Flat File Schema Developer's Guide

Innovation Release

Version 10.4

April 2019

WEBMETHODS

This document applies to webMethods Integration Server and Software AG Designer Version 10.4 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IS-FFS-DG-104-20190415

Table of Contents

About this Guide

The webMethods Integration Server provides the WmFlatFile package to enable you to exchange and process flat files using its built–in services. This guide contains information about how Integration Server processes flat files and how to create services that send and receive flat files.

To use this guide effectively, you should be familiar with the basic concepts described in *webMethods Integration Server Administrator's Guide* and *webMethods Service Development Help*.

Note: This guide describes features and functionality that may or may not be available with your licensed version of webMethods Integration Server. For information about the licensed components for your installation, see the **Settings > License** page in the webMethods Integration Server Administrator.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service* , APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.

Convention	Description
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at "http://documentation.softwareag.com". The site requires credentials for Software AG's Product Support site Empower. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to "empower@softwareag.com" with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at "https://empower.softwareag.com/".

You can find product information on the Software AG Empower Product Support website at "https://empower.softwareag.com".

To submit feature/enhancement requests, get information about product availability, and download products, go to "Products".

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the "Knowledge Center".

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at "https://empower.softwareag.com/public_directory.asp" and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at "http://techcommunity.softwareag.com". You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.

- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Concepts

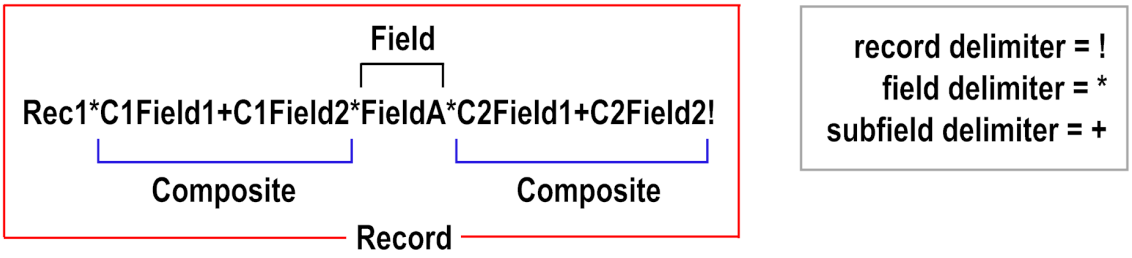
What is a Flat File?

Flat files present complex hierarchical structural data in a record–based storage format. Unlike XML, flat files do not embed structural data (metadata) within the data. The data in the flat file has been "flattened" by removing the hierarchical relationship between records, leaving the records intact as a single logical record of application data.

All flat files consist of a list of records containing fields and composites:

- *Fields* are atomic pieces of data (for example, ID and ID qualifier, Date and time).
- *Composites* contain multiple fields (for example, ID and ID qualifier, Date and time). The fields within a composite are referred to as *subfields*.
- *Records* (also known as segments) are sequences of fields and/or composites.

For example, the following flat file data and its list of delimiters enables you to see how *elements* (records, composites, and fields) within a flat file can be identified.



To communicate using flat files, you must create a *flat file schema* that contains a particular flat file's structural information, including how to identify records and separate those records into fields.

The flat file schema enables receivers of flat file documents to parse and validate inbound flat files for use by their back–end systems. The flat file schema also enables senders of documents to convert outbound documents to flat file format. Once converted to flat file format, the files can be delivered to another back–end system.

What is a Flat File Schema?

A flat file schema is the blueprint that contains the instructions for parsing or creating a flat file and is created as a namespace element in the Integration Server. This blueprint details the structure of the document, including delimiters, records, and repeated record structures. A flat file schema also acts as the model against which you can validate an inbound flat file.

A flat file schema consists of hierarchical *elements* that represent each record, field, and subfield in a flat file. Each element is a *record*, *composite*, or *field*. Additionally, each element is either a *reference* or a *definition*. A definition is an element that is defined in

the flat file schema. A reference is an element that is defined in a flat file dictionary and referenced from the flat file schema. You configure each element with the necessary constraints.

Flat file schemas contain information about the constraints at the document, record, and field levels of flat files. A flat file schema can define three types of constraints:

- **Structural constraints** describe the sequence of records, composites, and fields in a flat file. For example, a flat file schema might describe an address record that consists of a Street Address field, followed by a City field, State field, and ZIP Code field. You define structural constraints for each parent element (an element that contains other records, composites, or fields) in the flat file schema.
- **Content type constraints** describe the possible values or minimum/maximum values for elements in a flat file. For example, the State field might have a content constraint specifying that its value must be exactly two digits. These types of constraints are referred to as *validators*.
- **Conditional constraints** describe the valid combinations of fields and subfields allowed in records and composites. For example, you can use the Required (R) conditional validator to require that at least one of the specified fields be present.

When validating an inbound flat file, Integration Server compares the records, composites, and fields in a flat file with the constraints described for those records and fields in the flat file schema. An inbound flat file is considered valid when it complies with the constraints outlined in its related flat file schema. For more information about data validation, see the section *Validation Errors*.

How Is a Flat File Schema Used to Parse Records?

To parse a flat file, use the pub.flatFile:convertToValues service in the WmFlatFile package. This service uses a flat file schema to parse flat files inbound to the Integration Server. The convertToValues service uses a *record parser* (Delimited, Fixed Length, or Variable Length) to parse the records in a flat file. For more information about the pub.flatFile:convertToValues service, see the *webMethods Integration Server Built-In Services Reference*.

After each record in a flat file has been parsed, each record must be identified. This is done using the *record identifier*. Identifying the record provides the definition of that record, as defined in the flat file schema using the flat file schema editor. The field values then are pulled from the record using the *extractors* defined in the flat file schema.

If the record cannot be identified (known as *undefined data*) and you have selected a *default record*, the field values are extracted from the record using the definition of the default record. If the record cannot be identified and you have not selected a default record, the field values are not extracted from the record. You can also format flat file data to meet the requirements of your back–end system or application using the **Format Service** property in the **Flat File Structure** tab of the flat file schema editor. For information about how to specify field format services, see the section *Format Services*.

Finally, the parsed record is placed in the output data based on the structure defined in the flat file schema. This process is repeated until all records in the flat file have been parsed.

Record Parsers

A record parser breaks a flat file into individual records. When defining a flat file schema, you can specify one of the following record parsers:

- **Delimited Record Parser.** This parser expects a specific delimiter to indicate the end of a record. For a record delimiter, you can specify:
  - A character (for example, !) or character representation (for example, `\r\n` for carriage return)
  - Hexadecimal value (for example, `0X09`)
  - Octal value (for example, `009`)
  - Unicode characters (for example, `\u`*XXXX*, where *XXXX* represents the unicode value of the character)

  The occurrence of the record delimiter signals the end of one record and the beginning of the next record.
- **Fixed Length Record Parser.** This parser splits a file into records of the same pre-specified length.

  Note: This parser measures the lengths and positions of records in terms of *bytes*, rather than *characters*. Prior to 6.5, Service Pack 2, the parser measured lengths and positions in terms of characters. This change does *not* affect users who currently parse/compose flat files using single-byte encodings because one byte equals one character. In this case, there is no functional difference between specifying bytes or characters.

  The parser supports both single-byte and multi-byte encodings. Multi-byte encoded files must run on JVM version 1.4 or later. (The behavior of delimited fields and records remain the same.) For details, see *Flat File Byte Count Parser*.
- **Variable Length Record Parser.** This parser expects each record to be preceded by two bytes that indicate the length of the record. Each record may be a different length.
- **EDI Document Type Record Parser.** This parser is used only for EDI flat files and provides additional functionality needed to properly parse EDI documents. Documents using this record parser should be created using the webMethods Module for EDI, not the WmFlatFile package. For more information, see the *webMethods Module for EDI Installation and User's Guide*.

Record Identifiers

A record identifier looks at a record and extracts an identifier out of the data. Integration Server uses that identifier to connect the record definition in a flat file schema with a particular record in the flat file. The name of the record definition must match the value obtained by the record identifier. When creating a flat file schema, you can choose from one of two methods of record identification:

- **Starts at position** record identifiers compare the value that occurs in the record, at the specified offset, to all the record names defined in the flat file schema. Note that the **Starts at position** identifier cannot distinguish between all types of record names. For example, if you name records "Rec1" and "Rec," some instances of "Rec1" may be identified as "Rec," because "Rec1" begins with "Rec."
- **Nth Field** record identifiers use the value of the specified field as the record identifier. These identifiers count from zero (0). For example, if 2 is specified, the third field is used as the record identifier.

Integration Server processes the longer record identifiers first and then the shorter record identifiers.

Integration Server does not perform any kind of normalization on input provided in the flat file schema or when comparing or processing values retrieved from a file. You must enter carefully the exact Unicode character values you want to search for in an instance of the flat file you are describing. For example, you should differentiate wide (sometimes called multi–byte or *zenkaku*) characters from their narrow (or single–byte) equivalents when processing Asian characters. Another example is the use of Unicode combining and pre–composed sequences. In all cases, fields are matched using a strict binary comparison of Unicode character values in the separate fields.

Note: Exercise care when selecting the encoding of the file being processed. Some encodings are very similar to one another. For example, the Shift–JIS and MS932 encodings commonly used with Japanese language text are very similar, but they map some characters differently. This can result in Integration Server not finding a match where you otherwise would expect one to exist.

Extractors

Integration Server uses extractors take data out of a parsed record and place it in the output of the pub.flatFile:convertToValues service. If extractors are not defined in the flat file schema, the parser returns a series of records that contain no fields or composites. For information about the pub.flatFile:convertToValues service, see *webMethods Integration Server Built-In Services Reference*.

Integration Server extracts fields and composites from a record based on the position of the field delimiter. From the beginning of the record to the first field delimiter is the first field in the record. From the first field delimiter to the second field delimiter is the

second field in the record, and so on. Everything from the last field delimiter to the record delimiter (the end of the record) is considered part of the final field.

Note: Although fixed length and variable length record parsers do not use record delimiters, they can have field delimiters.

Integration Server can also extract fields from a record based on a substring of the original record starting at a particular byte count and ending at a particular byte count. You specify the start of the field by specifying the number of bytes from the beginning of the record to the start of the field and specifying the number of bytes from the beginning of the record to the end of the field.

If a value is a composite, it is simply a field that can be further parsed into more fields. Instead of a field delimiter, a subfield delimiter is used to separate a composite into fields. Fields are extracted from a composite based on either the position of the subfield delimiter in the composite or on a substring of the composite. Keep in mind that all positions used in extracting fields from a composite are referenced from the beginning of the composite, not the start of the record.

Undefined Data

In some cases, the data resulting from the pub.flatFile:convertToValues service might contain records that were not recognized, known as *undefined data*. This could be a result of the complexity of the flat file, or of the lack of detail in the flat file schema (such as no record identifier). When the pub.flatFile:convertToValues service processes an unrecognized record, it puts a placeholder named *unDefData* in the resulting IS document (IData object) and stores the record as a string in the pipeline.

You can select a default record when creating a flat file schema. Any record that cannot be recognized will be parsed using this default record. If a default record is not selected, unrecognized records will be placed into the output IS document in the *unDefData* field, which might produce errors.

Note: If the file is encoded using a multi-byte encoding, and if you use a fixed length or variable length parser, the service puts *two* placeholders into the pipeline: *unDefData* and *unDefBytes* .

For some types of documents, undefined data should not be allowed in the document and if encountered should generate validation errors in the output. In other cases, undefined data is expected and should not generate validation errors. In still other cases, all unidentified records should be parsed using a default record definition, as described in *Default Records*.

You can choose whether you want errors to be generated for a flat file schema when undefined data is encountered. For more information about configuring a flat file schema to allow undefined data, see *webMethods Service Development Help*.

To work with undefined data produced by processing a flat file, you must:

1. Configure the flat file schema and the records in the flat file schema to allow undefined data.
2. In the service that processes the flat file, parse the undefined data in each record based on the document structure.
3. In the service that processes the flat file, write logic to handle the parsed data, such as logic to map the parsed data to another service or perform error handling.

Default Records

If the pub.flatFile:convertToValues service cannot recognize a record (for example, the record does not have a record identifier), the record will be treated as undefined data. To avoid this, you can specify a default record definition. The pub.flatFile:convertToValues service uses the default record definition to parse all unrecognized records of a particular type. In fact, if your flat file schema does not contain any record identifiers, you *must* specify a default record.

Note: A default record can only be defined in a flat file dictionary.

For example, if you have a CSV (comma separated values) flat file that simply lists the name, address, and phone number of a group of customers, none of the records would have a record identifier:

```
John Doe, 123 Main St Washington DC, 888-555-1111;
Jane Smith, 321 Oak Dr Memphis TN, 800-555-2222;
```

If you specified a default record definition to parse this particular type of record, the file can be properly parsed and converted to an IS document (IData object) named *recordWithNoID* . By default, each *recordWithNoID* document appears as a child of the document above it, in an array. To modify this default behavior, see *Customizing the Flat File Configuration Settings*.

If a default record has not been specified, the record will be treated as undefined data, as described in *Undefined Data*.

What is a Flat File Dictionary?

A flat file schema can contain either record definitions or references to record definitions that are stored elsewhere in the namespace in a *flat file dictionary*. A flat file dictionary is simply a repository for elements that you reference from flat file schemas. This allows you to create record definitions in a dictionary that can be used across multiple flat file schemas. Reusing record definitions reduces the amount of memory consumed by a flat file schema.

