server.
- an HTML version of the XML list document, which you can read for debugging.
- a log of the scan that Arbortext PE server performs when it builds the Publishing Configuration document, which can also be used for debugging.

Publishing Configuration on the Server

The Publishing Configuration documents are generated by the Arbortext PE server each time it starts. The documents are stored in a cache for retrieval by all Arbortext Editor clients.

You can force the Arbortext PE server to regenerate the Publishing Configuration documents without restarting by selecting the **Rescan Publishing Configuration** link on the Arbortext PE server index page (refer to Monitoring and Reporting Using a Web Browser on page 26 for information). For example, you would need to regenerate if you installed a new document type or other publishing-related information on the server.

You can examine the Publishing Configuration document by selecting the **short**, **detailed**, or **log** links on the Arbortext PE server index page.

The log is the primary tool for debugging problems in the Publishing Configuration document. For example, you can use the log to determine why something you installed on the server doesn't appear in the Publishing Configuration document retrieved by an Arbortext Editor client.

Client Use of Publishing Configuration

An Arbortext Editor client retrieves the Publishing Configuration document from the Arbortext PE server each time it enables publishing against an Arbortext PE server. under the following circumstances:

- when Arbortext Editor starts and Arbortext Publishing Engine publishing was enabled during its last session
- When the **Use Publishing Engine** preference is changed from cleared to checked and the **URL** preference value is set to a valid URL of an Arbortext PE server (**Tools ▸Preferences ▸Publishing Engine**)
- When the**URL** preference value is changed to a valid URL of a different Arbortext PE server while the **Use Publishing Engine** box is checked (also in **Tools ▸ Preferences ▸Publishing Engine**)
- when the **`set`** command option **`peservices`** is changed to `on` and **`peserverurl`** is set to a valid URL of an Arbortext PE server
- when the **`set`** command option **`peserverurl`** changes to the valid URL of a different Arbortext PE server while the set option **`peservices`** is set to `on`

A good way to force Arbortext Editor to again retrieve the Publishing Configuration document from the Arbortext PE server is to clear the **Use Publishing Engine** check box and then reopen **Preferences ▸Publishing Engine** and check the box.

You can display the Arbortext Editor version of the Publishing Configuration document by selecting **Help ▸About Arbortext Editor ▸PE Configuration**. Either the Publishing Configuration document will be displayed, or an error dialog box will appear explaining why Arbortext Editor is unable to communicate with the Arbortext PE server.

Content of the Publishing Configuration Document

The XML, detailed XML, and HTML versions of the publishing configuration document contain the same information. Each contains global information about the Arbortext PE server; then presents lists of composers, files in the composer path, document types, stylesheets, framesets, import templates, PDF configuration files, APP configuration files, installed applications, and supported Arbortext Editor client versions.

General Server Information

The general information section of the Publishing Configuration document contains the following global information about the Arbortext PE server:

- Host Type (operating system)

- Whether Arbortext APP is installed
- Whether the HTML Help compiler is installed
- Whether the Arbortext Import/Export feature is installed
- Whether an Arbortext Print Composer license is installed
- Whether an Arbortext Web/Wireless Composer license is installed
- Whether to use Arbortext APP or the FOSI print engine to generate PostScript and PDF by default
- The build name, date, and version for the Arbortext PE Request Manager
- The build name, date, and version for Arbortext PE sub-processes

Composers

The Composer section of the Publishing Configuration document lists every composer configuration file (**`.ccf`**) defined on the Arbortext PE server. For each composer file, the document lists the absolute path to the **`.ccf`** file and every parameter defined for the composer. For each parameter, the document lists the parameter name, whether the parameter is required, the default value if one exists, the parameter type, and (for enumerated parameters only) legal parameter values.

Files in the Composition Path

The Composition Path section of the Publishing Configuration document lists the directories in the composer path on the Arbortext PE server. For each directory, the Publishing Configuration document lists the name of every file available in the directory.

Document Types

The Document Type section lists each document type installed on the Arbortext PE server. A document type can be a DTD or schema. For each document type, the Publishing Configuration document lists the following information:

- the absolute path to the directory containing the document type definition.
- a list of public IDs that map to the document type
- a list of URIs that map to the document type
- a list of stylesheets for the document type
- a list of composers associated with the document type
- a list frame sets for the document type
- a list of system ID mappings for the document type if available
- a list of stylesheet orderings if available

Path to Document Type

The path to the document type directory is the unique identifier for each document type.

Public ID Lists, URI Lists, and System ID Mappings

To Arbortext Editor or an Arbortext PE sub-process, a document type is simply a directory containing information about how to process a particular kind of document. The document type directory contains a Document Type Definition (**`.dtd`**) file or a Schema (**`.xsd`**) file. It also contains stylesheets, framesets, and other information about the document type. The only truly unique identifier for a document type is the absolute path to the directory in which it resides.

An SGML or XML document specifies its document type somewhat indirectly, by specifying a Public ID, a System ID, or a URI. A given document type may be identified by any number of public IDs, System IDs, or URIs. These identifiers are associated with the document type directory by directives in one or more catalog files, which are read by Arbortext Editor and by Arbortext PE sub-processes as they initialize.

