Example of a File System Client in Java

This page presents the source code for a very simple client to access a server that implements the file system we developed in Slice for a Simple File System. The Java code hardly differs from the code you would write for an ordinary Java program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Java 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. This is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

We now have seen enough of the client-side Java 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:

Java

```
import Filesystem.*;

public class Client {

    // 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).

    static void
    listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for (int i = 0; i < contents.length; ++i) {
            DirectoryPrx subdir = DirectoryPrxHelper.checkedCast(contents[i]);
            FilePrx file = FilePrxHelper.uncheckedCast(contents[i]);
```

```
            System.out.println(indent + contents[i].name() +
                (subdir != null ? " (directory):" : " (file):"));
            if (subdir != null) {
                listRecursive(subdir, depth);
            } else {
                String[] text = file.read();
                for (int j = 0; j < text.length; ++j)
                    System.out.println(indent + "\t" + text[j]);
            }
        }
    }

    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            // Create a communicator
            //
            ic = Ice.Util.initialize(args);

            // Create a proxy for the root directory
            //
            Ice.ObjectPrx base = ic.stringToProxy("RootDir:default -p 10000");
            if (base == null)
                throw new RuntimeException("Cannot create proxy");

            // Down-cast the proxy to a Directory proxy
            //
            DirectoryPrx rootDir = DirectoryPrxHelper.checkedCast(base);
            if (rootDir == null)
                throw new RuntimeException("Invalid proxy");

            // Recursively list the contents of the root directory
            //
            System.out.println("Contents of root directory:");
            listRecursive(rootDir, 0);
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

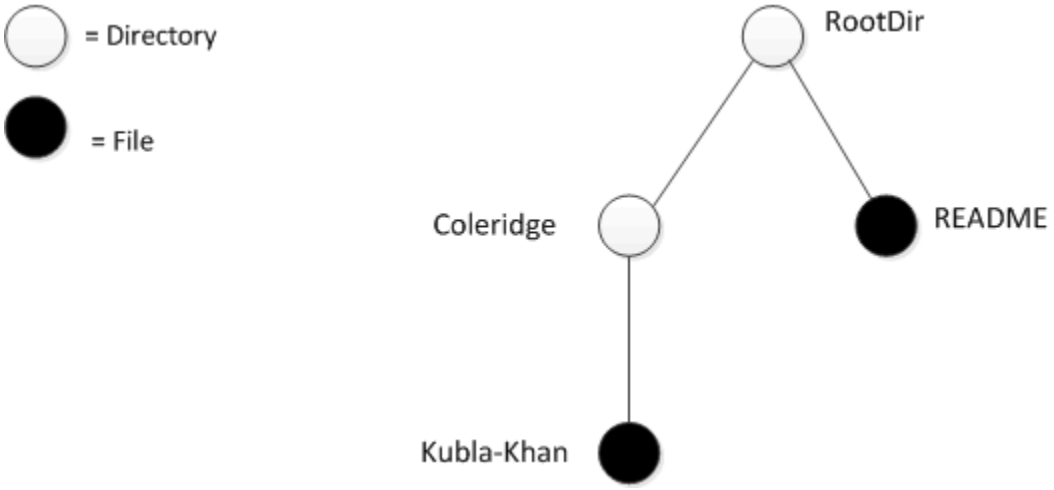
After importing the `Filesystem` package, the `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `main`, which is the main program. Let us look at `main` first:

1. The structure of the code in `main` follows what we saw in Hello World Application. After initializing the run time, the client 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 `DirectoryPrx` 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 does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `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 `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.
   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `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:
   - 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 two files and 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 (and server) are 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
- Example of a File System Server in Java
- Object Life Cycle
- IceGrid