Flat file dictionaries are created as namespace elements in the Integration Server and contain definitions of records, composites, and fields. When you change a definition in a flat file dictionary that is referenced in multiple flat file schemas, the element definition is updated automatically in all of the flat file schemas.

Note: You can reference a flat file dictionary definition in any flat file schema regardless of whether the dictionary and schema are located in the same package.

When creating an element definition in a flat file dictionary, you specify only certain properties. You then specify the remaining properties in the instance of the element definition in a particular flat file schema.

When Should I Create a Flat File Dictionary?

The decision to define records in a flat file dictionary versus in a flat file schema depends on the type of flat files that you intend to parse. The Electronic Document Interchange (EDI) ANSI X12 standard defines a large set of document structures that reuse the same record, field, and composite definitions many times. Defining these records, fields, and composites in a dictionary allows for them to be reused throughout the entire set of EDI ANSI X12 document flat file schemas. Reusing definitions reduces the amount of memory consumed by Integration Server.

EDI ANSI X12 also has different versions of these documents (for example, 4010). Each version of the document set should have its own dictionary. In this way, you can be certain that changes to a record, field, or composite between versions are maintained.

A more complex scenario would involve multiple families of documents and multiple versions of those families. An example of this is EDI ANSI X12 and UN/EDIFACT documents. One dictionary should be created for each version of EDI ANSI X12 documents and one dictionary should be created for each version of EDI UN/EDIFACT documents. A separate dictionary would not be required for each flat file schema in the same version. All flat file schemas in one version of the same family should use the same dictionary.

In a scenario in which you intend to parse only one flat file, or flat files that do not share record, composite, or field definitions, you can define these elements directly in the flat file schema. This allows for the entire document to be edited in a single view, without referencing a flat file dictionary.

If a clear choice does not exist between these two scenarios, the best approach is to create the definitions in the flat file dictionary and reference them in a flat file schema. The definitions then can be reused at a later time.

2 Processing Flat Files Overview

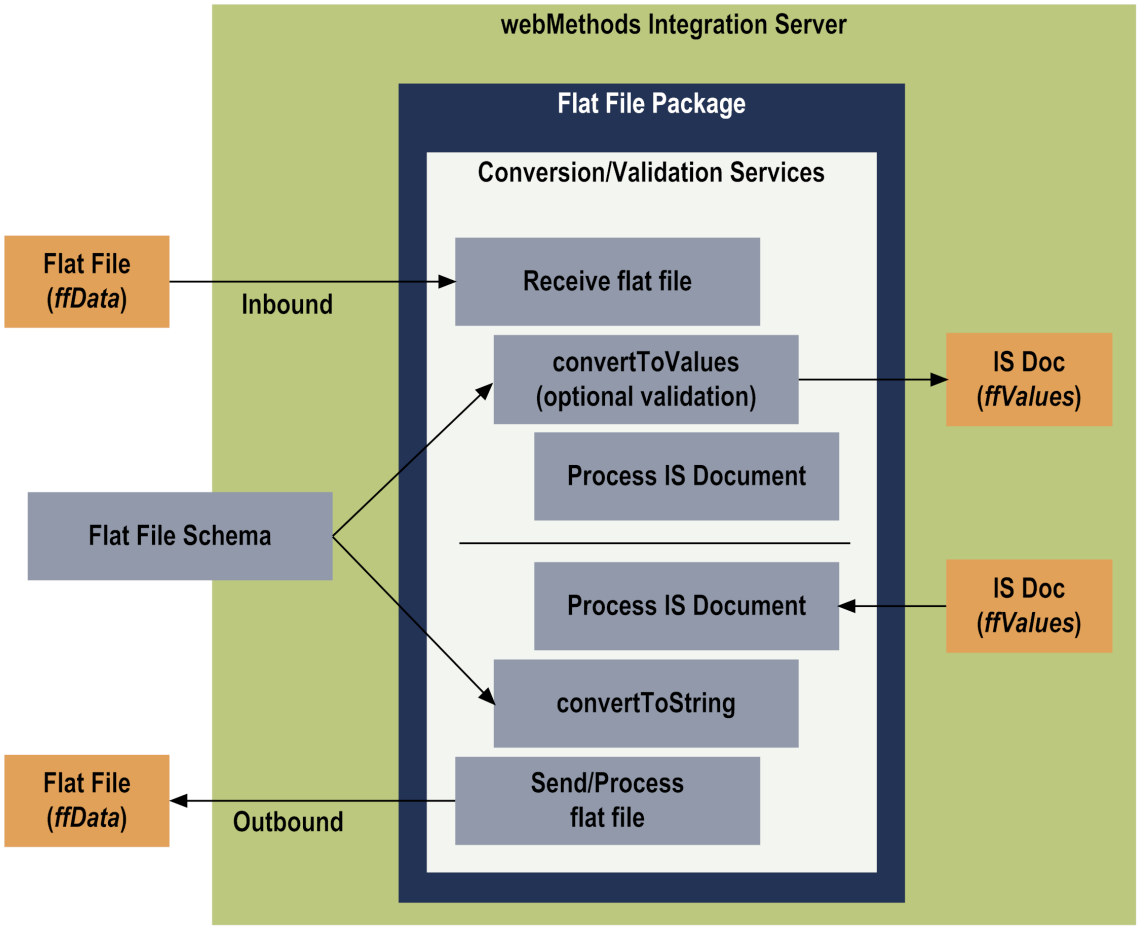
Overview of Processing Flat Files

Using the services in the webMethodsWmFlatFile package, you can exchange all types of flat files and process certain types of flat files. You create flat file schemas to convert and validate inbound flat files and to create outbound flat files.

The following diagram summarizes the services that the WmFlatFile package provides to process flat files: pub.flatFile:convertToValues and pub.flatFile:convertToString. For descriptions, see the table that follows the diagram below.

Note: If you are also using Trading Networks, see the sections about flat file TN document types in *webMethods Trading Networks User's Guide* and *webMethods Trading Networks Administrator's Guide* for information about how Trading Networks processes flat files.

Processing Flat Files



Item	Description
Flat File (*ffData*)	Flat file input with type of String, InputStream, or ByteArray (received by the Integration Server via File Polling, HTTP, FTP, etc. For more information, see "Sending and Receiving Flat Files" on page 45.)
Conversion/ Validation Services	WmFlatFile package provides services (pub.flatFile:convertToValues and pub.flatFile:convertToString) that parse, validate, and convert inbound flat files to IS documents (IData objects), and construct outbound documents from IS documents. For more information about these services, see *webMethods Integration Server Built-In Services Reference*.
Flat File Schema	Flat File schema containing the structure of the flat file.
IS Doc (*ffValues*)	An IS document (IData object) represents the structure and values of your flat file for eventual mapping or other processing. All inbound documents must be converted to IS documents before being processed further.

Formatting Inbound and Outbound Data

You can format flat file data to meet the requirements of your back–end system or application. The **Format Service** property in the flat file schema editor enables you to specify the service you want to invoke for a particular field when convertToValues or convertToString is called. For example, when using the convertToValues service to convert a date field in a flat file into an IS document, you can specify that a particular format service be invoked for that field during parsing. To alter the format of the date field, you might invoke a service that changes the format of the field from YYYYMMDD to MMDDYYYY.

For information about how to specify field format services, see "Format Services" on page 37.

Processing Inbound Flat Files

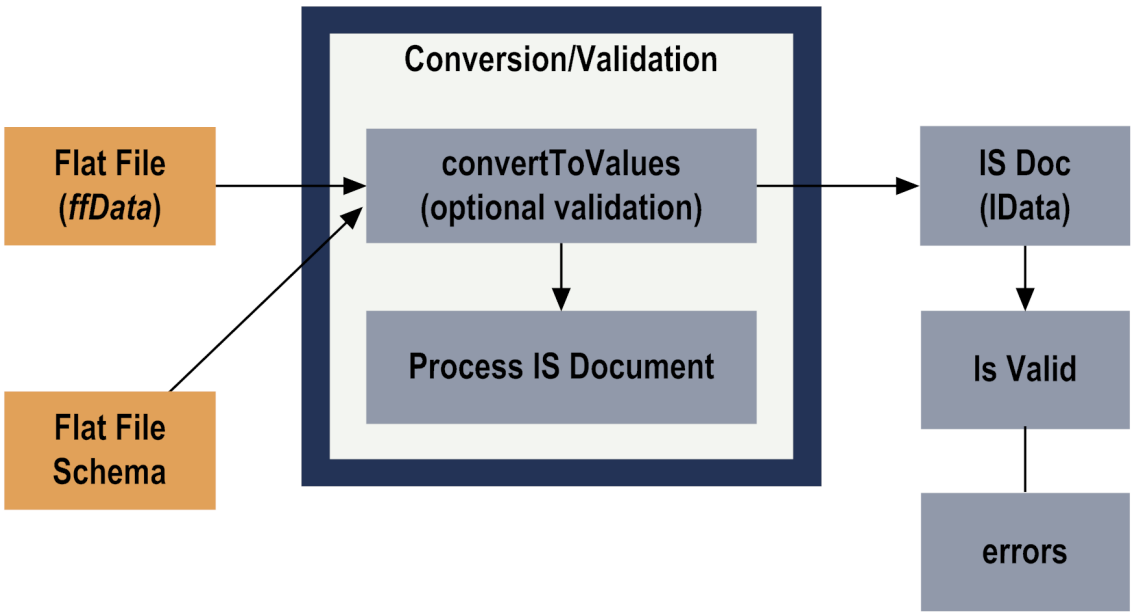
To process an inbound flat file, use the pub.flatFile:convertToValues service. This service uses a flat file schema to parse flat files inbound to an Integration Server. The input of this service is a flat file and the name of a flat file schema, and the output of this service is an IS document (IData object). The output IS document is created based on the structure defined in the flat file schema. You can also validate the flat file against the flat file schema. During validation, the constraints placed on the records and fields

cause the flat file schema to generate errors as output. For more information about the pub.flatFile:convertToValues service, see *webMethods Integration Server Built-In Services Reference*.

Note: Validation is turned off by default. To enable validation, you must set the *validate* variable of the convertToValues service to `true`.

Invoke the pub.flatFile:convertToValues service as part of customized flow services. You can then use Integration Server services to process the resulting data as necessary, (for example, map data from one format to another). The general process for inbound flat files is as follows:

Inbound Flat File Process



The flat file schema provides the pub.flatFile:convertToValues service with the information necessary to detect the boundaries between records and fields. After a record boundary is detected, the service matches the record in the flat file with a record defined in the flat file schema using the record identifier.

Note: When parsing a file, the flat file parser ignores any data that appears before the first valid record in the file.

Processing a flat file consists of the following basic steps:

Step 1 **Create a flat file schema.** Before you start working with an inbound flat file, you must create a flat file schema to be used for conversion and validation of the document. Use the Software AG Designer to create the flat file schemas used to parse, convert, and

validate inbound flat files. For steps to create flat file schemas, see *webMethods Service Development Help*.

Step 2 **Integration Server receives the flat file.** Integration Server provides a number of ways to send and receive flat files. For more information, see "Sending and Receiving Flat Files" on page 45.

Step 3 **Parse, convert, and validate flat file.** To parse a flat file using a flat file schema and to convert the data into IS documents (IData objects), you call the pub.flatFile:convertToValues service.

During flat file parsing, you have the option of validating the flat file against the document, record, and/or field constraints you defined in a flat file schema.

To validate a flat file against the flat file schema used to parse the file, you must set the *validate* variable of the pub.flatFile:convertToValues service to `true`, which returns errors describing how the given *ffData* violates the constraints described in the flat file schema. For a list of validation error codes and messages, see "Validation Errors" on page 55.

For more information about the pub.flatFile:convertToValues service, see *webMethods Integration Server Built-In Services Reference*.

Step 4 **Process the IS document produced by the pub.flatFile:convertToValues service.** In many cases it is necessary to map the output of the *ffValues* document produced by the pub.flatFile:convertToValues service to another format, such as your back-end or user-defined format.

To assist with mapping the data, create an IS document type from your flat file schema. You can add a document reference variable that references the resulting IS document type to the pipeline. You can then map the *ffValues* document to the document reference variable.

Handling Large Flat Files

By default, Integration Server processes all flat files in the same manner, regardless of their size. Integration Server receives a flat file and keeps the file content in memory during processing. However, if you receive large files, Integration Server can encounter problems when working with these files because the system does not have enough memory to hold the entire parsed file.

If some or all of the flat files that you process encounter problems because of memory constraints, you can set the *iterator* variable in the pub.flatFile:convertToValues service to `true` to process top level records (children of the document root) in the flat file schema one at a time. After all child records of the top level record are parsed, the

pub.flatFile:convertToValues service returns and the *iterator* moves to the top level of the next record in the schema, until all records are parsed. This parsing should be done in a flow service using a REPEAT step where each time the pub.flatFile:convertToValues service returns, the results are mapped and dropped from the pipeline to conserve memory. If the results were kept in the pipeline, out–of–memory errors might occur.

The pub.flatFile:converToValues service generates an output object (*ffIterator* variable) that encapsulates and keeps track of the input records during processing. When all input data has been parsed, this object becomes null. When the *ffIterator* variable is null, you should use an EXIT step to exit from the REPEAT step and discontinue processing.