An SGML or XML document usually has either a document type declaration or a schema declaration that specifies its document type. A document type declaration specifies a Public ID, a System ID, or possibly both. A schema declaration specifies a URI. A public ID is simply a text string; a system ID is an absolute or relative path to the **`.dtd`** or **`.xsd`** file. A URI is a URL for locating an **`.xsd`** file on the internet.

When Arbortext Editor or an Arbortext PE sub-process initializes, it scans a list of directories for catalog files. Catalog files may contain PUBLIC, URI, or SYSTEM directives (as well as others). A PUBLIC directive maps a Public ID to a System ID, which allows Arbortext Editor or an Arbortext PE sub-process to map a Public ID string to the path to a **`.dtd`** or **`.xsd`** file. A URI directive maps a schema URI to a System ID, which allows Arbortext Editor or an Arbortext PE sub-process to map a URI to an **`.xsd`** file. A SYSTEM directive maps one system ID to another; that is, a directive such as:

```
SYSTEM a b
```

directs Arbortext Editor or an Arbortext PE sub-process to look for file `b` instead of looking for file `a`.

When Arbortext Editor or an Arbortext PE sub-process opens an SGML or XML document, it finds the relevant document type directory by looking for a document type or schema declaration. If the document contains a schema declaration, Arbortext Editor or an Arbortext PE sub-process looks for the schema-to-directory mapping provided by a URI directive in a catalog file. If the document contains a document type declaration with a Public ID, Arbortext Editor or an Arbortext PE sub-process looks for a public ID-to-directory mapping provided by a PUBLIC directive in a catalog file. If the document contains a document type declaration with only a System ID, Arbortext Editor or an Arbortext PE sub-process uses the System ID directly. In all cases, if a URI or a Public ID can be translated to a System ID, or if a System ID is specified directly, Arbortext Editor or an Arbortext PE sub-process checks to see if the System ID is mapped to another location by a SYSTEM directive.

When the Publishing Configuration document is assembled, the Arbortext PE sub-process looks for PUBLIC, URI, and SYSTEM directives in every installed catalog file; each directive maps a Public ID, a URI, or a System ID to a document type directory. The Arbortext PE sub-process adds the mapping to the appropriate list for the document type.

When an Arbortext Editor client tries to publish a document using an Arbortext PE server, it must be able to locate the document's document type on the Arbortext PE server. To find the server-based document type, it looks in the Publishing Configuration document for a document type with a matching Public ID, URI, or system ID.

System ID Mapping Between Client and Server

Some document types are defined and then referenced only by System ID (in other words, by an absolute path). This complicates using Arbortext PE server for publishing operations, because the document type is likely to be installed in a different location on the server than on the client.

To get around this problem, you can add a catalog file to a document type directory on the Arbortext PE server that contains SYSTEM declarations for these document types. Each SYSTEM declaration should map an absolute path to the document type on a client to the absolute path to the document type on the Arbortext PE server.

You'll want to follow the practice of reducing the number of mappings needed if you're using the Arbortext PE server. Installing document types in the same location on every client machine will help minimize the total number of SYSTEM mappings you need to define.

Stylesheet List

The Stylesheets section of the Publishing Configuration document lists each of the stylesheets associated with a document type. For each stylesheet, the document lists the name, the absolute path to the stylesheet, and the types of publishing that the stylesheet supports. The document also indicates whether a stylesheet supports the FOSI engine, APP, or both.

Composer Associations

In the Publishing Configuration document, the Composer Associations section of the Document Type lists composer override information for a document type. A document type can specify a different composer (**`.ccf`**) file to use when publishing its documents to a particular output format. It also allows the document type to change the default parameter values used by the composer.

Frame Sets

The Frame Sets section of the Publishing Configuration document lists additional frame sets associated with a document type.

Stylesheet Orderings

The Publishing Configuration document, the Stylesheet Orderings section of the Document Type contains stylesheet order preferences for the document type. This section displays any configuration of the order of stylesheets presented in the Arbortext Editor stylesheet selection list. The order is controlled by configuring **PEStylesheetOrder** in the document type's **`.dcf`**.

Framesets

This section of the Publishing Configuration document lists frame sets installed on the Arbortext PE server that can be used when publishing to the Web output format. These framesets are not associated with any particular document type.

PDF Configuration Files

This section of the Publishing Configuration document lists PDF configuration files. The Arbortext PDF generator (PubPDF) uses these files when translating a document to PDF.

APP Configuration Files

This section of the Publishing Configuration document lists APP configuration files. The Arbortext Advanced Print Publisher uses these files when translating a document to PostScript or PDF.

Import Templates

The Import Templates section of the Publishing Configuration document lists templates for the Arbortext Import/Export feature that are available for importing a non-XML document to XML. For more information about Arbortext Import/Export, refer to the Arbortext online help and the Arbortext Import/Export documentation, including *Tutorial for Arbortext Import*, *Reference Guide to Arbortext Import*, and the *Customizer's Guide*.

Applications

This section of the Publishing Configuration document lists the optional applications installed on the Arbortext PE server, along with the platforms, application versions and client versions they support.

Client Versions

The Client Versions section of the Publishing Configuration document lists the versions of Arbortext Editor that the Arbortext PE server supports as publishing clients.

