File2

File Component

The File component provides access to file systems, allowing files to be processed by any other Camel Components or messages from other components to be saved to disk.

URI format

file:directoryName[?options]

or

file://directoryName[?options]

Where **`directoryName`** represents the underlying file directory.

You can append query options to the URI in the following format, **`?option=value&option=value&...`**

Only directories
Camel supports only endpoints configured with a starting directory. So the **`directoryName`** must be a directory. If you want to consume a single file only, you can use the **fileName** option e.g., by setting **`fileName=thefilename`**. Also, the starting directory must not contain dynamic expressions with **`${}`** placeholders. Again use the **`fileName`** option to specify the dynamic part of the filename.
Avoid reading files currently being written by another application
Beware the JDK File IO API is a bit limited in detecting whether another application is currently writing/copying a file. And the implementation can be different depending on OS platform as well. This could lead to that Camel thinks the file is not locked by another process and start consuming it. Therefore you have to do you own investigation what suites your environment. To help with this Camel provides different **`readLock`** options and **`doneFileName`** option that you can use. See also the section *Consuming files from folders where others drop files directly*.

URI Options

Common

confluenceTableSmall

Name	Default Value	Description
`autoCreate`	`true`	Automatically create missing directories in the file's path name. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.
`bufferSize`	`128kb`	Write buffer sized in bytes.
`fileName`	`null`	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the **`CamelFileName`** header. (**Note:** The header itself can also be an Expression). The expression options support both **`String`** and **`Expression`** types. If the expression is a **`String`** type, it is **always** evaluated using the File Language. If the expression is an **`Expression`** type, the specified **`Expression`** type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: **`mydata-${date:now:yyyyMMdd}.txt`**. From **Camel 2.11** onward the producers support the **`CamelOverruleFileName`** header which takes precedence over any existing **`CamelFileName`** header; the **`CamelOverruleFileName`** is a header that is used only once, and makes it easier as this avoids to temporary store **`CamelFileName`** and have to restore it afterwards.
`flatten`	`false`	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to `true` on the producer enforces that any file name received in **`CamelFileName`** header will be stripped for any leading paths.
`charset`	`null`	**Camel 2.9.3:** this option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. See further below for a examples and more important details.
`copyAndDeleteOnRenameFail`	`true`	**Camel 2.9**: whether to fallback and do a copy and delete file, in case the file could not be renamed directly. This option is not available for the FTP component.
`renameUsingCopy`	`false`	**Camel 2.13.1**: Perform rename operations using a copy and delete strategy. This is primarily used in environments where the regular rename operation is unreliable e.g., across different file systems or networks. This option takes precedence over the **`copyAndDeleteOnRenameFail`** parameter that will automatically fall back to the copy and delete strategy, but only after additional delays.

Consumer

confluenceTableSmall

Name	Default Value	Description
initialDelay	1000	Milliseconds before polling the file/directory starts.
delay	500	Milliseconds before the next poll of the file/directory.
useFixedDelay		Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details. In **Camel 2.7.x** or older the default value is **`false`.** From **Camel 2.8** onward the default value is **`true`.**
runLoggingLevel	TRACE	**Camel 2.8:** The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.
recursive	false	If a directory, will look for files in all the sub-directories as well.
delete	false	If **`true`**, the file will be deleted **after** it is processed successfully.
noop	false	If **`true`**, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If **`noop=true`**, Camel will set **`idempotent=true`** as well, to avoid consuming the same files over and over again.
preMove	null	Expression (such as File Language) used to dynamically set the filename when moving it **before** processing. For example to move in-progress files into the **`order`** directory set this value to **`order`**.
move	.camel	Expression (such as File Language) used to dynamically set the filename when moving it **after** processing. To move files into a **`.done`** subdirectory just enter **`.done`**.
moveFailed	null	Expression (such as File Language) used to dynamically set a different target directory when moving files *in case of* processing (configured via `move` defined above) failed. For example, to move files into a **`.error`** subdirectory use: **`.error`**. **Note**: When moving the files to the "fail" location Camel will **handle** the error and will not pick up the file again.
include	null	Is used to include files, if filename matches the regex pattern (matching is case in-sensitive from Camel **2.17** onward).
exclude	null	Is used to exclude files, if filename matches the regex pattern (matching is case in-sensitive from Camel **2.17** onward).
antInclude	null	**Camel 2.10:** Ant style filter inclusion, for example **`antInclude=*/.txt`**. Multiple inclusions may be specified in comma-delimited format. See below for more details about ant path filters.
antExclude	null	**Camel 2.10:** Ant style filter exclusion. If both **`antInclude`** and **`antExclude`** are used, **`antExclude`** takes precedence over **`antInclude`**. Multiple exclusions may be specified in comma-delimited format. See below for more details about ant path filters.
antFilterCaseSensitive	true	**Camel 2.11:** Ant style filter which is case sensitive or not.
idempotent	false	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. Will by default use a memory based LRUCache that holds 1000 entries. If **`noop=true`** then idempotent will be enabled as well to avoid consuming the same files over and over again.
idempotentKey	Expression	**Camel 2.11:** To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: idempotentKey=${file:name}-${file:size}
idempotentRepository	null	A pluggable repository `org.apache.camel.spi.IdempotentRepository` which by default use **`MemoryMessageIdRepository`** if none is specified and **`idempotent`** is **`true`.**
inProgressRepository	memory	A pluggable in-progress repository `org.apache.camel.spi.IdempotentRepository` . The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.
filter	null	Pluggable filter as a **`org.apache.camel.component.file.GenericFileFilter`** class. Will skip files if filter returns **`false`** in its **`accept()`** method. More details in section below.