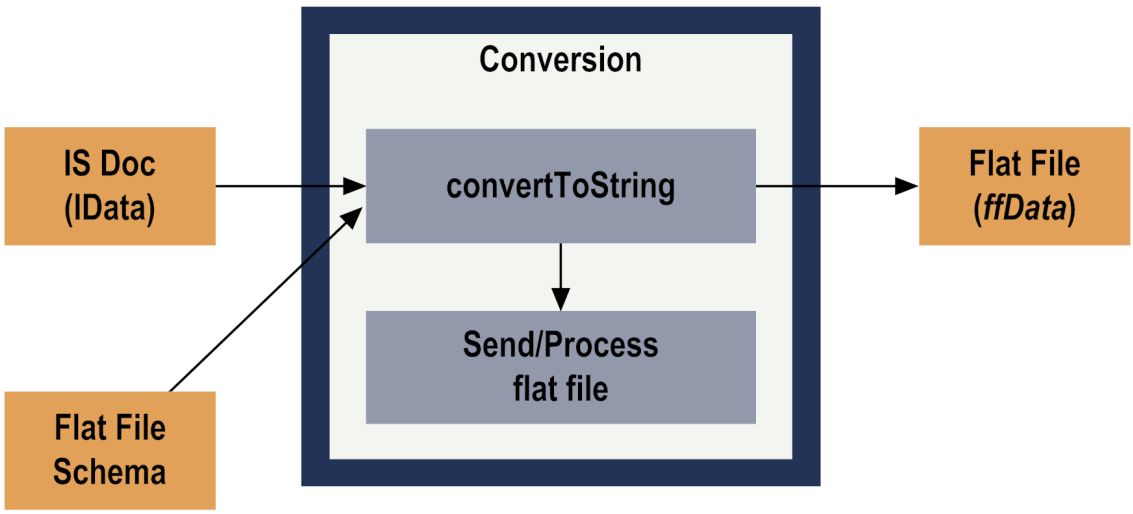
Processing Outbound Flat Files

To process an outbound flat file, use the pub.flatFile:convertToString service. This service uses a flat file schema to create a flat file outbound from Integration Server. The input of this service is an IS document (IData object) and a flat file schema, and the output of this service is a flat file. For more information about the pub.flatFile:convertToString service, see *webMethods Integration Server Built-In Services Reference* .

Note: The flat file resulting from pub.flatFile:convertToString will be only as complete as the information translated from the original flat file to the IS document. If you specified only certain parts of the flat file to translate to the IS document (via the flat file schema), only those parts will be translated back to the flat file.

The general process for outbound documents is as follows:

Outbound Flat File Process



To create and send an outbound flat file, use the following basic steps:

Step 1 **Create a flat file schema to be used for conversion of the object.** You use Software AG Designer to create the flat file schemas used to

create outbound flat files. For steps to create flat file schemas, see *webMethods Service Development Help*.

Step 2 **Convert the IS document to a flat file.** Invoke the pub.flatfile:convertToString service to convert an IS document (IData object) to an outbound flat file using the flat file schema you created. This service enables you to specify delimiters to be used as record, field, and subfield separators as well as optionally specify an *outputFileName* so that the output also will be written to a file.

> **Note:** You can build the document (IData object) to be converted to a flat file using IS document type from the flat file schema. Add a document reference variable that points to the IS document type in the pipeline. Build the document by using the pipeline to map data into the document reference variable. Then, in the pipeline for the pub.flatFile:convertToString service, map the document reference variable to the *ffValues* input variable.

Step 3 **Send the flat file.** You can send a flat file using many methods, including via HTTP, SMTP, and FTP. For more information, see "Sending and Receiving Flat Files" on page 45.

3 Working with Elements in Flat File Schemas and Dictionaries

Overview

This chapter explains the following flat file schema and dictionary elements and provides examples and points to keep in mind when creating these elements:

- Floating records
- Extractors
- Validators
- Format services

Floating Records

You can use the **Floating Record** property to designate any single record of a given flat file schema to be a floating record. By designating a floating record, you enable that record to appear in any position within a flat file without causing a parsing validation error.

If you do not use this property, validation errors will occur if the record structure of an inbound document does not match the record structure defined in its flat file schema. For example, consider the following EDI document and its flat file schema (where all records are mandatory except the record NTE):

Flat file schema definition	Input EDI document	Output IS document

`Rec1`
`Rec2`
`Rec3`
 	
NTE
	`Rec1`
	`NTE`
	`Rec2`
	
	Rec3
		`Rec1`
		`NTE`
		
		The parser generates a validation error because it expects NTE to be the last record, not the second record. In addition, it ignores Rec2 and Rec3.

To solve this problem, you would designate NTE to be a floating record so it may appear anywhere in the EDI document without causing a validation error. (The position in which it appears in the parsed record is explained in the following sections.) The parser creates a container for NTE records, and places each NTE record that appears in an EDI document into this container at run time. If a flat file schema contains multiple instances of the designated floating record at multiple record levels, the system creates a container for each record level. This property appears in the Properties view of the flat file schema editor.

This problem can also occur while converting outbound IS documents (IData objects) into string documents. The parser supports floating records for both

inbound and outbound parsing (i.e., while executing the pub.flatFile:convertToValues and pub.flatFile:convertToString services, respectively).

For examples, see "Examples: Parsing Inbound Floating Records" on page 29 and "Examples: Parsing Outbound Floating Records" on page 31.

Examples: Parsing Inbound Floating Records

When the pub.flatFile:convertToValues service receives a document that includes floating records, the parser handles them as shown in the following examples. Assume that the **Floating Record** property is set to the record NTE.

Example 1

Flat file schema definition	Input EDI document	Output IS document

Rec1
NTE[] Max Occurs 99
Rec2
  Rec3
  Rec4
Rec5
	Rec1
	Rec2
	Rec3
	Rec4
	NTE (1)
	NTE (2)
	NTE (3)
	Rec5
		Rec1
		NTE []
		  NTE (1)
		  NTE (2)
		  NTE (3)
		Rec2
		  Rec3
		  Rec4
		Rec5

In accordance with the schema definition, the parser creates an NTE array as the first record, and places `NTE (1)`, `NTE (2)`, and `NTE (3)` into it. A validation error is generated if more than 99 NTE records appear in the document. NTE records are not allowed prior to the `Rec1` record.

Example 2

Flat file schema definition	Input EDI document	Output IS document

Rec1
NTE [] Max Occurs 99
Rec2 [] Max Occurs 99
  NTE [] Max Occurs 99
  Rec3
  Rec4
Rec5
	Rec1
	NTE (1)
	Rec2
	Rec3
	NTE (2)
	Rec4
	NTE (3)
	Rec2
	NTE (4)
	Rec5
	NTE (5)
		Rec1
		NTE []
		  NTE (1)
		  NTE (5)
		Rec2 []
		  Rec2
		  NTE []
		  NTE (2)
		  NTE (3)
		  Rec3
		  Rec4
		  Rec2
		  NTE []
		  NTE (4)
		Rec5

How Example 2 was Parsed

1. The first six records of the EDI document were parsed as follows:

Input EDI document	Output IS document

Rec1
NTE (1)
Rec2
Rec3
NTE (2)
Rec4
	Rec1
	NTE [] <== NTE container for Rec1.
	  NTE (1)
	Rec2 []
	  Rec2
	  NTE [] <== NTE container for Rec2 array.
	  NTE (2)
	  Rec3
	  Rec4

   a. The positions of the first three records (`Rec1`, `NTE (1)`, and `Rec2`) match the schema. In accordance with the schema definition, the parser:
      - Creates an NTE container (an array) under `Rec1`, and places `NTE (1)` into it.
      - Creates a `Rec2` array, and places `Rec2` into it.
   b. Even though the schema definition says the next record should be an `NTE` (followed by `Rec3`), the next record in the document is `Rec3` (followed by `NTE (2)`). This is valid; the parser simply:
      - Places `Rec3` into the `Rec2` array, as specified in the schema definition.
      - Creates another NTE container (an array) under `Rec2`, as specified in the schema definition. To determine which container to put `NTE (2)` in, he parser looks for the last defined container at the current level. Thus, the parser places `NTE (2)` in this container.
   c. The parser places the next record, `Rec4`, in the `Rec2` array, in accordance with the schema definition.
2. Parsing `NTE (3)`:

Input EDI document	Output IS document

Rec1
NTE (1)
Rec2
Rec3
NTE (2)
Rec4
NTE (3)
	Rec1
	NTE []
	  NTE (1)
	Rec2 []
	  Rec2
	  NTE []
	  NTE (2)
	  NTE (3) <== added to this container at
	  current level
	  Rec3
	  Rec4

   To determine which container to put `NTE (3)` in, the parser looks for the last defined container at the current level. Thus, the parser places `NTE (3)` in the same container as `NTE (2)`.

3. Parsing `Rec2` and `NTE (4)`:

Input EDI document	Output IS document

Rec1
NTE (1)
Rec2
Rec3
NTE (2)
Rec4
NTE (3)
Rec2
NTE (4)
	Rec1
	NTE []
	  NTE (1)
	Rec2 []
	  Rec2
	  NTE []
	  NTE (2)
	  NTE (3)
	  Rec3
	  Rec4
	  Rec2
	  NTE [] <== creates an NTE container for
	  second Rec2
	NTE (4)

In accordance with the schema definition, the parser adds another `Rec2` to the `Rec2` array. Just as with the first `Rec2` in the array, the parser also creates another NTE container, for the second `Rec2`.

4. Parsing `Rec5` and `NTE (5)`:

Input EDI document	Output IS document

Rec1
NTE (1)
Rec2
Rec3
NTE (2)
Rec4
NTE (3)
Rec2
NTE (4)
Rec5
NTE (5)
	Rec1
	NTE []
	  NTE (1)
	  NTE (5) <== this container is same level as
	  Rec5 (current level)
	Rec2 []
	  Rec2
	  NTE []
	  NTE (2)
	  NTE (3)
	  Rec3
	  Rec4
	  Rec2
	  NTE []
	  NTE (4)
	Rec5

   a. In accordance with the schema definition, the parser places `Rec5` under `Rec1`.

   b. To determine which container to put `NTE (5)` in, the parser looks for the last defined container at the current level. Thus, the parser places `NTE (5)` in the same container as `NTE (1)`.

Examples: Parsing Outbound Floating Records

When the pub.flatFile:convertToString service converts an IS document (an IData object) to a string, the parser places each floating record immediately following its parent record (as defined in the IS document).

In the following examples, assume that the **Floating Record** property is set to the record NTE.

Example 1:

Flat file schema definition	Input IS document	Output string

Rec1
Rec2
NTE []
Rec3
Rec4
	Rec1
	Rec2
	Rec3
	  NTE []
	  NTE(1)
	  NTE(2)
	Rec4
		Rec1!Rec2!Rec3!NTE(1)!NTE(2)!
		Rec4

Because `NTE(1)` and `NTE(2)` are children of `Rec3`, the parser places them after Rec3 in the string.

Example 2:

Flat file schema definition	Input IS document	Output string

Rec1
Rec2
NTE []
Rec3
Rec4
	Rec1
	Rec2
	Rec3
	Rec4
	NTE []
	  NTE(1)
	  NTE(2)
		Rec1!NTE(1)!NTE(2)!Rec2!Rec3!
		Rec4

Because `NTE(1)` and `NTE(2)` are children of `Rec1`, the parser places them after `Rec1` in the string.

Extractors

Extractors represent the location of a field within a record, or of a subfield within a composite. You extract fields from a composite using a Nth field extractor, while you extract fields from a record using **Nth field**, **ID Node**, or **Fixed Position** extractors.

Note: To use **Nth Field** or **ID Node** extractors for a field or composite in a record, the field delimiter must be specified. To use these extractors for a field in a composite, the subfield delimiter must be specified. If these extractors are used without the appropriate delimiter set, the field values will not be returned correctly.

- **Nth Field.** Counting from zero (0), **Position** indicates the position of the field that you want to extract from the record. This value cannot be null and must be an integer greater than or equal to zero (0). For example, if you type 1, the second field will be extracted. This option is available only if you specified a field delimiter when configuring your flat file schema. This extractor returns the field as a key–value

pair. The key is the name of the field. The value is the String value of the field. Software AG recommends that you use this extractor instead of ID node.

- **ID Node.** This extractor is a variation of the **Nth Field** extractor and is available for backward compatibility for users of the webMethods Module for EDI. Counting from zero (0), **Position** indicates the position of the field that you want to extract from the record. This value cannot be null and must be an integer greater than or equal to zero (0). This extractor returns the field as a key–value pair. The key is the name of the field. The value is an IS document (IData object) containing a key–value pair in which the key is always "code," and the value is the String value of the field. Software AG does not recommend that you use the ID node extractor.
- **Fixed Position.** Counting from zero (0), extracts a fixed number of bytes from a record.
  - **Start Position.** First byte to extract from the record.
  - **End Position.** First byte that is not included in the extraction. If you enter a negative number (for example, –1), the extractor returns all bytes from the byte specified in the **Start Position** to the last byte in the record or composite.

For example:

record = webMethods
Start Position = `1`
End Position = `7`
Extracted bytes = `ebMeth`

Validators

Different types of validators are available for records, composites, fields, and multi-byte encoded records. You validate:

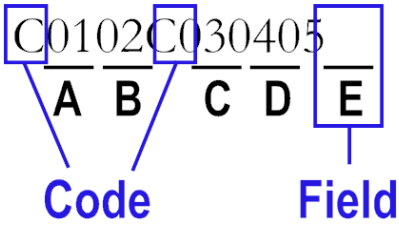
- Records and composites using conditional validators.
- Fields using either length or code list validators.
- Multi-byte encoded records, you use the byte count validator.