VI

Implementing Programs and Scripts for Arbortext Publishing Engine

22 Custom Applications

Overview of Custom Programs and Scripts

The Arbortext Editor and Arbortext Publishing Engine installations have directory structures within them where you can place your custom scripts and programs. The **`custom`** and the **`application`** directories are described in the following sections.

The Custom Directory Structure

The ***`Arbortext-path`*`\custom`** directory has a subdirectory structure designed to hold your custom programs and scripts and make them automatically available during the session. At startup, these subdirectories are searched for Java, JavaScript, JScript, VBScript, ACL, and composer configuration files. You can also provide custom document types, entities, fonts, graphics, and native shared libraries and DLLs. The supported file types are automatically accessed if they reside in the appropriate subdirectory. Implementing your custom files using this approach takes advantage of the startup sequence to automatically locate your custom files. The ***`Arbortext-path`*`\custom`** directory and its subdirectories are explained in detail in this chapter.

The Application Directory Structure

The ***`Arbortext-path`*`\application`** subdirectory can contain custom applications as well as application software distributed by Arbortext. The **`application`** directory must have one or more uniquely named subdirectories, each containing a specific configuration file, **`application.xml`**, that conforms to a specific format. At startup, the **`application`** directory is searched for subdirectories and the presence of a valid **`application.xml`** file. In the uniquely named subdirectory, all subdirectories of the **`custom`** directory are supported. The custom application in a **`application`** then uses these subdirectories in the same way as the **`custom`** directory structure. You can also have additional subdirectories needed to support the implementation of this type of custom application. Implementing your custom application using this approach takes advantage of the startup sequence, supports delivering a completely self-contained custom application, and offers the option of setting the conditions for whether the application should be loaded. The **`application`** directory is also explained in this chapter.

Description of the Custom Directory Structure

When Arbortext Editor or an Arbortext PE sub-process starts, it can access custom files placed in specific directories. At startup, it automatically looks for compiled Java files (**`.class`** and **`.jar`** files), JavaScript, JScript, VBScript, ACL, document type, publishing configuration and other types of files within the ***Arbortext-path*\custom** directory structure.

You can have one or more **custom** directories outside the ***Arbortext-path*** install tree. To specify a path list for their locations, set the **APTCUSTOM** environment variable. The **custom** directory must be located using a file system; HTTP references are not supported.

At startup, some search paths are automatically prepended with the path to a **custom** subdirectory. Startup automatically sets some of these search paths using a symbolic variable as a path specification. You can use symbolic parameters to represent a search path in the context of the default search path, the location of the install tree, or the locale.

If a directory supports more than one type of file, the file types are processed in the following order:

- **.acl** (Arbortext Command Language) files
- **.js** (JavaScript or JScript) files
- **.class** (Java) files
- **.vbs** (VBScript) files

For each file type, its files are processed in alphabetical order by file name.

The ***Arbortext-path*\custom** directory is processed at startup. If you add custom applications and document types after startup, they're not recognized during the session. If you're using Arbortext Editor, it needs to be closed and restarted. If you're using Arbortext Publishing Engine, you need to stop and restart the Arbortext Publishing Engine to re-initialize the Arbortext PE sub-processes.

