Example of a File System Client in PHP

This page presents a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The PHP code shown here hardly differs from the code you would write for an ordinary PHP program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local PHP object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs.

We now have seen enough of the client-side PHP mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

PHP

```
<?php
require 'Ice.php';
require 'Filesystem.php';

// Recursively print the contents of directory "dir"
// in tree fashion. For files, show the contents of
// each file. The "depth" parameter is the current
// nesting level (for indentation).

function listRecursive($dir, $depth = 0)
{
    $indent = str_repeat("\t", ++$depth);

    $contents = $dir->_list(); // list is a reserved word in PHP

    foreach ($contents as $i) {
        $dir = Filesystem_DirectoryPrxHelper::checkedCast($i);
        $file = Filesystem_FilePrxHelper::uncheckedCast($i);
        echo $indent . $i->name() . ($dir ? " (directory):" : " (file):") . "\n";
        if ($dir) {
            listRecursive($dir, $depth);
        } else {
```


Copyright © 2017, ZeroC, Inc.

```
            $text = $file->read();
            foreach ($text as $j)
                echo $indent . "\t" . $j . "\n";
        }
    }
}

$ic = null;
try
{
    // Create a communicator
    //
    $ic = Ice_initialize();

    // Create a proxy for the root directory
    //
    $obj = $ic->stringToProxy("RootDir:default -p 10000");

    // Down-cast the proxy to a Directory proxy
    //
    $rootDir = Filesystem_DirectoryPrxHelper::checkedCast($obj);

    // Recursively list the contents of the root directory
    //
    echo "Contents of root directory:\n";
    listRecursive($rootDir);
}
catch(Ice_LocalException $ex)
{
    print_r($ex);
}

if($ic)
{
    // Clean up
    //
    try
    {
        $ic->destroy();
    }
    catch(Exception $ex)
    {
        print_r($ex);
    }
}
?>
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main program follows. Let us look at the main program first:

1. The client first creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to the `Directory` interface and passes that proxy to `listRecursive`, which prints the contents of the file system.

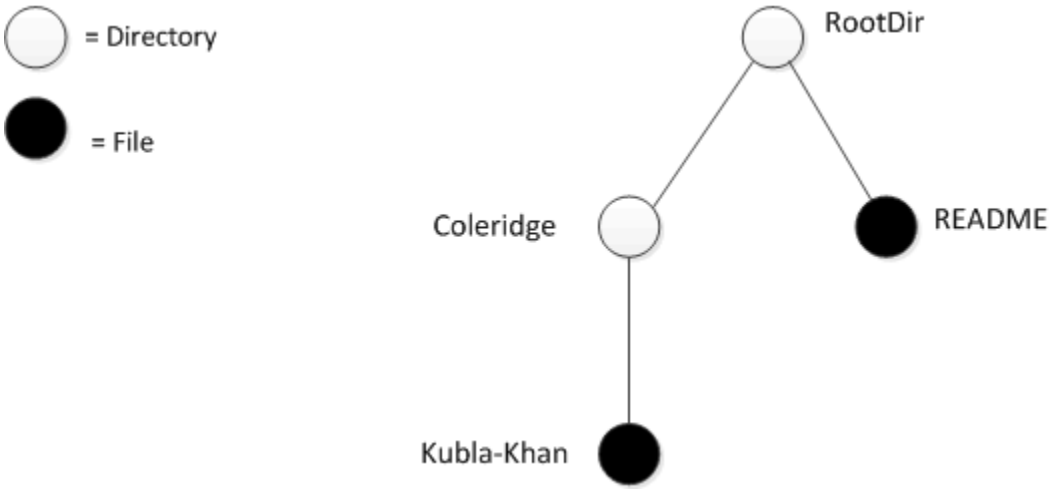
Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code uses `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, and uses `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the proxy returned by `checkedCast`; if `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, `uncheckedCast` is sufficient to get a `File` proxy.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use `uncheckedCast` instead of `checkedCast` because `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:

Copyright © 2017, ZeroC, Inc.

- If it is a directory, the code recurses, incrementing the indent level.
- If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:



A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client is not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in our discussions of IceGrid and object life cycle.

See Also

- Hello World Application
- Slice for a Simple File System
- Object Life Cycle
- IceGrid

Copyright © 2017, ZeroC, Inc.