Conditional Validators

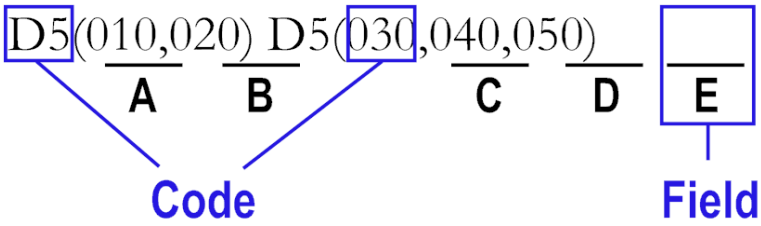
Regardless of what type of flat file you are processing, you follow the syntax rules created by the EDI Standards bodies ANSI X12 or UN/EDIFACT to specify conditional validators for a flat file record or composite. In these rules, *0n* and *0n0* represent the number of the field defined in the record or composite, where *n* is the number of the field in the composite or record. For example, in the following flat file:

```
Record1*FieldA*FieldB*FieldC*FieldD*FieldE;
```

To validate fields using the ANSI X12 conditional validator, you might enter the following in the **Validator** property:



To validate the appropriate fields using the UN/EDIFACT conditional validator, you might enter the following in the **Validator** property:



The record definition has five fields defined. Each with an Nth Field extractor, the first extracting field A, the second field B and so on. Both of these validators indicate that if Field A is present, Field B is required. If Field C is present, Field D and Field E are required. However, if Field D and Field E are present, Field C need not be present.

The conditional validator refers to the location of the field in the definition, not the location of the data in the flat file. For example, if you switched the extractors for the first and second field such that the first field extracted the value Field B and the second field extracts the value Field A, the conditional validator would indicate that if Field B is present Field A is required.

Example:

Record1 - conditional validator `C0102`
FieldA - Nth field extractor, position 1
FieldB - Nth field extractor, position 2

This indicates that if FieldA is present Field B also must be present. If you switched the position of these fields in the record, you would get the following.

Example:

Record1 - conditional validator `C0102`
FieldB - Nth field extractor, position 2
FieldA - Nth field extractor, position 1

This now indicates that if Field B is present Field A also must be present.

ANSI X12 Rule Name (and Code)	UN/EDIFACT Rule Name (and Code)	Definition
Required (R)	One or more (D3)	At least one of the specified fields must be present. **ANSI X12 Example:** R030405 **UN/EDIFACT Example:** D3(030,040,050)
Conditional (C)	If first, then all (D5)	If a specified field is present, all of the fields specified in a dependent set of fields also must be present. If a dependent field is present, the first specified field need not be present. **ANSI X12 Example:** C030405 **UN/EDIFACT Example:** D5(030,040,050)
Exclusion (E)	One or none (D4)	At most, only one of the specified fields can be present. **ANSI X12 Example:** E030405 **UN/EDIFACT Example:** D4(030,040,050)
List Conditional (L)	If first, then at least one (D6)	If a specified field is present, at least one of the fields specified in a dependent set of fields also must be present. If a dependent field is present, the first specified field need not be present. **ANSI X12 Example:** L030405 **UN/EDIFACT Example:** D6(030,040,050)
Paired (P)	All or none (D2)	All of the specified fields must be present or absent. **ANSI X12 Example:** P030405 **UN/EDIFACT Example:** D2(010,020)
N/A	One and only one (D1)	Exactly one field may be present. **UN/EDIFACT Example:** D1(030,040,050)

ANSI X12 Rule Name (and Code)	UN/EDIFACT Rule Name (and Code)	Definition
NA	If first, then none (D7)	If the first field is present, all others cannot be included. **UN/EDIFACT Example:** D7(030,040,050)

Field Validators

You validate fields using either a length validator or a code list validator.

Length Validator

For a length validator, you specify the maximum and minimum number of characters this field can contain to be considered valid.

- **Minimum length.** Minimum number of characters that this field can contain to be considered valid. If the field contains fewer characters than the number entered, an error will be generated. You can enter zero (0).
- **Maximum length.** Maximum number of characters that this field can contain to be considered valid. If the field contains more characters than the number entered, an error will be generated. If you enter a negative number (for example, –1), a maximum length error will never be generated for this field, regardless of its value.

Code List Validator

In the **Conditions** box, type the appropriate code list conditions. The code list is a comma–separated list of the allowed values for this field. If the value of the field is not contained in the code list, errors will be generated.

Note: This validator validates only in ascending order and according to the ASCII values.

You can specify three different types of code lists:

- **Simple Code List.** A comma separated list of valid codes.
- **Range List.** A range list, which can occur within any type of code list, validates that a string value is between a given start string and end string. The range is specified as *start string:end string* . Values are considered to be valid for a given range if all of the following conditions are true:
  - The value is the same length as the start and end strings of the range.
  - The character value in the same position in the start and end string comes from the same character set (a–z, A–Z or 0–9).

- The value is lexically greater than or equal to the start string and lexically less than or equal to the value of the end string.

For example `A0:Z9` is valid, because the first character of each string comes from the set A–Z, and the second character of each string comes from the set 0–9. `A2:2B` is not valid because the first character of the start string does not come from the same character set as the first character of the end string.

- **Partition List.** A partition joins together multiple code lists within one field, separated by a pipe (`|`). For example, if you enter `ABC,DEF|1,2`, the allowed values of the field are ABC1, ABC2, DEF1, and DEF2. Each set of code lists must have the same number of bytes.

  The code list validator allows for the right hand side of the partition to be *optional*. For example, if you enter `ABC,DEF|1,,2`, the allowed values of the field are:

  ABC1, ABC2, ABC, DEF1, DEF2, and DEF.

  The extra comma in `1,,2` indicates that the values `1` and `2` are optional.

Byte Count Validator

You use the byte count validator for multi-byte encoded records. You specify the maximum and minimum number of bytes this field can contain to be considered valid.

- **Minimum length.** Minimum number of bytes that this field can contain to be considered valid. If the field contains fewer bytes than the number entered, an error will be generated. You can enter zero (0).
- **Maximum length.** Maximum number of bytes that this field can contain to be considered valid. If the field contains more bytes than the number entered, an error will be generated. If you enter a negative number (for example, –1), a maximum length error will never be generated for this field, regardless of its value.

Format Services

You can use the **Format Service** property in the Properties view of the flat file dictionary editor or flat file schema editor to specify the service you want to invoke for a particular field when pub.flatFile:convertToValues or pub.flatFile:convertToString is called.

Important: If a particular field does not have a value (that is, a value is not returned in an IS document (IData object) for pub.flatFile:convertToValues or is not present in the input data for pub.flatFile:convertToString) the format service assigned to that field will not be executed.

A format service formats the string and optionally ensures that the value of the string meets the restrictions specified in the format service. For example, when using the pub.flatFile:convertToValues service to convert a date field in a flat file into an IS document (IData object), you can specify that a particular format service be invoked for that field

during parsing. To alter the format of the date field, you might invoke a service that changes the format of the field from YYYYMMDD to MMDDYYYY.

Note: If you are using webMethods Module for EDI, you can use the built-in format services provided in the WmEDI package to format fields. For more information, see the *webMethods Module for EDI Installation and User's Guide* and the *webMethods Module for EDI Built-In Services Reference*.

Creating Format Services

To create a format service for use with a field in a flat file schema or dictionary, the service must implement the pub.flatFile:FormatService service as its template specification for the format service (located on its **Input/Output** tab).

Working with Format Error Messages

If the *validate* input variable of your format service was set to `true` to validate the format and value of a field, and that field does not meet the criteria specified in the service, you will receive an error message in the *errorMessage* output variable of the format service indicating why this field could not be validated.

Disabling Format Services

To disable the format service for a particular field, you simply clear the text box and save the flat file schema or dictionary. To disable all format services within the WmFlatFile package regardless of what is specified in each field definition, open *Integration Server_directory*\instances*instance_name* \packages\WmFlatFile\config \ff.cnf and set the *enableFormatting* variable to `false`.

For more detailed information about format services, see the description of pub.flatFile:FormatService in *webMethods Integration Server Built-In Services Reference*

Managing Dictionary Dependencies on Format Services

If you move or rename a format service, the Integration Server's dependency manager will ask whether it should update all the flat file schemas and flat file dictionaries that use that format service with the new location or name of the format service.

If you delete a format service, the dependency manager will list all schemas and dictionaries that will be impacted by the deletion, and prompts you to confirm the deletion.

Managing Flat File Dictionary Dependencies

If you move or rename a dictionary, Integration Server's dependency manager will ask whether it should update all the flat file schemas and dictionaries that use this dictionary with the new location or name of the dictionary.

The dependency manager also manages dictionary dependencies on format services in a similar way, see *Managing Dictionary Dependencies on Format* .

Customizing the Flat File Configuration Settings

You can customize some aspects of your flat file experience.

The WmFlatFile package provides the following flags you can use to customize some aspects of the package's default behavior. You implement these flags in the configuration file:

Integration Server_directory\instances*instance_name* \packages\WmFlatFile\config \ff.cnf

Flag	Value
recWithNoIDLike46	Determines how the pub.flatFile:convertToValues service handles *recordWithNoID* records. ■ If `recWithNoIDLike46=false` (the default), each *recordWithNoID* record appears as a child of the record above it, in an array. ■ If `recWithNoIDLike46=true`, the services mimics the handling of *recordWithNoID* that was implemented in version 4.6 of the Integration Server. That is, all *recordWithNoID* records appeared as children of the root. In addition, when the pub.flatFile:convertToValues service returned only one *recordWithNoID* record, it returned it as a single record, *not* as an array.
alwaysSortFields	Determines whether the pub.flatFile:convertToString service checks the order of the fields in composites and records that it receives as input (in *ffValues*) against the flat file schema that is specified in the *ffSchema* input parameter. ■ If `alwaysSortFields=false`, the pub.flatFile:convertToString service does not check the order of the fields in the composites and records. It

Flag	Value
	assumes that the fields in composites and records are in the same order as the flat file schema. This is the default. ■ If `alwaysSortFields=true`, the pub.flatFile:convertToString service checks the order of the fields in the composites and records. It attempts to match the order of the fields in composites and records to the order of the fields in composites and records in the flat file schema.
allowSubfieldDelimiterIn NonCompositeField	Determines whether Integration Server Flat File Adapter allows subfield delimiters in non-composite elements. ■ If `allowSubfieldDelimiterInNonCompositeField=false`, subfield delimiters are not allowed in a non-composite element. This is the default. ■ If `allowSubfieldDelimiterInNonCompositeField=true`, subfield delimiters are allowed in a non-composite element.
dontTrimControlCharacters	Determines how whitespace and control characters are handled by the service wm.b2b.edi:convertToString when the parameter *spacePad* is set to left or right. ■ If `dontTrimControlCharacters=true`, and *spacePad* is set to left or right, the service does not truncate control characters from delimited fields. ■ If `dontTrimControlCharacters=false`, and *spacePad* is set to left or right, control characters are treated as whitespace and truncated.
recordIdentifierCheck	Determines whether the pub.flatFile:convertToString service includes the record identifier in the string output parameter. ■ If `recordIdentifierCheck=false`, the pub.flatFile:convertToString service includes the record identifier in the string output parameter. This is the default.

Flag	Value
	■ If `recordIdentifierCheck=true`, the pub.flatFile:convertToString service does not include the record identifier in the string output parameter.

spacePadJustifies	Controls how the input parameter *spacePad* justifies fixed length fields handled by the services pub.flatFile:convertToString and wm.b2b.edi:convertToString.
	■ If `spacePadJustifies=true` and:
	  ■ *spacePad* is set to left, fixed length fields are left justified.
	  ■ *spacePad* is set to right, fixed length fields are right justified.
	■ If `spacePadJustifies=false` and:
	  ■ *spacePad* is set to left, fixed length fields are *right* justified.
	  ■ *spacePad* is set to right, fixed length fields are *left* justified.
	  This is the default.
	spacePadJustifies has no effect when *spacePad* is set to none.

UNAOnlyForImmediateUNB	Controls how EDIFACT documents with multiple interchanges are validated when an interchange does not contain a UNA segment. When none of the interchanges contain a UNA segment, each interchange is validated against the default delimiters regardless of the value of this property. ■ If `UNAOnlyForImmediateUNB=true`, each UNB segment is validated against its corresponding UNA segment. When there is no corresponding UNA segment, EDI Module uses the default delimiters for validation. ■ If `UNAOnlyForImmediateUNB=false`, each UNB segment is validated against its corresponding UNA segment. When there is no corresponding UNA segment, EDI Module uses the delimiters set in the previous UNA segment for validation. This is the default.