custom.xml File

At the top level of the **custom** directory is the **custom.xml** file. Following is the default version of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Arbortext, Inc., 1988-2009, v.4002-->
<ApplicationConfiguration
   xmlns="http://www.arbortext.com/namespace/doctypes/appcfg">
     <Information>
     <!--The following name will be shown in the New dialog
         as the category for all document types in this
         custom directory that do not specify a category.-->
      <Name>Custom Directory Name</Name>
   </Information>
</ApplicationConfiguration>
```

This file is only used when you have a custom document type in the **custom \doctypes** subdirectory, and you have not designated a category name for the document type in the associated document type configuration (**.dcf**) file's **NewDialog** element. In this case, the name in the **custom.xml** file's **Name** element is used as the **Category** name for the document type(s) in the **custom\doctypes** subdirectory in the **New Document** dialog box.

Subdirectory Structure

The following list describes each **custom** subdirectory and how it's used. Arbortext Editor and Arbortext Publishing Engine look in these directories for any references that use a relative path or have no specified path.

- **classes** subdirectory

  Holds compiled Java **.class** and **.jar** files.

  The Arbortext Editor and Arbortext Publishing Engine JVM Java class path holds a list of directories and paths to **.jar** files. Any files matching ***.jar** are prepended to the JVM Java class path. Then the **classes** parent directory is prepended, putting it first in the JVM Java class path.

  In cases where a class file occurs in more than one **.jar** file, you can extract the preferred **.class** file from its **.jar** file and place it in a subdirectory path of the **classes** directory to control which one takes precedent.

- **composer** subdirectory

  Holds publishing configuration files (**.ccf**, **.ent**, and **.xml** files) and can support a **catalog** file. Supports one level of subdirectories.

  The default path is ***Arbortext-path*\composer**. If there are any subdirectories of the **custom\composer** directory, those subdirectories are prepended to the publishing configuration path. Then the **custom\composer** parent directory is prepended to the path. If the **custom\composer** directory contains a **catalog** file, that directory is also prepended to the catalog path.

- **datamerge** subdirectory

  Holds data merge configuration (**.dmf**) files specifying queries and their components. The **.dmf** file structure is discussed in the *Customizer's Guide*.

- **dialogs** subdirectory

  Holds dialog files that can be accessed from custom applications, such as one that uses the AOM **Application.createDialogFromFile** method.

  The ***Arbortext-path*\samples\XUI\preferences\pref_exts.zip** contains a sample application that adds a tab to the Preferences window as a way to extend preferences for custom applications. Refer to the **readme.txt** file for more information.

  If there are any subdirectories of the **custom\dialogs** directory, those subdirectories are prepended to the dialog path. Then the **custom\dialogs** parent directory is prepended to the dialog path.

- **ditarefs** subdirectory

  Holds content referenced by DITA documents when the reference is not specified as either an absolute path name or a path name relative to the current document directory. For example, the **ditarefs** subdirectory could hold content referenced by topic references, content references, and so forth. Supports one level of subdirectories.

  The default DITA reference path is ***Arbortext-path*\ditarefs**. The DITA references path can be set in the **File Locations** category of the **Tools ▸ Preferences** dialog box. You can also use the **set ditapath** option or the **APTDITAPATH** environment variable to set the default path for DITA references. If there are any subdirectories of the **custom\ditarefs** directory, those subdirectories are prepended to the path. Then the **custom\ditarefs** parent directory is prepended to the path.

  Note

  Graphic references from DITA documents are resolved using the graphics path list.

- **dictionaries** subdirectory

  Holds user-defined dictionary files that can be used by the spelling checker. Supports one level of subdirectories.

  The default path is ***Arbortext-path*\lib\proximity\userdict**. If there are any subdirectories of the **custom\dictionaries** directory, those subdirectories are prepended to the dictionary path. Then the **custom\dictionaries** parent directory is prepended to the dictionary path.

- **doctypes** subdirectory

  Holds a custom **catalog** file and document type files. Supports one level of subdirectories. Each document type should reside in a uniquely named subdirectory of **doctypes**. The subdirectory should also contain a **catalog** file for the custom document type. A **doctypes** subdirectory can also contain a subset of the complete document type file set. You can place a document type configuration file **.dcf** or stylesheets in a **\custom\doctypes\*doctype*** directory.

  You can add a stylesheet to the list of stylesheets that displays when you make a publishing request using one of the **File ▸Publish** choices. Arbortext Editor and Arbortext Publishing Engine search each **\custom\doctypes\*doctype*** directory and aggregate the list of stylesheets. For example, you can add stylesheets for the Arbortext Simplified XML DocBook Article built-in document type (**asdocbook**) by placing them in ***Arbortext-path*\custom\doctypes\asdocbook**.

  If a document does not specify an Editor view stylesheet with a stylesheet association PI, Arbortext Editor will first search first the document directory, then the relevant **`\custom\doctypes\`*`doctype`*** directory, and finally the original location for the ***`doctype`*** directory.

If the subdirectory contains only a **`.dcf`** file, it must conform to a naming convention that expects the subdirectory and **`.dcf`** file name to reflect the base document type name. For example, you could customize the default Arbortext Simplified XML DocBook Article **`asdocbook.dcf`** file, and put it in ***`Arbortext-path`*`\custom\doctypes\asdocbook\asdocbook.dcf`** to override the built-in **`.dcf`**. Note that the document type subdirectory and file name must be the same as the default document type name for Arbortext Editor and Arbortext Publishing Engine to find all the relevant document type files.

A DCF file can reference other files, such as the **`.pcf`**, **`demo.xml`**, and **`template.xml`** files. Custom versions of these files can be placed with the **`.dcf`** in **`\custom\doctypes\`*`doctype`***. If Arbortext Editor and Arbortext Publishing Engine find a **`.dcf`** in the **`\custom\doctypes\`*`doctype`*** location, relative path references are resolved by first searching the same directory as the **`.dcf`** and then by searching the document type directory in the original location.

