IO Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

List of Examples

Chapter 1. Introduction

The Fabric Engine high-performance computing platform would not be complete without the ability to read and write external data. Fabric IO collectively refers to a set of features that allows interaction with external data, including both local files and files obtained over HTTP, in the context of both KL programs and Fabric clients. Fabric IO is a mix of Fabric client functions, specialized dependency graph features and functionality provided by Fabric extensions. An important feature of Fabric IO is a secure IO model for the browser plugin client.

Chapter 2. FabricFileHandle

A FabricFileHandle is a KL `String` which represents a local file or folder, along with a "writable" attribute. Depending on the client security model, a FabricFileHandle can be either a local file path (standalone or encoded as a URI) or a special URI that referes to a local file resource through a secure "proxy" handle.

2.1. Security model

The enforcement of local file system security and privacy by Fabric depends on the client type. In the context of an end-user browser plugin, Fabric IO will only grant access to the files and folders which have been explicitely selected by the user, along with its permission to modify them. However, this restriction does not apply to the Python and Node.js Fabric clients, and can be bypassed by trusted browser plugin configurations by means of the Fabric FILESYSTEM extension. In order to support both restricted and unrestricted local file system access, Fabric IO features share the FabricFileHandle concept that permits this abstraction.

2.2. Forms

Depending on the security model, a FabricFileHandle any of the following forms:

`fabricio://`*`[encoded random bytes]`*	A URI-like cryptic String in the context of a secure browser plugin client. The content of the FabricFileHandle is generated by Fabric, and is associated to the actual file or folder path and attributes. Example: `fabricio://aWpv1PBn6caEMVFEBAJ+YF7Jj34FamQc` In the case of a writable folder, it is possible to append a relative path at the end of the FabricFileHandle, in which case it takes the form `fabricio://`*`[encoded random bytes][relative path]`*. Example: `fabricio://aWpv1PBn6caEMVFEBAJ+YF7Jj34FamQc/images/portrait.jpeg`
`file://`*`[local file path]`	A local file or folder URI, valid in the context of a trusted client. Example: `file://C/images/portrait.jpeg` (Windows), `file://~/images/portrait.jpeg` (Mac OS X/Linux)
`[local file path]`	Direct local file or folder path, valid in the context of a trusted client. Example: `C:\images\portrait.jpeg` (Windows), `~/images/portrait.jpeg` (Mac OS X/Linux)

2.3. Associated attributes

When a FabricFleHandle is constructed, it is associated with the following attributes:

local path	A local filesystem path.
writeAccess attribute	If writable, Fabric IO functions are allowed to modify the content of the file or folder.
folder attribute	A FabricFileHandle represents either a file or a folder. This distinction is made since the ability to change the contents of a folder has different security implications from the ability to change the contents of a file.

- A FabricFileHandle is valid only within the Fabric context in which it is built, and it remains valid until that context is destroyed.

- It is valid to build a FabricFileHandle representing a non-existing file or folder. For example, it can be used to store the path of a file that will be created in the future.

Chapter 3. The Dependency Graph ResourceLoadNode

The ResourceLoadNode is a specialized dependency graph Node that enables reading of external data asynchronously. The source data location is specified in its `url` member (of type `String`) which can contain either an HTTP URL (currently supported only by the browser plugin client) or a FabricFileHandle. Once available, the ResourceLoadNode's retrieved external data is stored in its `resource` member (of type `FabricResource`) and a notification callback is issued; details are discussed below.

3.1. Data storage

Depending on ResourceLoadNode's `storeDataAsFile` member value, the retrieved data can either be stored in a memory buffer or in a local file. In either case, the results are stored as part of the `resource` member, which is of type `FabricResource` and contains the following fields:

`data`	A `Byte` variable-length array containing the retrieved data. Only valid if ResourceLoadNode's `storeDataAsFile` member is `false` (the default).
`dataExternalLocation`	A FabricFileHandle (`String`) associated with a local fila containing the retrieved data. Only valid if ResourceLoadNode's `storeDataAsFile` member is `true`.
`url`	A `String` containing a copy of ResourceLoadNode's `url` member (source data URI).
`extension`	A `String` containing the source file's extension (or a MIME type's subtype from an HTTP request). Example: `jpeg`
`mimeType`	A `String` containing the source's full MIME type. Example: `image/jpeg`

3.2. Fabric Client Interface

A ResourceLoadNode is created using a Fabric client's `DG.createResourceLoadNode(`*`name`*`)` function. Once the `url` member of the ResourceLoadNode is set, the first evaluation of the node will issue a request for the external data; this call may return before the data is retrieved (ie. asynchronously). The `resource` member's data will remain empty until the data is fully retrieved. Once the data is set into the `resource` member, the dependency graph will not re-evaluate automatically; instead, notifications will be sent to the fabric client. The Fabric client's ResourceLoadNode provides the following methods to register for these notifications:

`addOnLoadSuccessCallback`	Registers a callback function on the ResourceLoadNode that will be triggered if the data is retrived successfully. The callback has takes parameter: the source ResourceLoadNode.
`addOnLoadFailureCallback`	Registers a callback function on the ResourceLoadNode that will be triggered if there is an error retrieving the data. The callback takes one parameter: the source ResourceLoadNode.
`addOnLoadProgressCallback`	Registers a callback function on the ResourceLoadNode that will be triggered a few times per second to provide progress information before the `addOnLoadSuccessCallback`- or `addOnLoadFailureCallback`- callback is triggered. The callback takes two parameters: the source ResourceLoadNode and a (dynamic language) object containing a `total` field and a `received` field, respectively the total number of bytes to be retrieved and the current retrived amount.

To complement the ResourceLoadNode features, each Node created through the dependency graph has a `putResourceToFile(fabricFileHandle, member)` method that saves the data from a `FabricResource` type member (such as ResourceLoadNode's `resource` member) to a local file associated with a FabricFileHandle.

3.2.1. Example

The following code is an example of using a ResourceLoadNode in the context of a Node.js Fabric client. In order to trace both the client and the KL runtime aspect of ResourceLoadNode's behavior, an operator is bound to the ResourceLoadNode to provide a trace, and the client registers to the ResourceLoadNode notification callbacks. Although the example was built for the Node.js Fabric client, it is relevant to other Fabric clients types such as Python.

Example 3.1. Building and using a ResourceLoadNode

```
FABRIC = require('Fabric').createClient();
node = FABRIC.DependencyGraph.createResourceLoadNode("node");

//Add an operator to the ResourceLoadNode so we can trace its runtime behavior
op = FABRIC.DependencyGraph.createOperator("loadop");
op.setEntryPoint("onLoad");
op.setSourceCode('\
operator onLoad( io FabricResource resource )\n\
{\n\
  if( resource.dataExternalLocation != "" || resource.data.size() != 0 ) {\n\
    report("Loaded " + resource.url);\n\
    if( resource.dataExternalLocation )\n\
      report( "Data loaded into file handle " + resource.dataExternalLocation );\n\
    else\n\
      report( "Data loaded in data buffer of size " + resource.data.size() );\n\
  }\n\
  else\n\
    report("Data not loaded yet");\n\
}\n\
');
binding = FABRIC.DG.createBinding();
binding.setOperator(op);
binding.setParameterLayout([
  "self.resource"
]);
node.bindings.append(binding);

//Trace the progress from the client side by registering to notification callbacks
node.addOnLoadFailureCallback( function( srcNode ) {
  console.log( 'Error loading ' + srcNode.getData('url', 0) );
  FABRIC.close();
});

node.addOnLoadProgressCallback( function( srcNode, progressInfo ) {
  console.log( 'Load progress for ' + srcNode.getData('url', 0) + ' : total = ' +
 progressInfo.total + ', received = ' + progressInfo.received );
});

node.addOnLoadSuccessCallback( function( srcNode ) {
  node.evaluate();//Ensure we reevaluate now that the data has loaded
  console.log( 'Loading of ' + srcNode.getData('url', 0) + ' succeeded' );
  var targetFileHandle = FABRIC.IO.buildFileHandleFromRelativePath('file://./loadedImage.jpeg');
  srcNode.putResourceToFile(targetFileHandle, 'resource');
  console.log( 'Loaded data written to file ./loadedImage.jpeg' );
  FABRIC.close();
});

node.setData('url',0,'file://image.jpeg');
node.evaluate();
```


Output:

```
[FABRIC] [MT] Data not loaded yet
[FABRIC] [MT] Loaded file://image.jpeg
[FABRIC] [MT] Data loaded in data buffer of size 829667
Loading of file://image.jpeg succeeded
Loaded data written to file ./loadedImage.jpeg
```


Chapter 4. Fabric.IO Helper Functions

A Fabric client's `IO` module contains several helper functions to support the FabricFileHandle concept. They are most useful for the JavaScript browser plugin client (NPAPI) since they use FabricFileHandle's secure form for their results. The following helper functions are available and described later:

- IO.queryUserFileHandle
- IO.queryUserFileAndFolderHandle
- IO.buildFileHandleFromRelativePath
- IO.buildFolderHandleFromRelativePath
- IO.getFileHandleInfo
- IO.getTextFileContent
- IO.putTextFileContent