Flag	Value
useAlternateCodeSets	Determines whether to enable the use of alternate code sets. ■ If `useAlternateCodeSets=true`, use of alternate code sets is enabled. This is the default. ■ If `useAlternateCodeSets=false`, use of alternate code sets is disabled.

useG11NEncodings and *alwaysUseWebMEncodings*	Determines whether the pub.flatFile:convertToValues and pub.flatFile:convertToString services use the webMethods charset or the default charset provided by JVM.
	■ If `useG11NEncodings=true` and either the encoding name begins with `WM_` OR `alwaysUseWebMEncodings=true`, the pub.flatFile:convertToValues and pub.flatFile:convertToString services use the webMethods charset.
	■ If `useG11NEncodings=false`, the pub.flatFile:convertToValues and pub.flatFile:convertToString services use the default charset provided by JVM, regardless of the value assigned to `alwaysUseWebMEncodings`.
	The default value for `alwaysUseWebMEncodings` is `false` and `useG11NEncodings` is `true`.
	Example:
	`alwaysUseWebMEncodings=false`
	`useG11NEncodings=true`

useReadableDelimiterReporting	Controls how the delimiter record parser parses non-readable delimiters, such as `\r` (carriage return), `\n` (line feed), or `\t` (tab). ■ `true` (the default): The parser reports the delimiters as human-readable characters, e.g., the carriage return character is represented as `\r`. This variable does not impact human-readable delimiters such as `|` or `*`. ■ `false`: The parser reports the delimiters as the actual delimiter character. For example, if the delimiter is a carriage return, the value reported will be the non-readable carriage return code (decimal 13 or hexadecimal OD).

Flag	Value

	Note: If you use webMethods Module for EDI, set this variable to false.
	
	This impacts how you can use the delimiter information in flow mapping. If you use the delimiters directly in flows with variable substitution, non-printable characters will not be substituted correctly when *useReadableDelimiterReporting=true* .
	
	For example, when the output of the pub.flatFile:convertToValues service shows that the record delimiter is `\n` and a user flow has an input set to the following:
	
	abc%@delimiters\record%def
	
	the resulting value will be as follows:
	
	If *useReadableDelimiterReporting=true* , the resulting value is `abc\ndef`.
	
	If *useReadableDelimiterReporting=false* , the resulting value is
	
	`abc`
	`def`

useAlternateNameForSegments	Determines whether the flat file parser uses an alternate name for segments, if an alternate name is provided. If *useAlternateNameForSegments=true* and an alternate name for a segment is provided, the flat file parser uses an alternate name for segments. If an alternate name is not provided, the flat file parser uses the existing segment name. If *useAlternateNameForSegments=false* , the flat file parser always uses the existing segment name. This is the default.

Note: When processing many small files, a file polling port might become disabled and the following exception might be returned: java.lang.IllegalArgumentException: Comparison method violates its general contract. If this occurs, modify the custom_wrapper.conf file to pass the following Java system property to Integration Server: java.util.Arrays.useLegacyMergeSort=true. For information about how to pass Java system properties to Integration Server, see *webMethods Integration*

*Server Administrator's Guide*For Microservices Runtime, add the property to JAVA_CUSTOM_OPTS in *Integration Server_directory*/bin/server.bat(sh).

4 Sending and Receiving Flat Files

Overview

The webMethods Integration Server provides the following automated mechanisms for sending and receiving any type of flat file.

- Submit a flat file to a service in a String variable (of any name), which the target service can convert into an IData object using pub.flatFile:convertToValues.
- Post a flat file to a service via HTTP.
- FTP a flat file to a service.
- Submit a flat file to a service via File Polling.
- Send a flat file to a service as an e-mail message attachment.

For general steps to configure ports, which is not included in this guide, see *webMethods Integration Server Administrator's Guide*.

Note: If you use Trading Networks, you can send flat files directly to Trading Networks. For more information about how Trading Networks processes flat files, see sections about flat file TN document types in *webMethods Trading Networks User's Guide* and *webMethods Trading Networks Administrator's Guide*.

Flat File Content Handler and Content Type

You submit a flat file to a flow service using one of the methods discussed in this chapter. The WmFlatFile package registers a content handler for the content type "application/x–wmflatfile." This content handler passes the contents of the flat file to the service specified in the submit method in an InputStream named *ffdata* . If an InputStream or ByteArray named *ffreturn* is present when the service completes execution, the InputStream is read and the results are returned to the invoker of the service, for example, HTTP, FTP, etc.

The WmFlatFile package provides services that can be called within customized flow services or the File Polling processing service to initially accept and consume inbound flat files. These services are described in "Choosing the Service to Receive the Flat File from the Content Handler" on page 46. For more information about File Polling, see "Submitting a Flat File to Integration Server via File Polling" on page 52.

Choosing the Service to Receive the Flat File from the Content Handler

Those methods that use the flat file content handler must do the following:

- Accept an InputStream object called *ffdata* .

- Return data in an InputStream or a ByteArray called *ffreturn* .

The WmFlatFile package provides the pub.flatFile:convertToValues service to be used within flow services to initially accept inbound flat files. Details about using this service to process inbound documents are provided in "Processing Inbound Flat Files" on page 21.

Submitting a Flat File to Integration Server in a String Variable

One way to submit a flat file to the Integration Server is to pass the entire document to a service in a String variable. This approach does not use the flat file content handler, so the restrictions outlined in the previous section do not apply. Instead, the invoked service must expect the contents of the flat file to be in a specific pipeline variable. The Java client must place the source file in the pipeline as a String. The invoked service then processes the input data and can place parameters in the pipeline that will be returned to the Java client as determined by the specific scenario.

When you use this approach, you should code the target service to execute pub.flatFile:convertToValues to convert the String variable (i.e., the flat file) to an IS document (IData object). For more information about using convertToValues, see *webMethods Integration Server Built-In Services Reference*.

The following code fragment illustrates how a Java client might submit a flat file to the purch:postOrder service on the Integration Server. In this example, the client 1) loads the flat file into a String, 2) puts the String into the element orders in an IS document (IData object) named "inputs," and 3) invokes purch:postOrder on the server at localhost:5555.

```
import com.wm.app.b2b.client.*;
import com.wm.util.*;
import com.wm.data.*;
import java.io.*;
public class ArbitraryFlatFileClient
.
.
//—Load FF into orders string
String orders = YourLoadFlatFileMethod(orderFile);
//—Put input values into IData object
IData inputs = IDataFactory.create();
IDataCursor inputsCursor = inputs.getcursor();
inputsCursor.last();
inputsCursor.insertAfter("orders", orders);
inputsCursor.insertAfter("authCode", authCode);
//—Submit request to server
c.connect("localhost:5555", "null", null);
IData outputs = c.invoke("purch", "postOrder", inputs);
c.disconnect();
if (inputsCursor.first("response"))
{
//...process response
}
```

The Integration Server invokes postOrder to pass the flat file to convertToValues. This will produce an "orders" string that can be mapped to the *ffdata* input variable in the

convertToValues service. As discussed in "Processing Flat Files Overview" on page 19, this service will produce an IS document (IData object).

Submitting a Flat File to Integration Server via HTTP

A client can post a flat file to a service via HTTP. To use this approach, you must have a client that can do the following:

- Send a string of data (a flat file) to the Integration Server using the HTTP POST method.

  –AND–

- Set the value of the *Content–Type request–header field* to "application/x–wmflatfile."

When the Integration Server receives an HTTP POST request where *Content–Type* is application/x–wmflatfile, it passes the body of the request as an InputStream to the service specified in the request's URL. Because most browsers do not allow you to modify the Content–Type header field, they are not suitable clients for this type of submission. Clients that you might use to submit a flat file in this manner include PERL scripts (which allow you to build and issue HTTP requests) or the Integration Server service, pub.client:http.

Building a Client that Posts a Flat File to a Service

Regardless of which client you use, if you want to post a flat file to a service through HTTP, you must specify certain information.

To build a client

1. Submit a POST request to the Integration Server.
2. Address the request to the URL of an service (for example, `http://rubicon:5555/invoke/purch/postOrder`).
3. Set the *Content–Type* header field to "application/x–wmflatfile."
4. Contain flat file data in the body of the message. The data must be the only text that appears in the body of the request. Do not assign it to a *name=value* pair.

The following example describes the values that you set if you use pub.client:http to POST a flat file to a service.

Set this variable...	To
url	This is the URL of the service that you want to invoke. webMethods services can be invoked via a URL. The format for specifying the URL is http://*hostname:port*/ invoke/ *folder.subfolder/service name* , where *hostname* is the name of the machine running the Integration Server, *port* is the port

Set this variable...	To
	on which the Integration Server is running, *folder.subfolder* is the path to the service folder names, and *service name* is the name of the service to invoke. The following value would invoke the "ProcessFlatFile" flow service located in the "MyFFProcessing" package on the "rubicon" server with port number "5555." For example, `http://rubicon:5555/invoke/MyFFProcessing/ProcessFlatFile`
method	"post"
headers.Content–Type	*Name:* Content–type *Value:* The specific content type for the document (application/x-wmflatfile)
data.string –OR– *data.bytes*	This is the flat file that you want to post.

You also will set any optional HTTP variables, such as authorization information, that are required by your application. The invoked service will have an InputStream *ffdata* object in the pipeline that contains the flat file data.

Submitting a Flat File to Integration Server via FTP

You can FTP a flat file to the Integration Server's FTP listening port. By default the FTP port is assigned to port "8021." However, this assignment is configurable, so you should check with your Integration Server administrator to see which port is used for FTP communications on your Integration Server.

When the Integration Server receives a flat file on the FTP listening port, it passes it as an InputStream to the service in the directory to which the file was FTP'd.

To submit a flat file to the Integration Server via FTP, the service to which you want to pass the document must take an InputStream as input for the *ffdata* variable.

If you want to submit a flat file to a service through FTP, the application must specify certain information.

To submit a flat file via FTP

1. Initiate an FTP session on the Integration Server's FTP listening port.
2. Point to the directory that contains the service to which you want to pass the flat file. For example,

```
cd \ns\Purchasing\SubmitOrder
```

Note: Note that the root directory for this operation is your Integration Server's namespace directory (ns), not the root directory of the target machine. Therefore, if you want to FTP a file to a service in the Purchasing folder, you use \ns\Purchasing*ServiceName* as the path to that service, not *Integration Server_directory*\instances*instance_name* \ns \Purchasing*ServiceName* . Each folder in Designer is treated as a directory when FTPing to the Integration Server. For example, for the purchasing.purchaseRequests:rfq service, the directory name would be \ns \purchasing\purchaseRequests\rfq.

3. Copy the flat file to this directory using the following command:

   `put`*sourceflatfile.txt destinationflatfile.txt;file content type*

   Where *sourceflatfile.txt* is the name of source file that you want to pass to the Integration Server, *destinationflatfile.txt* is the name of the destination file, and *application:x–wmflatfile* indicates the content type of the file submitted. For example, to put a document called `PurchaseOrder.txt` on the Integration Server as `Partner1PO.txt`, you would use the following FTP command:

   ```
   put PurchaseOrder.txt Partner1PO.txt;application:x-wmflatfile
   ```

   Note that the file you FTP to the Integration Server is never actually written to the server's file system. The file you send and the output file it produces (see below) are written to a virtual directory system maintained in your Integration Server session.

FTPing a File From a webMethods Integration Server

The pub.client folder contains built–in services that you can use to FTP a file from the Integration Server. To use the pub.client:ftp service, you must set the variables listed in the following table.

Set this variable...	To set...
serverhost	This is the name of the machine running the Integration Server.
serverport	This is the port on the Integration Server that is listening for FTP traffic.
username	This is the valid user name on the target machine.
password	This is the valid password for the username in *username* .
command	"put"

Set this variable...	To set...
dirpath	If you are sending to the Integration Server, this is the local path to the service that should be invoked after receiving the flat file. For example, `\ns\Purchasing\SubmitOrder`. If sending to another type of server, this is the working directory of the FTP server.
localfile	This is the name of the source file.
remotefile	This is the name of the remote file in which you want to save the data you sending. For example, `destinationflatfile.txt`. If you are sending to the Integration Server, the file name should be followed by the content–type. For example, `destinationflatfile.txt;application:x-wmflatfile`.

For more information about these services, see the *webMethods Integration Server Built-In Services Reference*.

Getting Results from an FTP'd Document

The results from a service executed by an FTP request are written to the same virtual directory to which the flat file was initially FTP'd. The name of the output file to which results are written is *`sourcefile`*`.txt.out`.

You retrieve this document using the FTP "get" command. For example, if you put a document called "PurchaseOrder.txt" on the Integration Server, you would use the following FTP command to get its results:

```
getPurchaseOrder.txt.out
```

"PurchaseOrder.txt.out" is the name of the flat file initially FTP'd to the service. This file contains the value of the output variable *ffreturn* of the invoked service.

It is a good practice to make each file name that you FTP to the Integration Server unique (perhaps by attaching a timestamp to the name) so that you do not inadvertently overwrite other FTP'd documents or their results during an FTP session.

Important: When you end the FTP session, the Integration Server automatically deletes the original file and its results from your session.

Submitting a Flat File to Integration Server via File Polling

You can send a flat file to the Integration Server via a File Polling processing service. For general steps to configure the File Polling listener port, see *webMethods Integration Server Administrator's Guide*.

To poll specifically for flat files, you must specify the Default Content Type as "application/x–wmflatfile". When a file with this content type is posted to the directory on the Integration Server that you are monitoring for flat files (Monitoring Directory), the Integration Server executes the service that you specified as the Processing Service.

Because the content type is "application/x–wmflatfile", when the Integration Server receives the flat file in the appropriately configured monitoring directory, it passes the document as an InputStream to the service configured in the pipeline variable *ffdata* .

The File Polling processing service of Integration Server processes files in the order in which they are received. Integration Server determines the order for processing files by comparing the timestamps of the files in the monitoring directory. In a high speed environment, where many files are placed in the monitoring directory at once, resulting in some of the files having the same timestamp, Integration Server processes the files alphabetically.