The default catalog path is ***`Arbortext-path`*`\doctypes`**. If there are any subdirectories of the **`custom\doctypes`** directory that contain a **`catalog`** file, those subdirectories are prepended to the catalog path. Then the **`custom\doctypes`** parent directory is prepended to the catalog path.

You can place custom tag template files (**`.tpl`**) in a **`custom\doctypes\`*`doctype`*`\tagtemplates`** directory. The **`custom\tagtemplates`** directory can also be used as a more generally available location for tag templates.

Any document type from the **`custom\doctypes`** directory is also added to the list of available document types that are displayed in the **File ▸New** dialog box.

- **`entities`** subdirectory

  Holds file entities. Supports one level of subdirectories.

  A file entity is any structurally complete document unit saved as a file. File entities commonly have an **`.xml`** file extension.

  The default entity path is ***`Arbortext-path`*`\entities`**. If there are any subdirectories of the **`custom\entities`** directory, those subdirectories are prepended to the entity path. Then the **`custom\entities`** parent directory is prepended to the entities path.

- **`fonts`** subdirectory

  Holds custom AFM or TFM font metric files (**`.afm`** and **`.tfm`**).

  The default fonts path is ***`Arbortext-path`*`\fonts`**. If there are fonts in **`custom\fonts`**, the path is prepended. If the APTTEXFONTS environment variable is set, the **`custom\fonts`** directory is prepended to it.

- **`formats`** subdirectory

  Holds custom PubTex format files (**`.fmt`**).

The default PubTex format path is ***Arbortext-path*\formats**. If there are **.fmt** files in **custom\formats**, the path is prepended. If the APTTEXFMTS environment variable is set, the **custom\formats** directory is prepended to it.

- **framesets** subdirectory

  Holds custom framesets for **Publish ▸For Web**. Supports one level of subdirectories. Framesets are defined in the document type configuration file.

  The default frameset path is ***Arbortext-path*\framesets**. If there are any subdirectories of the **custom\framesets** directory, those subdirectories are prepended to the framesets path. Then the **custom\framesets** parent directory is prepended to the frameset path.

- **graphics** subdirectory

  Holds graphic files. Supports one level of subdirectories.

  The default graphics path is ***Arbortext-path*\graphics**. If there are any subdirectories of the **custom\graphics** directory, those subdirectories are prepended to the graphics path. Then the **custom\graphics** parent directory is prepended to the graphics path.

- **importexport** subdirectory

  Holds Arbortext Import/Export Import project files.

- **inputs** subdirectory

  Holds source files for custom macros, program fixes, or other customizations in a **custom.tmx**. Refer to Using **.tmx** files for more information. Document type and document **.tmx** files can be placed in the **custom\doctypes** directory.

  Also holds **.tex** files and source files for hyphenation exception and pattern rules in **.exc** and **.pat** files.

  The default source path is ***Arbortext-path*\inputs**. Then the ***Arbortext-path*\custom\inputs** directory is prepended to it.

- **lib** subdirectory

  Holds custom versions of the **.pdfcf** PDF configuration file. The default path for **.pdfcf** files is ***Arbortext-path*\lib**. Then the ***Arbortext-path*\custom\lib** directory is prepended to it. For more information on creating **.pdfcf** files, refer to the *Customizer's Guide*.

  In addition, the **lib** subdirectory can hold **.wcf** files for custom window classes. For more information on creating **.wcf** files for window classes, refer to the *Creating custom window class preferences files* in the Arbortext Editor help.

  The **lib** subdirectory can also hold custom versions of the following files:

  charent.cf

  charmap.cf

installprefs.acl

prted.pro

pubview.cf

pubview.fnt

tfmfont.cf

tfmscaling.cf

tfontsub.cf

wcharset.cf

wfontsub.cf

xcharset.cf

xfontsub.cf

You can specify more than one **charent.cf** file, as the effects are cumulative. Refer to the *Setting paths for new character set files* and *APTCUSTOM environment variable* topics in the online help for more information.

The **custom\lib** directory also has **locale*locale-name*** subdirectories. The default path is the appropriate locale subdirectory of ***Arbortext-path*\lib\locale**. The locale-specific subdirectory of the **custom\lib\locale** directory is prepended to the default locale path.

The **locale*locale-name*** can hold custom versions of the **.pdfcf** PDF configuration file. For more information on creating **.pdfcf** files, refer to the *Customizer's Guide*.

Each **locale*locale-name*** directory can hold custom versions of the following files:

charent.cf

installprefs.acl

ixlang.cf

pubview.cf

pubview.fnt

tfmfont.cf

tfmscaling.cf

tfontsub.cf

wcharset.cf

wfontsub.cf

xcharset.cf

xfontsub.cf

The **custom\lib** directory also has a subdirectory to hold native shared libraries for platform-specific use:

– **dll**

  Holds Windows dynamic link libraries, or DLL files (**.dll**).

  The path to this directory is prepended to the system **PATH** environment variable.

The **custom\lib** directory can have an **ixlang** subdirectory, which holds a custom **ixlang.cf** file and index mapping files like those found in ***Arbortext-path*\lib\ixlang**.

- **publishingrules** subdirectory

  Holds publishing rules **.prcf** files which contain definitions of publishing rules and publishing rule sets.

- **pubview** subdirectory

  Holds **pubview.cf** and **pubview.fnt** files.

  The default path is ***Arbortext-path*\pubview**. Then the ***Arbortext-path*\custom\pubview** directory is prepended to it.

- **scripts** subdirectory

  Holds **.acl** (Arbortext Command Language), **.vbs** (VBScript), and **.js** (JavaScript and JScript) files. Supports one level of subdirectories.

  The scripts in this directory can be called from scripts or applications in the **custom\init** directory, which is processed at startup time. Scripts placed here can be accessed using the **source** or **require** ACL commands. A customized menu item or button can call a script in **custom\scripts** when invoked.

  If there are any subdirectories of the **custom\scripts** directory, those subdirectories are prepended to the load path. Then the **custom\scripts** parent directory is prepended to the load path.

- **stylermodules** subdirectory

  Holds Arbortext Styler stylesheet modules. Any modules stored in this directory are automatically available to Arbortext Styler.

- **tagtemplates** subdirectory