`filterDirectory`	`null`	**Camel 2.18**: Filters the directory based on Simple language. For example to filter on current date, you can use a simple date pattern such as **`${date:now:yyyMMdd}`.**
`filterFile`	`null`	**Camel 2.18:** Filters the file based on Simple language. For example to filter on file size, you can use **`${file}:size > 5000`**.
`shuffle`	`false`	**Camel 2.16:** To shuffle the list of files (sort in random order).
`sorter`	`null`	Pluggable sorter as a `java.util.Comparator<org.apache.camel.component.file.GenericFile>` class.
`sortBy`	`null`	Built-in sort using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date. See sorting section below for details.
`readLock`	`none`	Used by consumer, to only poll the files if it has exclusive read-lock on the file e.g., the file is not in-progress or being written. Camel will wait until the file lock is granted. This option provides the built-in strategies: • **`none`** is for no read locks at all. • **`markerFile`** Camel creates a marker file **`fileName.camelLock`** and then holds a lock on it. This option is **not** available for the FTP component. • **`changed`** is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec. to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option **`readLockCheckInterval`** can be used to set the check frequency. This option is **only** avail for the FTP component from **Camel 2.8** onward. Note: from **Camel 2.10.1** onward the FTP option **`fastExistsCheck`** can be enabled to speedup this **`readLock`** strategy, if the FTP server support the LIST operation with a full file name (some servers may not). • **`fileLock`** is for using **`java.nio.channels.FileLock`**. This option is **not** avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. • **`rename`** is for using a try to rename the file as a test if we can get exclusive read-lock. • **`idempotent` Camel 2.16** (only file component) is for using a **`idempotentRepository`** as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. • **`idempotent-changed` Camel 2.19** (only file component) is for using a **`idempotentRepository`** and changed as combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. • **`idempotent-rename` Camel 2.19** (only file component) is for using a **`idempotentRepository`** and rename as combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. **Warning**: most of the read lock strategies are not suitable for use in clustered mode. That is, you cannot have multiple consumers attempting to read the same file in the same directory. In this case, the read locks will not function reliably. The idempotent read lock supports clustered reliably if you use a cluster aware idempotent repository implementation such as from Hazelcast Component or Infinispan.
`readLockTimeout`	`10000`	Optional timeout in milliseconds for the **`readLock`**, if supported. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of **`0`** or lower to indicate forever. In **Camel 2.0** the default value is **`0`**. Starting with **Camel 2.1** the default value is **`10000`**. Currently **`fileLock`,** `changed` and **`rename`** support the timeout. **Note:** for FTP the default **`readLockTimeout`** value is **`20000`** instead of **`10000`**. The **`readLockTimeout`** value must be higher than **`readLockCheckInterval`**, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the **`readLockCheckInterval`**. This is needed to ensure that ample time is allowed for the read lock process to try to grab the lock before the timeout was hit.
`readLockCheckInterval`	`1000`	**Camel 2.6:** Interval in milliseconds for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the `changed` read lock, you can set a higher interval period to cater for *slow writes*. The default of 1 sec. may be *too fast* if the producer is very slow writing the file. For FTP the default **`readLockCheckInterval`** is **`5000`**. The **`readLockTimeout`** value must be higher than **`readLockCheckInterval`**, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the **`readLockCheckInterval`**. This is needed to ensure that ample time is allowed for the read lock process to try to grab the lock before the timeout was hit.
`readLockMinLength`	`1`	**Camel 2.10.1:** This option applied only for **`readLock=changed`**. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is **`1`**. You can set this option to zero, to allow consuming zero-length files.
`readLockMinAge`	`0`	**Camel 2.15**: This option applies only to **`readLock=change`**. This option allows you to specify a minimum age a file must be before attempting to acquire the read lock. For example, use **`readLockMinAge=300s`** to require that the file is at least 5 minutes old. This can speedup the poll when the file is old enough as it will acquire the read lock immediately. Notice for FTP: file timestamps reported by FTP servers are often reported with resolution of minutes, so **`readLockMinAge`** parameter should be defined in minutes, e.g. **60000** for 1 minute. Notice that Camel supports specifying this as **`60s`**, or **`1m`**, etc.