Example 4.1. Fabric.IO functions example

This example illustrates the use of most of the Fabric.IO helper functions, in the context of a browser plugin Fabric client (secure model).

```
var handles = FABRIC.IO.queryUserFileAndFolderHandle( FABRIC.IO.forSave, "Fabric IO text file
 test", "txt", "testfile" );
console.log( "Returned file & folder cryptic file handles (random): " + JSON.stringify( handles ) );

FABRIC.IO.putTextFileContent( handles.file, "Test file body", false );
console.log( "File content: " + FABRIC.IO.getTextFileContent( handles.file ) );

var selectedFileInfo = FABRIC.IO.getFileHandleInfo( handles.file );
console.log( "File info: " + JSON.stringify( selectedFileInfo ) );

//In this example we requested the permission to write to the folder. Let's add another file:
var otherFileHandle = FABRIC.IO.buildFileHandleFromRelativePath( handles.folder + '/Other' +
 selectedFileInfo.fileName );

//Append some text content to Other file
FABRIC.IO.putTextFileContent( otherFileHandle, FABRIC.IO.getTextFileContent( handles.file ), false );
FABRIC.IO.putTextFileContent( otherFileHandle, " with appended content", true );
var otherContent = FABRIC.IO.getTextFileContent( otherFileHandle );
console.log( "Other file content: " + FABRIC.IO.getTextFileContent( otherFileHandle ) );
```

Output:

```
Returned file & folder cryptic file handles (random): {"file":"fabricio://
enXGaI42F3rAmT9hE3j8U8oPUnShOyIa","folder":"fabricio://EqJs7FBMvWuRzQj8ob5BXuW86/hAnj8I"}
File content: Test file body
File info: {"type":"file","writeAccess":true,"exists":true,"fileName":"testfile.txt","fileSize":14}
Other file content: Test file body with appended content
```

4.1. IO.queryUserFileHandle

```
FabricFileHandle queryUserFileHandle(mode,
                                     uiTitle,
                                     extension,
                                     defaultFileName);
```

[browser plugin client only] Opens a file browser allowing the user to select a local file, which is returned as a FabricFileHandle. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *mode* : Access mode requested for the selected file. It can be one of the following:
  - `IO.forOpen` : open an existing file as read-only
  - `IO.forOpenWithWriteAccess` : open an existing file with read-write access
  - `IO.forSave` : creates a new file (or overwrites an exiting one) with write access
- *uiTitle* : *[optional]* title for the file browser dialog window. Note that for clarity and security reasons, Fabric will automatically be prepend it with "Open File: " (*mode* = `IO.forOpen`), "Save File: " (*mode* = `IO.forSave`) or "Open File with write access: " (*mode* = `IO.forOpenWithWriteAccess`).
- *extension* : *[optional]* default extension for the files to be opened or saved, for example: “*.jpeg”
- *defaultFileName* : *[optional]* default file name the file to be opened or saved, for example: “image.jpeg”
- *Returned value* : a FabricFileHandle associated to the selected file. If *mode* = `IO.forSave`, the file might not have beeen created yet.

4.2. IO.queryUserFileAndFolderHandle

```
Object queryUserFileAndFolderHandle(mode,
                                    uiTitle,
                                    extension,
                                    defaultFileName);
```

[browser plugin client only] Opens a file browser allowing the user to select both local file and its folder, which are returned as an object containing a FabricFileHandle for the file and another one for the folder. Note that the file browser title will warn the user that a premission to read or write the folder is requested. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *mode* : Access mode requested for the selected file and folder. It can be one of the following:
  - `IO.forOpen` : open an existing file and folder as read-only
  - `IO.forOpenWithWriteAccess` : open an existing file and folder with read-write access
  - `IO.forSave` : creates a new file (or overwrites an exiting one) with write access to it and its folder
- *uiTitle* : *[optional]* title for the file browser dialog window. Note that for clarity and security reasons, Fabric will automatically be prepend it with "Open File with read access to folder: " (*mode* = `IO.forOpen`), "Save File

with access to folder: " (*mode* = `IO.forSave`) or "Open File with read & write access to folder: " (*mode* = `IO.forOpenWithWriteAccess`).

- *extension* : *[optional]* default extension for the files to be opened or saved, for example: “*.jpeg”
- *defaultFileName* : *[optional]* default file name the file to be opened or saved, for example: “image.jpeg”
- *Returned value* : an `Object` containing the following members:
  - `.file` : member of type FabricFileHandle associated to the selected file , and containing a `folder` member of type FabricFileHandle associated to its containing folder. If *mode* = `IO.forSave`, the file might not have beeen created yet.
  - `.folder` : member of type FabricFileHandle associated to file's containing folder

4.3. IO.buildFileHandleFromRelativePath

```
FabricFileHandle buildFileHandleFromRelativePath(handleWithRelativePath);
```

Builds a file FabricFileHandle from a folder FabricFileHandle to which a relative path was appended. This method is mostly useful when FabricFileHandles are in a cryptic String form in the context of a secure model. The “writable” attribute of the folder handle gets propagated to the returned file handle. The method doesn't validate the existance of the file. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *handleWithRelativePath* : A FabricFileHandle to which a relative file path was appended, for example: `fabricio://aWpv1PBn6caEMVFEBAJ+YF7Jj34FamQc/images/portrait.jpeg`
- *Returned value* : a FabricFileHandle associated to the file of the provided relative path.