  Holds **.tpl** files. You can also put custom tag templates you want associated with a particular document type into a **custom\doctypes\*doctype*\tagtemplates** directory or in the original location of the document type's ***doctype*\tagtemplates** directory.

  If the **APTTAGTPLDIR** environment variable is set, this path is prepended to it.

- **init** subdirectory

Holds **`.acl`**, **`.js`**, **`.class`**, and **`.vbs`** files.

The **`init`** subdirectory is processed last at startup time. All files of the supported application types are executed. No nested subdirectories of **`custom\init`** are supported. This directory is processed after the other ***`Arbortext-path`*** **`\custom`** subdirectories so that its scripts and applications can rely on paths already established during startup.

If you are putting custom applications on the Arbortext PE server, use the **`init`** directory for your custom **`.acl`**, **`.js`**, **`.class`** files.

In the startup process, the **`custom\init`** directory is processed after **`_main.acl`** but before **`arbortext.wcf`**. See the online help topic *Startup command files* for complete startup processing information.

The supported application types are:

– **`.acl`** (Arbortext Command Language) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

– **`.js`** (JavaScript or JScript) files

  Errors are reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP clients. You need to specify the JavaScript interpreter engine to use in processing **`.js`** files. Refer to Specifying the JavaScript Interpreter Engine on page 296 for more information.

– **`.class`** (Java) files

  Java **`.class`** files in this directory must be compiled Java classes that are not part of a named package. You can also put a **`.class`** file in **`custom\init`** that calls into a **`.jar`** file located in the **`custom\classes`** directory.

  The Java class must also implement a **public static void main(String[] args)** method, which will be called with an empty string array. If the **`.class`** file does not implement this method, an error is reported to Arbortext Editor or recorded by Arbortext Publishing Engine to be sent to its HTTP client.

– **`.vbs`** (VBScript) files

  Errors are reported to Arbortext Editor.

- **`editinit`** subdirectory

  Holds **`.acl`**, **`.js`**, **`.class`**, and **`.vbs`** files. Note that when you run Arbortext Editor with the **-c** option, any applications in this subdirectory are not executed at startup.

  All files of the supported application types are executed each time a non-ASCII document is opened for editing. Files in this directory act on a document opened in the Edit window. File in this directory act on a document opened using ACL when the `0x8000` flag is used with the **doc_open** function. File in this directory act on a document opened using AOM when the `OPEN_EDITINIT` flag is used with the **Application.openDocument** method.

The **`editinit`** subdirectory is processed before any document type command files, document type instance command files, and document command files.

The supported application types are:

– **`.acl`** (Arbortext Command Language) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

– **`.js`** (JavaScript or JScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

– **`.class`** (Java) files

  Java **`.class`** files in this directory must be compiled Java classes that are not part of a named package. The Java class must also implement a **public static void main(String[] args)** method, which is called with an empty string array. You can put a **`.class`** file in **`custom\init`** that calls into a **`.jar`** file located in the **`custom\classes`** directory. Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

– **`.vbs`** (VBScript) files

  Errors will be reported if the interface is running interactively, otherwise they will be suppressed.

Error Reporting for the custom\init Directory

Errors caused by mistakes in custom code in the ***`Arbortext-path`*****`\custom\init`** directory are reported with both the error message and the name of the initialization file causing the error. Note the following:

- If Arbortext Editor is not running interactively (batch mode), no errors are reported and the errors are not logged.
- Arbortext Publishing Engine records errors and reports them to its HTTP clients in an HTML error page.
- ACL, JavaScript, and Java class errors are reported to the Arbortext Editor interface or held by Arbortext Publishing Engine to be sent to HTTP clients making requests.

Additional Information

If you are using the AOM, refer to the documentation for **Application. getCustomDirectory**. Refer to the XUI section of the *Customizer's Guide* for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following **`set`** command options and environment variables affect custom path search lists. They are documented in the online help.

`set catalogpath`

`set composerpath`

`set dialogspath`

`set ditapath`

`set entitypath`

`set framesetpath`

`set graphicspath`

`set javaclasspath`

`set libpath`

`set loadpath`

`set pdfconfigfile`

`set tagtemplatepath`

`set userdictpath`

Using the Custom Directory for Custom Applications

The ***`Arbortext-path`*****`\custom`** subdirectory structure provides the means to implement custom applications. Where your application should be placed depends on the application purpose and programming language.

If you're implementing custom applications or scripts, the following information will assist you in determining the approach and location for your files:

- A custom Java program can be placed in **`custom\init`**, which supports a **`.class`** file that must implement a **public static void main (String[] args)** method. The method will be called at startup with no arguments (an empty `String` array). If an error occurs, it's reported interactively for Arbortext Editor or sent to the HTTP client for the Arbortext Publishing Engine.

  A custom Java program can also be placed in **`custom\classes`**, which supports **`.class`** or **`.jar`** files.

We recommend putting Java applications in the **custom\classes** directory and calling or initializing them from the **custom\init** directory.

Paths to **.jar** files in **custom\classes** are automatically prepended to the embedded Arbortext Editor Java class path. Then the path to **custom\classes** is prepended, putting it first in the search order.

- A custom JavaScript, JScript, VBScript, or ACL application can be placed in **custom\init** or in **custom\scripts**. If you place your scripts in the **custom\scripts** directory, you can call them from a script or scripts you place in **custom\init** (which is processed at startup). Any code that exists outside a function definition in a script from **custom\init** is executed at startup time. Errors are reported if running interactively, otherwise they're suppressed.

You can create a simple JavaScript example file called **simple_init.js**. The script should contain the following line:

```
Application.alert("Hello from JavaScript");
```

Put the **simple_init.js** file in ***Arbortext-path*****\custom\init**.

When the startup process loads scripts from **custom\init**, you will see a dialog box showing the `Hello from JavaScript` message.

Description of the Application Directory Structure

The ***Arbortext-path*****\application** subdirectory supports installing an application into the Arbortext Editor and Arbortext Publishing Engine install trees. Arbortext Editor and the Arbortext Publishing Engine automatically search for subdirectories of the **application** directory at startup.

Arbortext-path**\application** must contain a uniquely named subdirectory for each distributed application. Arbortext recommends using the naming pattern for a unique qualified Java class name:

com.company-name.application-name

Each unique subdirectory of the **application** directory must also contain an **application.xml** configuration file which describes various aspects of the application, such as its release version and supported versions of Arbortext products. At startup, Arbortext Editor and the Arbortext Publishing Engine search the **application** directory for any subdirectories containing an **application.xml** configuration file. The **application.xml** file contents provide the criteria to determine whether the application should be loaded. The **application** directory must be located using a file system; HTTP references are not supported.

Subdirectory Structure

A subdirectory of the **`application`** directory can be structured the same as the **`custom`** directory to take advantage of automatic Arbortext Editor and Arbortext Publishing Engine startup processes. For example, if the uniquely named directory contains **`graphics`** or **`entities`** directories, those directories are automatically added to the search paths constructed at startup.

An application path could be something like:

`application\`*`com.company-name.application-name`*

Refer to the Description of the custom directory structure for the names and descriptions of each supported subdirectory.

Note

When Arbortext Editor or the Arbortext Publishing Engine constructs search paths, subdirectories of the **`custom`** *directory take precedence over any corresponding subdirectories under the* **`application`** *directory. When search lists are constructed at startup, the first path in any search list will be the appropriate* **`custom`** *directory followed by any applicable directory under the* **`application`** *directory. For example, in constructing the graphics search path list at startup,* **`custom\graphics`** *would precede* **`application\com.arbortext.sample\graphics`***. An* **`application\graphics`** *directory with no* **`application.xml`** *file will be ignored during startup.*

When implementing a custom application using the **`application`** directory structure, you can add supplemental directories as needed to support your application. However, your application code must be aware of these directories and how to use them.

Application Startup File

The ***`Arbortext-path`*`\doctypes\appcfg\application.xml`** file provides a basic template for defining information about the custom application. You can make a copy of **`doctypes\appcfg\application.xml`** to use as a template to create the file that will eventually be distributed with the application. The **`application.xml`** file must be placed in the application's top level directory, for example:

`Arbortext-path``\application\com.`***`company.application-package-name`***`\application.xml`

In the template **`application.xml`** file, you can specify a list of elements that describe the application. If the custom application determines its criteria is not met and the application is not to be loaded, then these values are ignored. The base element for the file is the **ApplicationConfiguration** element. This element has a required attribute called *installType* that determines the type of Arbortext Editor installation for which this application is supported. The default value is `any` meaning the application is supported in both the full and compact installations of Arbortext Editor. The other supported value is `full` meaning the application is only supported in the full installation of Arbortext Editor.

The following other elements are supported in the **`application.xml`** file:

- **Name** (required)
- **Description**
- **LicenseNumber** is only for an application distributed by Arbortext
- **Version** (required)
- **Date**
- **Copyright**
- **Vendor**
- **RequiredApplications** is for other applications that are required for this application to run correctly. You must enter the qualified name for the application in the *qualifiedName* attribute and a human-readable name in the *name* attribute.
- **SupportedProducts**

  A **Product** element has attributes for specifying the name (required), minimum version (required), and maximum version of the Arbortext product that supports the custom application or application. The **Product** specification helps the launching Arbortext product determine whether it should load this custom application by matching criteria specified in this section.

  The name must be one or more of the following:

  – Arbortext Editor

  – Arbortext Publishing Engine

  – Arbortext Architect

  – Arbortext Editor with Styler

  The version must follow the convention used by Arbortext products, such as 5.2, 5.2 M040, or 5.3.
- **SupportedPlatforms**

  The section is reserved for future use. Windows is currently the only supported platform.
- **GlobalParameters**

  Parameter contains **ParameterName** and **ParameterValue** elements for specifying any global variables that the application may need when it's launched.

Related Topics

If you are using ACL, refer to the following ACL function descriptions:

- **application_name** function
- **get_custom_dir** function
- **get_custom_property** function
- **get_user_property** function
- **set_user_property** function

If you are using the AOM, refer to the documentation for **Application.getCustomDirectory**. Refer to the XUI section of the *Customizer's Guide* for information on extending the Arbortext Editor **Preferences** dialog box for your custom application.

The following attributes from the **Application** interface are also useful:

- `haveWindows`
- `initDone`
- `isE3`
- `customProperties`
- `userProperties`
- `name`

Using the Application Directory for Custom Applications

The ***`Arbortext-path`*****`\application`** subdirectory provides the means to implement a custom application that uses a special configuration file to determine whether it should be loaded at startup. The **`application`** directory uses the same principles of structure as the **`custom`** directory.