`readLockLoggingLevel`	`WARN`	**Camel 2.12:** Logging level used when a read lock could not be acquired. By default a **`WARN`** is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for the **`readLock`** types: • **`changed`** • **`fileLock`** • **`rename`**
`readLockMarkerFile`	`true`	**Camel 2.14:** Whether to use marker file with the **`changed`**, **`rename`**, or **`exclusive`** read lock types. By default a marker file is used as well to guard against other processes picking up the same files. This behavior can be turned off by setting this option to **`false`**. For example if you do not want to write marker files to the file systems by the Camel application.
`readLockRemoveOnRollback`	`true`	**Camel 2.16:** This option applied only for **`readLock=idempotent`**. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file failed and a rollback happens. If this option is false, then the file name entry is confirmed (as if the file did a commit).
`readLockRemoveOnCommit`	`false`	**Camel 2.16:** This option applied only for **`readLock=idempotent`**. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file succeeded and a commit happens. By default the file is not removed which ensures that any race-condition do not occur so another active node may attempt to grab the file. Instead the idempotent repository may support eviction strategies that you can configure to evict the file name entry after X minutes - this ensures no problems with race conditions.
`readLockDeleteOrphanLockFiles`	`true`	**Camel 2.16:** Whether or not read lock with marker files should upon startup delete any orphan read lock files, which may have been left on the file system, if Camel was not properly shutdown (such as a JVM crash). If turning this option to false then any orphaned lock file will cause Camel to not attempt to pickup that file, this could also be due another node is concurrently reading files from the same shared directory.
`directoryMustExist`	`false`	**Camel 2.5:** Similar to **`startingDirectoryMustExist`** but this applies during polling recursive sub directories.
`doneFileName`	`null`	**Camel 2.6:** If provided, Camel will only consume files if a *done* file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The *done* file is **always** expected in the same folder as the original file. See *using done file* and *writing done file* sections for examples.
`exclusiveReadLockStrategy`	`null`	Pluggable read-lock as a **`org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy`** implementation.
`maxMessagesPerPoll`	`0`	An integer to define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. **`1000`** to avoid when starting up the server that there are thousands of files. Set a value of **`0`** or negative to disable it. See more details at Batch Consumer. **Notice:** If this option is in use then the File and FTP components will limit **before** any sorting. For example if you have 100000 files and use **`maxMessagesPerPoll=500`**, then only the first 500 files will be picked up, and then sorted. You can use the **`eagerMaxMessagesPerPoll`** option and set this to **`false`** to allow to scan all files first and then sort afterwards.
`eagerMaxMessagesPerPoll`	`true`	**Camel 2.9.3:** Allows for controlling whether the limit from **`maxMessagesPerPoll`** is eager or not. If eager then the limit is during the scanning of files. Where as `false` would scan all files, and then perform sorting. Setting this option to `false` allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.
`minDepth`	`0`	**Camel 2.8**: The minimum depth to start processing when recursively processing a directory. Using **`minDepth=1`** means the base directory. Using **`minDepth=2`** means the first sub directory. This option is supported by FTP consumer from **Camel 2.8.2, 2.9** onward.
`maxDepth`	`Integer.MAX_VALUE`	**Camel 2.8:** The maximum depth to traverse when recursively processing a directory. This option is supported by FTP consumer from **Camel 2.8.2, 2.9** onward.
`processStrategy`	`null`	A pluggable **`org.apache.camel.component.file.GenericFileProcessStrategy`** allowing you to implement your own **`readLock`** option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special *ready* file exists. If this option is set then the **`readLock`** option does not apply.
`startingDirectoryMustExist`	`false`	**Camel 2.5:** Whether the starting directory must exist. Mind that the **`autoCreate`** option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable **`autoCreate`** and enable this to ensure the starting directory must exist. Will thrown an exception if the directory doesn't exist.

pollStrategy	null	A pluggable **org.apache.camel.spi.PollingConsumerPollStrategy** allowing you to provide your custom implementation to control error handling that may occur during the **poll** operation but *before* an Exchange has been created and routed by Camel. In other words the error occurred while the polling was gathering information e.g., access to a file network failed so Camel cannot access it to scan for files. The default implementation will log the caused exception at **WARN** level and ignore it.
sendEmptyMessageWhenIdle	false	**Camel 2.9:** If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
consumer.bridgeErrorHandler	false	**Camel 2.10:** Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while trying to pickup files, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the **org.apache.camel.spi.ExceptionHandler** to deal with exceptions, that by default will be logged at **WARN**/**ERROR** level and ignored. See the following section for more details: *How to use the Camel error handler to deal with exceptions triggered outside the routing engine*.
scheduledExecutorService	null	**Camel 2.10:** Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple file consumers.
scheduler	null	**Camel 2.12:** To use a custom scheduler to trigger the consumer to run. See more details at Polling Consumer, for example there is a Quartz2, and Spring based scheduler that supports CRON expressions.
backoffMultiplier	0	**Camel 2.12:** To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then **backoffIdleThreshold** and/or **backoffErrorThreshold** must also be configured. For more details see: Polling Consumer.
backoffIdleThreshold	0	**Camel 2.12:** The number of subsequent idle polls that should happen before the **backoffMultipler** should kick-in.
backoffErrorThreshold	0	**Camel 2.12:** The number of subsequent error polls (failed due some error) that should happen before the **backoffMultipler** should kick-in.
onCompletionExceptionHandler		**Camel 2.16:** To use a custom **org.apache.camel.spi.ExceptionHandler** to handle any thrown exceptions that happens during the file on completion process where the consumer does either a commit or rollback. The default implementation will log any exception at **WARN** level and ignore.
probeContentType	false	**Camel 2.17:** Whether to enable probing of the content type. If enable then the consumer uses **Files#probeContentType(java.nio.file.Path)** to determine the content-type of the file, and store that as a header with key **Exchange#FILE_CONTENT_TYPE** on the Message. **Camel 2.15**-**2.16.x** the default is true.
extendedAttributes	null	**Camel 2.17**: To enable gathering extended file attributes through **java.nio.file.attribute** classes using **Files.getAttribute(ava.nio.file.Path, java.lang.String attribute)** or **Files.readAttributes(ava.nio.file.Path, java.lang.String attributes)** depending on the option value. This option supports a comma delimited list of attributes to collect e.g., **basic:creationTime**, **posix:group** or simple wildcard e.g., **posix:***. If the attribute name is not prefixed, the basic attributes are queried. The result is stored as a header with key **CamelFileExtendedAttributes** and it is of type **Map<String, Object>** where the key is the name of the attribute e.g., **posix:group** and the value is the attributed returned by the call to **Files.getAttribute()** or **Files.readAttributes**.

Default behavior for file consumer

- By default the file is **not** locked for the duration of the processing.
- After the route has completed, files are moved into the **.camel** subdirectory, so that they appear to be deleted.
- The File Consumer will always skip any file whose name starts with a dot, such as **.**, **.camel**, **.m2** or **.groovy.**
- Only files (not directories) are matched for valid filename, if options such as: **include** or **exclude** are used.