Submitting a Flat File to Integration Server via an E-mail Message

You can send a flat file document as an attachment to an e–mail message to an e–mail mailbox and have the Integration Server automatically retrieve the e–mail message and process the flat file it contains. To do this, your Integration Server must be configured with an e–mail port that monitors the mailbox to which the flat file will be sent. (Consult your Integration Server administrator to see whether an e-mail port has been set up on your Integration Server.)

When a flat file arrives in the e–mail port's mailbox, the Integration Server automatically retrieves the message and passes that document as an InputStream to the service specified on the e–mail's subject line (or, if a service is not specified on the subject line, the e–mail port's default service).

Requirements for Submitting a Flat File Document via an E-mail Message

To submit a flat file to the Integration Server via an e–mail message, your client program must specify certain information.

To submit a flat file via e-mail

1. Put the flat file in an e–mail attachment.
2. Set the e–mail's Content–Type header to "application/x–wmflatfile."
3. Specify the name of the service that will process the file in the e–mail's subject line. If you leave the subject line empty, the document will be processed by the global service if one is defined or, if not, by the default service assigned to the e–mail port (if one has been assigned). For more information about specifying the port's default service, see *webMethods Integration Server Administrator's Guide*.

The service that will process the flat file must take an InputStream as input in the *ffdata* variable.

The following example describes the values that you would set if you used pub.client:smtp to e-mail a flat file to a service. For more information about using this service, see the *webMethods Integration Server Built-In Services Reference*.

Set this variable...	To...
to	A String specifying the e–mail address monitored by the Integration Server's e–mail port.
subject	A String specifying the fully qualified name of the service to which the Integration Server will pass the attached document. For example, `orders:ProcessPO` If you do not specify *subject* , the e–mail port invokes its default service (if one has been assigned to it).
from	A String containing the e–mail address to which the results of the service will be sent.
attachments.contenttype	The specific content type for the document ("application/x–wmflatfile").
attachments.filename	A string specifying the fully qualified name of the file containing the flat file.

Getting Results from an E-mailed Document

If your e–mail port has been configured to return results, the results from a service invoked through the port are written to *sourcefile* .txt.out, and then sent as an attachment of an e–mail message to the sender of the original message.

This file contains the value of the output variable *ffreturn* of the invoked service. *ffreturn* can be either an InputStream or a byte array.

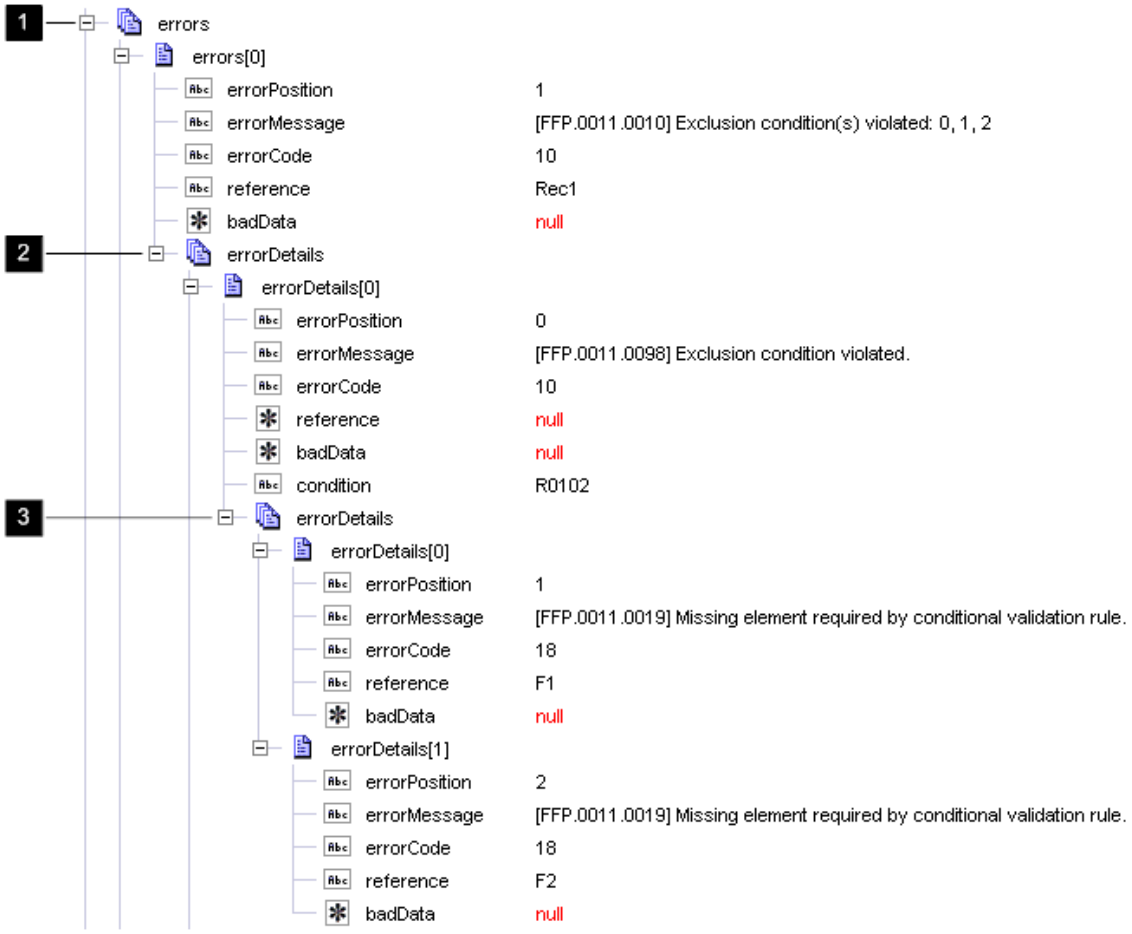
Important: By default, the e–mail port does not return any results from requests that it receives. If you want the port to return results, you must explicitly configure it to do so. For more information about configuring the e–mail port to return results, see *webMethods Integration Server Administrator's Guide*.

A Validation Errors

Validation Error Contents

When the *validate* variable of the pub.flatFile:convertToValues service is set to `true` and an object within the flat file does not conform to the flat file schema, the service generates errors when validating the flat file. If the service finds that an object is invalid, it returns validation errors in the *errors* output of the convertToValues service.

Example of Validation Results



This *errors* array contains the following types of information about the errors that occurred in a record during validation.

- Each error in a record generates one entry in the *errors* array. For more information about the fields in the *errors* array, see .
- Each error in a child record also generates one item within the parent record, which appears nested in the *error* array in the variable *childErrors* .
- Errors with conditional validators can generate detail information about the conditions violated in a child *errorDetails* array, one entry in the array for each

violated condition. The pub.flatFile:convertToValues service only generates details about violated conditions when the *flag/detailedErrors* input variable of the pub.flatFile:convertToValues service is set to `true`. For more information, see *Entities for Conditional Validator Errors in the errorDetails Array*.

General Error Entries in the *errors* Array

An entry in the *errors* array contains the following information. For information about errorDetails entries, see .

Variable in *errors*	Description
errorPosition	Indicates where the error occurred within the parent element. For a record, this indicates the position of the record within the flat file. For a composite, this indicates the position of the field within the composite. For a nested child record, this indicates the position of the field within the record.
errorMessage	A brief description of the error.
errorCode	A number indicating the type of error. For a list, see "Validation Error Codes" on page 60.
reference	The name of the element in the flat file schema that contained the error.
badData	The value of the data that failed validation.
childErrors	(Array). Indicates that a field or subfield within the record or composite generated a validation error.

Entries for Conditional Validator Errors in the *errorDetails* Array

The *errorDetails* array includes detail information about the conditions that were violated when the following are true:

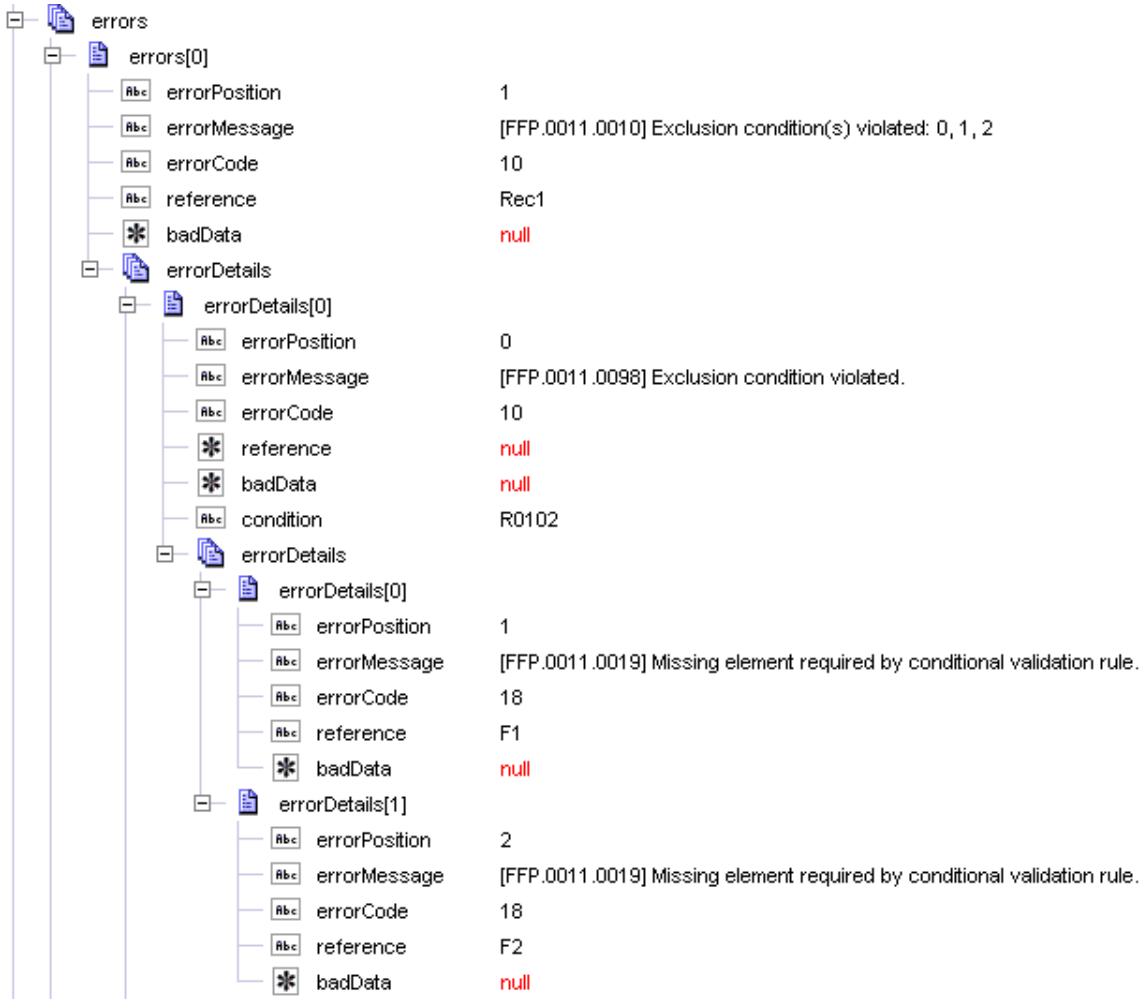
- When you set the *validate* and *flag/detailedErrors* input variables of the pub.flatFile:convertToValues service to `true`.

  –AND–

- The pub.flatFile:convertToValues service encounters errors with conditional validator.

The following shows a sample of the *errors* array that includes the *errorDetails* array, which contains details about violated conditions. See the table below the sample for more information.

Example of Validation Results with Conditional Validation Errors



Portion of the array	Description
errors	For a description of this portion of the *errors* array, see "General Error Entries in the errors Array" on page 57. When the *flag/detailedErrors* input variable of the pub.flatFile:convertToValues service is set to `false`, this is the only information generated in the *errors* array about violated conditions.

Portion of the array	Description
	When the *flag/detailedErrors* is `true`, the pub.flatFile:convertToValues service generates the detail in *errorDetails* array described below.
errorDetails	To provide information about the conditions that were violated, the convertToValues service generates an entry in the *errorDetails* array for each violated condition.

Variable in *errorDetails*	Description
errorPosition	Always zero; the *errorPosition* is not meaningful at this level of the *errorDetails* array.
errorMessage	A brief description of the condition that was violated.
errorCode	A number indicating the type of error. For a list, see "Validation Error Codes" on page 60.
reference	Always null; the *reference* is not meaningful at this level of the *errorDetails* array.
badData	Always null; the *badData* is not meaningful at this level of the *errorDetails* array.
condition	The condition that was violated.
errorDetails	Details about how the condition was violated. See the description below.

Portion of the array	Description
errorDetails	The child *errorDetails* array contains detail about how the condition was violated. The convertToValues service generates an entry for each way the condition was violated.

Variable in *errorDetails*	Description

Portion of the array	Description	
	errorPosition	Indicates the position in the record that contains the field that caused the condition to be violated.
	errorMessage	A brief description of the condition that was violated.
	errorCode	A number indicating the type of error. For a list, see "Validation Error Codes" on page 60.
	reference	The name of the element in the flat file schema that contained the error.
	badData	The value of the data that failed validation.

Validation Error Codes

The following table describes the validation error codes that you might receive when the *validate* variable of the pub.flatFile:convertToValues service is set to `true` and you are validating a flat file or testing a flat file schema.