The ***`Arbortext-path`*****`\application`** directory is processed at startup. If you add a custom application after startup, you must exit and restart Arbortext Editor or stop and restart the Arbortext Publishing Engine to have it recognized. You also have the option to issue the **f=init** function to re-initialize the Arbortext PE sub-processes. Refer to *Configuration Guide for Arbortext Publishing Engine* for more information.

Rules for using the **`application`** directory are:

- Your custom application must be contained in a uniquely named subdirectory of the **`application`** directory.
- You must have an **`application.xml`** configuration file in the uniquely named subdirectory that sets the conditions for loading the application.
- The same set of subdirectories supported by the **`custom`** directory are supported for the uniquely named subdirectory of the **`application`** directory. At startup,

the supported directories are automatically detected and used in constructing search paths.

- Any other subdirectory of the **`application`** directory will be ignored at startup. For example, an **`application\graphics`** subdirectory with no **`application.xml`** file will be ignored during startup.

Arbortext has developed proprietary custom applications that are deployed using the **`application`** subdirectory structure. A uniquely named subdirectory contains all the necessary components to run an application within Arbortext Editor as well as the Arbortext Publishing Engine.

The following information will help determine an approach for a custom application.

- You can have additional subdirectories for your custom application. You are not limited to the subdirectories supported by the **`custom`** directory. However, these additional directories are not automatically recognized during the startup process.
- Processing each unique application's subdirectories follows the same rules for processing **`custom`** subdirectories. Recall that the application's subdirectories come after the **`custom`** subdirectories in constructing any applicable search paths for the session.
- If you decide not to use a particular supported subdirectory, you can improve performance by omitting the directory to reduce the length of a search path that would contain it.
- You can use the **APTAPPLICATION** environment variable to set the path to one or more **`application`** directories.
- An application should not write data to its own application directory. An application user may not have write permission access to this application directory, for example, any **`C:\Program Files`** directories on Windows (the location where Arbortext Editor and the Arbortext Publishing Engine are typically installed).

Deploying Zipped Customizations

You can deploy not only **`custom`** directories, but also **`application`** and content management system adapters directories in a compressed zip file. Using a zip file to distribute your customizations has the following advantages:

- You can host your customizations on a web server.

  In this case, use the HTTP or HTTPS URL to the zip file as the value for the **APTCUSTOM** environment variable.
- Your customizations will be available to users when they cannot access your network.

If you use a shared network folder to host your customizations, users do not have access to those customizations when the network is unavailable. If you use a zip file to distribute your customizations, Arbortext Editor unzips those customizations to a directory in the Arbortext Editor cache directory (**`.aptcache\zc`**). At start up, Arbortext Editor checks to see whether the zip file has been updated. If it has, Arbortext Editor downloads and uncompresses the updated customizations. If not, Arbortext Editor continues to use the customizations stored in the local cache. If the network is unavailable to a user, your customizations are still available to that user in the local cache. Note that the user must also have a fixed Arbortext Editor license on their system to work away from the network.

- Network traffic might be reduced.

  Since the zip file containing your customizations is only downloaded once over the network, and then only if it has been updated, traffic on your network might be reduced. If you store your unzipped customizations in a shared network folder, Arbortext Editor might have to access that folder several times over the course of a session.
- Customizations stored in a compressed zip file are harder to change accidentally than customizations stored in a directory structure.

Note that you cannot use a zip file to distribute a customized **`installprefs.acl`** in the **`custom\lib`** directory. You can use the **APTINSTALLPREFS** environment variable to specify the location of a custom **`installprefs.acl`** file.

Note also that you cannot include the following font configuration files in the **`lib`** subdirectory of a zipped **`custom`** directory:

- **`charent.cf`**
- **`wcharent.cf`**
- **`wfontsub.cf`**
- **`charmap.cf`**

These files are processed before a zipped **`custom`** directory when Arbortext Editor starts up, so the files cannot be processed when deployed in that way.

Specifying the JavaScript Interpreter Engine

Both JavaScript and JScript files have a **`.js`** file extension. By default, Arbortext Editor and the Arbortext Publishing Engine interpret **`.js`** files as Rhino JavaScript files. You should specify the JavaScript interpreter for a JavaScript or JScript **`.js`** file. This is especially important if you have **`.js`** files of both types.

We recommend adding a comment line to your script that specifies either the Rhino JavaScript engine (the default) or the Microsoft JScript engine as shown in the following examples. The first line of your **`.js`** file must be a comment starting with `//`.

To specify the Rhino JavaScript interpreter:

```
// type="text/javascript"
```

To specify the Microsoft JScript interpreter:

```
// type="application/jscript"
```

The specification can be enclosed in a script tag. Both of the following examples are a valid specification for JScript:

```
// <script type="application/jscript">

// type="application/jscript"
```

You can also specify the JavaScript interpreter using the ACL **`set javascriptinterpreter`** command. You can specify it in an ACL file placed in the ***`Arbortext-path`*** **`\custom\init`** directory, where it will be processed at startup. For information on setting the interpreter using ACL, see the online help topic for **`set javascriptinterpreter`**.

Index