Producer

confluenceTableSmall

Name	Default Value	Description

`fileExist`	`Override`	What to do if a file already exists with the same name. The following values can be specified: • **`Override`** replaces the existing file. • **`Append`** adds content to the existing file. • **`Fail`** throws a **`GenericFileOperationException`** indicating that there is already an existing file. • **`Ignore`** silently ignores the problem and **does not** override the existing file, but assumes everything is okay. • **`Move`** (**Camel 2.10.1** onward) requires that the option **`moveExisting`** be configured as well. The **`eagerDeleteTargetFile`** can be used to control what to do if moving the file, and there already exists a file, otherwise causing the move operation to fail. The **`Move`** option will move any existing files, before writing the target file. • **`TryRename`** **(Camel 2.11.1** onward) is only applicable if **`tempFileName`** option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.
`tempPrefix`	`null`	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written to and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.
`tempFileName`	`null`	**Camel 2.1:** The **same** as **`tempPrefix`** option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.
`moveExisting`	`null`	**Camel 2.10.1:** Expression (such as File Language) used to compute file name to use when **`fileExist=Move`** is configured. To move files into a **`backup`** subdirectory just enter **`backup`**. This option only supports the following File Language tokens: • **`file:name`** • **`file:name.ext`** • **`file:name.noext`** • **`file:onlyname`** • **`file:onlyname.noext`** • **`file:ext`** • **`file:parent`** **Note**: the **`file:parent`** token is not supported by the FTP component which can only move files to a directory relative to the *current* directory.
`keepLastModified`	`false`	**Camel 2.2:** Will keep the last modified timestamp from the source file (if any). Will use the **`Exchange.FILE_LAST_MODIFIED`** header to located the timestamp. This header can contain either a **`java.util.Date`** or **`long`** with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. **Note:** This option only applies to the **file** producer. It *cannot* be used with any of the FTP producers.
`eagerDeleteTargetFile`	`true`	**Camel 2.3:** Whether or not to eagerly delete any existing target file. This option only applies when you use **`fileExists=Override`** and the **`tempFileName`** option as well. You can use this to disable (set it to **`false`**) deleting the target file before the temp file is written. For example you may write big files and want the target file to exist while the temp file is being written. This ensures that the target file is only deleted at the very last moment, just before the temp file is being renamed to the target filename. From **Camel 2.10.1** onward this option is also used to control whether to delete any existing files when **`fileExist=Move`** is enabled, and an existing file exists. If this option **`copyAndDeleteOnRenameFail`** is **`false`**, then an exception will be thrown if an existing file existed. When **`true`** the existing file is deleted before the move operation.
`doneFileName`	`null`	**Camel 2.6:** If provided, then Camel will write a second file (called *done file*) when the original file has been written. The *done file* will be empty. This option configures what file name to use. You can either specify a fixed name, or you can use dynamic placeholders. The *done file* will **always** be written in the same folder as the original file. See *writing done file* section for examples.
`allowNullBody`	`false`	**Camel 2.10.1:** Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a **`GenericFileWriteException`** the a message 'Cannot write null body to file' will be thrown. If **`fileExist=Override`** the file will be truncated. If **`fileExist=append`** the file will remain unchanged.
`forceWrites`	`true`	**Camel 2.10.5/2.11:** Whether to force syncing writes to the file system. You can turn this off if you do not want this level of guarantee, for example if writing to logs / audit logs etc; this would yield better performance.
`chmod`	`null`	**Camel 2.15.0**: Specify the file permissions which is sent by the producer, the chmod value must be between **`000`** and **`777`**; If there is a leading digit like in **`0755`** we will ignore it.
`chmodDirectory`	`null`	**Camel 2.17.0:** Specify the directory permissions used when the producer creates missing directories, the chmod value must be between **`000`** and **`777`**; If there is a leading digit like in **`0755`** we will ignore it.

Default behavior for file producer

- By default it will override any existing file, if one exist with the same name.

Move and Delete operations

Any move or delete operations is executed after (post command) the routing has completed; so during processing of the **`Exchange`** the file is still located in the inbox folder.

Lets illustrate this with an example:

javafrom("file://inbox?move=.done") .to("bean:handleOrder");

When a file is dropped in the **`inbox`** folder, the file consumer notices this and creates a new **`FileExchange`** that is routed to the **`handleOrder`** bean. The bean then processes the **`File`** object. At this point in time the file is still located in the **`inbox`** folder. After the bean completes, and thus the route is completed, the file consumer will perform the move operation and move the file to the **`.done`** sub-folder.

The **`move`** and the **`preMove`** options are considered as a directory name though if you use an expression such as File Language, or Simple then the result of the expression evaluation is the file name to be used e.g., if you set

move=../backup/copy-of-${file:name}

then that's using the File Language which we use return the file name to be used), which can be either relative or absolute. If relative, the directory is created as a sub-folder from within the folder where the file was consumed.

By default, Camel will move consumed files to the **`.camel`** sub-folder relative to the directory where the file was consumed.

If you want to delete the file after processing, the route should be:

javafrom("file://inobox?delete=true") .to("bean:handleOrder");

We have introduced a **pre** move operation to move files **before** they are processed. This allows you to mark which files have been scanned as they are moved to this sub folder before being processed.

javafrom("file://inbox?preMove=inprogress") .to("bean:handleOrder");

You can combine the **pre** move and the regular move:

javafrom("file://inbox?preMove=inprogress&move=.done") .to("bean:handleOrder");

So in this situation, the file is in the **`inprogress`** folder when being processed and after it's processed, it's moved to the **`.done`** folder.

Fine Grained Control Using The `move` and `preMove` Options

The **move** and **preMove** options are Expression-based, so we have the full power of the File Language to do advanced configuration of the directory and name pattern.
Camel will, in fact, internally convert the directory name you enter into a File Language expression. So when we enter **`move=.done`** Camel will convert this into: **`${file:parent}/.done/${file:onlyname}`**. This is only done if Camel detects that you have not provided a **${}** in the option value yourself. So when you enter a **${}** Camel will **not** convert it and thus you have the full power.