Error Code	Description
1	In the flat file schema, the **Mandatory** drop-down menu was set to true for this element, but the element does not occur in the flat file.
2	Reserved for future use.
3	Unexpected element. This field is not allowed in the record or composite in which it appears.
4	In the flat file schema, a length validator was specified in the **Validator** property for this field. The value in the flat file exceeded the maximum length specified.
5	In the flat file schema, a length validator was specified in the **Validator** property for this field. The value in the flat file did not meet the minimum length specified.

Error Code	Description
6	Reserved for future use.
7	In the flat file schema, a code list validator was specified in the **Validator** property for this field. The value in the flat file was not listed in the flat file schema as a valid value.
8	Reserved for future use.
9	Reserved for future use.
10	In the flat file schema, a conditional validator was specified in the **Validator** property for this composite or record. The contents of this composite or record did not meet the conditional requirements. The *errorMessage* variable contains the number of the condition that failed. If you had a validation string of `C010203R0405` and both conditions failed, the error message would state that rules 0 and 1 were violated. If only the second is violated, it states that the rule 1 was violated. If you require more detail about conditional validator errors, set the *flag/detailedErrors* input variable of the pub.flatFile:convertToValues service to `true`. For a description of the error details that convertToValues service generates when you set *flag/detailedErrors* to `true`, see "Entries for Conditional Validator Errors in the errorDetails Array" on page 57.
11	Indicates that this record is not defined anywhere in the flat file schema. You will receive this error only when you have not specified a **Default Record** or selected **Allow Undefined Data** where this record appears in the flat file. If you have specified either of these properties, you will not receive this error.
12	Indicates that this record is defined in this flat file schema, but it occurs in the flat file in a location prohibited by the flat file schema.
13	Reserved for future use.
14	In the flat file schema, you specified a maximum repeat value in the **Max Repeat** property for a particular record. The number of record instances in the flat file exceeded the specified maximum number of repeats.
15	Reserve for future use.

Error Code	Description
16	Within a record, this indicates that the record contains a composite or field error. For a composite, this indicates the that composite contains a subfield error.
17	A string could not be formatted into the intended format. A format service reported that the data could not be formatted properly. For information about field format services, see "Format Services" on page 37.
18	Indicates that a conditional validation rule requires a field, but that field is *not* present in the data.
19	Indicates that a field is excluded by a conditional validation rule, but the field *is* present in the data.

B Programming Creating Flat File Schemas and Dictionaries

Overview

You can use the services in the pub.flatFile.generate folder of the WmFlatFile package to programmatically create, modify, and delete:

- Flat file dictionary entries
- Entire flat file dictionaries
- Flat file schemas

To specify the data for the flat file dictionary entries, entire flat file dictionaries, or flat file schemas, you can:

- **Create the information in an XML file** and execute a service you create that retrieves the XML data from the XML file, then invokes the appropriate services in the pub.flatFile.generate folder to make the changes in the Integration Server namespace. When creating the XML data, it should conform to the following XML schema:

  Integration Server_directory\instances*instance_name*\packages\WmFlatFile\pub\FFGeneration.xsd

- **Create a service that maps the data to an IS document (IData object),** convert the IData object to an XML string, and then invoke the appropriate services in the pub.flatFile.generate folder to make the changes in the Integration Server namespace. The IS document type to which the IData object must conform is provided in the WmFlatFile package:
  - For a flat file dictionary entry or dictionary, use the pub.flatFile.generate:FFDictionary IS document type.
  - For a flat file schema, use the pub.flatFile.generate:FFSchema IS document type.

The sample.flatFile.generateFFSchema folder, which is in the WmFlatFileSamples package, contains sample services that illustrate how to use the services in the pub.flatFile.generate folder. The sample services are the sample.flatFile.generateFFSchema:delimited service and the sample.flatFile.generateFFSchema:fixedLength service.

For detailed descriptions of the services in the sample.flatFile folder and the services in the pub.flatFile folder listed in the following sections in this chapter, see the *webMethods Integration Server Built-In Services Reference*.

Creating Flat File Dictionary Entries, Dictionaries, and Schemas

To create a flat file dictionary entry, entire flat file dictionary, or flat file schema, you supply the data for the item you want to create; then invoke the appropriate service to create the item in the Integration Server namespace.

Creating Flat File Dictionary Entries

The following describes the basic steps to create flat file dictionary entries when using either an XML file or mapping data.

Using this method	Follow this procedure
XML File	1. Supply the data for the entries you want to create by creating the XML file that contains the data and that conforms to the XML schema: *Integration Server_directory*\instances*instance_name*\packages\WmFlatFile\pub\FFGeneration.xsd 2. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74. 3. Create the entries in the Integration Server namespace by invoking the pub.flatFile.generate:updateFFDictionaryEntryFromXML service.
Mapping Data	1. Map the data for the entries you want to create to an IData object that conforms to the pub.flatFile.generate:FFDictionary IS document type. 2. Convert the IData object to an XML string by invoking the pub.xml:documentToXMLString service. When you invoke this service: ■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFDictionary`. For more information about the pub.xml:documentToXMLString service, see the *webMethods Integration Server Built-In Services Reference*. 3. Create the entries in the Integration Server namespace by invoking the pub.flatFile.generate:updateFFDictionaryEntryFromXML service.

Creating an Entire Flat File Dictionary with Data

The following describes the basic steps to create an entire flat file dictionary when using either an XML file or mapping data. This procedure describes how to create a flat file

dictionary that contains data. If you want to create an empty flat file dictionary, see "Creating an Empty Flat File Dictionary" on page 67.

Important: The flat file dictionary you are creating must not already exist in the Integration Server namespace. If the flat file dictionary already exists, the pub.flatFile.generate:saveXMLAsFFDictionary service throws an exception.

Using this method	Follow this procedure
XML File	1. Supply the data that includes *all* entries for the dictionary you want to create by creating an XML file that contains the data and that conforms to the XML schema: *Integration Server_directory*`\instances\`*`instance_name`*`\packages\`WmFlatFile`\pub\FFGeneration.xsd` 2. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74. 3. Create the dictionary in the Integration Server namespace by invoking the pub.flatFile.generate:saveXMLAsFFDictionary service.
Mapping Data	1. Map the data for all entries for the dictionary you want to create to an IData object that conforms to the pub.flatFile.generate:FFDictionary IS document type. 2. Convert the IData object to an XML string by invoking the pub.xml:documentToXMLString service. When you invoke this service: ■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to `pub.flatFile.gnerate:FFDictionary`. For more information about the pub.xml:documentToXMLString service, see the *webMethods Integration Server Built-In Services Reference*. 3. Create the dictionary in the Integration Server namespace by invoking the pub.flatFile.generate:saveXMLAsFFDictionary service.

Creating an Empty Flat File Dictionary

The following describes how to create an empty flat file dictionary. If you want to create a flat file dictionary that contains data, see "Creating an Entire Flat File Dictionary with Data" on page 65.

Important: The flat file dictionary you are creating must not already exist in the Integration Server namespace. If the flat file dictionary already exists, the pub.flatFile.generate:createFFDictionary service throws an exception.

Follow this procedure
1. Create the empty dictionary in the Integration Server namespace by invoking the pub.flatFile.generate:createFFDictionary service.

Creating a Flat File Schema

The following describes the basic steps to create a flat file schema when using either an XML file or mapping data:

Important: The flat file schema you are creating must not already exist in the Integration Server namespace. If the flat file schema already exists, the pub.flatFile.generate:saveXMLAsFFSchema service throws an exception.

Using this method	Follow this procedure
XML File	1. Supply the data for the flat file schema you want to create by creating an XML file that contains the data and that conforms to the XML schema: *Integration Server_directory*\instances*instance_name*\packages\WmFlatFile\pub\FFGeneration.xsd 2. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74.
Mapping Data	1. Map the data for the flat file schema you want to create to an IData object that conforms to the pub.flatFile.generate:FFSchema IS document type. 2. Convert the IData object to an XML string by invoking pub.xml:documentToXMLString service. When you invoke this service:

Using this method	Follow this procedure
	■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFSchema`. For more information about the pub.xml:documentToSXLString service, see the *webMethods Integration Server Built-In Services Reference* 3. Create the dictionary in the Integration Server namespace by invoking the pub.flatFile.generate:saveXMLAsFFSchema service.

Modifying Flat File Dictionary Entries, Dictionaries, and Schemas

This section describes how to modify an existing flat file dictionary entry, entire flat file dictionary, or flat file schema.

Modifying an Existing Flat File Dictionary Entry

To modify an existing flat file dictionary entry, you first retrieve from the Integration Server namespace the dictionary entry that you want to modify. You make your modifications to the data; then invoke the appropriate service to write the changes back to the Integration Server namespace. The following describes the basic steps to modify a dictionary entry either using an XML file or mapping data.

Using this method	Follow this procedure
XML File	1. Retrieve the existing information for the dictionary entry from the Integration Server namespace by invoking the pub.flatFile.generate:getFFDictionaryEntryAsXML service and write it to an XML file. For more information, see "Retrieving Namespace Data to Write to an XML File" on page 75. 2. Update the data for the dictionary entry in the XML file. The XML file must conform to the XML schema: *Integration Server_directory*`\instances\`*`instance_name`*`\packages\`WmFlatFile`\pub\FFGeneration.xsd` 3. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74.

Using this method	Follow this procedure
	4. Update the dictionary entry in the Integration Server namespace by invoking the pub.flatFile.generate:updateFFDictionaryEntryFromXML service.
Mapping Data	1. Retrieve the existing information for the dictionary entry from the Integration Server namespace by invoking the pub.flatFile.generate:getFFDictionaryEntryAsXML service. The data is returned as an XML string in the *FFXML* variable. 2. To convert the XML string in the *FFXML* variable to an IData object: a. Invoke the pub.xml:xmlStringToXMLNode service to convert the XML string to an XML node. b. Invoke the pub.xml:XMLNodeToDocumentservice to convert the XML node to an IData object. When you invoke this service: ■ Set the *makeArrays* input variable to `false`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFDictionary`. This creates an IData object that conforms to the pub.flatFile.generate:FFDictionary IS document type. For more information about the pub.xml:xmlStringToXMLNode and pub.xml:XMLNodeToDocument services, see the *webMethods Integration Server Built-In Services Reference*. 3. Map data to the IData object to make your changes. 4. Convert the IData object to an XML string by invoking the pub.xml:documentToXMLString service. When you invoke this service: ■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to pub.flatFile.generate:FFDictionary. For more information about the pub.xml:documentToXMLString service, see the *webMethods Integration Server Built-In Services Reference*. 5. Update the dictionary entry in the Integration Server namespace by invoking the pub.flatFile.generate:updateFFDictionaryEntryFromXML service.

Modifying an Existing Flat File Dictionary

To modify an existing flat file dictionary, you first retrieve from the Integration Server namespace the item that you want to modify. You make your modifications to the data. Delete the dictionary from the namespace before invoking the appropriate service to write the changes back to the Integration Server namespace. The following describes the basic steps to modify a flat file dictionary either using an XML file or mapping data.

Using this method	Follow this procedure
XML File	1. Retrieve the existing information for the flat file dictionary from the Integration Server namespace by invoking the pub.flatFile.generate:findDependants service and write it to an XML file. For more information, see "Retrieving Namespace Data to Write to an XML File" on page 75. 2. Update the data for the flat file dictionary in the XML file. The XML file must conform to the XML schema: *Integration Server_directory*\instances*instance_name*\packages\WmFlatFile\pub\FFGeneration.xsd 3. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74. 4. Delete the existing flat file dictionary from the Integration Server namespace by invoking the pub.flatFile.generate:deleteFFDictionary service. 5. Create the flat file schema in the Integration Server namespace again by invoking the pub.flatFile.generate:saveXMLAsFFDictionary service.
Mapping Data	1. Retrieve the existing information for the flat file dictionary from the Integration Server namespace by invoking the pub.flatFile.generate:findDependants service. The data is returned as an XML string in the *FFXML* variable. 2. To convert the XML string in the *FFXML* variable to an IData object: a. Invoke the pub.xml:xmlStringToXMLNode service to convert the XML string to an XML node. b. Invoke the pub.xml:XMLNodeToDocument service to convert the XML node to an IData object. When you invoke this service:

Using this method	Follow this procedure
	■ Set the *makeArrays* input variable to `false`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFDictionary`. This creates an IData object that conforms to the pub.flatFile.generate:FFDictionary IS document type. For more information about the pub.xml:xmlStringToXMLNode and pub.xml:XMLNodeToDocument services, see the *webMethods Integration Server Built-In Services Reference*. 3. Map data to the IData object to make your changes. 4. Convert the IData object to an XML string by invoking the pub.xml:documentToXMLString service. When you invoke this service: ■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFDictionary`. For more information about the pub.xml:documentToXMLString service, see the *webMethods Integration Server Built-In Services Reference*. 5. Delete the existing flat file dictionary from the Integration Server namespace by invoking the pub.flatFile.generate:deleteFFDictionary service. 6. Create the flat file dictionary in the Integration Server namespace again by invoking the pub.flatFile.generate:saveXMLAsFFDictionary service.

Modifying an Existing Flat File Schema

To modify an existing flat file dictionary or flat file schema, you first retrieve from theIntegration Server namespace the item that you want to modify. You make your modifications to the data. Delete the item from the namespace before invoking the appropriate service to write the changes back to the Integration Server namespace. The following describes the basic steps to modify a flat file schema either using an XML file or mapping data.