4.4. IO.buildFolderHandleFromRelativePath

```
FabricFileHandle buildFolderHandleFromRelativePath(handleWithRelativePath);
```

Builds a subfolder FabricFileHandle from a parent folder FabricFileHandle to which a relative path was appended. This method is mostly useful when FabricFileHandles are in a cryptic String form in the context of a secure model. The “writable” attribute of the parent folder handle gets propagated to the returned subfolder handle. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *handleWithRelativePath* : A FabricFileHandle to which a subfolder path was appended, for example: `fabricio://aWpv1PBn6caEMVFEBAJ+YF7Jj34FamQc/images`
- *Returned value* : a FabricFileHandle associated to the subfolder of the provided relative path.

4.5. IO.getFileHandleInfo

```
Object getFileHandleInfo(fabricFileHandle);
```

Returns an `Object` containing information associated to a FabricFileHandle. Note that, for security and privacy reasons, private information such as the full local file or folder path is not part of the returned object. Relative paths are supported. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *fabricFileHandle* : the FabricFileHandle for which the information is requested
- *Returned value* : an Object containing the following members:
  - .type : a String member indicating the type of the input FabricFileHandle. Its values can be “file”, “folder” or “unknown”. “unknown” will be returned if the input FabricFileHandle has a relative path pointing to a non-existing file or folder.
  - .writeAccess : a Boolean member indicating if the associated file or folder can be modified or written to
  - .exists : a Boolean member indicating if the associated file or folder currently exists in the local file system
  - .fileName : a String member indicating the name of the associated file (provided only if .type is “file”)
  - .fileSize : a String member indicating the size of the associated file (provided only if .type is “file” and .exists is True)

4.6. IO.getTextFileContent

```
String getTextFileContent(fabricFileHandle);
```

Reads the content of a file FabricFileHandle and returns it as a String. This method is not well suited for reading binary data. In the context of a dependency graph, binary data associated to a FabricFileHandle should rather be read using a ResourceLoadNode or the FabricFILESTREAM extension. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *fabricFileHandle* : A FabricFileHandle associated to the file to be read.
- *Returned value* : a String containing the source file content.

4.7. IO.putTextFileContent

```
putTextFileContent(fabricFileHandle,
                   content,
                   append);
```

Writes a String to the file associated to the input FabricFileHandle, by replacing the file content or be appending to it. Note that the input FabricFileHandle must have a “writable” attribute. Example: see Example 4.1, “Fabric.IO functions example”

Parameters

- *fabricFileHandle* : A FabricFileHandle associated to the file to be written.
- *content* : the String which will be written to the file
- *append* : a Boolean indicating if the *content* String is appended at the end of the file, else it will replace the file's content.

Chapter 5. KL Extensions for IO

Fabric extensions are always trusted code in the sense that they have the same access priviliges as the process in which Fabric is running. As such, they are given access to the private information associated to the FabricFileHandle, in particular the associated file or folder full path along with other attributes, through the EDK::FileHandleWrapper C++ helper class.

Fabric is built with the *FabricFILESTREAM* and *FabricFILESYSTEM* C++ extensions in order to enable runtime IO operations directly from KL. However, only the FileSTREAM extension is installed with the browser plugin Fabric client, since the FileSYSTEM extension would enable applications to bypass the browser's security model using KL. In cases where this is acceptable, such as for the installation of a private, trusted Fabric application, the FileSYSTEM extension can installed separately.

5.1. The FabricFILESTREAM Extension

The FabricFILESTREAM extension provides the ability to read and write to files directly from KL though a FabricFileStream object that is initialized from a FabricFileHandle. Additionally, this extension gives the ability to read and write compressed binary data. This extension is considered secure because it respects the read-write permissions associated with the FabricFileHandle. See the FabricFILESTREAM reference guide for more details about the provided functions.

5.2. The FabricFILESYSTEM Extension

The FabricFILESYSTEM extension enables browsing and modification of the local file system's folder hierarhchy. Because it enables full access to the local file system, this extension is not considered secure and is not installed by default with the Fabric browser plugin. However, it can be downloaded and installed separately. See the FabricFILESYSTEM reference guide for more details about the provided functions.