So if we want to move the file into a backup folder with today's date as the pattern, we can do:

move=backup/${date:now:yyyyMMdd}/${file:name}

About `moveFailed`

The **`moveFailed`** option allows you to move files that **could not** be processed successfully to another location such as a error folder of your choice. For example to move the files in an error folder with a timestamp you can use **`moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${file:ext}`**.

See more examples at File Language

Message Headers

The following headers are supported by this component:

File producer only

confluenceTableSmall

Header	Description
`CamelFileName`	Specifies the name of the file to write (relative to the endpoint directory). This name can be a `String`; a `String` with a File Language or Simple expression; or an Expression object. If it's **`null`** then Camel will auto-generate a filename based on the message unique ID.
`CamelFileNameProduced`	The absolute file path (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.

`CamelOverruleFileName`	**Camel 2.11:** Is used for overruling `CamelFileName` header and use the value instead (but only once, as the producer will remove this header after writing the file). The value can be only be a String. Notice that if the option **`fileName`** has been configured, then this is still being evaluated.

File consumer only

confluenceTableSmall

Header	Description
`CamelFileName`	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
`CamelFileNameOnly`	Only the file name (the name with no leading paths).
`CamelFileAbsolute`	A `boolean` option specifying whether the consumed file denotes an absolute path or not. Should normally be `false` for relative paths. Absolute paths should normally not be used but we added to the move option to allow moving files to absolute paths. But can be used elsewhere as well.
`CamelFileAbsolutePath`	The absolute path to the file. For relative files this path holds the relative path instead.
`CamelFilePath`	The file path. For relative files this is the starting directory + the relative filename. For absolute files this is the absolute path.
`CamelFileRelativePath`	The relative path.
`CamelFileParent`	The parent path.
`CamelFileLength`	A `long` value containing the file size.
`CamelFileLastModified`	A `Long` value containing the last modified timestamp of the file. In **Camel 2.10.3 and older** the type is `Date`.

Batch Consumer

This component implements the Batch Consumer.

Exchange Properties, file consumer only

As the file consumer implements the **`BatchConsumer`** it supports batching the files it polls. By batching we mean that Camel will add the following additional properties to the Exchange, so you know the number of files polled, the current index, and whether the batch is already completed.

confluenceTableSmall

Property	Description
`CamelBatchSize`	The total number of files that was polled in this batch.
`CamelBatchIndex`	The current index of the batch. Starts from 0.
`CamelBatchComplete`	A `boolean` value indicating the last Exchange in the batch. Is only `true` for the last entry.

This allows you for instance to know how many files exist in this batch and for instance let the Aggregator2 aggregate this number of files.

Using `charset`

Available as of Camel 2.9.3
The **`charset`** option allows for configuring an encoding of the files on both the consumer and producer endpoints. For example if you read utf-8 files, and want to convert the files to iso-8859-1, you can do:

from("file:inbox?charset=utf-8") .to("file:outbox?charset=iso-8859-1")

You can also use the **`convertBodyTo`** in the route. In the example below we have still input files in utf-8 format, but we want to convert the file content to a byte array in iso-8859-1 format. And then let a bean process the data. Before writing the content to the outbox folder using the current charset.

from("file:inbox?charset=utf-8") .convertBodyTo(byte[].class, "iso-8859-1") .to("bean:myBean") .to("file:outbox");

If you omit the charset on the consumer endpoint, then Camel does not know the charset of the file, and would by default use "UTF-8". However you can configure a JVM system property to override and use a different default encoding with the key **`org.apache.camel.default.charset`**.

In the example below this could be a problem if the files is not in UTF-8 encoding, which would be the default encoding for read the files.
In this example when writing the files, the content has already been converted to a byte array, and thus would write the content directly as is (without any further encodings).

from("file:inbox") .convertBodyTo(byte[].class, "iso-8859-1") .to("bean:myBean") .to("file:outbox");

You can also override and control the encoding dynamic when writing files, by setting a property on the exchange with the key **`Exchange.CHARSET_NAME`**.
For example in the route below we set the property with a value from a message header.

from("file:inbox") .convertBodyTo(byte[].class, "iso-8859-1") .to("bean:myBean") .setProperty(Exchange.CHARSET_NAME, header ("someCharsetHeader")) .to("file:outbox");

We suggest to keep things simpler, so if you pickup files with the same encoding, and want to write the files in a specific encoding, then favor to use the **`charset`** option on the endpoints.

Notice that if you have explicit configured a **`charset`** option on the endpoint, then that configuration is used, regardless of the **`Exchange.CHARSET_NAME`** property.

If you have some issues then you can enable **`DEBUG`** logging on **`org.apache.camel.component.file`**, and Camel logs when it reads/write a file using a specific charset.
For example the route below will log the following:

from("file:inbox?charset=utf-8") .to("file:outbox?charset=iso-8859-1")

And the logs:

DEBUG GenericFileConverter - Read file /Users/davsclaus/workspace/camel/camel-core/target/charset/input/input.txt with charset utf-8 DEBUG FileOperations - Using Reader to write file: target/charset/output.txt with charset: iso-8859-1

Common gotchas with folder and filenames

When Camel is producing files (writing files) there are a few gotchas affecting how to set a filename of your choice. By default, Camel will use the message ID as the filename, and since the message ID is normally a unique generated ID, you will end up with filenames such as: **`ID-MACHINENAME-2443-1211718892437-1-0`**. If such a filename is not desired, then you must provide a filename in the **`CamelFileName`** message header. The constant, **`Exchange.FILE_NAME`**, can also be used.