Using this method	Follow this procedure
XML File	1. Retrieve the existing information for the flat file schema from the Integration Server namespace by invoking the pub.flatFile.generate:getFFSchemaAsXML service and write it

Using this method	Follow this procedure
	to an XML file. For more information, see"Retrieving Namespace Data to Write to an XML File" on page 75. 2. Update the data for the flat file schema in the XML file. The XML file must conform to the XML schema: *Integration Server_directory*\instances*instance_name*\packages\WmFlatFile\pub\FFGeneration.xsd 3. Retrieve the contents of the XML file as an XML string. For more information, see "Creating a Service that Retrieves the XML File" on page 74. 4. Delete the existing flat file schema from the Integration Server namespace by invoking the pub.flatFile.generate:deleteFFSchema service. 5. Create the flat file schema in the Integration Server namespace again by invoking the pub.flatFile.generate:saveXMLAsFFSchema service.
Mapping Data	1. Retrieve the existing information for the flat file schema from the Integration Server namespace by invoking the pub.flatFile.generate:getFFSchemaAsXML service. The data is returned as an XML string in the *FFXML* variable. 2. To convert the XML string in the *FFXML* variable to an IData object: a. Invoke the pub.xml:xmlStringToXMLNode service to convert the XML string to an XML node. b. Invoke the pub.xml:XMLNodeToDocument service to convert the XML node to an IData object. When you invoke this service: ■ Set the *makeArrays* input variable to `false`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFSchema`. This creates an IData object that conforms to the pub.flatFile.generate:FFSchema IS document type. For more information about the pub.xml:xmlStringToXMLNode and pub.xml:XMLNodeToDocument services, see the *webMethods Integration Server Built-In Services Reference*. 3. Map data to the IData object to make your changes.

Using this method	Follow this procedure
	4. Convert the IData object to an XML string by invoking the pub.xml:documentToXMLString service. When you invoke this service: ■ Set the *encode* input variable to `true`. ■ Set the *documentTypeName* input variable to `pub.flatFile.generate:FFSchema`. For more information about the pub.xml:documentToXMLString service, see the *webMethods Integration Server Built-In Services Reference*. 5. Delete the existing flat file schema from the Integration Server namespace by invoking the pub.flatFile.generate:deleteFFSchema service. 6. Create the flat file schema in the Integration Server namespace again by invoking the pub.flatFile.generate:saveXMLAsFFSchema service.

Deleting Flat File Dictionary Entries, Dictionaries, and Schemas

To delete a flat file dictionary entry, entire flat file dictionary, or flat file schema, you invoke the appropriate service.

For this item...	Use this service to delete the item from the namespace
Flat file dictionary entry	**pub.flatFile.generate:deleteFFDictionaryEntry**
Entire flat file dictionary	**pub.flatFile.generate:deleteFFDictionary**
Flat file schema	**pub.flatFile.generate:deleteFFSchema**

Sample Flow Services for Working with XML Files

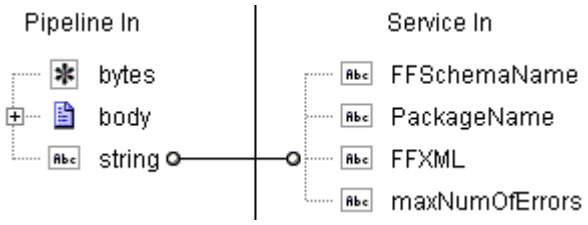
This section shows sample flow services that show how to retrieve data from an XML file in the local file system and how to retrieve data from the Integration Server namespace that can be written to a file in the local file system.

Creating a Service that Retrieves the XML File

The following shows a sample flow service for retrieving data from an XML file.

Sample code for retrieving data from an XML file

1 pub.file:getFile
2 pub.string:bytesToString
3 pub.flatFile.generate:saveXMLAsFFSchema

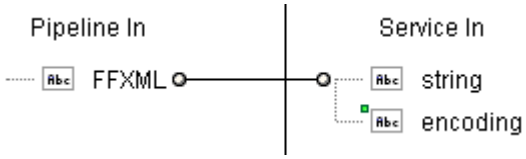
Flow Operation	Description
1	Invoke the pub.file:getFile service to retrieve the XML file from the local file system. The XML data must conform to the following XML schema: *Integration Server_directory*`\instances\`*`instance_name`*`\packages\WmFlatFile\pub\FFGeneration.xsd` For more information about this service, see the *webMethods Integration Server Built-In Services Reference*.
2	Invoke the pub.string:bytesToString service to convert the file (in bytes format) to a String. For more information about this service, see the *webMethods Integration Server Built-In Services Reference*.
3	Invoke the appropriate service to make your changes to the Integration Server namespace. In this sample, the flow service invokes the pub.flatFile.generate:saveXMLAsFFSchema to save the XML data as a flat file schema in the Integration Server namespace. With the INVOKE saveXMLAsFFSchema flow operation selected, in the pipeline, map the output from the pub.string:bytesToString service to the FFXML input of the pub.flatFile.generate:saveXMLAsFFSchema service. 

Retrieving Namespace Data to Write to an XML File

The following shows a sample flow service that retrieves data from the Integration Server namespace that can be written to an XML file.

Sample code for retrieving namespace data that can be written to an XML file

1 ➡ pub.flatFile.generate:getFFDictionaryAsXML
2 ➡ pub.string:stringToBytes
3 ➡ custom:writeToByteToFile

Flow Operation	Description
1	Invoke the appropriate service to retrieve the Integration Server namespace data for a flat file dictionary entry, entire flat file dictionary, or flat file schema. In this sample, the flow service invokes the pub.flatFile.generate:findDependants service to retrieve the data for an entire flat file dictionary.
2	Invoke the pub.string:stringToBytes service to convert the namespace data (in String format) to a byte[]. For more information about this service, see the *webMethods Integration Server Built-In Services Reference*. With the INVOKE stringToBytes flow operation selected, in the pipeline, map the *FFXML* output from the pub.flatFile.generate:findDependants service to the input of the stringToBytes service. 
3	Add code to or invoke a service that you create to write the byte[] to an XML file.

C Flat File Byte Count Parser

Overview

The flat file parser measures the lengths and positions of all records and fields in flat file schemas in terms of *bytes*, not *characters*. Prior to Version 6.5, Service Pack 2, the parser measured all lengths and positions in characters.

Note: This change does *not* affect users who currently parse flat files using single-byte encodings because one byte equals one character. Thus, there is no functional difference between parsing bytes and characters.

This parser supports both single-byte encodings (equivalent to character encodings) and multi-byte encodings. With multi-byte encodings, a byte count can differ from a character count since a single character can be encoded to multiple bytes. Both the pub.flatFile:convertToValues and the pub.flatFile:convertToString services support byte-count parsing for multi-byte encodings. In addition, pub.flatFile:convertToString includes an optional setting that returns a document as a byte array instead of a string. For more information about pub.flatFile:convertToString, see *webMethods Integration Server Built-In Services Reference*.

Important: Multi-byte encoded files must run on JVM version 1.4 or later.

There is no change in the way in which the parser handles the following entities:

- Delimited fields and records.

  A delimiter may be a single character, but may *not* be byte based. Delimiter and release character extractors still extract the specified character from the input file based on a character position.

- The field length validator.

  The parser still uses the String.length() method to validate field lengths (in characters). However, the byte count validator validates the total number of bytes in a field.

Important: The parser does *not* support the writing or reading of binary data (e.g., COMP-3 COBOL fields), nor does it support multiple kinds of encodings within one file.

Configuring the flat file byte count parser

To process multi-byte encoded files, you must configure the flat file byte count parser to read the webMethods character set. The webMethods character set is included in the Extended Character Set file which can be installed using the Software AG Installer.

For more information about Extended Character Set, see the *Installing Software AG Products* document.

To install the Extended Character Set

1. Download Installer from the "Empower Product Support website".
2. If you are installing the Extended Character Set on an existing Integration Server, shut down the Integration Server.
3. Start the Installer.
4. Choose the webMethods release that includes the Integration Server on which you want to install the Extended Character Set.
5. Specify the webMethods installation directory that contains the host Integration Server.
6. In the product selection list, select **Infrastructure** > **Extended Character Set**.
7. After the installation completes, close the Installer and start the host Integration Server.

To verify that the host Integration Server contains the Extended Character Set, run the built-in service, pub.flatFile:getSupportedEncodings. If the encoding output variable contains an encoding with the prefix, "WM_" it indicates that the Extended Character Set is installed.

For example, WM_UTF8 is the webMethods encoding for the Java default UTF-8.

Handling Partial Characters

The parser reads and writes only complete characters. Partial characters are characters that violate character boundary conditions, as described below.

Reading Partial Characters

The following table describes how the parser reads partial characters:

Character Boundary Condition	How the Parser Handles the Condition
Reading a fixed position field that begins in the middle of a multi-byte character.	The field starts on the next complete character. The partial character is ignored.
Reading a fixed position field that ends in the middle of a multi-byte character.	The field ends on the previous complete character. The partial character is not included in the field.

Character Boundary Condition	How the Parser Handles the Condition
Reading a fixed position record that begins or ends in the middle of a multi-byte character.	If the bytes encode to an invalid character, an exception is thrown. **Note** If the bytes that begin or end the record are encoded to a valid character, it will be the *wrong* character.

To illustrate a case where the parser reads fixed position records that begin and end in the middle of a multi-byte character, consider the following multi-byte encoding:

```
12121212
```

These eight bytes represent four two-byte characters. If we specified a fixed length file with a record length of *three* bytes instead of two bytes, the parser would read this encoding as follows, producing an undesirable result:

Record 1: 121

Record 2: 212

Record 3: 12

Note that:

- Record 1 contains one character, formed by the first two bytes of the file. The last byte is a partial character; the parser ignores it.
- Record 2 contains one character, formed by the fifth and sixth bytes of the file. The character formed by the third and fourth bytes is lost because these bytes span Records 1 and 2, and cannot be properly decoded.
- Record 3 contains one character, formed by the seventh and eighth bytes.

Writing Partial Character Encodings

Partial characters present a similar problem when writing to a fixed length file. The parser considers all fixed position fields to be "self contained". This means that all encoding information for a fixed position field is contained in the byte range specified for the field. Keep this in mind when writing multi-byte encodings to fixed length fields because it is possible to specify a field or record that does not end on a character boundary.

Consider a fixed length field that is 10 bytes, but the string for that field encodes to more than 10 bytes. In this case, the parser will truncate the byte array to fit into 10 bytes. This could result in the creation of invalid characters. Thus, the parser always truncates a string on a character boundary; only complete characters are written to the output file.

The following table describes how the parser writes partial characters:

Character Boundary Condition	
Writing a string to a fixed position field where the string is longer than the fixed position field (where it breaks at a character boundary).	Truncates the string to fit the field.
Writing to a fixed position field that ends in the middle of a multi-byte character.	The field ends on the previous complete character. The partial character is not included in the field, and is replaced by one or more pad characters. For an example, see below
Writing to a fixed position field in the middle of a delimited field that contains a stateful encoding.	Does *not* generate an error during creation of the file. Parsing the created file will likely result in an encoding error.

To illustrate a case where the parser writes to a fixed position field that ends in the middle of a multi-byte character, consider the following multi-byte encoding:

Field	Number of Bytes in Field	Character 1	Character 2
Field1	4	12	12
Field2	4	12	345

The parser encodes this multi-byte encoding as follows:

Field	Value
Field1	1212
Field2	12PP

The parser encodes Field1 properly; it considers character 1 and character 2 to be complete characters.

The parser encodes Field2 as follows:

- It considers character 1 to be a complete character
- Since byte 3 does not begin on a character boundary, the parser considers character 2 to be a partial character. It truncates bytes, 3, 4, and 5 because those three bytes

would extend beyond the end of the field. It replaces these three bytes with two pad characters (represented by `PP`).

Stateful Encodings

The parser can read any well-formed document that adheres to the rules of the document's particular kind of encoding. The parser can only write to files that have fixed length fields or delimited records.

Note: Escape encodings behave similarly to stateful encodings. The parser truncates characters preceding the escape sequence so that the integrity of the escape sequence is maintained.

Stateful encodings have two specific bytes that indicate when the record's state has changed:

- `|` is a "shift-in" byte that indicates a change from the normal state to an alternate state
- O (the character O, not zero) is a "shift-out" byte that indicates a change from the alternate state to the normal state

Writing Stateful Encodings

Consider a record definition with the following four-byte fields:

Field	Start Position	End Position	Record Value
A	0	3	Character value "`AB`" encodes to 5 bytes: `|123O`. The character "`A`" is represented by bytes 1 and 2. The character "`B`" is represented by byte 3.
B	4	8	Character value "`C`" encodes to 4 bytes: `|45O`.

The parser encodes this record as follows:

`|12O|45O`

Notice that the parser truncated byte 3 (the character "`B`") from the first field so that the field would fit within the four-byte limit. This truncation occurred on the character boundary between the characters A and B, creating a properly encoded record.

Note: A different method of encoding, using a "padding" byte, would have produced this result:

```
|123450P
```

where `P` is an added padding byte, to extend the record length to the required eight bytes. The parser does *not* use this method of encoding. Using this method, extracting the value for the second field produces:

```
450P
```

The parser cannot properly write this byte sequence. All fixed position fields must be encoded as if they were a stand-alone sequence of bytes. No stateful encoding information may be carried from one fixed position field to any other fixed position field. However, *delimited* fields may carry stateful information from one field to the next field. In contrast, delimited *records* may *not* carry stateful information from one record to the next record. Thus, delimited fields are not necessarily "self contained", but delimited records are.