The sample code below produces files using the message ID as the filename:

from("direct:report") .to("file:target/reports");

To use **`report.txt`** as the filename you have to do:

from("direct:report") .setHeader(Exchange.FILE_NAME, constant("report.txt")) .to("file:target/reports");

... the same as above, but with **`CamelFileName:`**

from("direct:report") .setHeader("CamelFileName", constant("report.txt")) .to("file:target/reports");

And a syntax where we set the filename on the endpoint with the **`fileName`** URI option.

from("direct:report") .to("file:target/reports/?fileName=report.txt");

Filename Expression

Filename can be set either using the **expression** option or as a string-based File Language expression in the **`CamelFileName`** header. See the File Language for syntax and samples.

Consuming files from folders where others drop files directly

Beware if you consume files from a folder where other applications write files to directly. Take a look at the different **`readLock`** options to see what suits your use cases. The best approach is however to write to another folder and after the write move the file in the drop folder. However if you write files directly to the drop folder then the option changed could better detect whether a file is currently being written/copied as it uses a file changed algorithm to see whether the file size / modification changes over a period of time. The other **`readLock`** options rely on Java File API that sadly is not always very good at detecting this. You may also want to look at the **`doneFileName`** option, which uses a marker file (done file) to signal when a file is done and ready to be consumed.

Using 'done' Files

Available as of Camel 2.6

See also section *writing done files* below.

If you want only to consume files when a done file exists, then you can use the **`doneFileName`** option on the endpoint.

javafrom("file:bar?doneFileName=done");

Will only consume files from the bar folder, if a done *file* exists in the same directory as the target files. Camel will automatically delete the *done file* when it's done consuming the files. From Camel **2.9.3** onward Camel will not automatically delete the *done file* if **noop=true** is configured.

However it is more common to have one *done file* per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the **doneFileName** option. Currently Camel supports the following two dynamic tokens: **file:name** and **file:name.noext** which must be enclosed in **${}**. The consumer only supports the static part of the *done file* name as either prefix or suffix (not both).

javafrom("file:bar?doneFileName=${file:name}.done");

In this example only files will be polled if there exists a done file with the name *file name*.done. For example

- **hello.txt** - is the file to be consumed
- **hello.txt.done** - is the associated done file

You can also use a prefix for the done file, such as:

javafrom("file:bar?doneFileName=ready-${file:name}");

- **hello.txt** - is the file to be consumed
- **ready-hello.txt** - is the associated done file

Writing 'done' Files

Available as of Camel 2.6

After you have written a file you may want to write an additional *done file* as a kind of marker, to indicate to others that the file is finished and has been written. To do that you can use the **doneFileName** option on the file producer endpoint.

java.to("file:bar?doneFileName=done");

Will simply create a file named `done` in the same directory as the target file.

However it is more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the **doneFileName** option. Currently Camel supports the following two dynamic tokens: **file:name** and **file:name.noext** which must be enclosed in **${}**.

java.to("file:bar?doneFileName=done-${file:name}");

Will for example create a file named **done-foo.txt** if the target file was **foo.txt** in the same directory as the target file.

java.to("file:bar?doneFileName=${file:name}.done");

Will for example create a file named **foo.txt.done** if the target file was **foo.txt** in the same directory as the target file.

java.to("file:bar?doneFileName=${file:name.noext}.done");

Will for example create a file named **foo.done** if the target file was **foo.txt** in the same directory as the target file.

Examples

Read from a directory and write to another directory

javafrom("file://inputdir/?delete=true") .to("file://outputdir")

Read from a directory and write to another directory using a overrule dynamic name

javafrom("file://inputdir/?delete=true") .to("file://outputdir?overruleFile=copy-of-${file:name}")

Listen on a directory and create a message for each file dropped there. Copy the contents to the **outputdir** and delete the file in the **inputdir.**

Reading recursively from a directory and writing to another

javafrom("file://inputdir/?recursive=true&delete=true") .to("file://outputdir")

Listen on a directory and create a message for each file dropped there. Copy the contents to the **outputdir** and delete the file in the **inputdir**. Will scan recursively into sub-directories. Will lay out the files in the same directory structure in the **outputdir** as the **inputdir**, including any sub-directories.

inputdir/foo.txt inputdir/sub/bar.txt

Will result in the following output layout:

outputdir/foo.txt outputdir/sub/bar.txt

Using `flatten`

If you want to store the files in the **outputdir** directory in the same directory, disregarding the source directory layout e.g., to flatten out the path, you just add the **flatten=true** option on the file producer side:

javafrom("file://inputdir/?recursive=true&delete=true") .to("file://outputdir?flatten=true")

Will result in the following output layout:

outputdir/foo.txt outputdir/bar.txt

Reading from a directory and the default move operation

Camel will by default move any processed file into a **`.camel`** subdirectory in the directory the file was consumed from.

javafrom("file://inputdir/?recursive=true&delete=true") .to("file://outputdir")

Affects the layout as follows:
before

inputdir/foo.txt inputdir/sub/bar.txt

after

inputdir/.camel/foo.txt inputdir/sub/.camel/bar.txt outputdir/foo.txt outputdir/sub/bar.txt

Read from a directory and process the message in java

from("file://inputdir/").process(new Processor() { public void process(Exchange exchange) throws Exception { Object body = exchange.getIn().getBody(); // do some business logic with the input body } });

The body will be a **`File`** object that points to the file that was just dropped into the **`inputdir`** directory.

Writing to files

Camel is of course also able to write files, i.e. produce files. In the sample below we receive some reports on the SEDA queue that we process before they are being written to a directory.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/ToFileRouteTest.java}

Write to subdirectory using `Exchange.FILE_NAME`

Using a single route, it is possible to write a file to any number of subdirectories. If you have a route setup as such:

xml <route> <from uri="bean:myBean"/> <to uri="file:/rootDirectory"/> </route>

You can have **`myBean`** set the header **`Exchange.FILE_NAME`** to values such as:

Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt

This allows you to have a single route to write files to multiple destinations.

Writing file through the temporary directory relative to the final destination

Sometime you need to temporarily write the files to some directory relative to the destination directory. Such situation usually happens when some external process with limited filtering capabilities is reading from the directory you are writing to. In the example below files will be written to the **`/var/myapp/filesInProgress`** directory and after data transfer is done, they will be atomically moved to the **`/var/myapp/finalDirectory`** directory.

javafrom("direct:start") .to("file:///var/myapp/finalDirectory?tempPrefix=/../filesInProgress/");

Using Expressions for Filenames

In this sample we want to move consumed files to a backup folder using today's date as a sub-folder name:

javafrom("file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}") .to("...");

See File Language for more samples.

Avoiding reading the same file more than once (idempotent consumer)

Camel supports Idempotent Consumer directly within the component so it will skip already processed files. This feature can be enabled by setting the **`idempotent=true`** option.

javafrom("file://inbox?idempotent=true") .to("...");

Camel uses the absolute file name as the idempotent key, to detect duplicate files. From **Camel 2.11** onward you can customize this key by using an expression in the **`idempotentKey`** option. For example to use both the name and the file size as the key

xml <route> <from uri="file://inbox?idempotent=true&idempotentKey=${file:name}-${file:size}"/> <to uri="bean:processInbox"/> </route>

By default Camel uses a in memory based store for keeping track of consumed files, it uses a least recently used cache holding up to 1000 entries. You can plugin your own implementation of this store by using the **`idempotentRepository`** option using the # sign in the value to indicate it's a referring to a bean in the Registry with the specified **`id`**.

xml <!-- define our store as a plain spring bean --> <bean id="myStore" class="com.mycompany.MyIdempotentStore"/> <route> <from uri="file://inbox?idempotent=true&idempotentRepository=#myStore"/> <to uri="bean:processInbox"/> </route>

Camel will log at **DEBUG** level if it skips a file because it has been consumed before:

DEBUG FileConsumer is idempotent and the file has been consumed before. Will skip this file: target\idempotent\report.txt

Using a file based idempotent repository

In this section we will use the file based idempotent repository **org.apache.camel.processor.idempotent.FileIdempotentRepository** instead of the in-memory based that is used as default.
This repository uses a 1st level cache to avoid reading the file repository. It will only use the file repository to store the content of the 1st level cache. Thereby the repository can survive server restarts. It will load the content of the file into the 1st level cache upon startup. The file structure is very simple as it stores the key in separate lines in the file. By default, the file store has a size limit of 1mb. When the file grows larger Camel will truncate the file store, rebuilding the content by flushing the 1st level cache into a fresh empty file.

We configure our repository using Spring XML creating our file idempotent repository and define our file consumer to use our repository with the **idempotentRepository** using # sign to indicate Registry lookup:{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/idempotent/fileConsumerIdempotentTest.xml}

Using a JPA based idempotent repository

In this section we will use the JPA based idempotent repository instead of the in-memory based that is used as default.

First we need a persistence-unit in **META-INF/persistence.xml where we need to use the class org.apache.camel.processor.idempotent.jpa.MessageProcessed** as model.{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-jpa/src/test/resources/META-INF/persistence.xml}Next, we can create our JPA idempotent repository in the spring XML file as well:{snippet:id=jpaStore|lang=xml|url=camel/trunk/components/camel-jpa/src/test/resources/org/apache/camel/processor/jpa/fileConsumerJpaIdempotentTest-config.xml}And yes then we just need to refer to the **jpaStore** bean in the file consumer endpoint using the **idempotentRepository** using the # syntax option:

xml <route> <from uri="file://inbox?idempotent=true&idempotentRepository=#jpaStore"/> <to uri="bean:processInbox"/> </route>

Filter using `org.apache.camel.component.file.GenericFileFilter`

Camel supports pluggable filtering strategies. You can then configure the endpoint with such a filter to skip certain files being processed.

In the sample we have built our own filter that skips files starting with **skip** in the filename:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerFileFilterTest.java}And then we can configure our route using the **filter** attribute to reference our filter (using # notation) that we have defined in the spring XML file:

xml <!-- define our filter as a plain spring bean --> <bean id="myFilter" class="com.mycompany.MyFileFilter"/> <route> <from uri="file://inbox?filter=#myFilter"/> <to uri="bean:processInbox"/> </route>

Filtering using ANT path matcher

New options from Camel 2.10 onwards
There are now **antInclude** and **antExclude** options to make it easy to specify ANT style include/exclude without having to define the filter. See the URI options above for more information.

The ANT path matcher is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven.
The reasons is that we leverage Spring's AntPathMatcher to do the actual matching.

The file paths is matched with the following rules:

- ? matches one character
- * matches zero or more characters
- ** matches zero or more directories in a path

The sample below demonstrates how to use it:{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/file/SpringFileAntPathMatcherFileFilterTest-context.xml}

Sorting using `Comparator`

Camel supports pluggable sorting strategies. This strategy it to use the build in **java.util.Comparator** in Java. You can then configure the endpoint with such a comparator and have Camel sort the files before being processed.

In the sample we have built our own comparator that just sorts by file name:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileSorterRefTest.java}And then we can configure our route using the **sorter** option to reference to our sorter (`mySorter`) we have defined in the spring XML file:

xml <!-- define our sorter as a plain spring bean --> <bean id="mySorter" class="com.mycompany.MyFileSorter"/> <route> <from uri="file://inbox?sorter=#mySorter"/> <to uri="bean:processInbox"/> </route> URI options can reference beans using the # syntax
In the Spring DSL route above notice that we can refer to beans in the Registry by prefixing the id with **#**. So writing **sorter=#mySorter**, will instruct Camel to go look in the Registry for a bean with the ID, **mySorter**.

Sorting using `sortBy`

Camel supports pluggable sorting strategies. This strategy it to use the File Language to configure the sorting. The **sortBy** option is configured as follows:

sortBy=group 1;group 2;group 3;...

Where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:

sortBy=file:name

This will sort by file name, you can reverse the order by prefixing **`reverse:`** to the group, so the sorting is now Z..A:

sortBy=reverse:file:name

As we have the full power of File Language we can use some of the other parameters, so if we want to sort by file size we do:

sortBy=file:length

You can configure to ignore the case, using **`ignoreCase:`** for string comparison, so if you want to use file name sorting but to ignore the case then we do:

sortBy=ignoreCase:file:name

You can combine ignore case and reverse, however reverse must be specified first:

sortBy=reverse:ignoreCase:file:name

In the sample below we want to sort by last modified file, so we do:

sortBy=file:modified

And then we want to group by name as a 2nd option so files with same modifcation is sorted by name:

sortBy=file:modified;file:name

Now there is an issue here, can you spot it? Well the modified timestamp of the file is too fine as it will be in milliseconds, but what if we want to sort by date only and then subgroup by name?
Well as we have the true power of File Language we can use its date command that supports patterns. So this can be solved as:

sortBy=date:file:yyyyMMdd;file:name

Yeah, that is pretty powerful, oh by the way you can also use reverse per group, so we could reverse the file names:

sortBy=date:file:yyyyMMdd;reverse:file:name

Using `GenericFileProcessStrategy`

The option **`processStrategy`** can be used to use a custom **`GenericFileProcessStrategy`** that allows you to implement your own *begin*, *commit* and *rollback* logic.
For instance lets assume a system writes a file in a folder you should consume. But you should not start consuming the file before another *ready* file has been written as well.

So by implementing our own **`GenericFileProcessStrategy`** we can implement this as:

- In the **`begin()`** method we can test whether the special *ready* file exists. The begin method returns a `boolean` to indicate if we can consume the file or not.
- In the **`abort()`** method (**Camel 2.10**) special logic can be executed in case the **`begin`** operation returned **`false`**, for example to cleanup resources etc.
- In the **`commit()`** method we can move the actual file and also delete the *ready* file.

Using `filter`

The **`filter`** option allows you to implement a custom filter in Java code by implementing the **`org.apache.camel.component.file.GenericFileFilter`** interface. This interface has an **`accept`** method that returns a boolean. Return **`true`** to include the file, and **`false`** to skip the file. From Camel 2.10 onward, there is a **`isDirectory`** method on **`GenericFile`** whether the file is a directory. This allows you to filter unwanted directories, to avoid traversing down unwanted directories.

For example to skip any directories which starts with `"skip"` in the name, can be implemented as follows:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerDirectoryFilterTest.java}

How to use the Camel error handler to deal with exceptions triggered outside the routing engine

The file and ftp consumers, will by default try to pickup files. Only if that is successful then a Camel Exchange can be created and passed in the Camel routing engine. When the Exchange is processed by the routing engine, then the Camel Error Handling takes over e.g., the **`onException`** / **`errorHandler`** in the routes. However outside the scope of the routing engine, any exceptions handling is component specific. Camel offers a **`org.apache.camel.spi.ExceptionHandler`** that allows components to use that as a pluggable hook for end users to use their own implementation. Camel offers a default **`LoggingExceptionHandler`** that will log the exception at **`ERROR/WARN`** level.

For the file and ftp components this would be the case. However if you want to bridge the **`ExceptionHandler`** so it uses the Camel Error Handling, then you need to implement a custom **`ExceptionHandler`** that will handle the exception by creating a Camel Exchange and send it to the routing engine; then the error handling of the routing engine can get triggered.

Easier with Camel 2.10
The new option **`consumer.bridgeErrorHandler`** can be set to true, to make this even easier. See further below for more details.

Here is such an example based upon an unit test.

First we have a custom **`ExceptionHandler`** where you can see we deal with the exception by sending it to a Camel Endpoint named **`direct:file-error`**:{snippet:id=e1|title=MyExceptionHandler|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerCustomExceptionHandlerTest.java}

Then we have a Camel route that uses the Camel routing error handler, which is the **`onException`** where we handle any **`IOException`** being thrown. We then send the message to the same **`direct:file-error`** endpoint, where we handle it by transforming it to a message, and then being sent to a Mock endpoint. This is just for testing purpose. You can handle the exception in any custom way you want, such as using a Bean or sending an email, etc.

Notice how we configure our custom **`MyExceptionHandler`** by using the **`consumer.exceptionHandler`** option to refer to **`#myExceptionHandler`** which is a id of the bean registered in the Registry. If using Spring XML or OSGi Blueprint, then that would be a **`<bean id="myExceptionHandler" class="com.foo.MyExceptionHandler"/>`**:{snippet:id=e2|title=Camel route with routing engine error handling|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerCustomExceptionHandlerTest.java}

The source code for this example can be seen here

Using `consumer.bridgeErrorHandler`

Available as of Camel 2.10

If you want to use the Camel Error Handler to deal with any exception occurring in the file consumer, then you can enable the **`consumer.bridgeErrorHandler`** option as shown below:{snippet:id=e2|title=Using consumer.bridgeErrorHandler|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/file/FileConsumerBridgeRouteExceptionHandlerTest.java}So all you have to do is to enable this option, and the error handler in the route will take it from there.

Important when using consumer.bridgeErrorHandler
When using **`consumer.bridgeErrorHandler`**, then interceptors, OnCompletions does **not** apply. The Exchange is processed directly by the Camel Error Handler, and does not allow prior actions such as interceptors, **`onCompletion`** to take action.

Debug logging

This component has log level **`TRACE`** that can be helpful if you have problems.

Endpoint See Also

- File Language
- FTP
- Polling Consumer